Domain Specific Embedded Languages Chapter Nine

9.1 Chapter Overview

HLA's compile time language was designed with one purpose in mind: to give the HLA user the ability to change the syntax of the language in a user-defined manner. The compile-time language is actually so powerful that it lets you implement the syntax of other languages (not just an assembly language) within an HLA source file. This chapter discusses how to take this feature to an extreme and implement your own "mini-languages" within the HLA language.

9.2 Introduction to DSELs in HLA

One of the most interesting features of the HLA language is its ability to support *Domain Specific Embedded Languages* (or DSELs, for short, which you pronounce "D-cells"). A domain specific language is a language designed with a specific purpose in mind. Applications written in an appropriate domains specific language (DSL) are often much shorter and much easier to write than that same application written in a general purpose language (like C/C++, Java, or Pascal). Unfortunately, writing a compiler for a DSL is considerable work. Since most DSLs are so specific that few programs are ever written in them, it is generally cost-prohibitive to create a DSL for a given application. This economic fact has led to the popularity of domain specific *embedded* languages. The difference between a DSL and a DSEL is the fact that you don't write a new compiler for DSEL; instead, you provide some tools for use by an existing language translator to let the user extend the language as necessary for the specific application. This allows the language designer to use the features of the existing (i.e., *embedding*) language without having to write the translator for these features in the DSEL. The HLA language incorporates lots of features that let you extend the language to handle your own particular needs. This section discusses how to use these features to extend HLA as you choose.

As you probably suspect by now, the HLA compile-time language is the principle tool at your disposal for creating DSELs. HLA's multi-part macros let you easily create high level language-like control structures. If you need some new control structure that HLA does not directly support, it's generally an easy task to write a macro to implement that control structure. If you need something special, something that HLA's multi-part macros won't directly support, then you can write code in the HLA compile-time language to process portions of your source file as though they were simply string data. By using the compile-time string handling functions you can process the source code in just about any way you can imagine. While many such techniques are well beyond the scope of this text, it's reassuring to know that HLA can handle just about anything you want to do, even once you become an advanced assembly language programmer.

The following sections will demonstrate how to extend the HLA language using the compile-time language facilities. Don't get the idea that these simple examples push the limits of HLA's capabilities, they don't. You can accomplish quite a bit more with the HLA compile-time language; these examples must be fairly simple because of the assumed general knowledge level of the audience for this text.

9.2.1 Implementing the Standard HLA Control Structures

HLA supports a wide set of high level language-like control structures. These statements are not true assembly language statements, they are high level language statements that HLA compiles into the corresponding low-level machine instructions. They are general control statements, not "domain specific" (which is why HLA includes them) but they are quite typical of the types of statements one can add to HLA in order to extend the language. In this section we will look at how you could implement many of HLA's high-level control structures using the compile-time language. Although there is no real need to implement these statements in this manner, their example should provide a template for implementing other types of control structures in HLA.

The following sections show how to implement the FOREVER..ENDFOR, WHILE..ENDWHILE, and IF..ELSEIF..ELSE..ENDIF statements. This text leaves the REPEAT..UNTIL and BEGIN..EXIT..EXITIF..END statements as exercises. The remaining high level language control structures (e.g., TRY..ENDTRY) are a little too complex to present at this point.

Because words like "if" and "while" are reserved by HLA, the following examples will use macro identifiers like "_if" and "_while". This will let us create recognizable statements using standard HLA identifiers (i.e., no conflicts with reserved words).

9.2.1.1 The FOREVER Loop

The FOREVER loop is probably the easiest control structure to implement. After all, the basic FOREVER loop simply consists of a label and a JMP instruction. So the first pass at implementing _FOREVER.._ENDFOR might look like the following:

```
#macro _forever: topOfLoop;
    topOfLoop:

#terminator _endfor;
    jmp topOfLoop;

#endmacro;
```

Unfortunately, there is a big problem with this simple implementation: you'll probably want the ability to exit the loop via break and breakif statements and you might want the equivalent of a continue and continueif statement as well. If you attempt to use the standard BREAK, BREAKIF, CONTINUE, and CONTINUEIF statements inside this *_forever* loop implementation, you'll quickly discover that they do not work. Those statements are valid only inside an HLA loop and the *_forever* macro above is not an HLA loop. Of course, we could easily solve this problem by defining _FOREVER thusly:

```
#macro _forever;
    forever

#terminator _endfor;
    endfor;

#endmacro;
```

Now you can use BREAK, BREAKIF, CONTINUE, and CONTINUEIF inside the *_forever.._endfor* statement. However, this solution is ridiculous. The purpose of this section is to show you how you could create this statement were it not present in the HLA language. Simply renaming FOREVER to *_forever* is not an interesting solution.

Probably the best way to implement these additional statements is via KEYWORD macros within the *_forever* macro. Not only is this easy to do, but it has the added benefit of not allowing the use of these statements outside a *_forever* loop.

Implementing a *_continue* statement is very easy. Continue must transfer control to the first statement at the top of the loop. Therefore, the *_continue* #KEYWORD macro will simply expand to a single JMP instruction that transfers control to the *topOfLoop* label. The complete implementation is the following:

```
keyword _continue;
        jmp topOfLoop;
```

Implementing *_continueif* is a little bit more difficult because this statement must evaluate a boolean expression and decide whether it must jump to the *topOfLoop* label. Fortunately, the HLA JT (jump if true) pseudo-instruction makes this a relatively trivial task. The JT pseudo-instruction expects a boolean expression (the same that CONTINUEIF allows) and transfers control to the corresponding target label if the result of the expression evaluation is true. The *_continueif* implementation is nearly trivial with JT:

```
keyword _continueif( ciExpr );
       JT( ciExpr ) topOfLoop;
```

You will implement the *_break* and _breakif #KEYWORD macros in a similar fashion. The only difference is that you must add a new label just beyond the JMP in the *_endfor* macro and the break statements should jump to this local label. The following program provides a complete implementation of the *_forever.._endfor* loop as well as a sample test program for the *_forever* loop.

```
/************************************************/
/*                                              */
/* foreverMac.hla                               */
/*                                              */
/* This program demonstrates how to use HLA's   */
/* "context-free" macros, along with the JT     */
/* "medium-level" instruction to create         */
/* the FOREVER..ENDFOR, BREAK, BREAKIF,         */
/* CONTINUE, and CONTINUEIF control statements. */
/*                                              */
/************************************************/




program foreverDemo;
#include( "stdlib.hhf" )



    // Emulate the FOREVER..ENDFOR loop here, plus the
    // corresponding CONTINUE, CONTINUEIF, BREAK, and
    // BREAIF statements.


    macro _forever:foreverLbl, foreverbrk;

        // Target label for the top of the
        // loop.  This is also the destination
        // for the _continue and _continueif
        // macros.

        foreverLbl:


    // The _continue and _continueif statements
    // transfer control to the label above whenever
    // they appear in a _forever.._endfor statement.
    // (Of course, _continueif only transfers control
    // if the corresponding boolean expression evaluates
    // true.)

    keyword _continue;
        jmp foreverLbl;

    keyword _continueif( cifExpr );
        jt( cifExpr ) foreverLbl;
    // the _break and _breakif macros transfer
    // control to the "foreverbrk" label which
    // is at the bottom of the loop.

    keyword _break;
        jmp foreverbrk;

    keyword _breakif( bifExpr );
        jt( bifExpr ) foreverbrk;


    // At the bottom of the _forever.._endfor
    // loop this code must jump back to the
    // label at the top of the loop.  The
    // _endfor terminating macro must also supply
    // the target label for the _break and _breakif
    // keyword macros:

    terminator _endfor;
        jmp foreverLbl;
        foreverbrk:

    endmacro;




begin foreverDemo;

    // A simple main program that demonstrates the use of the
    // statements above.

    mov( 0, ebx );
    _forever

        stdout.put( "Top of loop, ebx = ", (type uns32 ebx), nl );
        inc( ebx );

        // On first iteration, skip all further statements.

        _continueif( ebx = 1 );

        // On fourth iteration, stop.

        _breakif( ebx = 4 );

        _continue;  // Always jumps to top of loop.
        _break;     // Never executes, just demonstrates use.

    _endfor;


end foreverDemo;
```

Program 9.1 Macro Implementation of the FOREVER..ENDFOR Loop

9.2.1.2 The WHILE Loop

Once the FOREVER..ENDFOR loop is behind us, implementing other control structures like the WHILE..ENDWHILE loop is fairly easy. Indeed, the only notable thing about implementing the *_while.._endwhile* macros is that the code should implement this control structure as a REPEAT..UNTIL statement for efficiency reasons. The implementation appearing in this section takes a rather lazy approach to implementing the DO reserved word. The following code uses a #KEYWORD macro to implement a "_do" clause, but it does not enforce the (proper) use of this keyword. Instead, the code simply ignores the *_do* clause wherever it appears between the *_while* and *_endwhile*. Perhaps it would have been better to check for the presence of this statement (not to difficult to do) and verify that it immediately follows the *_while* clause and associated expression (somewhat difficult to do), but this just seems like a lot of work to check for the presence of an irrelevant keyword. So this implementation simply ignores the *_do*. The complete implementation appears in Program 9.2:

```
/************************************************/
/*                                              */
/* whileMacs.hla                                */
/*                                              */
/* This program demonstrates how to use HLA's   */
/* "context-free" macros, along with the JT and */
/* JF "medium-level" instructions to create     */
/* the basic WHILE statement.                   */
/*                                              */
/************************************************/


program whileDemo;
#include( "stdlib.hhf" )


    // Emulate the while..endwhile loop here.
    //
    // Note that this code implements the WHILE
    // loop as a REPEAT..UNTIL loop for efficiency
    // (though it inserts an extra jump so the
    // semantics remain the same as the WHILE loop).


    macro _while( whlexpr ): repeatwhl, whltest, brkwhl;

        // Transfer control to the bottom of the loop
        // where the termination test takes place.

        jmp whltest;

        // Emit a label so we can jump back to the
        // top of the loop.

        repeatwhl:

    // Ignore the "_do" clause.  Note that this
    // macro should really check to make sure
    // that "_do" follows the "_while" clause.
    // But it's not semantically important so
    // this code takes the lazy way out.

    keyword _do;
```

```
        // If we encounter "_break" inside this
        // loop, transfer control to the first statement
        // beyond the loop.

        keyword _break;
            jmp brkwhl;

        // Ditto for "_breakif" except, of course, we
        // only exit the loop if the corresponding
        // boolean expression evaluates true.

        keyword _breakif( biwExpr );
            jt( biwExpr ) brkwhl;

        // The "_continue" and "_continueif" statements
        // should transfer control directly to the point
        // where this loop tests for termination.

        keyword _continue;
            jmp whltest;

        keyword _continueif( ciwExpr );
            jt( ciwExpr ) whltest;


        // The "_endwhile" clause does most of the work.
        // First, it must emit the target label used by the
        // "_while", "_continue", and "_continueif" clauses
        // above.  Then it must emit the code that tests the
        // loop termination condition and transfers control
        // to the top of the loop (the "repeatwhl" label)
        // if the expression evaluates false.  Finally,
        // this code must emit the "brkwhl" label the "_break"
        // and "_breakif" statements reference.


        terminator _endwhile;

            whltest:
            jt( whlexpr ) repeatwhl;
            brkwhl:

        endmacro;



    begin whileDemo;


        // Quick demo of the _while statement.
        // Note that the _breakif in the nested
        // _while statement only skips the
        // inner-most _while, just as you should expect.

        mov( 0, eax );
        _while( eax < 10 ) _do

            stdout.put( "eax in loop = ", eax, " ebx=" );
            inc( eax );
            mov( 0, ebx );
            _while( ebx < 4 ) _do

                stdout.puti32( ebx );
                _breakif( ebx = 3 );
                stdout.put( ", " );
                inc( ebx );

            _endwhile;
            stdout.newln();

            _continueif( eax = 5 );
            _breakif( eax = 8 );
            _continue;
            _break;

        _endwhile

    end whileDemo;
```

Program 9.2 Macro Implementation of the WHILE..ENDWHILE Loop

9.2.1.3 The IF Statement

Simulating the HLA IF..THEN..ELSEIF..ELSE..ENDIF statement using macros is a little bit more involved than the simulation of FOREVER or WHILE. The semantics of the ELSEIF and ELSE clauses complicate the code generation and require careful thought. While it is easy to write #KEYWORD macros for *_elseif* and *_else*, ensuring that these statements generate correct (and efficient) code is another matter altogether.

