

# Teaching Assembly Language Using HLA

Randall Hyde  
rhyde@cs.ucr.edu  
<http://webster.cs.ucr.edu>

I first began teaching assembly language programming at Cal Poly Pomona in the Winter Quarter of 1987. I quickly discovered that good pedagogical material was difficult to come by; even the textbooks available for the course left something to be desired. As a result, my students were learning very little assembly language in the ten weeks available to the course. After about two quarters, I decided to do something about the textbook problem, so I began writing a text I entitled “How to Program the IBM PC Using 8088 Assembly Language” (obviously, this was back in the days when schools still used PCs made by IBM and the main CPU you could always count on was the 8088). “How to Program...” became the epitome of a “work in progress.” Each quarter I would get feedback from the students, update the text, and give it to Kinko’s (and the UCR Printing and Reprographics Department) to run off copies for my students the very next quarter.

The original “How to Program...” text provide a basic set of library routines to print strings, input characters and lines of text, and a few other basic functions. This allowed the students to quickly begin writing programs without having to learn about the INT instruction, DOS, or BIOS. However, I discovered that students were spending a significant time each quarter writing their own numeric conversion routines, string manipulation routines, etc. One student commented on “how much easier it was to program in ‘C’ than assembly language since all those conversions and string operations were built into the language.” I replied that the real savings were due more to the ‘C’ standard library than the language itself and that a comparable library for assembly language programmers would make assembly language programming almost as easy as ‘C’ programming. At that moment a little light went on in my head and I sat down and wrote the first few routines of what ultimately became the “UCR Standard Library for 80x86 Assembly Language Programmers” (You can still get a copy of the UCR stdlib from webster at the URL given above). As I finished each group of routines in the standard library, I incorporated them into my courses. This reaped immediate benefits as students spent less time writing numeric conversion routines and spent more time learning assembly language. My students were getting into far more advanced topics than was possible before the advent of the UCR Stdlib.

In the early 1990’s, the 8088 CPU finally died off and IBM was no longer the major supplier of PCs. Not only was it time to change the title of my text, but I needed to update references to the 8088 (that were specific to that chip) and bring the text into the world of the 80386 and 80486 processors. DOS was still King and 16-bit code was still what everyone was writing, but issues of optimization and the like were a little outdated in the text. In addition to the changes reflecting the new Intel CPUs, I also incorporated the UCR Standard Library into the text since it dramatically improved the speed at which students progressed beyond the basic assembly programming skills. I entitled the new version of the text “The Art of Assembly Language Programming,” an obvious knock-off of Knuth’s series (“The Art of Computer Programming”).

In early 1996 it became obvious to me that DOS was finally dying and I needed to modify “The Art of Assembly Language Programming” (AoA) to use Windows as the development platform. I wasn’t interested in having students write Windows GUI applications in assembly language (the time spent teaching event-oriented programming would interfere with the teaching of basic machine organization and assembly language programming), but it was clear that the days of writing code that arbitrarily pokes around in memory and accesses I/O addresses directly (things that AoA taught) were nearly over. So I decided to get started on a new version of AoA that used Windows as the basic development environment with the emphasis on writing console applications.

The UCR Standard Library was the single most important pedagogical tool I’d discovered that dramatically improved my students’ progress. As I began work on a new version of AoA for Windows 3.1 my first task was to improve upon the UCR Standard Library to make it even easier to use, more flexible, more efficient, and more “high level.” After six months of part time work I eventually gave up on the UCR Stdlib v2.0. The idea was right, unfortunately the tools at my disposal (specifically, MASM 6.11) weren’t quite up to the task at hand. I was writing some really tricky macros, obviously exploiting code inside MASM that Microsoft’s engineers had never run (i.e., I discovered lots of bugs). I would code in some workarounds to the defects only to have the macro package break at the next minor patch of MASM (e.g., from MASM 6.11a to MASM 6.11b). There was also a robustness issue. Although MASM’s macro capabilities are quite powerful and it *almost* let me do everything I wanted, it was very easy to confuse the macro package and then MASM would generate some totally weird (but absolutely correct) diagnostic messages that correctly described what was going wrong in the macro but made absolutely no sense whatsoever at all to a beginning assembly language student who use using the macro to print some data to the console device. As it became clear that the UCR Stdlib v2.0 would never be robust enough for student use, I decide to take a different approach.

