Iterators Chapter Two

2.1 Chapter Overview

This chapter discusses the low-level implementation of iterators. Earlier, this text briefly discussed iterators and the FOREACH loop in the chapter on intermediate procedures (see “Iterators and the FOREACH Loop” on page 843). This chapter will review that information, discuss some uses for iterators, and then present the low-level implementation of this interesting control structure.

2.2 Review of Iterators

An iterator is a cross between a control structure and a function. Although common high level languages do not support iterators, they are present in some very high level languages[1]. Iterators provide a combination state machine/function call mechanism that lets a function pick up where it last left off on each new call. Iterators are also part of a loop control structure, with the iterator providing the value of the loop control variable on each iteration.

To understand what an iterator is, consider the following for loop from Pascal:

```
for I := 1 to 10 do <some statement>;
```

When learning Pascal you were probably taught that this statement initializes *i* with one, compares *i* with 10, and executes the statement if *i* is less than or equal to 10. After executing the statement, the *FOR* statement increments *i* and compares it with 10 again, repeating the process over and over again until *i* is greater than 10.

While this description is semantically correct, and indeed, it’s the way that most Pascal compilers implement the FOR loop, this is not the only point of view that describes how the for loop operates. Suppose, instead, that you were to treat the TO reserved word as an operator. An operator that expects two parameters (one and ten in this case) and returns the range of values on each successive execution. That is, on the first call the TO operator would return one, on the second call it would return two, etc. After the tenth call, the TO operator would fail which would terminate the loop. This is exactly the description of an iterator.

In general, an iterator controls a loop. Different languages use different names for iterator controlled loops, this text will just use the name FOREACH as follows:

```
foreach iterator() do
    statements;
endfor;
```

An iterator returns two values: a boolean success or failure value and a function result. As long as the iterator returns success, the FOREACH statement executes the statements comprising the loop body. If the iterator returns failure, the FOREACH loop terminates and executes the next sequential statement following the FOREACH loop’s body.

Iterators are considerably more complex than normal functions. A typical function call involves two basic operations: a call and a return. Iterator invocations involve four basic operations:

1) Initial iterator call
2) Yielding a value
3) Resumption of an iterator

---

[1] Ada and PL/I support very limited forms of iterators, though they do not support the type of iterators found in CLU, SETL, Icon, and other languages.

4) Termination of an iterator.

To understand how an iterator operates, consider the following short example:

```
iterator range( start:int32; stop:int32 );
begin range;

    forever

        mov( start, eax );
        breakif( eax > stop );
        yield();
        inc( start );

    endfor;

end range;
```

In HLA, iterator calls may only appear in the FOREACH statement. With the exception of the "yield();" statement above, anyone familiar with HLA should be able to figure out the basic logic of this iterator.

An iterator in HLA may return to its caller using one of two separate mechanisms, it can return to the caller by exiting through the "end Range;" statement or it may yield a value by executing the "yield();" statement. An iterator succeeds if it executes the "yield();" statement, it fails if it simply returns to the caller. Therefore, the FOREACH statement will only execute its corresponding statement if you exit an iterator with a "yield();". The FOREACH statement terminates if you simply return from the iterator. In the example above, the iterator returns the values *start..stop* via a "yield();" statement and then the iterator terminates. The loop

```
foreach Range(1,10) do

    stdout.put( (type uns32 eax ), nl );

endfor;
```

is comparable to the code:

```
for( mov( 1, eax ); eax <= 10; inc( eax )) do

    stdout.put( (type uns32 eax ), nl );

endfor;
```

When an HLA program first executes the FOREACH statement, it makes an initial call to the iterator. The iterator runs until it executes a "yield();" or it returns. If it executes the "yield();" statement, it returns the value in EAX as the iterator result and it succeeds. If it simply returns, the iterator returns failure and no iterator result. In the current example, the initial call to the iterator returns success and the value one.

Assuming a successful return (as in the current example), the FOREACH statement returns the current result in EAX and executes the FOREACH loop body. After executing the loop body, the FOREACH statement calls the iterator again. However, this time the FOREACH statement resumes the iterator rather than making an initial call. An iterator resumption continues with the first statement following the last "yield();" it executes. In the *range* example, a resumption would continue execution at the "inc(start);" statement. On the first resumption, the *range* iterator would add one to *start*, producing the value two. Two is less than ten (*stop's* value) so the FOREACH loop would repeat and the iterator would yield the value two. This process would repeat over and over again until the iterator yields ten. Upon resuming after yielding ten, the iterator would increment start to eleven and then return, rather than yield, since this new value is not less than or equal to ten. When the *Range* iterator returns (fails), the FOREACH loop terminates.

2.2.1 Implementing Iterators Using In-Line Expansion

The implementation of an iterator is rather complex. To begin with, consider a first attempt at an assembly implementation of the FOREACH statement above:

```
       push( 1 );    // Manually pass 1 and 10 as parameters.
       push( 10 );
       call Range_initial;
       jc Failure;
ForLp: stdout.put( (type uns32 eax), nl );
       call Range_Resume;
       jnc ForLp;
Failure:
```

Although this looks like a straight-forward implementation project, there are several issues to consider. First, the call to *Range_Resume* above looks simple enough, but there is no fixed address that corresponds to the resume address. While it is certainly true that this *Range* example has only one resume address, in general you can have as many "yield();" statements as you like in an iterator. For example, the following iterator returns the values 1, 2, 3, and 4:

```
iterator OneToFour;
begin OneToFour;

       mov( 1, eax ); yield();
       mov( 2, eax ); yield();
       mov( 3, eax ); yield();
       mov( 4, eax ); yield();

end OneToFour;
```

The initial call would execute the "mov(1, eax);" and "yield();" statements. The first resumption would execute the "mov(2, eax);" and "yield();" statements, the second resumption would execute "mov(3, eax);" and "yield();", etc. Obviously there is no single resume address the calling code can count on.