The basic *_if.._endif* statement, without the *_elseif* and *_else* clauses, is very easy to implement (even easier than the *_while.._endwhile* loop of the previous section). The complete implementation is

```
#macro _if( ifExpr ): onFalse;

    jf( ifExpr ) onFalse;

#keyword _then;  // Just ignore _then.

#terminator _endif;

    onFalse:

#endmacro;
```

This macro generates code that tests the boolean expression you supply as a macro parameter. If the expression evaluates false, the code this macro emits immediately jumps to the point just beyond the *_endif* terminating macro. So this is a simple and elegant implementation of the IF..ENDIF statement, assuming you don't need an ELSE or ELSEIF clause.

Adding an ELSE clause to this statement introduces some difficulties. First of all, we need some way to emit the target label of the JF pseudo-instruction in the *_else* section if it is present and we need to emit this label in the terminator section if the *_else* section is not present.

A related problem is that the code after the _if clause must end with a JMP instruction that skips the *_else* section if it is present. This JMP must transfer control to the same location as the current *onFalse* label.

Another problem that occurs when we use #KEYWORD macros to implement the *_else* clause, is that we need some mechanism in place to ensure that at most one invocation of the *_else* macro appears in a given *_if.._endif* sequence.

We can easily solve these problems by introducing a compile-time variable (i.e., VAL object) into the macro. We will use this variable to indicate whether we've seen an *_else* section. This variable will tell us if we have more than one *_else* clause (which is an error) and it will tell us if we need to emit the onFalse label in the *_endif* macro. A reasonable implementation might be the following:

```
#macro _if( ifExpr ): onFalse, ifDone, hasElse;

    ?hasElse := False;  // Haven't seen an _else clause yet.

    jf( ifExpr ) onFalse;

#keyword _then;  // Just ignore _then.

#keyword _else;

    // Check to see if this _if statement already has an _else clause:

    #if( hasElse )

        #error( "Only one _else clause is legal in an _if statement' )

    #endif

    ?hasElse := true;  //Let the world know we've see an _else clause.

    // Since we've just encountered the _else clause, we've just finished
    // processing the statements in the _if section.  The first thing we
    // need to do is emit a JMP instruction that will skip around the
    // _else statements (so the _if section doesn't fall in to the
    // _else code).

        jmp ifDone;

    // Okay, emit the onFalse label here so a false expression will transfer
    // control to the _else statements:

    onFalse:

#terminator _endif;

    // If there was no _else section, we must emit the onFalse label
    // so that the former JF instruction has a  proper destination.
    // If an _else section was present, we cannot emit this label
    // (since the _else code has already done so) but we must emit
    // the ifDone label.

    #if( hasElse )

        ifdone:

    #else

        onFalse:

    #endif

#endmacro;
```

Adding the *_elseif* clause to the *_if.._endif* statement complicates things considerably. The problem is that *_elseif* can appear zero or more times in an *_if* statement and each occurrence needs to generate a unique *onFalse* label. Worse, if at least one _elseif clause appears in the sequence, then the JF instruction in the *_if* clause must transfer control to the first *_elseif*, not to the *_else* clause. Also, the last _elseif clause must transfer control to the *_else* clause (or to the first statement beyond the *_endif* clause) if its expression evaluates false. A straight-forward implementation just isn't going to work here.

A clever solution is to create a string variable that contains the name of the previous JF target label. Whenever you encounter an *_elseif* or an *_else* clause you simply emit this string to the source file as the target label. Then the only trick is "how do we generate a unique label whenever we need one?". Well, let's suppose that we have a string that is unique on each invocation of the *_if* macro. This being the case, we can generate a (source file wide) unique string by concatenating a counter value to the end of this base string. Each time we need a unique string, we simply bump the value of the counter up by one and create a new string. Consider the following macro:

```
#macro genLabel( base, number );

    @text( base + string( number ));

#endmacro;
```

If the *base* parameter is a string value holding a valid HLA identifier and the *number* parameter is an integer numeric operand, then this macro will emit a valid HLA identifier that consists of the *base* string followed by a string representing the numeric constant. For example, 'genLabel("Hello", 52)' emits the label *Hello52*. Since we can easily create an *uns32* VAL object inside our *_if* macro and increment this each time we need a unique label, the only problem is to generate a unique base string on each invocation of the *_if* macro. Fortunately, HLA already does this for us.

Remember, HLA converts all local macro symbols to a unique identifier of the form "_xxxx_" where xxxx represents some four-digit hexadecimal value. Since local symbols are really nothing more than text constants initialized with these unique identifier strings, it's very easy to obtain an unique string in a macro invocation- just declare a local symbol (or use an existing local symbol) and apply the @STRING: operator to it to extract the unique name as a string. The following example demonstrates how to do this:

```
#macro uniqueIDs: counter, base;

    ?counter := 0;          // Increment this for each unique symbol you need.
    ?base := @string:base;  // base holds the base name to use.
        .
        .
        .

    // Generate a unique label at this point:

    genLabel( base, counter ):  // Notice the colon. We're defining a
    ?counter := counter + 1;    // label at this point!
        .
        .
        .
    genLabel( base, counter ):
    ?counter := counter + 1;
        .
        .
        .
    etc.

#endmacro;
```

Once we have the capability to generate a sequence of unique labels throughout a macro, implementing the *_elseif* clause simply becomes the task of emitting the last referenced label at the beginning of each _elseif (or _else) clause and jumping if false to the next unique label in the series. Program 9.3 implements the *_if.._then.._elseif.._else.._endif* statement using exactly this technique.

```
/*************************************************/
/*                                               */
/* IFmacs.hla                                    */
/*                                               */
/* This program demonstrates how to use HLA's    */
/* "context-free" macros, along with the JT and  */
/* JF "medium-level" instructions to create      */
/* an IF statement.                              */
/*                                               */
/*************************************************/

program IFDemo;
#include( "stdlib.hhf" )

    // genlabel-
    //
    // This macro creates an HLA-compatible
    // identifier of the form "_xxxx_n" where
    // "_xxxx_" is the string associated with
    // the "base" parameter and "n" represents
    // some numeric value that the caller.  The
    // combination of the base and the n values
    // will produce a unique label in the
    // program if base's string is unique for
    // each invocation of the "_if" macro.

    macro genLabel( base, number );

        @text( base + string( number ))

    endmacro;


    /*
    ** Emulate the if..elseif..else..endif statement here.
    */

    macro _if( ifexpr ):elseLbl, ifDone, hasElse, base;

        // This macro must create a unique ID string
        // in base.  One sneaky way to do this is
        // to use the converted name HLA generates
        // for the "base" object (this is generally
        // a string of the form "_xxxx_" where "xxxx"
        // is a four-digit hexadecimal value).

        ?base := @string:base;

        // This macro may need to generate a large set
        // of different labels (one for each _elseif
        // clause).  This macro uses the elseLbl
        // value, along with the value of "base" above,
        // to generate these unique labels.

        ?elseLbl := 0;
        // hasElse determines if we have an _else clause
        // present in this statement.  This macro uses
        // this value to determine if it must emit a
        // final else label when it encounters _endif.

        ?hasElse := false;


        // For an IF statement, we must evaluate the
        // boolean expression and jump to the current
        // else label if the expression evaluates false.

        jf( ifexpr ) genLabel( base, elseLbl );


    // Just ignore the _then keyword.
    // A slightly better implementation would require
    // this keyword, the current implementation lets
    // you write an "_if" clause without the "_then"
    // clause.  For that matter, the current implementation
    // lets you arbitrarily sprinkle "_then" clauses
    // throughout the "_if" statement; we will ignore
    // this for this example.

    keyword _then;


    // Handle the "_elseif" clause here.

    keyword _elseif(elsex);

            // _elseif clauses are illegal after
            // an _else clause in the statement.
            // Enforce that here.

            #if( hasElse )

                #error( "Unexpected '_elseif' clause" )

            #endif

            // We've just finished the "_if" clause
            // or a previous "_elseif" clause.  So
            // the first thing we have to do is jump
            // to the code just beyond this "_if"
            // statement.

            jmp ifDone;


            // Okay, this is where the previous "_if" or
            // "_elseif" statement must jump if its boolean
            // expression evaluates false.  Emit the target
            // label.  Next, because we're about to jump
            // to our own target label, bump up the elseLbl
            // value by one to prevent jumping back to the
            // label we're about to emit.  Finally, emit
            // the code that tests the boolean expression and
            // transfers control to the next _elseif or _else
            // clause if the result is false.
```

```
                genLabel( base, elseLbl ):
                    ?elseLbl := elseLbl+1;
                    jf(elsex) genLabel( base, elseLbl );


            keyword _else;

                    // Only allow a single "_else" clause in this
                    // "_if" statement:

                    #if( hasElse )

                        #error( "Unexpected '_else' clause" )

                    #endif


                    // As above, we've just finished the previous "_if"
                    // or "_elseif" clause, so jump directly to the end
                    // of the "_if" statement.

                    jmp ifDone;

                    // Okay, emit the current 'else' label so that
                    // the failure of the previous "_if" or "_elseif"
                    // test will transfer control here.  Also set
                    // 'hasElse' to true to catch additional "_elseif"
                    // and "_else" clauses.

                genLabel( base, elseLbl ):
                    ?hasElse := true;



            terminator _endif;

                // At the end of the _if statement we must emit the
                // destination label that the _if and _elseif sections
                // jump to.  Also, if there was no _else section, this
                // code has to emit the last deployed else label.

                ifDone:
                #if( !hasElse )

                    genLabel( base, elseLbl ):

                #endif

            endmacro;


        begin IFDemo;

            // Quick demo of the use of the above statements.

            for( mov( 0, eax ); eax < 5; inc( eax )) do

                _if( eax = 0 ) _then

                    stdout.put( "in _if statement" nl );
                _elseif( eax = 1 ) _then

                    stdout.put( "in first _elseif clause" nl );

                _elseif( eax = 2 ) _then

                    stdout.put( "in second _elseif clause" nl );

                _else

                    stdout.put( "in _else clause" nl );
                    _if( eax > 3 ) _then

                        stdout.put( "in second _if statement" nl );

                    _endif;

                _endif;

            endfor;


        end IFDemo;
```

Program 9.3 Macro Implementation of the IF..ENDIF Statement

9.2.2 The HLA SWITCH/CASE Statement

HLA doesn’t support a selection statement (SWITCH or CASE statement). Instead, HLA’s SWITCH..CASE..DEFAULT..ENDSWITCH statement exists only as a macro in the HLA Standard Library HLL.HHF file. This section discusses HLA’s macro implementation of the SWITCH statement.

The SWITCH statement is very complex so it should come as no surprise that the macro implementation is long, involved, and complex. The example appearing in this section is slightly simplified over the standard HLA version, but not by much. This discussion assumes that you’re familiar with the low-level implementation of the SWITCH..CASE..DEFAULT..ENDSWITCH statement. If you are not comfortable with that implementation, or feel a little rusty, you may want to take another look at “SWITCH/CASE Statements” on page 776 before attempting to read this section. The discussion in this section is somewhat advanced and assumes a fair amount of programming skill. If you have trouble following this discussion, you may want to skip this section until you gain some more experience.

There are several different ways to implement a SWITCH statement. In this section we will assume that the *_switch.._endswitch* macro we are writing will implement the SWITCH statement using a jump table. Implementation as a sequence of *if..elseif* statements is fairly trivial and is left as an exercise. Other schemes are possible as well, this section with not consider them.