About this time, I was talking with my Department Chair about the assembly language course. We were identifying some of the problems that students had learning assembly language. One problem, of course, was the paradigm shift- learning to solve problems using machine language rather than a high level language. The second problem we identified is that students get to apply very little of what they’ve learned from other courses to the assembly language class. A third problem was the primitive tools available to assembly language programmers. Energized by this discussion, I decided to see how I could solve these problems and improve the educational process.

Problem one, the paradigm shift, had to be handled carefully. After all, the whole purpose of having students take an assembly language programming course in the first place is to acquaint them with the low-level operation of the machine. However, I felt it was certainly possible to redefine parts of assembly language so that would be more familiar to students. For example, one might test the carry flag after an addition to determine if an unsigned overflow has occurred using code like the following:

```
add eax, 5
jnc NoOverflow
    << code to execute if overflow occurs >>
NoOverflow:
```

Although this code is fairly straight-forward, you would be surprised how many students cannot visualize this code on their own. On the other hand, if you feed them some pseudo code like:

```
add eax, 5
if( the carry flag is set ) then
    << code to execute if overflow occurs >>
endif
```

Those same students won't have any problems understanding this code. To take advantage of this difference in perspective, I decided to explore changing the definition of assembly language to allow the use of the "if condition then do something" paradigm rather than the "if a condition is false then skip over something" paradigm. Fundamentally, this does not change the material the student has to learn; it just presents it from a different point of view to which they're already accustomed. This certainly wasn't a gigantic leap away from assembly language as it existed in 1996. After all, MASM and other assemblers were already allowing statements like ".if" and ".endif" in the code. So I tried these statements out on a few of my students. What I discovered is that the students picked up the basic "high level" syntax very rapidly. Once they mastered the high level syntax, they were able to learn the low-level syntax (i.e., using conditional jumps) faster than ever before. What I discovered is something that Nicoderm CQ is pushing for their smoking cessation program: "learning assembly language in graduated steps (from high level to low level) is easier than going about it 'cold turkey.'"

The second problem, students not being able to leverage their programming skills from other classes, is largely linked to the syntax of Intel x86 assembly language. Many skills students pick up, such as programming style, indentation, appropriate programming construct selection, etc., are useless in a typically assembly language class. Even skills like commenting and choosing good variable names are slightly different in assembly language programs. As a result, students spend considerable (unproductive) time learning the new "rules of the game" when writing assembly language programs. This directly equates to less progress over the ten week quarter. Ideally, students should be able to applying knowledge like program style, commenting style, algorithm organization, and control construct selection they learned in a C/C++ or Pascal course to their assembly language programs. If they could, they'd be "up and writing" in assembly language much faster than before.

The third problem with teaching assembly language is the primitive state of the tools. While MASM provides a wonderful set of high level language control constructs, very little else about MASM supports this "brave new world" of assembly language I want to teach. For example, MASM's variable declarations leave a lot to be desired (the syntax is straight out of the 1960's). As I noted earlier, as powerful as MASM's macro facilities are, they weren't sufficient to develop a robust library package for my students. I briefly looked at TASM, but it's "ideal" mode fared little better than MASM. Likewise, while development environments for high level languages have been improving by leaps and bounds (e.g., Delphi and C++ Builder), assembly language programmers are still using the same crude command line tools popularized in the early 1970's. Code-

view, which is practically useless under Windows, is the most advanced tool Microsoft provides specifically for assembly language programmers.

Faced with these problems, I decided the first order of business was to create a new x86 assembly language and write a compiler for it. I decided to give this language the somewhat-less-than-original name of “the High Level Assembler,” or HLA (IBM and Motorola both already have assemblers that use a variant of this name). It took three years, but the first version of HLA was ready for public consumption in September of 1999.