There are a couple of additional details left to consider. First, an iterator is free to call procedures and functions. If such a procedure or function executes the "yield();" statement then resumption by the FOREACH statement continues execution within the procedure or function that executed the "yield();"[2]. Second, the semantics of an iterator require all local variables and parameters to maintain their values until the iterator terminates. That is, yielding does not deallocate local variables and parameters. Likewise, any return addresses left on the stack (e.g., the call to a procedure or function that executes the "yield();" statement) must not be lost when a piece of code yields and the corresponding FOREACH statement resumes the iterator. In general, this means you cannot use the standard call and return sequence to yield from or resume to an iterator because you have to preserve the contents of the stack.

While there are several ways to implement iterators in assembly language, perhaps the most practical method is to have the iterator call the loop controlled by the iterator and have the loop return back to the iterator function. Of course, this is counter-intuitive. Normally, one thinks of the iterator as the function that the loop calls on each iteration, not the other way around. However, given the structure of the stack during the execution of an iterator, the counter-intuitive approach turns out to be easier to implement.

Some high level languages support iterators in exactly this fashion. For example, Metaware's Professional Pascal Compiler for the PC supports iterators[3]. Were you to create a Professional Pascal code sequence as follows:

```
iterator OneToFour:integer;
begin
   yield 1;
```

---

[2] This requires the use of nested procedures, a subject we will discuss in a later chapter.

[3] Obviously, this is a non-standard extension to the Pascal programming language provided in Professional Pascal.

```
        yield 2;
        yield 3;
        yield 4;
    end;
```

and call it in the main program as follows:

```
    for i in OneToFour do writeln(i);
```

Professional Pascal would completely rearrange your code. Instead of turning the iterator into an assembly language function and calling this function from within the FOR loop body, this code would turn the FOR loop body into a function, expand the iterator in-line (much like a macro) and call the FOR loop body function on each yield. That is, Professional Pascal would probably produce assembly language that looks something like the following:

```
// The following procedure corresponds to the for loop body
// with a single parameter (I) corresponding to the loop
// control variable:

procedure ForLoopCode( i:int32 ); nodisplay;
begin ForLoopCode;

    mov( i, eax );
    stdout.put( i, nl );

end ForLoopCode;

// The follow code would be emitted in-line upon encountering the
// for loop in the main program, it corresponds to an in-line
// expansion of the iterator as though it were a macro,
// substituting a call for the yield instructions:

        ForLoopCode( 1 );
        ForLoopCode( 2 );
        ForLoopCode( 3 );
        ForLoopCode( 4 );
```

This method for implementing iterators is convenient and produces relatively efficient (fast) code. It does, however, suffer from a couple drawbacks. First, since you must expand the iterator in-line wherever you call it, much like a macro, your program could grow large if the iterator is not short and you use it often. Second, this method of implementing the iterator completely hides the underlying logic of the code and makes your assembly language programs difficult to read and understand.

2.2.2 Implementing Iterators with Resume Frames

In-line expansion is not the only way to implement iterators. There is another method that preserves the structure of your program at the expense of a slightly more complex implementation. Several high level languages, including Icon and CLU, use this implementation.

To start with, you will need another stack frame: the *resume frame*. A resume frame contains two entries: a yield return address (that is, the address of the next instruction after the yield statement) and a *dynamic link*, that is a pointer to the iterator's activation record. Typically the dynamic link is just the value in the EBP register at the time you execute the yield statement. This version implements the four parts of an iterator as follows:

1) A CALL instruction for the initial iterator call,
2) A CALL instruction for the YIELD statement,
3) A RET instruction for the resume operation, and

4) A RET instruction to terminate the iterator.

To begin with, an iterator will require two return addresses rather than the single return address you would normally expect. The first return address appearing on the stack is the termination return address. The second return address is where the subroutine transfers control on a *yield* operation. The calling code must push these two return addresses upon initial invocation of the iterator. The stack, upon initial entry into the iterator, should look something like Figure 2.1.

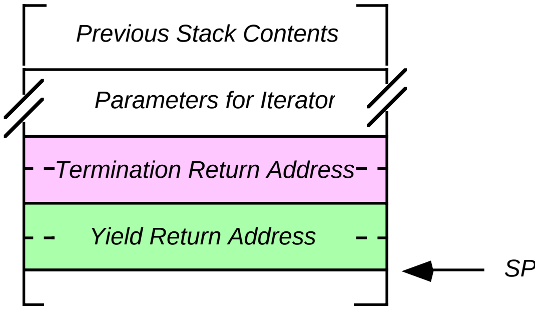

Figure 2.1 Iterator Activation Record

As an example, consider the *Range* iterator presented earlier. This iterator requires two parameters, a starting value and an ending value:

```
foreach range(1,10) do

    stdout.put( i, nl );

endfor;
```

The code to make the initial call to the *range* iterator, producing a stack like the one above, could be the following:

```
        push( 1 );        // Push start parameter value.
        push( 10 );       // Push stop parameter value.
        push( &ForDone);  // Push termination address.
        call range;       // Call the iterator.
            .
            .
            .
ForDone:
```

fordone is the first statement immediately following the FOREACH loop, that is, the instruction to execute when the iterator returns failure. The FOREACH loop body must begin with the first instruction following the call to *range*. At the end of the FOREACH loop, rather than jumping back to the start of the loop, or calling the iterator again, this code should just execute a RET instruction. The reason will become clear in a moment. So the implementation of the above FOREACH statement could be the following:

```
push( 1 );             // Push start parameter value.
push( 10 );            // Push stop parameter value.
push( &ForDone);       // Push termination address.
call range;            // Call the iterator.
push( ebp );           // Preserve iterator's ebp value.
mov( [esp+8], ebp );   // Get original EBP value passed to us by range.
stdout.put( i, nl );   // Display i's value.
```

```
        pop( ebp );             // Restore iterator's EBP value.
        ret(4 );                // Return and clean EBP value off stack.
ForDone:
```

Granted, this doesn't look anything at all like a loop. However, by playing some major tricks with the stack, you'll see that this code really does iterate the loop body (*stdout.put*) as intended.

