HLA v2.0 Lexical Analyzer Design Documentation

This document describes the internal operation of HLA v2.0's lexical analyzer (scanner). This document assumes that the reader is familiar with compiler theory and terminology. This document also assumes that the reader is comfortable with the HLA language and 80x86 assembly language in general.

The HLA lexical analyzer (*lexer*) is responsible for scanning the source input file and translating text sequences ("lexemes") into small objects that the compiler can easily process (e.g., integers). These small values are often called "tokens" (although, technically, a *token* is all the data associated with a lexeme, not just the small integer encoding of that lexeme). The lexical analyzer is also responsible for converting sequences of digits into their numeric form as well as processing other literal constants, for removing comments and whitespace from the source file, and for taking care of many other mechanical details.

Lexical analysis, or *scanning*, is one of the most time-consuming components of the compilation process. Therefore, care must be taken when writing the lexer or the resulting compiler/assembler will be slower than it needs to be. One way to speed up software, of course, is to write that code in assembly language rather than a high level (or very high level language). High performance lexical analyzers, in particular, can make use of algorithms that are more easily written in assembly language than in higher-level languages. HLA v1.x's lexer, for example, was written using the LEX/FLEX language. HLA v2.x's lexical analyzer was written in assembly language (HLA). Although HLA v2.x's lexer isn't written in the most hand-optimized form, it is often an order of magnitude (or better) faster than HLA v1.x's lexer. Part of the reason for the speed improvement is due to the use of assembly language (and the fact that it is easier and more natural to express high-performance algorithms in assembly), the other part of the reason is that HLA v2.x's lexer uses better algorithms.

One problem with highly-optimized algorithms is that they are often difficult to understand and modify. This is especially true for hand-optimized assembly code (where the programmer counts cycles, rearranges the instructions to speed up the code, and uses non-obvious instruction sequences to achieve some goal). HLA v2.x's lexer was not written this way. The lexical analzyer probably sacrifices on the order of 2x or 3x performance so that the code is readable and easy to maintain. The lexer really is fast enough on most modern machines. Since I expect the need to modify HLA's lexer will arise in the future, I did not feel that it was wise to sacrifice readability for a factor of two or three speed, especially when that performance would not be noticed on faster machines (hey, once compile times are less than a second, no one really notices if one version of the code is twice as fast as another). Some may consider this a cop-out; if you're offended by this attitude, the sources are there, feel free to modify them yourself.

As far as performance is concerned, perhaps the most important issue is how efficiently the lexer processes the source file is the speed with which the lexer can read the individual characters from the source file. Simple-minded lexical analyzers (e.g., those you would expect a student to write in a compiler construction course) typically make an OS call like "getc" to read each individual character from the source file. This is an extremely slow approach though it does have the benefit of producing easy to read and understand code; too slow for HLA, though (by a couple of orders of magnitude).

A better solution is to use "blocked I/O" and read the source file from secondary storage in blocks roughly equivalent to the disk's sector size. To retrieve characters from the source file the lexer calls a "getc" function (or, better yet, a macro) that retrieves a single byte from the buffer on each call and reloads the buffer each time it retrieve's a sector's worth of data. This scheme is often hundreds of times faster but is slightly more complex since the code has to check for the end of the buffer and transparently reload the buffer when it is empty. Blocked I/O is more efficient because it avoids the overhead of an operating system call (which is very expensive) on each character and amortizes the cost of the OS call across hundreds or thousands of characters (the sector size).

One problem with the blocked I/O scheme is that it must check for the end of the buffer before processing each character so it can reload the buffer when the buffer is empty. This may not seem like much (it's basically a load, compare, and conditional jump), but this turns out to be a fair portion of the code associated with a "getc" function, especially if you implement the function as a macro. Another problem with this scheme is that lexers sometimes read one or two characters too many and have to put some characters back into the buffer to reread them later. This creates a problem if the character you're trying to put back is the last character of the previous buffer read from disk. It takes extra logic to handle this and lexers often have to execute some code to handle this situation on every character they read. A technique some higher-performance lexers use is to employ "double-buffering." Here the lexer alternates

between two buffers while processing the source file. If it needs to put back some text that appeared in the previous buffer, well, that buffer is still in memory so there isn't a problem.

HLA's lexer takes the concept of double-buffering to its logical extreme. HLA uses memory-mapped files to process the source file. Therefore, as far as HLA is concerned, the entire source file is just a big array in memory. Processing characters in the file, to HLA, is nothing more than incrementing a pointer and accessing the byte whose address the pointer contains (okay, HLA still has to check for the end of the file). This reduces the overhead of the "getc" operation to three instructions: an INC instruction, an instruction to load the pointer into a register (often this is unnecessary since HLA generally keeps this pointer in ESI), and an instruction to copy the byte from this address into the AL register.