I began using HLA in my CS 61 course (machine organization and assembly language programming) at UCR in the Fall Quarter, 1999. With no pedagogical material other than a roughly written reference guide to the language, I was expecting a complete disaster. It turns out that I was pleasantly surprised. Although the students did have major problems, the course went far more smoothly than I anticipated and we managed to cover about the same material I normally covered when using MASM.

Although things were going far better than I expected, this is not to say that things were going great, or even as smoothly as I would have liked. The major problem, of course, was the lack of a textbook. The only material the students had to study from were their lecture notes. Clearly something needed to be done about this. Of course, the whole reason for spending three years writing HLA was to allow me to write a new version of AoA. So in November, 1999, I began work on the new edition of the text. By the start of the Winter Quarter in January, 2000, I had roughed together five chapters, about 50% of the material was brand new, the other 50% was cut, pasted, and updated from the older version of the text. During the quarter I rushed out two more chapters bringing the total to seven. The Winter Quarter went far more smoothly than the Fall Quarter. Student projects were much better and the progress of the class outstripped any assembly language course I’d taught prior to that point. Clearly the class was benefiting from the use of HLA.

By the start of the Spring Quarter in April, 2000, I’d managed to make one proofreading pass over the first six chapters and I’d written the first draft of the eighth chapter. By the Winter Quarter 2001, I’d split the text into volumes and supplied five chapters for volume one, eight for volume two, and thirteen chapters each for volumes three and four. Although this is far more material than one course can cover, the extra material gives instructors flexibility with respect to what they want to teach. Certainly the first four volumes of AoA cover all the essential material most instructors want to teach in an assembly course.

Well, this has been a long-winded report of HLA’s justification. You’re probably wondering what HLA is and whether it is applicable to you (especially if you’re a programmer rather than an educator). Fair enough, the rest of this article will discuss the HLA system and how you would use it.

HLA is a technically a compiler, not an assembler. HLA v1.x converts an HLA source file into a MASM-compatible assembly language source file. This MASM file is then assembled and linked to produce a Win32 executable file. The HLA compiler automatically runs the assembler and linker, so these steps are transparent to the HLA user (other than the few extra seconds it takes to assemble and link the output file). This whole process takes only a few seconds (for example,

compiling, assembling, and linking the 750-line “x2p.hla” program in the HLA examples directory only takes about two seconds on a 266 MHz Pentium II system with UW SCSI drives). I am planning to emit object code directly in version 2.0 of HLA. Until then, an HLA user will need Microsoft’s MASM and linker. For those who would prefer to have HLA generate code for TASM, NASM, or some other assembler, the HLA compiler source code is available, have fun :-).

HLA is a Win32 console application and it generates Win32 applications. By default, it generates console applications although it does not restrict you to writing console applications under Windows. There is absolutely no support for DOS applications. While it is possible to write Linux applications with only minor changes to HLA, the development process for Linux applications is convoluted and hardly worthwhile. HLA v2.0 will address portability across 32-bit x86 operating systems. For now, using HLA is practical only under Win32 OSes (Win 95, 98, NT, and 2000).

When designing the HLA language, I chose a syntax that is very similar to common imperative high level languages like Pascal/Delphi, Ada, Modula-2, FORTRAN77, C/C++, and Java. That is not to say that HLA compiles Pascal programs, but rather, a Pascal programmer will note many similarities between Pascal and HLA (and ditto for the other languages). HLA stole many of the ideas for data declarations from the Algol based languages (Pascal, Modula-2, and Ada), it grabbed the ideas for many of its control structures from FORTRAN77, Ada, and C/C++/Java, and the structure of the HLA Standard Library is based on the C Standard Library. So regardless of which high level language you’re most comfortable with in this set, you’ll certainly recognize some elements of your favorite HLL in HLA.