Now consider the *range* iterator itself, here's the (mostly) low-level code to do the job:

```
iterator range( start:int32; stop:int32 ); @nodisplay; @noframe;
begin range;

    push( ebp );      // Standard Entry Sequence
    mov( esp, ebp );

    ForEverLbl:

        mov( start, eax );
        cmp( eax, stop );
        jng ForDone;

        yield();
        inc( start );
        jmp ForEverLbl;

    ForDone:
        pop( ebp );
        add( 4, esp );
        ret( 8 );

end range;
```

Although this routine is rather short, don't let its size deceive you; it's quite complex. The best way to describe how this iterator operates is to take it a few instructions at a time. The first two instructions are the standard entry sequence for a procedure. Upon execution of these two instructions, the stack looks like that in Figure 2.2.

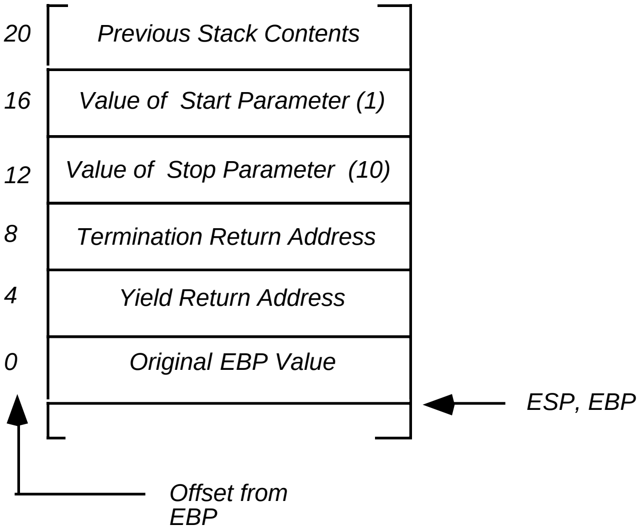

Figure 2.2 Range Activation Record

The next three statements in the *range* iterator, at label *ForEverLbl*, implement the termination test of the loop. When the *start* parameter contains a value greater than the *stop* parameter, control transfers to the *ForDone* label at which point the code pops the value of EBP off the stack, pops the success return address off the stack (since this code will not return back to the body of the iterator loop) and then returns via the termination return address that is immediately above the success return address on the stack. The return instruction also pops the two parameters (eight bytes) off the stack.

The real work of the iterator occurs in the body of the loop. The main question here is "what is this *yield* procedure and what is it doing?". To understand what *yield* is, we must consider what it is that *yield* does. Whenever the iterator executes the *yield* statement, it calls the body of the FOREACH loop that invoked the iterator. Since the body of the FOREACH loop is the first statement following the call to the iterator, it turns out that the iterator’s return address points at that body of code. Therefore, the *yield* statement does the unusual operation of calling a subroutine pointed at by the iterator’s (success) return address.

Simply calling the body of the FOREACH loop is not all the *yield* call must do. The body of the FOREACH loop is (probably) in a different procedure with its own activation record. That FOREACH loop body may very well access variables that are local to the procedure containing that loop body; therefore, the *yield* statement must also pass the original procedure’s EBP value as a parameter so that the loop body can restore EBP to point at the FOREACH loop body’s activation record (while, of course, preserving EBP’s value within the iterator itself). The caller’s EBP value (also known as the dynamic link) was the value the iterator pushes on the stack in the standard entry sequence. Therefore, [EBP+0] will point at this dynamic link value. To properly implement the yield operation, the iterator must emit the following code:

```
push( ebp );                // Save iterator’s activation record pointer.
call((type dword [ebp+4])); // Call the return address.
```

The PUSH and CALL instructions build the resume frame and then return control to the body of the FOREACH loop. The CALL instruction is not calling a subroutine. What it is really doing here is finishing off the resume frame (by storing the *yield* resume address into the resume frame) and then it returns control back to the body of the FOREACH loop by jumping indirect through the success return address pushed on the stack by the initial call to the iterator. After the execution of this call, the stack frame looks like that in Figure 2.3.

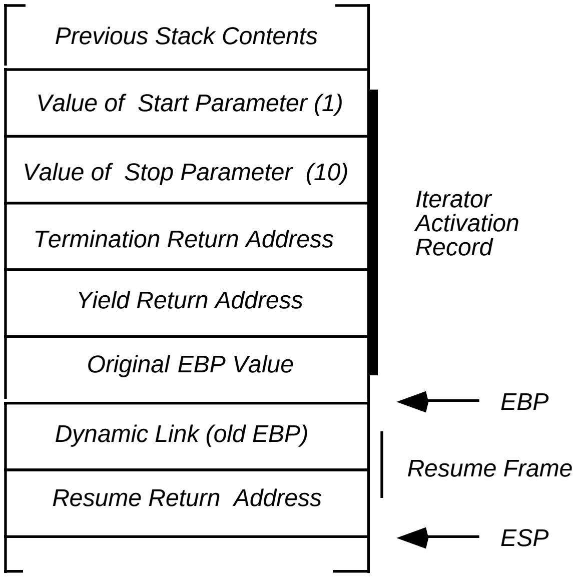

Figure 2.3 Range Resume Record

By convention, the EAX register contains the return value for the iterator. As with functions, EAX is a good place to return the iterator return result.