A typical SWITCH statement implementation might look like the following:

```
readonly
    JmpTbl:dword[3] := [ &Stmt5, &Stmt6, &Stmt7 ];
         .
         .
         .
```

```
// switch( i )

    mov( i, eax );              // Check to see if "i" is outside the range
    cmp( eax, 5 );              // 5..7 and transfer control directly to the
    jb EndCase                  // DEFAULT case if it is.
    cmp( eax, 7 );
    ja EndCase;
    jmp( JmpTbl[ eax*4 - 5*@size(dword)] );


// case( 5 )
        Stmt5:
            stdout.put( "I=5" );
            jmp EndCase;

// Case( 6 )
        Stmt6:
            stdout.put( "I=6" );
            jmp EndCase;

// Case( 7 )
        Stmt7:
            stdout.put( "I=7" );

    EndCase:
```

If you study this code carefully, with an eye to writing a macro to implement this statement, you'll discover a couple of major problems. First of all, it is exceedingly difficult to determine how many cases and the range of values those cases cover before actually processing each CASE in the SWITCH statement. Therefore, it is really difficult to emit the range check (for values outside the range 5..7) and the indirect jump before processing all the cases in the SWITCH statement. You can easily solve this problem, however, by moving the checks and the indirect jump to the bottom of the code and inserting a couple of extra JMP instructions. This produces the following implementation:

```
readonly
    JmpTbl:dword[3] := [ &Stmt5, &Stmt6, &Stmt7 ];
         .
         .
         .

// switch( i )

    jmp DoSwitch;              // First jump inserted into this code.

// case( 5 )
        Stmt5:
            stdout.put( "I=5" );
            jmp EndCase;

// Case( 6 )
        Stmt6:
            stdout.put( "I=6" );
            jmp EndCase;

// Case( 7 )
        Stmt7:
            stdout.put( "I=7" );
            jmp EndCase;        // Second jump inserted into this code.

DoSwitch:                      // Insert this label and move the range
    mov( i, eax );              // checks and indirect jump down here.
    cmp( eax, 5 );
    jb EndCase
    cmp( eax, 7 );
    ja EndCase;
    jmp( JmpTbl[ eax*4 - 5*@size(dword)] );

// All the cases (including the default case) jump down here:

EndCase:
```

Since the range check code appears after all the cases, the macro can now process those cases and easily determine the bounds on the cases by the time it must emit the CMP instructions above that check the bounds of the SWITCH value. However, this implementation still has a problem. The entries in the *JmpTbl* table refer to labels that can only be determined by first processing all the cases in the SWITCH statement. Therefore, a macro cannot emit this table in a READONLY section that appears earlier in the source file than the SWITCH statement. Fortunately, HLA lets you embed data in the middle of the code section using the READONLY..ENDREADONLY and STATIC..ENDSTATIC directives[1]. Taking advantage of this feature allows use to rewrite the SWITCH implementation as follows:

```
// switch( i )

    jmp DoSwitch;              // First jump inserted into this code.

// case( 5 )
        Stmt5:
            stdout.put( “I=5” );
            jmp EndCase;

// Case( 6 )
        Stmt6:
            stdout.put( “I=6” );
            jmp EndCase;

// Case( 7 )
        Stmt7:
            stdout.put( “I=7” );
            jmp EndCase;         // Second jump inserted into this code.

DoSwitch:                      // Insert this label and move the range
    mov( i, eax );             // checks and indirect jump down here.
    cmp( eax, 5 );
    jb EndCase
    cmp( eax, 7 );
    ja EndCase;
    jmp( JmpTbl[ eax*4 - 5*@size(dword)] );

// All the cases (including the default case) jump down here:

EndCase:

readonly
    JmpTbl:dword[3] := [ &Stmt5, &Stmt6, &Stmt7 ];
endreadonly;
```

HLA’s macros can produce code like this when processing a SWITCH macro. So this is the type of code we will generate with a *_switch.._case.._default.._endswitch* macro.

Since we’re going to need to know the minimum and maximum case values (in order to generate the appropriate operands for the CMP instructions above), the *_case* #KEYWORD macro needs to compare the

1. HLA actually moves the data to the appropriate segment in memory, the data is not stored directly in the CODE section.

current case value(s) against the global minimum and maximum case values for all cases. If the current case value is less than the global minimum or greater than the global maximum, then the *_case* macro must update these global values accordingly. The *_endswitch* macro will use these global minimum and maximum values in the two CMP instructions it generates for the range checking sequence.

For each case value appearing in a *_switch* statement, the *_case* macros must save the case value and an identifying label for that case value. This is necessary so that the *_endswitch* macro can generate the jump table. What is really needed is an arbitrary list of records, each record containing a value field and a label field. Unfortunately, the HLA compile-time language does not support arbitrary lists of objects, so we will have to implement the list using a (fixed size) array of record constants. The record declaration will take the following form:

```
caseRecord:
    record
        value:uns32;
        label:uns32;
    endrecord;
```

The *value* field will hold the current case value. The *label* field will hold a unique integer value for the corresponding *_case* that the macros can use to generate statement labels. The implementation of the *_switch* macro in this section will use a variant of the trick found in the section on the *_if* macro; it will convert a local macro symbol to a string and append an integer value to the end of that string to create a unique label. The integer value appended will be the value of the *label* field in the *caseRecord* list.

Processing the *_case* macro becomes fairly easy at this point. All the *_case* macro has to do is create an entry in the *caseRecord* list, bump a few counters, and emit an appropriate case label prior to the code emission. The implementation in this section uses Pascal semantics, so all but the first case in the *_switch.._endswitch* statement must first emit a jump to the statement following the *_endswitch* so the previous case's code doesn't fall into the current case.

The real work in implementing the *_switch.._endswitch* statement lies in the generation of the jump table. First of all, there is no requirement that the cases appear in ascending order in the *_switch.._endswitch* statement. However, the entries in the jump table must appear in ascending order. Second, there is no requirement that the cases in the *_switch.._endswitch* statement be consecutive. Yet the entries in the jump table must be consecutive case values[2]. The code that emits the jump table must handle these inconsistencies.

The first task is to sort the entries in the *caseRecord* list in ascending order. This is easily accomplished by writing a little *SortCases* macro to sort all the *caseRecord* entries once the *_switch.._endswitch* macro has processed all the cases. *SortCases* doesn't have to be fancy. In fact, a bubblesort algorithm is perfect for this because:

- Bubble sort is easy to implement
- Bubble sort is efficient when sorting small lists and most SWITCH statements only have a few cases.
- Bubble sort is especially efficient on nearly sorted data and most programmers put their cases in ascending order.

After sorting the cases, only one problem remains: there may be gaps in the case values. This problem is easily handled by stepping through the *caseRecord* elements one by one and synthesizing consecutive entries whenever a gap appears in the list. Program 9.4 provides the full *_switch.._case.._default.._endswitch* macro implementation.

```
/**************************************************/
/*                                                */
/* switch.hla-                                    */
/*                                                */
```

2. Of course, if there are gaps in the case values, the jump table entries for the missing items should contain the address of the default case.

```
/* This program demonstrates how to implement the */
/* _switch.._case.._default.._endswitch statement */
/* using macros.                                  */
/*                                                */
/**************************************************/


program demoSwitch;
#include( "stdlib.hhf" )

const

    // Because this code uses an array to implement
    // the caseRecord list, we have to specify a fixed
    // number of cases.  The following constant defines
    // the maximum number of possible cases in a
    // _switch statement.

    maxCases := 256;

type

    // The following data type hold the case value
    // and statement label information for each
    // case appearing in a _switch statement.

    caseRecord:
        record

            value:uns32;
            lbl:uns32;

        endrecord;


// SortCases
//
//  This routine does a bubble sort on an array
// of caseRecord objects.  It sorts in ascending
// order using the "value" field as the key.
//
// This is a good old fashioned bubble sort which
// turns out to be very efficient because:
//
//  (1) The list of cases is usually quite small, and
//  (2) The data is usually already sorted (or mostly sorted).

macro SortCases( sort_array, sort_size ):
    sort_i,
    sort_bnd,
    sort_didswap,
    sort_temp;

    ?sort_bnd := sort_size - 1;
    ?sort_didswap := true;
    #while( sort_didswap )

        ?sort_didswap := false;
        ?sort_i := 0;
        #while( sort_i < sort_bnd )
```

```
            #if
            (
                sort_array[sort_i].value >
                    sort_array[sort_i+1].value
            )

                ?sort_temp := sort_array[sort_i];
                ?sort_array[sort_i] := sort_array[sort_i+1];
                ?sort_array[sort_i+1] := sort_temp;
                ?sort_didswap := true;

            #elseif
            (
                sort_array[sort_i].value =
                    sort_array[sort_i+1].value
            )

                #error
                (
                    "Two cases have the same value: (" +
                    string( sort_array[sort_i].value ) +
                    ")"
                )

            #endif
            ?sort_i := sort_i + 1;

        #endwhile
        ?sort_bnd := sort_bnd - 1;

    #endwhile;


endmacro;
```

```
// HLA Macro to implement a C SWITCH statement (using
// Pascal semantics). Note that the switch parameter
// must be a 32-bit register.

macro _switch( switch_reg ):
    switch_minval,
    switch_maxval,
    switch_otherwise,
    switch_endcase,
    switch_jmptbl,
    switch_cases,
    switch_caseIndex,
    switch_doCase,
    switch_hasotherwise;        // Just used to generate unique names.


    // Verify that we have a register operand.
```

```
        #if( !@isReg32( switch_reg ) )

            #error( "Switch operand must be a 32-bit register" )

        #endif

        // Create the switch_cases array.  Allow, at most, 256 cases.

        ?switch_cases:caseRecord[ maxCases ];

        // General initialization for processing cases.

        ?switch_caseIndex := 0;         // Index into switch_cases array.
        ?switch_minval := $FFFF_FFFF;   // Minimum case value.
        ?switch_maxval := 0;            // Maximum case value.
        ?switch_hasotherwise := false;  // Determines if DEFAULT section present.



        // We need to process the cases to collect information like
        // switch_minval prior to emitting the indirect jump.  So move the
        // indirect jump to the bottom of the case statement.

        jmp switch_doCase;


        // "case" keyword macro handles each of the cases in the
        // case statement.  Note that this syntax allows you to
        // specify several cases in the same _case macro, e.g.,
        // _case( 2, 3, 4 ).  Such a situation tells this macro
        // that these three values all execute the same code.

    keyword _case( switch_parms[] ):
        switch_parmIndex,
        switch_parmCount,
        switch_constant;

        ?switch_parmCount:uns32;
        ?switch_parmCount := @elements( switch_parms );

        #if( switch_parmCount <= 0 )

            #error( "Must have at least one case value" );
            ?switch_parms:uns32[1] := [0];

        #endif

        // If we have at least one case already, terminate
        // the previous case by transfering control to the
        // first statement after the endcase macro.  Note
        // that these semantics match Pascal's CASE statement,
        // not C/C++'s SWITCH statement which would simply
        // fall through to the next CASE.

        #if( switch_caseIndex <> 0 )

            jmp switch_endcase;

        #endif

        // The following loop processes each case value
        // supplied to the _case macro.

        ?switch_parmIndex:uns32;
        ?switch_parmIndex := 0;
        #while( switch_parmIndex < switch_parmCount )

            ?switch_constant: uns32;
            ?switch_constant: uns32 :=
                uns32( @text( switch_parms[ switch_parmIndex ]));

            // Update minimum and maximum values based on the
            // current case value.

            #if( switch_constant < switch_minval )

                ?switch_minval := switch_constant;

            #endif
            #if( switch_constant > switch_maxval )

                ?switch_maxval := switch_constant;

            #endif

            // Emit a unique label to the source code for this case:

            @text
            (
                    "_case"
                +   @string:switch_caseIndex
                +   string( switch_caseIndex )
            ):

            // Save away the case label and the case value so we
            // can build the jump table later on.

            ?switch_cases[ switch_caseIndex ].value := switch_constant;
            ?switch_cases[ switch_caseIndex ].lbl := switch_caseIndex;

            // Bump switch_caseIndex value because we've just processed
            // another case.

            ?switch_caseIndex := switch_caseIndex + 1;
            #if( switch_caseIndex >= maxCases )

                #error( "Too many cases in statement" );

            #endif

            ?switch_parmIndex := switch_parmIndex + 1;

        #endwhile



        // Handle the default keyword/macro here.

    keyword _default;

        // If there was not a preceding case, this is an error.
        // If so, emit a jmp instruction to skip over the
        // default case.

        #if( switch_caseIndex < 1 )

            #error( "Must have at least one case" );

        #endif

            jmp switch_endcase;


        // Emit the label for this default case and set the
        // switch_hasotherwise flag to true.

        switch_otherwise:
        ?switch_hasotherwise := true;


        // The endswitch terminator/macro checks to see if
        // this is a reasonable switch statement and emits
        // the jump table code if it is.

    terminator _endswitch:
        switch_i_,
        switch_j_,
        switch_curCase_;


        // If the difference between the smallest and
        // largest case values is great, the jump table
        // is going to be fairly large.  If the difference
        // between these two values is greater than 256 but
        // less than 1024, warn the user that the table will
        // be large.  If it's greater than 1024, generate
        // an error.
        //
        // Note: these are arbitrary limits.  Feel free to
        // adjust them if you like.

        #if( (switch_maxval - switch_minval) > 256 )

            #if( (switch_maxval - switch_minval) > 1024 )

                // Perhaps in the future, this macro could
                // switch to generating an if..elseif..elseif...
                // chain if the range between the values is
                // too great.

                #error( "Range of cases is too great" );

            #else

                #print( "Warning: Range of cases is large" );

            #endif

        #endif

        // Table emission algorithm requires that the switch_cases
        // array be sorted by the case values.
        SortCases( switch_cases, switch_caseIndex );


        // Build a string of the form:
        //
        //      switch_jmptbl:dword[ xx ] := [&case1, &case2, &case3...&casen];
        //
        // so we can output the jump table.

        readonly

            switch_jmptbl:dword[ switch_maxval - switch_minval + 2] := [

            ?switch_i_ := 0;
            #while( switch_i_ < switch_caseIndex )

                ?switch_curCase_ := switch_cases[ switch_i_ ].value;
                // Emit the label associated with the current case:

                @text
                (
                        "&"
                    +   "_case"
                    +   @string:switch_caseIndex
                    +   string( switch_cases[ switch_i_ ].lbl )
                    +   ","
                )

                // Emit "&switch_otherwise" table entries for any gaps present
                // in the table:

                ?switch_j_ := switch_cases[ switch_i_ + 1 ].value;
                ?switch_curCase_ := switch_curCase_ + 1;

                #while( switch_curCase_ < switch_j_ )

                    &switch_otherwise,
                    ?switch_curCase_ := switch_curCase_ + 1;

                #endwhile
                ?switch_i_ := switch_i_ + 1;

            #endwhile

            // Emit a dummy entry to terminate the table:

            &switch_otherwise];


        endreadonly;

        #if( switch_caseIndex < 1 )

            #error( "Must have at least one case" );

        #endif

            // After the default case, or after the last
            // case entry, jump over the code that does
            // the conditional jump.
            jmp switch_endcase;

        // Okay, here's the code that does the conditional jump.

        switch_doCase:

            // If the minimum case value is zero, we don't
            // need to emit a CMP instruction for it.

            #if( switch_minval <> 0 )

                cmp( switch_reg, switch_minval );
                jb switch_otherwise;

            #endif
            cmp( switch_reg, switch_maxval );
            ja switch_otherwise;
            jmp( switch_jmptbl[ switch_reg*4 - switch_minval*4 ] );



        // If there was no default case, transfer control
        // to the first statement after the "endcase" clause.

        #if( !switch_hasotherwise )

            switch_otherwise:

        #endif

        // When each of the cases complete execution,
        // transfer control down here.

        switch_endcase:

        // The following statement deallocates the storage
        // assocated with the switch_cases array (this saves
        // memory at compile time, it does not affect the
        // execution of the resulting machine code).

        ?switch_cases := 0;



    endmacro;


    begin demoSwitch;


        // A simple demonstration of the _switch.._endswitch statement:

        for( mov( 0, eax ); eax < 8; inc( eax )) do

            _switch( eax )

                _case( 0 )

                    stdout.put( "eax = 0" nl );

                _case( 1, 2 )
```

```
                    stdout.put( "eax = 1 or 2" nl );

                _case( 3, 4, 5 )

                    stdout.put( "eax = 3, 4, or 5" nl );

                _case( 6 )

                    stdout.put( "eax = 6" nl );

                _default

                    stdout.put( "eax is not in the range 0-6" nl );

            _endswitch;

        endfor;

end demoSwitch;
```