A carefully written HLA program will look almost exactly like a high level language program. Consider the following sample program:

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

const
    HelloWorld := "Hello World";

begin SampleHLApgm;

    stdout.put
    (
        "The classical 'Hello World' program: ",
        HelloWorld,
        nl
    );

end SampleHLApgm;
```

This program does the obvious thing. Anyone with any high level language background can probably figure out everything except the purpose of “nl” (which is the newline string imported by the

standard library). This certainly doesn't look like an assembly language program; there isn't even a real machine instruction in sight. Of course, this is a trivial example; nonetheless, I've managed to write reasonable HLA programs that were just over 1,000 lines of code that contained only one or two identifiable machine language instructions. If it's possible to do this, how can I get away with calling HLA an assembly language?

The truth is, you can actually write a very similar looking program with MASM. Here's an example I trot out for unbelievers. This code is compilable with MASM (assuming you include the UCR Standard Library v2.0 and some additional code I've cut out for brevity:

```
var
    enum colors,<red,green,blue>

    colors c1, c2

endvar

Main
    proc
    mov     ax, dseg
    mov     ds, ax
    mov     es, ax

    MemInit
    InitExcept
    EnableExcept

    finit

    try

        cout     "Enter two colors:"
        cin       c1, c2
        cout      "You entered ",c1," and ",c2,nl
        .if       c1 == red

            cout   "c1 was red"

        .endif

    except $Conversion
        cout      "Conversion error occurred",nl

    except $Overflow
        cout      "Overflow error occurred",nl
```



```

                                endtry
                                CleanUpEx
                                ExitPgm                                ;DOS macro to quit program.
Main                            endp

```

As you can see, the only identifiable machine instructions here are the ones that initialize the segment registers at the beginning of the program (which is unnecessary in a Win32 environment). So let me blunt criticism from “die-hard” assembly fans right at the start: HLA doesn’t open up all kinds of new programming paradigms that weren’t possible before. With some really clever macros (e.g., `enum`, `cout`, and `cin` in the MASM code), it is quite possible to do some really amazing things. If you’re wondering why you should bother with HLA if MASM is so wonderful, don’t forget my comments about the robustness of these macros. Both HLA and MASM (with the UCR Standard Library v2.0) work great as long as you write perfect code and don’t make any mistakes. However, if you do make mistakes, the MASM macro scheme gets ugly real quick.

The “die-hard” assembly fan will probably make the observation that they would never write code like the MASM code I’ve presented above; they would write traditional assembly code. They *want* to write traditional code. They *don’t* want this high level syntax forced upon them. Well, HLA doesn’t force you to use high level control structures rather than machine instructions. You can always write the low level code if you prefer it that way. Here is the original HLA program rewritten to use familiar machine instructions:

```

program SampleHLApgm2;
#include( "stdlib.hhf" )

data
    dword 37, 37;
    TchWpStr: dword;
               byte  "The classical 'Hello World' program: ",0,0,0;

    dword 11, 11;
    HWstr:     dword;
               byte  "Hello World",0;

begin SampleHLApgm2;

    lea( eax, TchWpStr );
    push( eax );
    call stdout.puts;

    lea( eax, HWstr );
    push( eax );
    call stdout.puts;

    call stdout.newln;

```

```
end SampleHLApgm2;
```

The `stdout.puts` and `stdout.newln` procedures come from the HLA Standard Library. I will leave it up to the interested reader to translate these into Win API Write calls if this code isn't sufficiently low level to satisfy. Note that HLA strings are not simple zero terminated strings like C/C++. This explains the extra zeros and dword values in the DATA section (the dword values hold the string lengths; I offer these without further explanation, see the HLA documentation for more details on HLA's string format).

One thing you've probably noticed from this second example is that HLA uses a functional notation for assembly language statements. That is, the instruction mnemonics look like function calls in a high level language and the operands look like parameters to those functions. The neat thing about this notation is that it easily allows the use of "*instruction composition*." Instruction composition, like functional composition, means that you get to use one instruction as the operand of another. For example, an instruction like "`mov( mov( 0, eax ), ebx );`" is perfectly legal in HLA. The HLA compiler will compile the innermost instruction first and then substitute the destination operand of the innermost instruction for the operand position occupied by the instruction. HLA's MOV instruction takes the generic form "`MOV( source, destination );`" so the former instruction translates to the following two instruction sequence:

```
mov( 0, eax );    // intel syntax:  mov eax, 0
mov( eax, ebx ); // intel syntax:  mov ebx, eax
```

By and of itself, instruction composition is somewhat interesting, but programmers striving to write readable code need to exercise caution when using instruction composition. It is real easy to write some really unreadable code if you abuse instruction composition. E.g., consider:

```
mov( add( mov( 0, eax ), sub( ebx, ecx)), edx ), mov( i, esi )); //Hopefully I got this right!
```

Egads! What does this mess do? Some might consider the inclusion of instruction composition in HLA to be a fault of the language if it allows you to write such unreadable code. However, I've never felt it was the language syntax's job to enforce good programming style. If there's really a reason for writing such messy code, the compiler shouldn't prevent it.

Although you can produce some truly unreadable messes with instruction composition, if you use it properly it can enhance the readability of your programs. For example, HLA lets you associate an arbitrary string with a procedure that HLA will substitute for that procedure name when the procedure call appears as an operand of another instruction. Most functions that return a value in a register specify that register name as their "returns" string (the string HLA substitutes for the procedure call). For example, the "`str.eq( str1, str2)`" function compares the two string operands and returns true or false in AL depending on the result of the comparison. This allows you to write code like the following:

```
if( str.eq( str1, "Hello" )) then
```



```
    stdout.put( "str1 = 'Hello'" nl );  
  
endif;
```

HLA directly translates the IF statement into the following sequence:

```
str.eq( str1, "Hello" );  
if( al ) then  
  
    stdout.put( "str1= 'Hello'" nl );  
  
endif;
```

(If a register name appears where a boolean expression is expected, as AL does in the IF statement above, HLA emits a TEST instruction to see if the register contains a non-zero value.)

Arguably, the former version is a little more readable than the latter version. Instruction composition, when you use it in this fashion, lets you write code that “looks” a little more high level without the compiler having to generate lots of extra code (as it would if HLA supported a generalized arithmetic expression parser).

Like MASM, HLA supports a wide variety of high level control structures. HLA’s set is both higher level and lower level at the same time. There are two reasons HLA’s control structures aren’t always as powerful as MASM’s. First, with the sole exception of object method invocations, I made a rule that HLA’s high level control structures would not modify any general purpose registers behind the programmer’s back. MASM, for example, may modify the value in EAX for certain boolean expressions it must compute. Second, remember that the primary goal of HLA is to teach assembly language; yes, it’s supposed to ease the learning curve, but still the goal is to teach assembly language. It is possible to get carried away with the high level language features and then wind up with an “assembler” that lets students write their assembly language programs in a high level language. In general, most HLA boolean expressions compile into two instructions: a CMP and a conditional jump.

Although I designed HLA as a tool to teach assembly language programming, this is also a tool that I intend to use so I included lots of goodies for advanced assembly language programmers. For example, HLA’s macro facilities are more powerful than I’ve seen in any programming language based macro processor. One unique feature of HLA’s macro preprocessor is the ability to create “context free” control structures using macros. For example, suppose that you decide that you need a new type of looping construct that HLA doesn’t provide; let’s say, a loop that will repeat once for each character in a string supplied as a parameter to the loop. Let’s call this loop “OnceForEachChar” and decide on the following syntax:

```
OnceForEachChar( SomeString )  
  
    << Loop Body >>
```

```
endOnceForEachChar;
```

On each iteration of this loop, the AL register will contain the corresponding character from the string specified as the OnceForEachChar operand. You can easily implement this loop using the following HLA macro:

```
macro OnceForEachChar( SomeString ): TopOfLoop, LoopExit;

    pushd( -1 );          // index into string.

    TopOfLoop:
        inc( (type dword [esp] ));    // Bump up index into string.
        #if( @IsConst( SomeString ) )

            lea( eax, SomeString );    // Load address of string
                                         // constant into EAX.

        #else

            mov( SomeString, eax );    // Get ptr to string.

        #endif
        add( [esp], eax );             // Point at next available
                                         // character
        mov( [eax], al );             // Get the next available
                                         // character
        cmp( al, 0 );                 // See if we're at the end
                                         // of the string
        je LoopExit;

    terminator endOnceForEachChar;

        jmp TopOfLoop;               // Return to the top of the
                                         // loop and repeat.