HLA's lexer only checks for the end of a file between tokens. This could create a problem if EOF occurs in the middle of a lexeme (e.g., an identifier ends with EOF). To avoid this problem, HLA assumes that all source files end with a new line sequence; actually, HLA checks each file when it opens the file to ensure that the file ends with a new line and displays a warning message if this is not the case. Strictly speaking, most of the time HLA doesn't actually require a new line at the end of the file, but in some very rare situations it is possible for HLA to crash if the source file doesn't end with a new line, hence the warning. The alternative is to check for EOF after each character the lexer reads; this activity, however, nearly doubles the time HLA needs to process characters in identifiers and certain other multi-character sequences.

The "getc" function has traditionally been the greatest time sink in a typical lexical analyzer. HLA has reduced the cost of this function to almost nothing and this is largely responsible for the high performance achieved by the lexer. Once you speed up the "getc" function, the next two areas where lexers have performance problems is separating lexemes by their types and differentiating reserved words from identifiers.

To separate lexemes you begin by looking at the first character of each lexeme. HLA uses a traditional mechanism to do this: a jump table (i.e., SWITCH statement). When HLA scans each new lexeme, it fetches a character from the file and uses that character's ASCII code as an index into a jump table and transfers to a section of code that processes lexemes beginning with (or consisting only of) that particular character. Although memory accesses (the jump table) are somewhat expensive, this scheme is much faster than just about any other scheme that will differentiate the many different lexeme classes that HLA supports.

Reserved Words and Identifiers

The other major time-consuming task is differentiating reserved words and identifiers. HLA supports three types of special identifiers (reserved words): compile-time statements, compile-time functions, and standard reserved words.

Compile-Time Statements

Compile-time statements (and a few other directives) always begin with a "#" and only compile-time statement begin with this symbol, so it is very easy to differentiate compile-time statements and other lexemes. To differentiate the individual compile-time statements, the HLA lexer uses a simple linear search. Linear searches, of course, are slow, but this doesn't impact the performance of the HLA lexer much because: (1) There are only a dozen or so compile-time statements, and (2) compile-time statements rarely occur in a source file. Here, the simplicity of "tokenizing" HLA's compile-time statements outweighs the dubious benefits of a higher-performance algorithm.

Compile-Time Function Names and Identifiers

HLA's compile-time function names and pseudo-variables aways begin with an "@" symbol. Like the compile-time statements, it's very easy to differentiate compile-time function names from other lexemes. Differentiating compile-time functions from one another, however, is not as easy as differentiating the compile-time statements. Rather than a dozen, there are almost 200 different compile-time function identifiers. Although these names don't appear that frequently in an HLA source file, the number of these functions prevents the use of a simple linear search.

The HLA lexer uses a combination hash/binary search algorithm to differentiate the compile-time function identifiers (or reject an illegal identifier). First, HLA removes the first character of the identifier (always an "@") and then transfers to one of sixteen different code sequences depending on the length of the identifier (compile-time function names must be 1..16 characters long). These code sequences then use a binary search algorithm to compare the cur-

rent lexeme against the known compile-time function names. It takes, at most, five string comparisons to accept or reject a compile-time function name.

The string comparison the compile-time function recognition code uses is extremely efficient. When the lexer determines the length of the purported compile-time function name, it also loads the characters of the string into the EAX, EBX, ECX, and EDX registers. Strings of length 1..4 are passed in EAX, strings of length 5..8 are passed in EAX and EBX, strings of length 9..12 are passed in EAX, EBX, and ECX, and, finally, strings of length 13..16 are passed in EAX, EBX, ECX, and EDX. Therefore, to compare strings, all the code needs do is compare one to four registers against a set of eight, sixteen, or thirty-two bit constants. The binary search is accomplished by the targets of the branches after these comparisons. A typical string comparison looks like the following:

```
cmpfuncs_6_7:
    cmp( eax, funcstr( "istype", 0, 4) );
    jb cmpfuncs_6_8;
    ja cmpfuncs_6_10;
    cmp( bx, funcstr( "istype", 4, 2) );
    jb cmpfuncs_6_8;
    ja cmpfuncs_6_10;
    mov( tkn_at_istype, eax );
    mov( tc_func, ebx );
    jmp funcsDone;
```