Program 9.4 Macro Implementation of the SWITCH..ENDSWITCH Statement

9.2.3 A Modified WHILE Loop

The previous sections have shown you how to implement statements that are already available in HLA or the HLA Standard Library. While this approach lets you work with familiar statements that you should be comfortable with, it doesn't really demonstrate that you can create *new* control statements with HLA's compile-time language. In this section you will see how to create a variant of the WHILE statement that is not simply a rehash of HLA's WHILE statement. This should amply demonstrate that there are some useful control structures that HLA (and high level languages) don't provide and that you can easily use HLA compile-time language to implement specialized control structures as needed.

A common use of a WHILE loop is to search through a list and stop upon encountering some desired value or upon hitting the end of the list. A typical HLA example might take the following form:

```
while( <<There are more items in the list>> ) do

    breakif( <<This was the item we're looking for>> );
    << select the next item in the list>>

endwhile;
```

The problem with this approach is that when the statement immediately following the ENDWHILE executes, that code doesn't know whether the loop terminated because it found the desired value or because it exhausted the list. The typical solution is to test to see if the loop exhausted the list and deal with that accordingly:

```
while( <<There are more items in the list>> ) do

    breakif( <<This was the item we're looking for>> );
    << select the next item in the list>>
```

```
endwhile;
if( <<The list wasn’t exhausted>> ) then

    << do something with the item we found >>

endif;
```

The problem with this "solution" should be obvious if you think about it a moment. We’ve already tested to see if the loop is empty, immediately after leaving the loop we repeat this same test. This is somewhat inefficient. A better solution would be to have something like an "else" clause in the WHILE loop that executes if you break out of the loop and doesn’t execute if the loop terminates because the boolean expression evaluated false. Rather than use the keyword ELSE, let’s invent a new (more readable) term: *onbreak*. The ONBREAK section of a WHILE loop executes (only once) if a BREAK or BREAKIF statement was the reason for the loop termination. With this ONBREAK clause, you could recode the previous WHILE loop a little bit more elegantly as follows:

```
while( <<There are more items in the list>> ) do

        breakif( <<This was the item we’re looking for>> );
        << select the next item in the list>>


    onbreak

        << do something with the item we found >>

endwhile;
```

Note that if the ONBREAK clause is present, the WHILE’s loop body ends at the ONBREAK keyword. The ONBREAK clause executes at most once per execution of this WHILE statement.

Implementing a _while.._onbreak.._endwhile statement is very easy using HLA’s multi-part macros. Program 9.5 provides the complete implementation of this statement:

```
/****************************************************/
/*                                                  */
/* while.hla                                        */
/*                                                  */
/* This program demonstrates a variant of the       */
/* WHILE loop that provides a special "onbreak"     */
/* clause.  The _onbreak clause executes if the     */
/* program executes a _break clause or it executes  */
/* a _breakif clause and the corresponding          */
/* boolean expression evaluates true.  The _onbreak */
/* section does not execute if the loop terminates  */
/* due to the _while boolean expression evaluating  */
/* false.                                           */
/*                                                  */
/****************************************************/

program Demo_while;
#include( "stdlib.hhf" )

// _while semantics:
//
// _while( expr )
//
//      << stmts including optional _break, _breakif
//          _continue, and _continueif statements >>
//
```

```
//    _onbreak  // This section is optional.
//
//      << stmts that only execute if program executes
//          a _break or _breakif (with true expression)
//          statement. >>
//
// _endwhile;

macro _while( expr ):falseLbl, breakLbl, topOfLoop, hasOnBreak;

    // hasOnBreak keeps track of whether we've seen an _onbreak
    // section.
    ?hasOnBreak:boolean:=false;

    // Here's the top of the WHILE loop.
    // Implement this as a straight-forward WHILE (test for
    // loop termination at the top of the loop).

    topOfLoop:
        jf( expr ) falseLbl;

  // Ignore the _do keyword.

  keyword _do;


  // _continue and _continueif (with a true expression)
  // transfer control to the top of the loop where the
  // _while code retests the loop termination condition.

  keyword _continue;
        jmp topOfLoop;

  keyword _continueif( expr1 );
        jt( expr1 ) topOfLoop;


  // Unlike the _break or _breakif in a standard WHILE
  // statement, we don't immediately exit the WHILE.
  // Instead, this code transfers control to the optional
  // _onbreak section if it is present.  If it is not
  // present, control transfers to the first statement
  // beyond the _endwhile.

  keyword _break;
        jmp breakLbl;

  keyword _breakif( expr2 );
        jt( expr2 ) breakLbl;


  // If we encounter an _onbreak section, this marks
  // the end of the while loop body.  Emit a jump that
  // transfers control back to the top of the loop.
  // This code also has to verify that there is only
  // one _onbreak section present.  Any code following
  // this clause is going to execute only if the _break
  // or _breakif statements execute and transfer control
  // down here.

  keyword _onbreak;
```

```
        #if( hasOnBreak )

            #error( "Extra _onbreak clause encountered" )

        #else

                jmp topOfLoop;
                ?hasOnBreak := true;

            breakLbl:

        #endif

      terminator _endwhile;

        // If we didn't have an _onbreak section, then
        // this is the bottom of the _while loop body.
        // Emit the jump to the top of the loop and emit
        // the "breakLbl" label so the execution of a
        // _break or _breakif transfers control down here.

        #if( !hasOnBreak )

            jmp topOfLoop;
            breakLbl:

        #endif
        falseLbl:

    endmacro;


    static
        i:int32;

    begin Demo_while;

        // Demonstration of standard while loop

        mov( 0, i );
        _while( i < 10 ) _do

            stdout.put( "1: i=", i, nl );
            inc( i );

        _endwhile;

        // Demonstration with BREAKIF:

        mov( 5, i );
        _while( i < 10 ) _do

            stdout.put( "2: i=", i, nl );
            _breakif( i = 7 );
            inc( i );

        _endwhile

        // Demonstration with _BREAKIF and _ONBREAK:

        mov( 0, i );
        _while( i < 10 ) _do

            stdout.put( "3: i=", i, nl );
            _breakif( i = 4 );
            inc( i );

          _onbreak

            stdout.put( "Breakif was true at i=", i, nl );

        _endwhile
        stdout.put( "All Done" nl );

    end Demo_while;
```

Program 9.5 The Implementation of _while.._onbreak.._endwhile

9.2.4 A Modified IF..ELSE..ENDIF Statement

The IF statement is another statement that doesn't always do exactly what you want. Like the *_while.._onbreak.._endwhile* example above, it's quite possible to redefine the IF statement so that it behaves the way we want it to. In this section you'll see how to implement a variant of the IF..ELSE..ENDIF statement that nests differently than the standard IF statement.

It is possible to simulate short-circuit boolean evaluation invovling conjunction and disjunction without using the "&&" and "||" operators if you carefully structure your code. Consider the following example:

```
// "C" code employing logical-AND operator:

if( expr1 && expr2 )
{
    << statements >>
}


// Equivalent HLA version:

if( expr1 ) then

    if( expr2 ) then

        << statements >>

    endif;

endif;
```

In both cases ("C" and HLA) the << *statements*>> block executes only if both *expr1* and *expr2* evaluate true. So other than the extra typing involved, it is often very easy to simulate logical conjunction by using two IF statements in HLA.