    LoopExit:
        add( 4, esp );               // Remove index into string from stack.

endmacro;
```

Anyone familiar with MASM's macro processor should be able to figure out most of this code. Note that the symbols "TopOfLoop" and "LoopExit" are local symbols to this macro. Hence, if you repeat this macro several times in the code, HLA will emit different actual labels for these symbols to the MASM output file. The "@IsConst" is an HLA compile-time function that returns true if its operand is a constant. Obtaining the address for a constant is fundamentally different than obtaining the address of a string variable (since HLA string variables are actually pointers to

the string data). The most interesting feature of this macro definition is the “terminator” line. This actually defines a second macro that is active only after HLA encounters the “OnceForEachChar” macro and control returns to the first statement after the OnceForEachChar invocation. Invocation of “context free” macros always occur in pairs; that is, for every “OnceForEachChar” invocation there must be a matching “endOnceForEachChar” invocation. The following program demonstrates this macro in use, it also demonstrates that you can nest this newly created control structure in your program:

```
program SampleHLApgm3;
#include( "stdlib.hhf" )

macro OnceForEachChar( SomeString ): TopOfLoop, LoopExit;

    pushd( -1 );          // index into string.

    TopOfLoop:
        inc( (type dword [esp] ) );
        #if( @IsConst( SomeString ) )

            lea( eax, SomeString );

        #else

            mov( SomeString, eax );

        #endif
        add( [esp], eax );
        mov( [eax], al );
        cmp( al, 0 );
        je LoopExit;

terminator endOnceForEachChar;

        jmp TopOfLoop;
LoopExit:
    add( 4, esp );

endmacro;

static
    strVar: string := ":" nl;

begin SampleHLApgm3;
```

```

OnceForEachChar( "Hello" )

    stdout.putc( al );
    OnceForEachChar( strVar )

        stdout.putc( al );

    endOnceForEachChar;

endOnceForEachChar;

end SampleHLApgm3;

```

This program produces the output:

```

H:
e:
l:
l:
o:

```

Here's the MASM code the compiler emits for the sequence above (the “strings” segment was moved for clarity):

```

strings          segment page public 'data'
                  align      4
?635_len         dword      5
                  dword      5
?635_str         byte       "Hello",0,0,0
strings          ends

                  pushd      -1

?634__0278_:
                  inc        dword ptr [esp+0]          ;(type dword [esp])
                  lea        eax, ?635_str
                  add        eax, [esp+0] ;[esp]
                  mov        al, [eax+0] ;[eax]
                  cmp        al, 0
                  je         ?636__0279_

```

```

        push    eax
        call    stdio_putc        ;putc
        pushd   -1

?639__027d_:
        inc     dword ptr [esp+0]      ;(type dword [esp])
        mov     eax, dword ptr ?630_strVar[0] ;strVar
        add     eax, [esp+0] ;[esp]
        mov     al, [eax+0] ;[eax]
        cmp     al, 0
        je      ?640__027e_
        push    eax
        call    stdio_putc        ;putc
        jmp     ?639__027d_

?640__027e_:
        add     esp, 4
        jmp     ?634__0278_

?636__0279_:
        add     esp, 4

```

In addition to the “terminator” clause, HLA macros also support a “keyword” clause that let you bury reserved words within a context-free language construct. For example, the HLA language does not provide a SWITCH/CASE statement. This omission was intentional. Rather than build the SWITCH/CASE statement into the HLA language, I implemented the SWITCH .. CASE .. DEFAULT .. ENDCASE statement using HLA’s macro facilities (as a demonstration of HLA’s power). An HLA SWITCH statement takes the following form:

```

switch( reg32 )

    case( constantList1 )
        << statements >>

    case (constantList2 )
        << statements >>
        .
        .
        .
    default // This is optional
        << statements >>

endswitch;

```

The switch macro implements the “switch” and “endswitch” reserved words using the macro and terminator clauses in the macro declaration. It implements the “case” and “default” reserved words using the HLA “keyword” clause in a macro definition. The “keyword” clause is similar to the “terminator” clause except it doesn’t force the end of the macro expansion in the invoking code. The actual code for the HLA SWITCH statement is a little too complex to present here, so I will extend the example of the OnceForEachChar macro to demonstrate how you code use the “keyword” clause in a macro.

Let’s suppose you wanted to add a “\_break” clause to the “OnceForEachChar” loop ( I’m using “\_break” with an underscore because “break” is an HLA reserved word). You could easily modify the “OnceForEachChar” macro to achieve this by using the following code:

```
macro OnceForEachChar( SomeString ): TopOfLoop, LoopExit;

    pushd( -1 );          // index into string.

    TopOfLoop:
        inc( (type dword [esp] ) );
        #if( @IsConst( SomeString )

            lea( eax, SomeString );

        #else

            mov( SomeString, eax );

        #endif
        add( [esp], eax );
        mov( [eax], al );
        cmp( al, 0 );
        je LoopExit;

keyword _break;
    jmp LoopExit;

terminator endOnceForEachChar;

    jmp TopOfLoop;
LoopExit:
    add( 4, esp );

endmacro;
```

The “keyword” clause defines a macro (“\_break”) that is active between the “OnceForEachChar” and “endOnceForEachChar” invocations. This macro simply expands to a jmp instruction that



exits the loop. Note that if you have nested “OnceForEachChar” loops and you “\_break” out of the innermost loop, the code only jumps out of the innermost loop, exactly as you would expect.

HLA’s macro facilities are part of a larger feature I refer to as the “HLA Compile-Time Language.” HLA actually contains a built-in interpreter than executes while it is compiling your program. The compile-time language provides conditional compilation ( the #IF..#ELSE..#ENDIF statements in the previous example), interpreted procedure calls (macros), looping constructs (#WHILE..#ENDWHILE), a very powerful constant expression evaluator, compile-time I/O facilities (#PRINT, #ERROR, #INCLUDE, and #TEXT..#ENDTEXT), and dozens of built-in compile time functions (like the @IsConst function above).

The HLA built-in string functions (not to be confused with the HLA Standard Library’s string functions) are actually powerful enough to let you write a compiler for a high level language completely within HLA. I mentioned earlier that it is possible to write an expression compiler within HLA; I was serious. The HLA compile-time language will let you write a sophisticated recursive descent parser for arithmetic expressions (and other context-free language constructs, for that matter).

HLA is a great tool for creating low-level Domain Specific Embedded Languages (DSELs). DSELs are mini-languages that you create on a project by project basis to help reduce development time. HLA’s compile time language lets you create some very high level constructs. For example, HLA implements a very powerful string pattern matching language in the “patterns” module found in the HLA Standard Library. This module lets you write pattern matching programs that use techniques found in language like SNOBOL4 and Icon. As a final example, consider the following HLA program that translate RPN (reverse polish notation) expressions into their equivalent assembly language (HLA) statements and displays the results to the standard output:

```
// This program translates user RPN input into an
// equivalent sequence of assembly language instrs (HLA fmt).

program RPNtoASM;

#include( "stdlib.hhf" );

static

    s:                string;
    operand:          string;
    StartOperand:     dword;

macro mark;

    mov( esi, StartOperand );
```

```

endmacro;

macro delete;

    mov( StartOperand, eax );
    sub( eax, esi );
    inc( esi );
    sub( s, eax );
    str.delete( s, eax, esi );

endmacro;

procedure length( s:string ); returns( "eax" ); nodisplay;
begin length;

    push( ebx );
    mov( s, ebx );
    mov( (type str.strRec [ebx]).length, eax );
    pop( ebx );

end length;

begin RPNtoASM;

    stdout.put( "-- RPN to assembly --" nl );
    forever

        stdout.put( nl nl "Enter RPN sequence (empty line to
quit): " );
        stdin.a_gets();
        mov( eax, s );
        breakif( length( s ) = 0 );
        while( length( s ) <> 0 ) do

            pat.match( s );