Immediately after yielding back to the FOREACH loop, the code must reload EBP with the original value prior to the iterator invocation. This allows the calling code to correctly access parameters and local variables in its own activation record rather than the activation record of the iterator. The FOREACH loop body begins by preserving EBP's value (the pointer to iterator's activation record) and then loading EBP with the value pushed on the stack by the *yield* statement. Of course, in this example reloading EBP isn't necessary because the body of the FOREACH loop does not reference any memory locations off the EBP register, but in general you will need to save EBP's value and load EBP with the value pushed on the stack by the yield statement.

At the end of the FOREACH loop body the "pop(ebp);" and "ret(4);" instructions restore EBP's value, cleans up the environment pointer passed as a parameter, and resumes the iterator. The RET instruction pops the return address off the stack which returns control back to the iterator immediately after the call emitted by the *yield* statement.

Of course, this is a lot of work to create a piece of code that simply repeats a loop ten times. A simple FOR loop would have been much easier and quite a bit more efficient that the FOREACH implementation described in this section. This section used the *range* iterator because it was easy to show how iterators work using *range*, not because actually implementing *range* as an iterator is a good idea.

Note that HLA does not provide an actual *yield* statement. If you look carefully at this code, and you think back to the last chapter, you will notice that the *yield* "statement" generates exactly the same code as a thunk invocation. Indeed, were you to dump the HLA symbol table when compiling a program containing the *range* iterator, you'd discover that *yield* is actually a local variable of type thunk within the *range* iterator code. The offset of this thunk is zero (from EBP) in the iterator's activation record (see Figure 2.4):

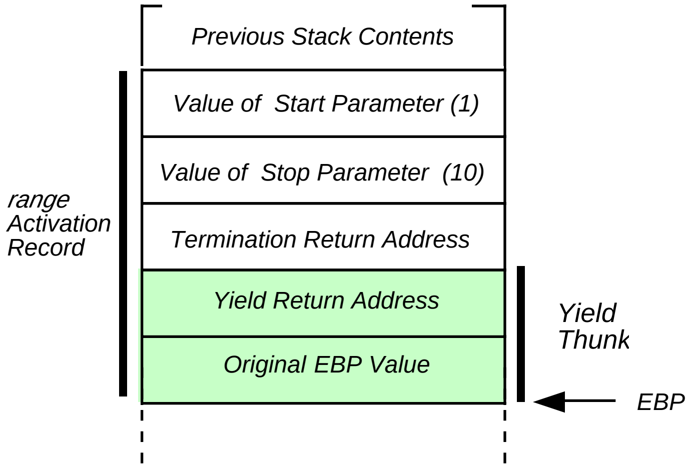

Figure 2.4 Yield Thunk in the *range* Iterator's Activation Record

This thunk value is somewhat unusual. If you look closely, you'll realize that this thunk's value is actually the return address the original call to *range* pushes along with the old EBP value that the *range* iterator code pushes as part of the standard entry sequence. In other words, the call to *range* and the standard entry sequence automatically initializes the *yield* thunk! How's that for elegance? All the HLA compiler has to do to create the *yield* thunk is to automatically create this "yield" symbol and associate *yield* with the address of the old EBP value that the *range* standard entry sequence pushes on the stack.

As you may recall from the chapter on Thunks, the calling sequence for a thunk is the following:

```
push( << Thunk's Environment Pointer>> );
call( << Thunk's Code Pointer >> );
```

In the case of the *yield* thunk in an iterator, the calling sequence looks like this:

```
pushd( [ebp] );               // Pass iterator caller's EBP as parameter.
call( (type dword [ebp+4] )); // Call the FOREACH loop's body.
```

The body of the FOREACH loop, like any thunk, must preserve EBP's value and load EBP with the value pushed on the stack in the code sequence above. Prior to returning (resuming) back to the iterator, the FOREACH loop body must restore the iterator's EBP value and it must remove the environment pointer parameter from the stack upon return. The code given earlier for the FOREACH loop does this:

```
// "FOREACH range( 1, 10) do"  statement:

    push( 1 );             // Push start parameter value.
    push( 10 );            // Push stop parameter value.
    push( &ForDone);       // Push termination address.
    call range;            // Call the iterator.

// FOREACH loop body (a thunk):

    push( ebp );           // Preserve iterator's ebp value.
    mov( [esp+8], ebp );   // Get original EBP value passed to us by range.
    stdout.put( i, nl );   // Display i's value.
    pop( ebp );            // Restore iterator's EBP value.
    ret(4 );               // Return and clean EBP value off stack.
    // endfor;

ForDone:
```

Of course, HLA does not make you write this low-level code. You can actually use a FOREACH statement in your program. The above code is the low-level implementation of the following high-level HLA code:

```
    foreach range( 1, 10 ) do

        stdout.put( i, nl );

    endfor;
```

The HLA compiler automatically emits the code to preserve and set up EBP at the beginning of the FOREACH loop's body; HLA also automatically emits the code to restore EBP and return to the iterator (removing the environment pointer parameter from the stack).

2.3 Other Possible Iterator Implementations

Thus far, this text has given two different implementations for iterators and the FOREACH loop. Although the resume frame/thunk implementation of the previous section is probably the most common implementation in HLA programs (since the HLA compiler automatically generates this type of code for FOREACH loops and iterators), don't get the impression that this is the only, or best, way to implement iterators and the FOREACH loop. Other possible implementations certainly exist and in some specialized situations some other implementation may offer some advantages. In this section we'll look at a couple of ways to implement iterators and the FOREACH loop.