There is one very big problem with this scheme. Consider what happens if you modify the "C" code to be the following:

```
// "C" code employing logical-AND operator:

if( expr1 && expr2 )
{
    << 'true' statements >>
}
else
{
    << 'false' statements >>
}
```

Before describing how to create this new type of IF statement, we must digress for a moment and explore an interesting feature of HLA's multi-part macro expansion: #KEYWORD macros do not have to use unique names. Whenever you declare an HLA #KEYWORD macro, HLA accepts whatever name you choose. If that name happens to be already defined, then the #KEYWORD macro name takes precedence as long as the macro is active (that is, from the point you invoke the macro name until HLA encounters the #TERMINATOR macro). Therefore, the #KEYWORD macro name hides the previous definition of that name until the termination of the macro. This feature applies even to the original macro name; that is, it is possible to define a #KEYWORD macro with the same name as the original macro to which the #KEYWORD macro belongs. This is a very useful feature because it allows you to change the definition of the macro within the scope of the opening and terminating invocations of the macro.

Although not pertinent to the IF statement we are constructing, you should note that parameter and local symbols in a macro also override any previously defined symbols of the same name. So if you use that symbol between the opening macro and the terminating macro, you will get the value of the local symbol, not the global symbol. E.g.,

```
var
    i:int32;
    j:int32;
        .
        .
        .
#macro abc:i;
    ?i:text := "j";
        .
        .
        .
#terminator xyz;
        .
        .
        .
#endmacro;
        .
        .
        .
    mov( 25, i );
    mov( 10, j );
    abc
        mov( i, eax );   // Loads j's value (10), not 25 into eax.
    xyz;
```

The code above loads 10 into EAX because the "mov(i, eax);" instruction appears between the opening and terminating macros *abc..xyz*. Between those two macros the local definition of *i* takes precedence over the global definition. Since *i* is a text constant that expands to *j*, the aforementioned MOV statement is really equivalent to "mov(j, eax);" That statement, of course, loads 10 into EAX. Since this problem is difficult to see while reading your code, you should choose local symbols in multi-part macros very carefully. A good convention to adopt is to combine your local symbol name with the macro name, e.g.,

```
#macro abc : i_abc;
```

You may wonder why HLA allows something to crazy to happen in your source code, in a moment you'll see why this behavior is useful (and now, with this brief message out of the way, back to our regularly scheduled discussion).

Before we digressed to discuss this interesting feature in HLA multi-part macros, we were trying to figure out how to efficiently simulate the conjunction and disjunction operators in an IF statement without actually using this operators in our code. The problem in the example appearing earlier in this section is that you would have to duplicate some code in order to convert the IF..ELSE statement properly. The following code shows this problem:

```
// "C" code employing logical-AND operator:

if( expr1 && expr2 )
{
   << 'true' statements >>
}
else
{
   << 'false' statements >>
}


// Corresponding HLA code using the "nested-IF" algorithm:

if( expr1 ) then

   if( expr2 ) then

      << 'true' statements >>

   else

      << 'false' statements >>

   endif;

else

   << 'false' statements >>

endif;
```

Note that this code must duplicate the "<< 'false' statements >>" section if the logic is to exactly match the original "C" code. This means that the program will be larger and harder to read than is absolutely necessary.

One solution to this problem is to create a new kind of IF statement that doesn't nest the same way standard IF statements nest. In particular, if we define the statement such that all IF clauses nested with an outer IF..ENDIF block share the same ELSE and ENDIF clauses. If this were the case, then you could implement the code above as follows:

```
if( expr1 ) then

   if( expr2 ) then

   << 'true' statements >>


else

   << 'false' statements >>

endif;
```

If *expr1* is false, control immediately transfers to the ELSE clause. If the value of *expr1* is true, the control falls through to the next IF statement.

If *expr2* evaluates false, then the program jumps to the single ELSE clause that all IFs share in this statement. Notice that a single ELSE clause (and corresponding 'false' statements) appear in this code; hence the code does not necessarily expand in size. If expr2 evaluates true, then control falls through to the 'true' statements, exactly like a standard IF statement.

Notice that the nested IF statement above does not have a corresponding ENDIF. Like the ELSE clause, all nested IFs in this structure share the same ENDIF. Syntactically, there is no need to end the nested IF statement; the end of the THEN section ends with the ELSE clause, just as the outer IF statement's THEN block ends.

Of course, we can't actually define a new macro named "if" because you cannot redefine HLA reserved words. Nor would it be a good idea to do so even if these were legal (since it would make your programs very difficult to comprehend if the IF keyword had different semantics in different parts of the program. The following program uses the identifiers "_if", "_then", "_else", and "_endif" instead. It is questionable if these are good identifiers in production code (perhaps something a little more different would be appropriate). The following code example uses these particular identifiers so you can easily correlate them with the corresponding high level statements.

```
/***********************************************/
/*                                             */
/* if.hla                                      */
/*                                             */
/* This program demonstrates a modification of */
/* the IF..ELSE..ENDIF statement using HLA's   */
/* multi-part macros.                          */
/*                                             */
/***********************************************/


program newIF;
#include( "stdlib.hhf" )



// Macro implementation of new form of if..then..else..endif.
//
// In this version, all nested IF statements transfer control
// to the same ELSE clause if any one of them have a false
// boolean expression.  Syntax:
//
//  _if( expression ) _then
//
//      <<statements including nested _if clauses>>
//
//  _else // this is optional
//
//      <<statements, but _if clauses are not allowed here>>
//
//  _endif
//
//
// Note that nested _if clauses do not have a corresponding
// _endif clause.  This is because the single _else and/or
// _endif clauses terminate all the nested _if clauses
// including the first one.  Of course, once the code
// encounters an _endif another _if statement may begin.
// Macro to handle the main "_if" clause.
// This code just tests the expression and jumps to the _else
// clause if the expression evaluates false.

macro _if( ifExpr ):elseLbl, hasElse, ifDone;

    ?hasElse := false;
    jf(ifExpr) elseLbl;


// Just ignore the _then keyword.

keyword _then;


// Nested _if clause (yes, HLA lets you replace the main
// macro name with a keyword macro).  Identical to the
// above _if implementation except this one does not
// require a matching _endif clause.  The single _endif
// (matching the first _if clause) terminates all nested
// _if clauses as well as the main _if clause.

keyword _if( nestedIfExpr );
    jf( nestedIfExpr ) elseLbl;

    // If this appears within the _else section, report
    // an error (we don't allow _if clauses nested in
    // the else section, that would create a loop).

    #if( hasElse )

        #error( "All _if clauses must appear before the _else clause" )

    #endif


// Handle the _else clause here.  All we need to is check to
// see if this is the only _else clause and then emit the
// jmp over the else section and output the elseLbl target.

keyword _else;
    #if( hasElse )

        #error( "Only one _else clause is legal per _if.._endif" )

    #else

        // Set hasElse true so we know that we've seen an _else
        // clause in this statement.

        ?hasElse := true;
        jmp ifDone;
        elseLbl:

    #endif

// _endif has two tasks.  First, it outputs the "ifDone" label
// that _else uses as the target of its jump to skip over the
// else section.  Second, if there was no else section, this
// code must emit the "elseLbl" label so that the false conditional(s)
// in the _if clause(s) have a legal target label.

terminator _endif;

    ifDone:
    #if( !hasElse )

        elseLbl:

    #endif

endmacro;


static
    tr:boolean := true;
    f:boolean := false;

begin newIF;

    // Real quick demo of the _if statement:

    _if( tr ) _then

        _if( tr ) _then
        _if( f ) _then

            stdout.put( "error" nl );

    _else

        stdout.put( "Success" );

    _endif

end newIF;
```

Program 9.6 Using Macros to Create a New IF Statement

Just in case you're wondering, this program prints "Success" and then quits. This is because the nested "_if" statements are equivalent to the expression "true && true && false" which, of course, is false. Therefore, the "_else" portion of this code should execute.

The only surprise in this macro is the fact that it redefines the *_if* macro as a keyword macro upon invocation of the main *_if* macro. The reason this code does this is so that any nested *_if* clauses do not require a corresponding *_endif* and don't support an *_else* clause.

Implementing an ELSEIF clause introduces some difficulties, hence its absence in this example. The design and implementation of an ELSEIF clause is left to the more serious reader[3].

3. I.e., I don't even want to have to think about this problem!

9.3 Sample Program: A Simple Expression Compiler

This program's sample program is a bit complex. In fact, the theory behind this program is well beyond the scope of this text (since it involves compiler theory). However, this example is such a good demonstration of the capabilities of HLA's macro facilities and DSEL capabilities, it was too good not to include here. The following paragraphs will attempt to explain how this compile-time program operates. If you have difficulty understanding what's going on, don't feel too bad, this code isn't exactly the type of stuff that beginning assembly language programmers would normally develop on their own.

This program presents a (very) simple *expression compiler.* This code includes a macro, *u32expr*, that emits a sequence of instructions that compute the value of an arithmetic expression and leave that result sitting in one of the 80x86's 32-bit registers. The syntax for the *u32expr* macro invocation is the following:

u32expr(*reg*$_{32}$, *uns32_expression*);

This macro emits the code that computes the following (HLL) statement:

reg$_{32}$:= *uns32_expression;*

For example, the macro invocation "u32expr(eax, ebx+ecx*5 - edi);" computes the value of the expression "ebx+ecx*5 - edi" and leaves the result of this expression sitting in the EAX register.

The *u32expr* macro places several restrictions on the expression. First of all, as the name implies, it only computes the result of an *uns32* expression. No other data types may appear within the expression. During computation, the macro uses the EAX and EDX registers, so expressions should not contain these registers as their values may be destroyed by the code that computes the expression (EAX or EDX may safely appear as the first operand of the expression, however). Finally, expressions may only contain the following operators:

```
<, <=, >, >=, <>, !=, =, ==
           +, -
           *, /
           (, )
```

The "<>" and "!=" operators are equivalent (not equals) and the "=" and "==" operators are also equivalent (equals). The operators above are listed in order of increasing precedence; i.e., "*" has a higher precedence than "+" (as you would expect). You can override the precedence of an operator by using parentheses in the standard manner.

It is important to remember that *u32expr* is a macro, not a function. That is, the invocation of this macro results in a sequence of 80x86 assembly language instructions that computes the desired expression. The *u32expr* invocation is not a function call. to some routine that computes the result.

To understand how this macro works, it would be a good idea to review the section on "Converting Arithmetic Expressions to Postfix Notation" on page 635. That section discusses how to convert floating point expressions to reverse polish notation; although the *u32expr* macro works with *uns32* objects rather than floating point objects, the approach it uses to translate expressions into assembly language uses this same algorithm. So if you don't remember how to translate expressions into reverse polish notation, it might be worthwhile to review that section of this text.

Converting floating point expressions to reverse polish notation is especially easy because the 80x86's FPU uses a stack architecture. Alas, the integer instructions on the 80x86 use a register architecture and efficiently translating integer expression to assembly language is a bit more difficult (see "Arithmetic Expressions" on page 597). We'll solve this problem by translating the expressions to assembly code in a somewhat less than efficient manner; we'll simulate an integer stack architecture by using the 80x86's hardware stack to hold temporary results during an integer calculation.

To push an integer constant or variable onto the 80x86 hardware stack, we need only use a PUSH or PUSHD instruction. This operation is trivial.