The code above compares the six-character string in BX:EAX against the string constant “istype”. This checks the string to see if it matches the compile-time function “@istype” (remember, the lexer strips off the “@” because all compile-time function identifiers begin with “@”). In this code, “funcstr” is a macro that converts one to four characters of a string constant to an integer value whose bytes correspond to the ASCII codes for the characters in the string. The last two parameters of the “funcstr” macro are the starting position and length values for substring extraction. For example, funcstr(“abcdefg”, 0, 4) extracts the string “abcd” and translates it to the 32-bit value $6162_6364 (the ASCII codes for “a”, “b”, “c”, and “d”). The JA and JB instructions above transfer control to an appropriate sequence if the function name appearing in BX:EAX is lexicographically less than or greater than “istype”. If the current identifier is equal to “istype” then control falls down to the code that returns the token values in the EAX and EBX registers.

Writing and maintaining the code that does the binary search by hand would be very difficult. Adding a new function to the compile-time function list could force you to rearrange many comparison sequence (if you want to keep the binary search tree balanced). To make maintenance of this code easy, this code is machine generated rather than hand-generated. There is a program in the “rw” directory of the HLA v2.x source distribution that automates the creation of the code above. This program is “funcs.exe” and the source code can be found in the “funcs.hla” file. This program reads the file “funcs.txt” and creates a file named “cmpfuncs.hla”. The “cmpfuncs.hla” file is the HLA code that searches for one of the compile-time function names using a binary search.

The “funcs.txt” file is a standard ASCII text file containing all the reserved compile-time function names. Each line in this file contains a single function name. The function names do not include the leading “@” character. Here are the first few lines of this file:

```
a
abs
addofs1st
ae
arity
b
be
bound
byte
c
```

These lines correspond to the HLA compile-time function IDs @a, @abs, @addofs1st, @ae, @arity, @b, @be, @bound, @byte, and @c, respectively.

The "funcs.exe" program requires that you sort the lines in the "funcs.txt" input file. If you want to add a single function name to "funcs.txt" it's probably easiest to manually insert the reserved word at an appropriate point in the file (take care, however, if you put the reserved word in the wrong spot the binary search may fail to locate that identifier and other identifiers in the file). When inserting more than a single reserved word into the file, it's best to have the computer sort the file for you. To do this, make a copy of the "funcs.txt" file (perhaps to the file "f.txt") and insert the new reserved words, one per line, at the beginning of the file (actually, you can insert them anywhere, but the beginning of the text file is a convenient place to put them). Then you can sort the file using the "SORT.EXE" program using the following command line sequence statement.

```
sort f.txt >funcs.txt
```

Before you do this, of course, make sure that you have backed up the "funcs.txt" file.

Once you've sorted the compile-time function names text file, the next step is to run the "funcs.exe" program to generate the code to perform the binary search on these strings. To do this, execute the following command (this assumes you've compiled the "funcs.hla" program):

```
funcs
```

This program should output 16 lines of text to the display describing the number of compile-time function identifiers whose IDs have lengths in the range 1..16. This program will also write the HLA code to compute the binary search to the "cmpfuncs.hla" source file. In addition to writing the HLA code for the binary search to the "cmpfuncs.hla" file, the "funcs.exe" program also creates the "fTokens.hhf" header file that contains "token" constant declarations for each of the reserved words. This header file contains lines like the following:

```
const

    tkn_at_a              := tkn_endTknList + $0000_0000;
    tkn_at_abs            := tkn_endTknList + $0000_0001;
    tkn_at_addofs1st      := tkn_endTknList + $0000_0002;
    tkn_at_ae             := tkn_endTknList + $0000_0003;
    tkn_at_arity          := tkn_endTknList + $0000_0004;
    tkn_at_b              := tkn_endTknList + $0000_0005;
    tkn_at_be             := tkn_endTknList + $0000_0006;
    tkn_at_bound          := tkn_endTknList + $0000_0007;
    tkn_at_byte           := tkn_endTknList + $0000_0008;
    tkn_at_c              := tkn_endTknList + $0000_0009;
```

As you can see from this example, each token identifier for the compile-time function names consists of the prefix "tkn_at_" followed by the compile-time function identifier. The "tkn_endTknList" identifier is created by a different program and is the value (plus one) of the last reserved word token (i.e., the values associated with HLA's standard reserved words). Near the end of the "fTokens.hhf" header file there is a statement similar to the following:

```
    tkn_endFTknList       := tkn_endTknList + $0000_00AC;
```

(the value "$0000_00AC" will vary depending on the number of actual items in the "funcs.txt" file). This provides a symbol the lexer can use to determine where to begin additional token constants that follow the compile-time function values. The "hlacompiler.hhf" header file (which is written by hand) includes the "fTokens.hhf" header file to define the token values for the the compile-time function identifiers. The "hlacompiler.hhf" header file is where many of the other token values are defined as well (and, hence, uses the value of the "tkn_endFTknList" identifier).