The standard HLA implementation of iterators uses two separate return addresses for an iterator call: a success/yield address and a failure address. This organization is elegant given the thunk implementation of HLA's FOREACH statement, but there are other ways to return success/failure from an iterator. For example, you could use the value of the carry flag upon return from the iterator call to denote success or failure. Then a call to the iterator might take the following form:

```
ForEachLoopLbl:
        << Push any Necessary Parameters >>
        call iter;
        jc iterFails;

        << Code for the body of the iterator >>

        jmp ForEachLoopLbl;

iterFails:
```

One problem with this approach is that the code reenters the iterator on each iteration of the loop. This means it always passes the same parameters and reconstructs the activation record on each call of the iterator. Clearly, you cannot use this scheme if the iterator needs to maintain state information in its activation record between calls to the iterator. Furthermore, if the iterator yields from different points, then transferring control to the first statement after each yield statement will be a problem. There is a trick you can pull to tell the iterator whether this is the first invocation or some other invocation - pass a special parameter to indicate the first call of an iterator. You can do this as follows:

```
        pushd( 0 );   // Zero indicates the first call to iter.
ForEachLoopLbl:
        << Push Any Other Necessary Parameters >>
        call iter;      // iter is a standard HLA procedure, not an iterator.
        jc iterFails;
        << Code for the body of the iterator >>

        pushd( 1 ); // One indicates a re-entry into the iterator.
        jmp ForEachLoopLbl;

iterFails:
```

Notice how this code pushes a zero as the first parameter on the first call to *iter* and it pushes a one on each invocation thereafter.

What if the iterator needs to maintain state information (i.e., local variable values) between calls? Well, the easiest way to handle this using the current scheme is to pass extra parameters by value and use those parameters as the local variables. When the iterator returns success, it should not clean up the parameters on the stack, instead, it will leave them there for the next iteration of the "FOREACH" loop. E.g., consider the following implementation and invocation of the *ArithRange* iterator (this iterator returns the sum of all the values between the *start* and *stop* parameters):

```
// Note: we have to use a procedure, not an iterator, here because we don’t
// want HLA generating funny code for us.
//
// "sum" is actually a local variable in which we maintain state information.
// It must contain zero on the first entry to denote the intial entry into
// this code.

procedure ArithRange( start:uns32; stop:uns32; sum:uns32 );
    @nodisplay;  @noframe;
begin ArithRange;

    push( ebp );          // Standard entry sequence.
    mov( esp, ebp );


    mov( start, eax );
    if( eax <= stop ) then

        add( sum, eax );  // Compute arithmetic sum of values.
        mov( eax, sum );  // Save for the next time through and return in EAX.
        pop( ebp );       // Restore pointer to caller’s environment.
        clc;              // Indicate success on return.
        ret();            // Note that we don’t pop parameters!

    endif;
    pop( ebp );          // Restore pointer to caller’s environment.
    stc;                 // Indicate return on failure.
    ret( 12 );           // On failure, remove the parameters from the stack.

end ArithRange;
        .
        .
        .
        pushd( 1 );        // Pass the start parameter value.
        pushd( 10 );       // Pass the stop parameter value.
        pushd( 0 );        // Must pass zero in for sum value.
ForEachLoop:
        call ArithRange;   // Call our "iterator".
        jc ForEachDone;    // Quit on failure, fall through on success.

        << Foreach loop body code >>

        jmp ForEachLoop;
```

```
ForEachDone:
```

Notice how this code pushes the parameters on the stack for the first invocation of the ArithRange iterator, but it does not push the parameters on the stack on successive iterations of the loop. That's because the code does not remove these parameter values until it fails. Therefore, on each iteration of the loop, the parameter values left on the stack by the previous invocation of *ArithRange* are still on the stack for the next invocation. When the iterator fails, it pops the parameters off the stack so the stack is clean on exit from the "FOREACH" loop above.

If you can spare a register, there is a slightly more efficient way to implement iterators and the FOREACH loop (HLA doesn't use this scheme by default because it promise not to monkey with your register set in the code generation for the high level control structures). Consider the following code that HLA emits for a *yield* call:

```
push( [ebp] );        // Pass FOREACH loop's EBP value as a parameter.
call( [ebp+4] );      // Call the success address.
```

The FOREACH loop body code looks like the following:

```
push( ebp );          // Save iterator's EBP value.
mov( [esp+8], ebp );  // Fetch our EBP value pushed by the iterator.
<< loop body >>
pop( ebp );           // Restore iterator's EBP value.
ret( 4 );             // Resume the iterator and remove EBP value.
```

This code can be improved slightly by preserving and setting the EBP value within the iterator. Consider the following yield and FOREACH loop body code:

```
push( ebp );          // Save iterator's EBP value.
mov( [ebp+4], edx );  // Put success address into an available register.
mov( [ebp], ebp );    // Set up FOREACH loop's EBP value.
call edx;             // Call the success address.
pop( ebp );           // Restore our EBP value.
```

Here's the corresponding FOREACH loop body:

```
<< Loop Body >>
ret();
```

These scheme isn't amazingly better than the standard resume frame approach (indeed, it is about the same). But in some situations the fact that the loop body code doesn't have to mess with the stack may be important.

2.4 Breaking Out of a FOREACH Loop

The BREAK, BREAKIF, CONTINUE, and CONTINUEIF statements are active within a FOREACH loop, but there are some problems you must consider if you attempt to break out of a FOREACH loop with a BREAK or BREAKIF statement. In this section we'll look at the problems of prematurely leaving a FOREACH loop.

Keep in mind that an iterator leaves some information on the stack during the execution of the FOREACH loop body. Remember, the iterator doesn't return back to the FOREACH loop in order to execute the loop body; it actually calls the FOREACH loop body. That call leaves a resume frame plus all parameters, local variables, and other information in the iterator's activation record on the stack when it calls the FOREACH loop body. If you attempt to bail out of the FOREACH loop using BREAK, BREAKIF, or (worse still) a conditional or unconditional jump, ESP does not automatically revert back to the value prior to the execution of the FOREACH statement. The HLA generated code can only clean up the stack properly if the iterator returns via the failure address.