To add two values sitting on the top of stack together, leaving their sum on the stack, all we need do is pop those two values into registers, add the register values, and then push the result back onto the stack. We can do this operation slightly more efficiently, since addition is commutative, by using the following code:

```
// Compute X+Y where X is on NOS (next on stack) and Y is on TOS (top of stack):

        pop( eax );                // Get Y’s value.
        add( eax, [esp] );         // Add with X’s value and leave sum on TOS.
```

Subtraction is identical to addition. Although subtraction is not commutative the operands just happen to be on the stack in the proper order to efficiently compute their difference. To compute "X-Y" where *X* is on NOS and *Y* is on TOS, we can use code like the following:

```
// Compute X-y where X is on NOS and Y is on TOS:

        pop( eax );
        sub( eax, [esp] );
```

Multiplication of the two items on the top of stack is a little more complicated since we must use the MUL instruction (the only unsigned multiplication instruction available) and the destination operand must be the EDX:EAX register pair. Fortunately, multiplication is a commutative operation, so we can compute the product of NOS (next on stack) and TOS (top of stack) using code like the following:

```
// Compute X*Y where X is on NOS and Y is on TOS:

        pop( eax );
        mul( [esp], eax );         // Note that this wipes out the EDX register.
        mov( eax, [esp] );
```

Division is problematic because it is not a commutative operation and its operands on the stack are not in a convenient order. That is, to compute X/Y it would be really convenient if X was on TOS and Y was in the NOS position. Alas, as you’ll soon see, it turns out that X is at NOS and Y is on the TOS. To resolve this issue requires slightly less efficient code that the sequences we’ve used above. Since the DIV instruction is so slow anyway, this will hardly matter.

```
// Compute X/Y where X is on NOS and Y is on TOS:

        mov( [esp+4], eax );           // Get X from NOS.
        xor( edx, edx );               // Zero-extend EAX into EDX:EAX
        div( [esp], edx:eax );         // Compute their quotient.
        pop( edx );                    // Remove unneeded Y value from the stack.
        mov( eax, [esp] );             // Store quotient to the TOS.
```

The remaining operators are the comparison operators. These operators compare the value on NOS with the value on TOS and leave true (1) or false (0) sitting on the stack based on the result of the comparison. While it is easy to work around the non-commutative aspect of many of the comparison operators, the big challenge is converting the result to true or false. The SETcc instructions are convenient for this purpose, but they only work on byte operands. Therefore, we will have to zero extend the result of the SETcc instructions to obtain an *uns32* result we can push onto the stack. Ultimately, the code we must emit for a comparison is similar to the following:

```
// Compute X <= Y where X is on NOS and Y is on TOS.

        pop( eax );
        cmp( [esp], eax );
        setbe( al );               // This instruction changes for other operators.
        movzx( al, eax );
```

```
mov( eax, [esp] );
```

As it turns out, the appearance of parentheses in an expression only affects the order of the instructions appearing in the sequence, it does not affect the number of type of instructions that correspond to the calculation of an expression. As you'll soon see, handling parentheses is an especially trivial operation.

With this short description of how to emit code for each type of arithmetic operator, it's time to discuss exactly how we will write a macro to automate this translation. Once again, a complete discussion of this topic is well beyond the scope of this text, however a simple introduction to compiler theory will certainly ease the understanding the *u32expr* macro.

For efficiency and reasons of convenience, most compilers are broken down into several components called *phases*. A compiler phase is collection of logically related activities that take place during compilation. There are three general compiler phases we are going to consider here: (1) *lexical analysis* (or *scanning*), (2) *parsing*, and (3) *code generation*. It is important to realize that these three activities occur concurrently during compilation; that is, they take place at the same time rather than as three separate, serial, activities. A compiler will typically run the lexical analysis phase for a short period, transfer control to the parsing phase, do a little code generation, and then, perhaps, do some more scanning and parsing and code generation (not necessarily in that order). Real compilers have additional phases, the *u32expr* macro will only use these three phases (and if you look at the macro, you'll discover that it's difficult to separate the parsing and code generation phases).

Lexical analysis is the process of breaking down a string of characters, representing the expression to compile, into a sequence of *tokens* for use by the parser. For example, an expression of the form "MaxVal - x <= $1c" contains five distinct tokens:

- MaxVal
- -
- x
- <=
- $1c

Breaking any one of these tokens into smaller objects would destroy the intent of the expression (e.g., converting MaxVal to "Max" and "Val" or converting "<=" into "<" and "="). The job of the lexical analyzer is to break the string down into a sequence of constituent tokens and return this sequence of tokens to the parser (generally one token at a time, as the parser requests new tokens). Another task for the lexical analyzer is to remove any extra white space from the string of symbols (since expressions may generally contain an arbitrary amount of white space).

Fortunately, it is easy to extract the next available token in the input string by skipping all white space characters and then look at the current character. Identifiers always begin with an alphabetic character or an underscore, numeric values always begin with a decimal digit, a dollar sign ("$"), or a percent sign ("%"). Operators always begin with the corresponding punctuation character that represents the operator. There are only two major issues here: how do we classify these tokens and how do we differentiate two or more distinct tokens that start with the same character (e.g., "<", "<=", and "<>")? Fortunately, HLA's compile-time functions provide the tools we need to do this.

Consider the declaration of the *u32expr* macro:

```
#macro u32expr( reg, expr ):sexpr;
```

The *expr* parameter is a text object representing the expression to compile. The *sexpr* local symbol will contain the string equivalent of this text expression. The macro translates the text *expr* object to a string with the following statement:

```
?sexpr := @string:expr;
```

From this point forward, the macro works with the string in *sexpr.*

The *lexer* macro (compile-time function) handles the lexical analysis operation. This macro expects a single string parameter from which it extracts a single token and removes the string associated with that token from the front of the string. For example, the following macro invocation returns "2" as the function result and leaves "+3" in the parameter string (*str2Lex*):

```
?str2Lex := "2+3";
?TokenResult := lexer( str2Lex );
```

The *lexer* function actually returns a little more than the string it extracts from its parameter. The actual return value is a record constant that has the definition:

```
tokType:
    record

        lexeme:string;
        tokClass:tokEnum;

    endrecord;
```

The *lexeme* field holds that actual string (e.g., "2" in this example) that the *lexer* macro returns. The *tokClass* field holds a small numeric value (see the *tokEnum* enumerated data type) that specifies that type of the token. In this example, the call to *lexer* stores the value *intconst* into the *tokClass* field. Having a single value (like *intconst*) when the *lexeme* could take on a large number of different forms (e.g., "2", "3", "4", ...) will help make the parser easier to write. The call to lexer in the previous example produces the following results:

```
str2lex : "+3"
TokenResult.lexeme: "2"
TokenResult.tokClass: intconst
```

A subsequent call to lexer, immediately after the call above, will process the next available character in the string and return the following values:

```
str2lex : "3"
TokenResult.lexeme: "+"
TokenResult.tokClass: plusOp
```

To see how *lexer* works, consider the first few lines of the *lexer* macro:

```
#macro lexer( input ):theLexeme,boolResult;

    ?theLexeme:string;      // Holds the string we scan.
    ?boolResult:boolean;    // Used only as a dummy value.

    // Check for an identifier.

    #if( @peekCset( input, tok1stIDChar ))

        // If it began with a legal ID character, extract all
        // ID characters that follow.  The extracted string
        // goes into "theLexeme" and this call also removes
        // those characters from the input string.

        ?boolResult := @oneOrMoreCset( input, tokIDChars, input, theLexeme );

        // Return a tokType constant with the identifier string and
        // the "identifier" token value:

        tokType:[ theLexeme, identifier ]



    // Check for a decimal numeric constant.
    #elseif( @peekCset( input, digits ))
        .
        .
        .
```

The real work begins with the #IF statement where the code uses the *@peekCset* function to see if the first character of the *input* parameter is a member of the *tok1stIDChar* set (which is the alphabetic characters plus an underscore, i.e., the set of character that may appear as the first character of an identifier). If so, the code executes the *@oneOrMoreCset* function to extract all legal identifier characters (alphanumerics plus underscore), storing the result in the *theLexeme* string variable. Note that this function call to *@oneOrMoreCset* also removes the string it matches from the front of the *input* string (see the description of *@oneOrMoreCset* for more details). This macro returns a tokType result by simply specifying a *tokType* constant containing *theLexeme* and the enum constant *identifier.*

If the first character of the input string is not in the *tok1stIDChar* set, then the *lexer* macro checks to see if the first character is a legal decimal digit. If so, then this macro processes that string of digits in a manner very similar to identifiers. The code handles hexadecimal and binary constants in a similar fashion. About the only thing exciting in the whole macro is the way it differentiates tokens that begin with the same symbol. Once it determines that a token begins with a character common to several lexemes, it calls *@matchStr* to attempt to match the longer tokens before settling on the shorter lexeme (i.e., *lexer* attempts to match "<=" or "<>" before it decides the lexeme is just "<"). Other than this complication, the operation of the lexer is really quite simple.

The operation of the parser/code generation phases is a bit more complex, especially since these macros are indirectly recursive; to simplify matters we will explore the parser/code generator in a bottom-up fashion.

The parser/code generator phases consist of four separate macros: *doTerms, doMulOps, doAddOps,* and *doCmpOps*. The reason for these four separate macros is to handle the different precedences of the arithmetic operators and the parentheses. An explanation of how these four macros handle the different arithmetic precedences is beyond the scope of this text; we'll just look at how these four macros do their job.

The *doTerms* macro is responsible for handling identifiers, numeric constants, and subexpressions surrounded by parentheses. The single parameter is the current input string whose first (non-blank) character sequence is an identifier, constant, or parenthetical expression. Here is the full text for this macro:

```
#macro doTerms( expr ):termToken;

    // Begin by removing any leading white space from the string:

    ?expr := @trim( expr, 0 );

    // Okay, call the lexer to extract the next token from the input:

    ?termToken:tokType := lexer( expr );

    // See if the current token is an identifier.  If so, assume that
    // it's an uns32 identifier and emit the code to push its value onto
    // the stack.

    #if( termToken.tokClass = identifier )

        // If we've got an identifier, emit the code to
        // push that identifier onto the stack.

        push( @text( termToken.lexeme ));

    // If it wasn't an identifier, see if it's a numeric constant.
    // If so, emit the code that will push this value onto the stack.
    #elseif( termToken.tokClass = intconst )

        // If we've got a constant, emit the code to push
        // that constant onto the stack.

        pushd( @text( termToken.lexeme ) );

    // If it’s not an identifier or an integer constant, see if it’s
    // a parenthesized subexpression.  If so, invoke the doCmpOps macro
    // to do the real work of code generation for the subexpression.
    // The call to the doCmpOps macro emits all the code needed to push
    // the result of the subexpression onto the stack;  note that this
    // macro doesn’t need to emit any code for the parenthetical expression,
    // all the code emission is handled by doCmpOps.

    #elseif( termToken.tokClass = lparen )

        // If we've got a parenthetical expression, emit
        // the code to leave the parenthesized expression
        // sitting on the stack.

        doCmpOps( expr );

        // We must have a closing right parentheses after the subexpression.
        // Skip any white space and check for the closing ")" here.

        ?expr := @trim( expr, 0 );
        ?termToken:tokType := lexer( expr );
        #if( termToken.tokClass <> rparen )

            #error( "Expected closing parenthesis: " + termToken.lexeme )

        #endif


    // If we get to this point, then the lexer encountered something besides
    // an identifier, a numeric constant, or a parenthetical expression.

    #else

        #error( "Unexpected term: '" + termToken.lexeme + "'" )

    #endif

#endmacro;
```

The *doTerms* macro is responsible for leaving a single item sitting on the top of the 80x86 hardware stack. That stack item is either the value of an *uns32* identifier, the value of an *uns32* expression, or the value left on the stack via a parenthesized subexpression. The important thing to remember is that you can think of *doTerms* as a function that emits code that leaves a single item on the top of the 80x86 stack.