            // Match identifiers and numeric constants

            mark;
            pat.zeroOrMoreWS();
            pat.oneOrMoreCset( { 'a'..'z', 'A'..'Z', '0'..'9',
'_' } );

            pat.a_extract( operand );
            stdout.put( "    pushd( ", operand, " );" nl );
            strfree( operand );

```

```

delete;

pat.alternate;

// Handle the "+" operator.

mark;
pat.zeroOrMoreWS();
pat.oneChar( '+' );
stdout.put
(
    "    pop( eax );" nl
    "    add( eax, [esp] );" nl
);
delete;

pat.alternate;

// Handle the '-' operator.

mark;
pat.zeroOrMoreWS();
pat.oneChar( '-' );
stdout.put
(
    "    pop( eax );" nl
    "    pop( ebx );" nl
    "    sub( eax, ebx );" nl
    "    push( ebx );" nl
);
delete;

pat.alternate;

// Handle the '*' operator.

mark;
pat.zeroOrMoreWS();
pat.oneChar( '*' );
stdout.put
(
    "    pop( eax );" nl
    "    imul( eax, [esp] );" nl
);
delete;

```

```

pat.alternate;

// handle the '/' operator.

mark;
pat.zeroOrMoreWS();
pat.oneChar( '/' );
stdout.put
(
    "    pop( ebx );" nl
    "    pop( eax );" nl
    "    cdq();" nl
    "    idiv( ebx, edx:eax );" nl
    "    push( ebx );" nl
);
delete;

pat.if_failure

// If none of the above, it must be an error.

stdout.put( nl "Illegal RPN Expression" nl );
mov( s, ebx );
mov( 0, (type str.strRec [ebx]).length );

pat.endmatch;

endwhile;

endfor;

end RPNtoASM;

```

Consider for a moment the code that matches an identifier or an integer constant:

```

mark;
pat.zeroOrMoreWS();
pat.oneOrMoreCset( {'a'..'z', 'A'..'Z', '0'..'9', '_' } );
pat.a_extract( operand );
stdout.put( "    pushd( ", operand, " );" nl );
strfree( operand );
delete;

```

The “mark;” invocation saves a pointer into the “s” string where the current identifier starts. The pat.ZeroOrMoreWS pattern matching function skips over zero or more whitespace characters.

The `pat.OneOrMoreCset` pattern match function matches one or more alphanumeric and underscore characters (a crude approximation for identifiers and integer constants). The `pat.a_extract` function makes a copy of the string between the “mark” and the “a\_extract” calls (this corresponds to the whitespace and identifier/constant). The `stdout.put` statement emits the HLA machine instruction that will push this operand on to the x86 stack for later computations. The remaining statements clean up allocated string storage space and delete the matched string from “s”.

Although the “`pat.xxxxx`” statements look like simple function calls, there’s actually a whole lot more going on here. HLA’s pattern matching facilities, like SNOBOL4 and Icon, support *success*, *failure*, and *backtracking*. For example, if the `pat.oneOrMoreChar` function fails to match at least one character from the set, control does not flow down to the `pat.a_extract` function. Instead, control flows to the next “`pat.alternate`” or “`pat.if_failure`” clause. Some calls to HLA pattern matching routines may even cause the program to back up in the code and reexecute previously called functions in an attempt to match a difficult pattern (i.e., the backtracking component). This article is not the place to get into the theory of pattern matching; however, these few examples should be sufficient to show you that something really special is going on here. *And all these facilities were developed using the HLA compile-time language.* This should give you a small indication of what is possible when using the HLA compile time language facilities.

The HLA language is far too rich to describe in this short article (the *\*very\** rough documentation for the language is nearly 300 pages long). For more information, check out the on-line documentation for HLA at <http://webster.cs.ucr.edu>. Someday, you’ll also be able to learn about HLA via “The Art of Assembly Language Programming, HLA/Windows version.” I will keep interested individuals updated on the progress of AoA at the Webster web site.

HLA is totally free. It is public domain software and there are no restrictions on its use, the use of the HLA standard library, or the HLA compiler source code. Do whatever you want with it and have a lot of fun!