Although HLA cannot clean up the stack for you, it is quite possible for you to clean up the stack yourself. The easiest way to do this is to store the value in ESP to a local variable immediately prior to the execution of the FOREACH statement. Then simply reload ESP from this value prior to prematurely leaving the FOREACH loop. Here’s some code that demonstrates how to do this:

```
mov( esp, espSave );
foreach range( 1, 10 ) do

    << foreach loop body >>

    if( some_condition ) then

        mov( espSave, esp );
        break;

    endif;

    << more loop body code >>

endfor;
```

By restoring ESP from the *espSave* variable, this code removes all the activation record information from the stack prior to leaving the FOREACH loop body. Notice the MOV instruction immediately before the FOREACH statement that saves the stack position prior to calling the *range* iterator.

2.5 An Iterator Implementation of the Fibonacci Number Generator

Consider for a moment the Fibonacci number generator from the chapter on Thunks (this is the slow, non-thunk, implementation):

```
// Standard fibonacci function using the slow recursive implementation.

procedure slowfib( n:uns32 ); nodisplay; returns( "eax" );
begin slowfib;

    // For n= 1,2 just return 1.

    if( n <= 2 ) then

        mov( 1, eax );

    else

        // Return slowfib(n-1) + slowfib(n-2) as the function result:

        dec( n );
        slowfib( n );   // compute fib(n-1)
        push( eax );    // Save fib(n-1);

        dec( n );       // compute fib(n-2);
        slowfib( n );

        add( [esp], eax );  // Compute fib(n-1) [on stack] + fib(n-2) [in eax].
        add( 4, esp );      // Remove old value from stack.

    endif;
```

```
end slowfib;
```

Program 2.1 Recursive Implementation of the Fibonacci Number Generator

This particular function generates the nth Fibonacci number by computing the first through the nth the Fibonacci number. If you wanted to generate a sequence of Fibonacci numbers, you could use a FOR loop as follows:

```
for( mov( 1, ebx ); ebx < n; inc( ebx )) do

    slowfib( ebx );
    stdout.put( "Fib(", (type uns32 ebx), ") = ", (type uns32 eax), nl );

endfor;
```

True to its name, this implementation of *slowfib* runs quite slowly as *n* gets larger. The reason this function takes so much time (as pointed out in the chapter on Thunks) is that each recursive call recomputes all the previous Fibonacci numbers. Given the (n-1)st and (n-2)nd Fibonacci numbers, computing the nth Fibonacci number is trivial and very efficient – you simply add the previous two values together. The efficiency loss occurs when the recursive implementation computes the same value over and over again. The chapter on Thunks described how to eliminate this recomputation by passing the computed values to other invocations of the functions. This allows the Fibonacci function to compute the nth Fibonacci number in *n* units of time rather than 2^n units of time, a dramatic improvement. However, since each call to the (efficient) Fibonacci generator requires *n* units of time to compute its result, the loop above (which repeats n times) requires approximately n^2 units of time to run. This does not seem reasonable since it clearly takes only a few instructions to compute a new Fibonacci number given the previous two values; that is, we should be able to compute the nth Fibonacci number in *n* units of time. Iterators provide a trivial way to implement a Fibonacci number generator that generates a sequence of *n* Fibonacci numbers in *n* units of time. The following program demonstrates how to do this.

```
program fibIter;
#include( "stdlib.hhf" )

// Fibonocci function using a thunk to calculate fib(n-2)
// without making a recursive call.

procedure fib( n:uns32; nm2:thunk ); nodisplay; returns( "eax" );
var
    n2: uns32;      // A recursive call to fib stores fib(n-2) here.
    t:  thunk;      // This thunk actually stores fib(n-2) in n2.

begin fib;

    // Special case for n = 1, 2.  Just return 1 as the
    // function result and store 1 into the fib(n-2) result.

    if( n <= 2 ) then

        mov( 1, eax );  // Return as n-1 value.
        nm2();          // Store into caller as n-2 value.

    else

        // Create a thunk that will store the fib(n-2) value
        // into our local n2 variable.
        thunk   t :=
                #{
                    mov( eax, n2 );
                }#;

        mov( n, eax );
        dec( eax );
        fib( eax, t );  // Compute fib(n-1).

        // Pass back fib(n-1) as the fib(n-2) value to a previous caller.

        nm2();


        // Compute fib(n) = fib(n-1) [in eax] + fib(n-2) [in n2]:

        add( n2, eax );

    endif;

end fib;


// Standard fibonocci function using the slow recursive implementation.

procedure slowfib( n:uns32 ); nodisplay; returns( "eax" );
begin slowfib;

    // For n= 1,2 just return 1.

    if( n <= 2 ) then

        mov( 1, eax );

    else

        // Return slowfib(n-1) + slowfib(n-2) as the function result:

        dec( n );
        slowfib( n );   // compute fib(n-1)
        push( eax );    // Save fib(n-1);

        dec( n );       // compute fib(n-2);
        slowfib( n );

        add( [esp], eax );  // Compute fib(n-1) [on stack] + fib(n-2) [in eax].
        add( 4, esp );      // Remove old value from stack.

    endif;

end slowfib;


// FibNum-
//
//  Iterator that generates all the fibonacci numbers between 1 and n.

iterator FibNum( n:uns32 ); nodisplay;
var
    Fibn_1: uns32;      // Holds Fib(n-1) for a given n.
```

```
        Fibn_2: uns32;      // Holds Fib(n-2) for a given n.
        CurFib: uns32;      // Current index into fib sequence.

    begin FibNum;

        mov( 1, Fibn_1 );   // Initialize these guys upon initial entry.
        mov( 1, Fibn_2 );
        mov( 1, eax );      // Fib(0) = 1
        yield();
        mov( 1, eax );      // Fib(1) = 1;
        yield();
        mov( 2, CurFib );
        forever

            mov( CurFib, eax );     // Compute sequence up to the nth #.
            breakif( eax > n );
            mov( Fibn_2, eax );     // Compute this result.
            add( Fibn_1, eax );

            // Recompute the Fibn_1 and Fibn_2 values:

            mov( Fibn_1, Fibn_2 );
            mov( eax, Fibn_1 );

            // Return current value:

            yield();

            // Next value in sequence:

            inc( CurFib );

        endfor;

    end FibNum;



    var
        prevTime:dword[2];      // Used to hold 64-bit result from RDTSC instr.
        qw: qword;              // Used to compute difference in timing.
        dummy:thunk;            // Used in original calls to fib.

    begin fibIter;

        // "Do nothing" thunk used by the initial call to fib.
        // This thunk simply returns to its caller without doing
        // anything.

        thunk dummy := #{ }#;


        // Call the fibonocci routines to "prime" the cache:

        fib( 1, dummy );
        slowfib( 1 );
        foreach FibNum( 1 ) do
        endfor;
```

```
// Okay, compute the running times for the three fibonocci routines to
// generate a sequence of n fibonacci numbers where n ranges from
// 1 to 32:

for( mov( 1, ebx ); ebx < 32; inc( ebx )) do

    // Emit the index:

    stdout.put( (type uns32 ebx):2, stdio.tab );

    // Compute the # of cycles needed to compute the Fib via iterator:

    rdtsc();
    mov( eax, prevTime );
    mov( edx, prevTime[4] );

    foreach FibNum( ebx ) do

    endfor;

    rdtsc();
    sub( prevTime, eax );
    sbb( prevTime[4], edx );
    mov( eax, (type dword qw));
    mov( edx, (type dword qw[4]));

    stdout.putu64Size( qw, 4, ' ' );
    stdout.putc( stdio.tab );




    // Read the time stamp counter before calling fib:

    rdtsc();
    mov( eax, prevTime );
    mov( edx, prevTime[4] );

    for( mov( 1, ecx ); ecx <= ebx; inc( ecx )) do

        fib( ecx, dummy );

    endfor;

    // Read the timestamp counter and compute the approximate running
    // time of the current call to fib:

    rdtsc();
    sub( prevTime, eax );
    sbb( prevTime[4], edx );
    mov( eax, (type dword qw));
    mov( edx, (type dword qw[4]));

    // Display the results and timing from the call to fib:

    stdout.putu64Size( qw, 10, ' ' );
    stdout.putc( stdio.tab );


    // Okay, repeat the above for the slowfib implementation:
            rdtsc();
            mov( eax, prevTime );
            mov( edx, prevTime[4] );

            for( mov( 1, ecx ); ecx <= ebx; inc( ecx )) do

                slowfib( ebx );

            endfor;

            rdtsc();
            sub( prevTime, eax );
            sbb( prevTime[4], edx );
            mov( eax, (type dword qw));
            mov( edx, (type dword qw[4]));

            stdout.putu64Size( qw, 10, ' ' );
            stdout.newln();



        endfor;

    end fibIter;
```