The *doMulOps* macro handles expressions consisting of a single term (items handled by the *doTerms* macro) optionally followed by zero or more pairs consisting of a multiplicative operator ("*" or "/") and a second term. It is especially important to remember that the *doMulOps* macro does not require the presence of a multiplicative operator; it will legally process a single term (identifier, numeric constant, or parenthetical expression). If one or more multiplicative operator and term pairs are present, the *doMulOps* macro will emit the code that will multiply the values of the two terms together and push the result onto the stack. E.g., consider the following:

```
X * 5
```

Since there is a multiplicative operator present ("*"), the *doMulOps* macro will call *doTerms* to process the two terms (pushing *X* and then *Y* onto the stack) and then the *doMulOps* macro will emit the code to multiply the two values on the stack leaving their product on the stack. The complete code for the *doMulOps* macro is the following:

```
#macro doMulOps( sexpr ):opToken;

    // Process the leading term (not optional).  Note that
    // this expansion leaves an item sitting on the stack.

    doTerms( sexpr );

    // Process all the MULOPs at the current precedence level.
    // (these are optional, there may be zero or more of them.)
    // Begin by removing any leading white space.

    ?sexpr := @trim( sexpr, 0 );
    #while( @peekCset( sexpr, MulOps ))

        // Save the operator so we know what code we should
        // generate later.

        ?opToken := lexer( sexpr );

        // Get the term following the operator.

        doTerms( sexpr );

        // Okay, the code for the two terms is sitting on
        // the top of the stack (left operand at [esp+4] and
        // the right operand at [esp]).  Emit the code to
        // perform the specified operation.

        #if( opToken.lexeme = "*" )

            // For multiplication, compute
            // [esp+4] = [esp] * [esp+4] and
            // then pop the junk off the top of stack.

            pop( eax );
            mul( (type dword [esp]) );
            mov( eax, [esp] );

        #elseif( opToken.lexeme = "/" )

            // For division, compute
            // [esp+4] = [esp+4] / [esp] and
            // then pop the junk off the top of stack.

            mov( [esp+4], eax );
            xor( edx, edx );
            div( [esp], edx:eax );
            pop( edx );
            mov( eax, [esp] );

        #endif
        ?sexpr := @trim( sexpr, 0 );

    #endwhile

#endmacro;
```

Note the simplicity of the code generation. This macro assumes that *doTerms* has done its job leaving two values sitting on the top of the stack. Therefore, the only code this macro has to generate is the code to pop these two values off the stack and them multiply or divide them, depending on the actual operator that is present. The code generation uses the sequences appearing earlier in this section.

The *doAddOps* and *doCmpOps* macros work in a manner nearly identical to doMulOps. The only difference is the operators these macros handle (and, of course, the code that they generate). See Program 9.7, below, for details concerning these macros.

Once we've got the lexer and the four parser/code generation macros written, writing the *u32expr* macro is quite easy. All that *u32expr* needs to do is call the *doCmpOps* macro to compile the expression and then pop the result off the stack and store it into the destination register appearing as the first operand. This requires little more than a single POP instruction.

About the only thing interesting in the *u32expr* macro is the presence of the RETURNS statement. This HLA statement takes the following form:

```
returns( { statements }, string_expression )
```

This statement simply compiles the sequence of statements appearing between the braces in the first operand and then it uses the second *string_expression* operand as the "returns" value for this statement. As you may recall from the discussion of instruction composition (see "Instruction Composition in HLA" on page 558), HLA substitutes the "returns" value of a statement in place of that statement if it appears as an operand to another expression. The RETURNS statement appearing in the *u32expr* macro returns the register you specify as the first parameter as the "returns" value for the macro invocation. This lets you invoke the *u32expr* macro as an operand to many different instructions (that accept a 32-bit register as an operand). For example, the following *u32expr* macro invocations are all legal:

```
mov( u32expr( eax, i*j+k/15 - 2), m );
if( u32expr( edx, eax < (ebx-2)*ecx) ) then ... endif;
funcCall( u32expr( eax, (x*x + y*y)/z*z ), 16, 2 );
```