Program 2.2 Fibonacci Iterator Example Program

The important concept here is that the *FibNum* iterator maintains its state across calls. In particular, it keeps track of the current iteration and the previous two Fibonacci values. Therefore, the iterator takes very little time to compute the result of each number in the sequence. This is far more efficient than either Fibonacci number generator from the chapter on Thunks, as the following table attests.

Table 1: CPU Cycle Times for Various Fibonacci Implementations

n	Iterator Implementation	Thunk Implementation	Recursive Implementation
1	156	233	98
2	148	98	77
3	178	221	271
4	193	376	399
5	213	509	879
6	213	712	1758
7	233	919	3493
8	252	1166	6531
9	271	1460	12568

Table 1: CPU Cycle Times for Various Fibonacci Implementations

n	Iterator Implementation	Thunk Implementation	Recursive Implementation
10	290	1759	22871
11	308	2139	40727
12	331	2503	71986
13	349	2938	126443
14	372	3341	220673
15	380	3877	382364
16	402	4326	660506
17	417	4977	1160660
18	452	5528	1954253
19	487	6222	3322819
20	479	6840	5685066
21	524	7691	9621772
22	545	8313	16339720
23	576	9292	27709571
24	599	10029	47036825
25	616	11274	80102556
26	650	12348	132583731
27	653	13172	222580780
28	683	14339	374788752
29	694	15394	627559062
30	722	16363	1054201515
31	732	17727	1756744511

2.6 Iterators and Recursion

It is completely possible, and sometimes very useful, to recursively call an iterator. In this section we’ll explore the syntax for this and present a couple of useful recursive iterators.

Although there is nothing stopping you from manually calling an iterator with the CALL instruction, the only valid (high level syntax) invocation of an iterator is via the FOREACH statement. Therefore, to recursively call an iterator, that iterator must contain a FOREACH loop to recursively call itself.

Iterators are especially useful for traversing tree and graph data structures. Some of the best (and most efficient) examples of recursive iterators are those that traverse such structures. Unfortunately, this text does not assume the prerequisite knowledge of such data structures, so it cannot use such examples to demonstrate recursive iterators. Nevertheless, it's worth mentioning this fact here because if you are familiar with graph traversal algorithms (or will be learning them in the future) you should consider using iterators for this purpose.