Well, without further ado, here's the complete code for the *u32expr* compiler and some test code that checks out the operation of this macro:

```
// u32expr.hla
//
// This program demonstrates how to write an "expression compiler"
// using the HLA compile-time language.  This code defines a macro
// (u32expr) that accepts an arithmetic expression as a parameter.
// This macro compiles that expression into a sequence of HLA
// machine language instructions that will compute the result of
// that expression at run-time.
//
//  The u32expr macro does have some severe limitations.
// First of all, it only support uns32 operands.
// Second, it only supports the following arithmetic
// operations:
//
//  +, -, *, /, <, <=, >, >=, =, <>.
//
//  The comparison operators produce zero (false) or
// one (true) depending upon the result of the (unsigned)
// comparison.
//
//  The syntax for a call to u32expr is
//
//  u32expr( register, expression )
```

```
//
//  The macro computes the result of the expression and
// leaves this result sitting in the register specified
// as the first operand.  This register overwrites the
// values in the EAX and EDX registers (though these
// two registers are fine as the destination for the
// result).
//
//  This macro also returns the first (register) parameter
// as its "returns" value, so you may use u32expr anywhere
// a 32-bit register is legal, e.g.,
//
//      if( u32expr( eax, (i*3-2) < j )) then
//
//          << do something if (i*3-2) < j >>
//
//      endif;
//
// The statement above computes true or false in EAX and the
// "if" statement processes this result accordingly.



program TestExpr;
#include( "stdlib.hhf" )

// Some special character classifications the lexical analyzer uses.

const

    // tok1stIDChar is the set of legal characters that
    // can begin an identifier.  tokIDChars is the set
    // of characters that may follow the first character
    // of an identifier.

    tok1stIDChar := { 'a'..'z', 'A'..'Z', '_' };
    tokIDChars := { 'a'..'z', 'A'..'Z', '0'..'9', '_' };

    // digits, hexDigits, and binDigits are the sets
    // of characters that are legal in integer constants.
    // Note that these definitions don't allow underscores
    // in numbers, although it would be a simple fix to
    // allow this.

    digits := { '0'..'9' };
    hexDigits := { '0'..'9', 'a'..'f', 'A'..'F' };
    binDigits := { '0'..'1' };


    // CmpOps, PlusOps, and MulOps are the sets of
    // operator characters legal at three levels
    // of precedence that this parser supports.

    CmpOps  := { '>', '<', '=', '!' };
    PlusOps := { '+', '-' };
    MulOps := { '*', '/' };



type
```

```
    // tokEnum-
    //
    // Data values the lexical analyzer returns to quickly
    // determine the classification of a lexeme.  By
    // classifying the token with one of these values, the
    // parser can more quickly process the current token.
    // I.e., rather than having to compare a scanned item
    // against the two strings "+" and "-", the parser can
    // simply check to see if the current item is a "plusOp"
    // (which indicates that the lexeme is "+" or "-").
    // This speeds up the compilation of the expression since
    // only half the comparisons are needed and they are
    // simple integer comparisons rather than string comparisons.

    tokEnum:    enum
                {
                    identifier,
                    intconst,
                    lparen,
                    rparen,
                    plusOp,
                    mulOp,
                    cmpOp
                };

    // tokType-
    //
    //  This is the "token" type returned by the lexical analyzer.
    // The "lexeme" field contains the string that matches the
    // current item scanned by the lexer.  The "tokClass" field
    // contains a generic classifcation for the symbol (see the
    // "tokEnum" type above).

    tokType:
        record

            lexeme:string;
            tokClass:tokEnum;

        endrecord;



// lexer-
//
// This is the lexical analyzer.  On each call it extracts a
// lexical item from the front of the string passed to it as a
// parameter (it also removes this item from the front of the
// string).  If it successfully matches a token, this macro
// returns a tokType constant as its return value.

macro lexer( input ):theLexeme,boolResult;

    ?theLexeme:string;      // Holds the string we scan.
    ?boolResult:boolean;    // Used only as a dummy value.

    // Check for an identifier.

    #if( @peekCset( input, tok1stIDChar ))
```

```
            // If it began with a legal ID character, extract all
            // ID characters that follow.  The extracted string
            // goes into "theLexeme" and this call also removes
            // those characters from the input string.

            ?boolResult := @oneOrMoreCset( input, tokIDChars, input, theLexeme );

            // Return a tokType constant with the identifier string and
            // the "identifier" token value:

            tokType:[ theLexeme, identifier ]



        // Check for a decimal numeric constant.

        #elseif( @peekCset( input, digits ))

            // If the current item began with a decimal digit, extract
            // all the following digits and put them into "theLexeme".
            // Also remove these characters from the input string.

            ?boolResult := @oneOrMoreCset( input, digits, input, theLexeme );

            // Return an integer constant as the current token.

            tokType:[ theLexeme, intconst ]



        // Check for a hexadecimal numeric constant.

        #elseif( @peekChar( input, '$' ))

            // If we had a "$" symbol, grab it and any following
            // hexadecimal digits.  Set boolResult true if there
            // is at least one hexadecimal digit.  As usual, extract
            // the hex value to "theLexeme" and remove the value
            // from the input string:

            ?boolResult := @oneChar( input, '$', input ) &
                           @oneOrMoreCset( input, hexDigits, input, theLexeme );

            // Returns the hex constant string as an intconst object:

            tokType:[ '$' + theLexeme, intconst ]



        // Check for a binary numeric constant.

        #elseif( @peekChar( input, '%' ))

            // See the comments for hexadecimal constants.  This boolean
            // constant scanner works the same way.

            ?boolResult := @oneChar( input, '%', input ) &
                           @oneOrMoreCset( input, binDigits, input, theLexeme );
            tokType:[ '%' + theLexeme, intconst ]
```

```
// Handle the "+" and "-" operators here.

#elseif( @peekCset( input, PlusOps ))

    // If it was a "+" or "-" sign, extract it from the input
    // and return it as a "plusOp" token.

    ?boolResult := @oneCset( input, PlusOps, input, theLexeme );
    tokType:[ theLexeme, plusOp ]



// Handle the "*" and "/" operators here.

#elseif( @peekCset( input, MulOps ))

    // If it was a "*" or "/" sign, extract it from the input
    // and return it as a "mulOp" token.

    ?boolResult := @oneCset( input, MulOps, input, theLexeme );
    tokType:[ theLexeme, mulOp ]



// Handle the "=" ("=="), "<>" ("!="), "<", "<=", ">", and ">="
// operators here.

#elseif( @peekCset( input, CmpOps ))

    // Note that we must check for two-character operators
    // first so we don't confuse them with the single
    // character opertors:

    #if
    (
            @matchStr( input, ">=", input, theLexeme )
        |   @matchStr( input, "<=", input, theLexeme )
        |   @matchStr( input, "<>", input, theLexeme )
    )

        tokType:[ theLexeme, cmpOp ]

    #elseif( @matchStr( input, "!=", input, theLexeme ))

        tokType:[ "<>", cmpOp ]

    #elseif( @matchStr( input, "==", input, theLexeme ))

        tokType:[ "=", cmpOp ]

    #elseif( @oneCset( input, {'>', '<', '='}, input, theLexeme ))

        tokType:[ theLexeme, cmpOp ]

    #else

        #error( "Illegal comparison operator: " + input )

    #endif
```

```
        // Handle the parentheses down here.

        #elseif( @oneChar( input, '(', input, theLexeme ))

            tokType:[ "(", lparen ]

        #elseif( @oneChar( input, ')', input, theLexeme ))

            tokType:[ ")", rparen ]


        // Anything else is an illegal character.

        #else

            #error
            (
                "Illegal character in expression: '" +
                @substr( input, 0, 1 ) +
                "' ($" +
                string( dword( @substr( input, 0, 1 ))) +
                ")"
            )
            ?input := @substr( input, 1, @length(input) - 1 );

        #endif

    endmacro;




    // Handle identifiers, constants, and sub-expressions within
    // paretheses within this macro.
    //
    //  terms-> identifier | intconst | '(' CmpOps ')'
    //
    // This compile time function does the following:
    //
    //  (1) If it encounters an indentifier, it emits the
    //      following instruction to the code stream:
    //
    //          push( identifier );
    //
    //  (2) If it encounters an (unsigned) integer constant, it emits
    //      the following instruction to the code stream:
    //
    //          pushd( constant_value );
    //
    //  (3) If it encounters an expression surrounded by parentheses,
    //      then it emits whatever instruction sequence is necessary
    //      to leave the value of that (unsigned integer) expression
    //      sitting on the top of the stack.
    //
    //  (4) If the current lexeme is none of the above, then this
    //      macro prints an appropriate error message.
    //
    //  The end result of the execution of this macro is the emission
    //  of some code that leaves a single 32-bit unsigned value sitting
    //  on the top of the 80x86 stack (assuming no error).
```

```
macro doTerms( expr ):termToken;

    ?expr := @trim( expr, 0 );
    ?termToken:tokType := lexer( expr );
    #if( termToken.tokClass = identifier )

        // If we've got an identifier, emit the code to
        // push that identifier onto the stack.

        push( @text( termToken.lexeme ));

    #elseif( termToken.tokClass = intconst )

        // If we've got a constant, emit the code to push
        // that constant onto the stack.

        pushd( @text( termToken.lexeme ) );

    #elseif( termToken.tokClass = lparen )

        // If we've got a parenthetical expression, emit
        // the code to leave the parenthesized expression
        // sitting on the stack.

        doCmpOps( expr );
        ?expr := @trim( expr, 0 );
        ?termToken:tokType := lexer( expr );
        #if( termToken.tokClass <> rparen )

            #error( "Expected closing parenthesis: " + termToken.lexeme )

        #endif


    #else

        #error( "Unexpected term: '" + termToken.lexeme + "'" )

    #endif

endmacro;



// Handle the multiplication, division, and modulo operations here.
//
// MulOps-> terms ( mulOp terms )*
//
// The above grammar production tells us that a "MulOps" consists
// of a "terms" expansion followed by zero or more instances of a
// "mulop" followed by a "terms" expansion (like wildcard filename
// expansions, the "*" indicates zero or more copies of the things
// inside the parentheses).
//
// This code assumes that "terms" leaves whatever operands/expressions
// it processes sitting on the 80x86 stack at run time.  If there is
// a single term (no optional mulOp/term following), then this code
// does nothing (it leaves the result on the stack that was pushed
// by the "terms" expansion).  If one or more mulOp/terms pairs are
// present, then for each pair this code assumes that the two "terms"
// expansions left some value on the stack.  This code will pop
// those two values off the stack and multiply or divide them and
// push the result back onto the stack (sort of like the way the
// FPU multiplies or divides values on the FPU stack).
//
// If there are three or more operands in a row, separated by
// mulops ("*" or "/") then this macro will process them in
// a left-to-right fashion, popping each pair of values off the
// stack, operating on them, pushing the result, and then processing
// the next pair.  E.g.,
//
//      i * j * k
//
//  yields:
//
//      push( i );  // From the "terms" macro.
//      push( j );  // From the "terms" macro.
//
//      pop( eax ); // Compute the product of i*j
//      mul( (type dword [esp]));
//      mov( eax, [esp]);
//
//      push( k );  // From the "terms" macro.
//
//      pop( eax );                 // Pop K
//      mul( (type dword [esp]));   // Compute K* (i*j) [i*j is value on TOS].
//      mov( eax, [esp]);           // Save product on TOS.


macro doMulOps( sexpr ):opToken;

    // Process the leading term (not optional).  Note that
    // this expansion leaves an item sitting on the stack.

    doTerms( sexpr );

    // Process all the MULOPs at the current precedence level.
    // (these are optional, there may be zero or more of them.)

    ?sexpr := @trim( sexpr, 0 );
    #while( @peekCset( sexpr, MulOps ))

        // Save the operator so we know what code we should
        // generate later.

        ?opToken := lexer( sexpr );

        // Get the term following the operator.

        doTerms( sexpr );

        // Okay, the code for the two terms is sitting on
        // the top of the stack (left operand at [esp+4] and
        // the right operand at [esp]).  Emit the code to
        // perform the specified operation.

        #if( opToken.lexeme = "*" )

            // For multiplication, compute
            // [esp+4] = [esp] * [esp+4] and
            // then pop the junk off the top of stack.
```

```
                pop( eax );
                mul( (type dword [esp]) );
                mov( eax, [esp] );

            #elseif( opToken.lexeme = "/" )

                // For division, compute
                // [esp+4] = [esp+4] / [esp] and
                // then pop the junk off the top of stack.

                mov( [esp+4], eax );
                xor( edx, edx );
                div( [esp], edx:eax );
                pop( edx );
                mov( eax, [esp] );

            #endif
            ?sexpr := @trim( sexpr, 0 );

        #endwhile

    endmacro;




    // Handle the addition, and subtraction operations here.
    //
    // AddOps-> MulOps ( addOp MulOps )*
    //
    // The above grammar production tells us that an "AddOps" consists
    // of a "MulOps" expansion followed by zero or more instances of an
    // "addOp" followed by a "MulOps" expansion.
    //
    // This code assumes that "MulOps" leaves whatever operands/expressions
    // it processes sitting on the 80x86 stack at run time.  If there is
    // a single MulOps item then this code does nothing.  If one or more
    // addOp/MulOps pairs are present, then for each pair this code
    // assumes that the two "MulOps" expansions left some value on the
    // stack.  This code will pop those two values off the stack and
    // add or subtract them and push the result back onto the stack.

    macro doAddOps( sexpr ):opToken;

        // Process the first operand (or subexpression):

        doMulOps( sexpr );

        // Process all the ADDOPs at the current precedence level.

        ?sexpr := @trim( sexpr, 0 );
        #while( @peekCset( sexpr, PlusOps ))

            // Save the operator so we know what code we should
            // generate later.

            ?opToken := lexer( sexpr );

            // Get the MulOp following the operator.

            doMulOps( sexpr );
            // Okay, emit the code associated with the operator.

            #if( opToken.lexeme = "+" )

                pop( eax );
                add( eax, [esp] );

            #elseif( opToken.lexeme = "-" )

                pop( eax );
                sub( eax, [esp] );

            #endif

        #endwhile

    endmacro;



    // Handle the comparison operations here.
    //
    // CmpOps-> addOps ( cmpOp AddOps )*
    //
    // The above grammar production tells us that a "CmpOps" consists
    // of an "AddOps" expansion followed by zero or more instances of an
    // "cmpOp" followed by an "AddOps" expansion.
    //
    // This code assumes that "MulOps" leaves whatever operands/expressions
    // it processes sitting on the 80x86 stack at run time.  If there is
    // a single MulOps item then this code does nothing.  If one or more
    // addOp/MulOps pairs are present, then for each pair this code
    // assumes that the two "MulOps" expansions left some value on the
    // stack.  This code will pop those two values off the stack and
    // add or subtract them and push the result back onto the stack.


    macro doCmpOps( sexpr ):opToken;

        // Process the first operand:

        doAddOps( sexpr );

        // Process all the CMPOPs at the current precedence level.

        ?sexpr := @trim( sexpr, 0 );
        #while( @peekCset( sexpr, CmpOps ))

            // Save the operator for the code generation task later.

            ?opToken := lexer( sexpr );

            // Process the item after the comparison operator.

            doAddOps( sexpr );


            // Generate the code to compare [esp+4] against [esp]
            // and leave true/false sitting on the stack in place
            // of these two operands.
            #if( opToken.lexeme = "<" )

                pop( eax );
                cmp( [esp], eax );
                setb( al );
                movzx( al, eax );
                mov( eax, [esp] );

            #elseif( opToken.lexeme = "<=" )

                pop( eax );
                cmp( [esp], eax );
                setbe( al );
                movzx( al, eax );
                mov( eax, [esp] );

            #elseif( opToken.lexeme = ">" )

                pop( eax );
                cmp( [esp], eax );
                seta( al );
                movzx( al, eax );
                mov( eax, [esp] );

            #elseif( opToken.lexeme = ">=" )

                pop( eax );
                cmp( [esp], eax );
                setae( al );
                movzx( al, eax );
                mov( eax, [esp] );

            #elseif( opToken.lexeme = "=" )

                pop( eax );
                cmp( [esp], eax );
                sete( al );
                movzx( al, eax );
                mov( eax, [esp] );

            #elseif( opToken.lexeme = "<>" )

                pop( eax );
                cmp( [esp], eax );
                setne( al );
                movzx( al, eax );
                mov( eax, [esp] );


            #endif

        #endwhile

    endmacro;



    // General macro that does the expression compliation.
    // The first parameter must be a 32-bit register where
    // this macro will leave the result.  The second parameter
    // is the expression to compile.  The expression compiler
    // will destroy the value in EAX and may destroy the value
    // in EDX (though EDX and EAX make fine destination registers
    // for this macro).
    //
    // This macro generates poor machine code.  It is more a
    // "proof of concept" rather than something you should use
    // all the time.  Nevertheless, if you don't have serious
    // size or time constraints on your code, this macro can be
    // quite handy.  Writing an optimizer is left as an exercise
    // to the interested reader.

    macro u32expr( reg, expr):sexpr;

        // The "returns" statement processes the first operand
        // as a normal sequence of statements and then returns
        // the second operand as the "returns" value for this
        // macro.

        returns
        (
            {

                ?sexpr:string := @string:expr;
                #if( !@IsReg32( reg ) )

                    #error( "Expected a 32-bit register" )

                #else

                    // Process the expression and leave the
                    // result sitting in the specified register.

                    doCmpOps( sexpr );
                    pop( reg );

                #endif
            },

            // Return the specified register as the "returns"
            // value for this compilation:

            @string:reg
        )



    endmacro;



    // The following main program provides some examples of the
    // use of the above macro:

    static
        x:uns32;
        v:uns32 := 5;


    begin TestExpr;
```

```
mov( 10, x );
mov( 12, ecx );

// Compute:
//
//  edi := (x*3/v + %1010 == 16) + ecx;
//
//  This is equivalent to:
//
//  edi := (10*3/5 + %1010 == 16) + 12
//      := ( 30/5 + %1010 == 16) + 12
//      := ( 6 + 10 == 16) + 12
//      := ( 16 == 16) + 12
//      := ( 1 ) + 12
//      := 13
//
//  This macro invocation emits the following code:
//
//  push(x);
//  pushd(3);
//  pop(eax);
//  mul( (type dword [esp]) );
//  mov( eax, [esp] );
//  push( v );
//  mov( [esp+4], eax );
//  xor edx, edx
//  div( [esp], edx:eax );
//  pop( edx );
//  mov( eax, [esp] );
//  pushd( 10 );
//  pop( eax );
//  add( eax, [esp] );
//  pushd( 16 );
//  pop( eax );
//  cmp( [esp], eax );
//  sete( al );
//  movzx( al, eax );
//  mov( eax, [esp+0] );
//  push( ecx );
//  pop( eax );
//  add( eax, [esp] );
//  pop( edi );


u32expr( edi, (x*3/v+%1010 == 16) + ecx );
stdout.put( "Sum = ", (type uns32 edi), nl );



// Now compute:
//
//  eax := x + ecx/2
//      := 10 + 12/2
//      := 10 + 6
//      := 16
//
// This macro emits the following code:
//
//  push( x );
//  push( ecx );
//  pushd( 2 );
```

```
        //  mov( [esp+4], eax );
        //  xor( edx, edx );
        //  div( [esp], edx:eax );
        //  pop( edx );
        //  mov( eax, [esp] );
        //  pop( eax );
        //  add( eax, [esp] );
        //  pop( eax );


        u32expr( eax, x+ecx/2 );
        stdout.put( "x=", x, " ecx=", (type uns32 ecx), " v=", v, nl );
        stdout.put( "x+ecx/2 = ", (type uns32 eax ), nl );


        // Now determine if (x+ecx/2) < v
        //  (it is not since (x+ecx/2)=16 and v = 5.)
        //
        //  This macro invocation emits the following code:
        //
        //  push( x );
        //  push( ecx );
        //  pushd( 2 );
        //  mov( [esp+4], eax );
        //  xor( edx, edx );
        //  div( [esp], edx:eax );
        //  pop( edx );
        //  mov( eax, [esp] );
        //  pop( eax );
        //  add( eax, [esp]);
        //  push( v );
        //  pop( eax );
        //  cmp( eax, [esp+0] );
        //  setb( al );
        //  movzx( al, eax );
        //  mov( eax, [esp+0] );
        //  pop( eax );


        if( u32expr( eax, x+ecx/2 < v ) ) then

            stdout.put( "x+ecx/2 < v" nl );

        else

            stdout.put( "x+ecx/2 >= v" nl );

        endif;

    end TestExpr;
```

Program 9.7 Uns32 Expression Compiler

9.4 Putting It All Together

The ability to extend the HLA language is one of the most powerful features of the HLA language. In this chapter you got to explore the use of several tools that allow you to extend the base language. Although a complete treatise on language design and implementation is beyond the scope of this chapter, further study in the area of compiler construction will help you learn new techniques for extending the HLA language. Later volumes in this text, including the volume on advanced string handling, will cover additional topics of interest to those who want to design and implement their own language constructs.