One useful iterator that doesn't require a tremendous amount of prerequisite knowledge is the traversal of a binary search tree implemented within an array. We won't go into the details of what a binary search tree is or why you would use it here other than to describe some properties of that tree. A binary search tree, implemented as an array, is a data structure that allows one to quickly search for some value within the structure. The values are arranged in the tree such that after each comparison you can eliminate half of the possible values with a single comparison. As a result, if the array contains *n* items, you can locate a particular item of interest in $\log_2 n$ units of time. To achieve this efficient search time, you have to arrange the data in the array in a particular fashion and then use a specific algorithm when searching through the tree. For the sake of our example, we'll assume that the data in the array is sorted (that is, a[0] < a[1] < a[2] < ... < a[n-1] for some definition of "less than"). The binary search algorithm then takes the following form:

```
1.   Set i = n
2.   Set j = i / 2  (integer/truncating division)
3.   Quit if i=j  (failed to find value in the search tree).
4.   Compare the key value (the one you're searching for) against a[i].
5.   If key > a[i] then set j = (i - j + 1)/2 + j
     else set i = j and then  j = i/2
6.   Go to step 3.
```

How this works and what it does is irrelevant here. What is important to this section is the arrangement of the data in the array that forms the binary search tree (specifically, the sorted nature of the data). Of course, if we wanted to generate a list of numbers in the sorted order, that would be especially trivial, all we would have to do is step through the array one element at a time. Suppose, however, that we wanted to generate a list of median values in this array. That is, the first value to generate would be the median of all the values, the second and third values would be the two median values of the array "slice" on either side of the original median. The next four values would be the medians of the array slices around the previous two medians and so on. If you're wondering what good such a sequence could be, well, were we to store this sequence into successive elements of an array, we could develop a binary search algorithm that is a little bit faster than the algorithm above (faster by some multiplicative constant). Hence, by running this iterator over the sorted data, we can come up with a slightly faster searching algorithm.

The following is the iterator that generates this particular sequence:

```
program RecIter;
#include( "stdlib.hhf" )

const
    MaxData := 17;      // # of data items to process.

type
    AryType: uns32[ MaxData ];

static

    // Here is the sorted data we'll process.  For simplicity, we'll just
    // fill the array with 1..MaxData at indices 0..MaxData-1.

    SortedData: AryType :=
        [
            // Fill this table with the values 1..MaxData:

            ?i := 1;
            #while( i < MaxData )

                i,
                ?i := i + 1;

            #endwhile
            i
        ];


/*******************************************************************/
/*                                                                 */
/* MedianVal iterator-                                             */
/*                                                                 */
/* Given a sorted array, this iterator yields the median value,    */
/* then it recursively yields a list of median values for the      */
/* array slice consisting of the array elements whose index is     */
/* less than the median value.  Finally, it yields the list of     */
/* median values for the array slice built from the array elements */
/* whose indices are greater than the median element.              */
/*                                                                 */
/* Inputs:                                                         */
/*  Ary-                                                           */
/*      The array whose elements we are to process.                */
/*                                                                 */
/*  start-                                                         */
/*      Starting index for the array slice.                        */
/*                                                                 */
/*  last-                                                          */
/*      Ending index (plus one) for the array slice.               */
/*                                                                 */
/* Yields:                                                         */
/*  A list of median values in EAX (one median value on            */
/*  each iteration of the corresponding FOREACH loop).             */
/*                                                                 */
/* Notes:                                                          */
/*  This iterator wipes out EBX.                                   */
/*                                                                 */
/*******************************************************************/


iterator MedianVal( var Ary: AryType; start:uns32; last:uns32 ); nodisplay;
var
    median: uns32;
begin MedianVal;

    mov( last, eax );
    sub( start, eax );
    if( @a ) then

        shr( 1, eax );              // Compute half the size.
        add( start, eax );          // Compute median index.
        mov( eax, median );         // Save for later.
```

```
            mov( Ary, ebx );            // Compute address of median element.
            mov( [ebx][eax*4], eax );   // Get median element.
            yield();


            // Recursively yield the medians for the elements at indices below
            // the element we just yielded.  Do this by recursively calling
            // the iterator to generate the list and then yield whatever value
            // the iterator returns.

            foreach MedianVal( Ary, start, median ) do

                yield();

            endfor;

            // Recursively yield the medians for the array slice whose indices
            // are greater than the current array element.  Note that we don't
            // include the median value itself in this list as we already
            // returned that value above.

            mov( median, eax );
            inc( eax );
            foreach MedianVal( Ary, eax, last ) do

                yield();

            endfor;

        endif;

    end MedianVal;


    // Main program that tests the functionality of this iterator.

    begin RecIter;

        foreach MedianVal( SortedData, 0, MaxData ) do

            stdout.put( "Value = ", (type uns32 eax), nl );

        endfor;

    end RecIter;
```

Program 2.3 Recursive Iterator to Rearrange Data for a Binary Search

The program above produces the following output:

```
Value = 9
Value = 5
Value = 3
Value = 2
Value = 1
Value = 4
Value = 7
Value = 6
Value = 8
Value = 14
Value = 12
Value = 11
Value = 10
Value = 13
Value = 16
Value = 15
Value = 17
```

2.7 Calling Other Procedures Within an Iterator

It is perfectly legal to call other procedures from an iterator. However, unless that procedure is nested within the iterator (see “Lexical Nesting” on page 1375), you cannot yield from that procedure using the iterator’s *yield* thunk unless you pass the thunk as a parameter to the other procedure. Other than the fact that you must remember that there is additional information on the stack, calling a procedure from an iterator is really no different than calling an iterator from any other procedure.

2.8 Iterators Within Classes

You can declare an iterator within a class. Iterators are called via the class’ virtual method table, just like methods. This means that you can override an iterator at run-time. See the chapter on classes for more details.

2.9 Putting It Altogether

This chapter provides a brief introduction to the low-level implementation of iterators and then discusses several different ways that you may use iterators in your programs. Although iterators are not as familiar as other program units, they are quite useful in many important situations. You should try to use iterators in your programs wherever appropriate, even if you’re not familiar with iterators from your high level language experiences.