The "funcs.exe" program does not generate all the code associated with scanning compile-time identifiers; it only generates the code that does a binary search for the identifier. To actually lex compile-time identifiers, some additional (hand-written) code is necessary. This code is the "CheckFuncs" procedure found in the "lexRW.hla" source file. The "CheckFuncs" procedure has several responsibilities. The main lexer calls the "CheckFuncs" proce-

dure whenever it finds the "@" character in the input stream. Upon entry, the first thing the "CheckFuncs" procedure has to do is to scan the identifier that immediately follows the "@". This is done with a simple loop that repeats for each alphanumeric character that immediately follows the "@" character (note that this code only allows alphanumeric characters, "_" and other symbols are not legal in a compile-time function identifier; you will have to change "CheckFuncs" if you want to allow other symbols). While collecting the characters following the "@", the "CheckFuncs" procedure also computes the length of the identifier. Valid compile-time function IDs have lengths in the range 1..16. While it is possible to modify the code to handle longer identifiers, doing so is a bit of work; so you should strive to keep your compile-time function identifiers below 17 characters in length so that major modifications to "CheckFuncs" won't be necessary.

After computing the length of the identifier (not including the "@"), the "CheckFuncs" procedure jumps to one of 17 code sequences depending on the length of the string (including one code sequence that handles all lengths outside the range 1..16). The code sequences corresponding to lengths 1..16 load the EAX, EBX, ECX, and EDX registers with the characters of the string and then jump to some statement within the "cmpfuncs.hla" code to begin the binary search.

The "cmpfuncs.hla" code actually contains 16 different binary search sequences, one for each string length. By only having to compare strings of a fixed length, the code is a bit more efficient. The initial comparison for each binary search sequence (one for each length) begins with an identifier of the following form:

cmpfuncs_*n*_0

where *n* represents an integer value in the range 1..16 corresponding to the length of the string. The second numeric value in the name ("0" in the case of these sixteen labels") is used internally by the "cmpfuncs.hla" code as branch targets for the binary search; the start of the binary search always begins with the label ending with a zero.

The "CheckFuncs" procedure returns the carry flag set if it cannot find the identifier it has scanned among those in the "funcs.txt" file; it returns the carry flag clear if it can find the current identifier in the list of compile-time function names. In addition, if the "CheckFuncs" procedure successfully matches an identifier, it returns the constant "tc_func" (token class "function name") in EBX and the token's code (which depends on the string matched) in the EAX register. Consider, again, the code given earlier for the "@istype" function identifier:

```
cmpfuncs_6_7:
    cmp( eax, funcstr( "istype", 0, 4) );
    jb cmpfuncs_6_8;
    ja cmpfuncs_6_10;
    cmp( bx, funcstr( "istype", 4, 2) );
    jb cmpfuncs_6_8;
    ja cmpfuncs_6_10;
    mov( tkn_at_istype, eax );
    mov( tc_func, ebx );
    jmp funcsDone;
```

Notice how this code compares the six characters in EAX and BX against the string. The "funcstr" macro translates up to four characters in the string appearing as its first parameter to an eight, 16, 24, or 32-bit unsigned integer value whose bytes correspond to the characters in the string (arranged so as to allow a numeric comparison that is equilvalent to a lexicographical comparison). Since the 80x86's 32-bit registers hold a maximum of four characters, the "funcstr" macro also behaves like a substring function; the second parameter specifies a starting position in the string while the third parameter specifies a length. Therefore, the first CMP instruction above compares EAX against the string "isty" while the second CMP instruction above compares the BX register against the string "pe".

HLA's lexical analyzer uses a case insensitive comparison when checking for compile-time function names. To achieve this, HLA logically OR's each character of the suspect identifier with $20 to convert any uppercase characters to lower case (note that logically ORing a digit with $20 leaves that digit unaffected). Therefore, the strings appearing in the "funcstr" macro invocations must always be lower case characters; the code sequences above will never match a "funcstr" operand containing upper case because the 80x86 registers will only contain numeric and lower case alphabetic characters.

See the "lexrw.hla" file for more information about the "CheckFuncs" procedure.

Standard HLA Reserved Words and Identifiers

The HLA v2.0 reserved words are a bit more numerous and appear in a typical HLA program source file far more frequently than the compile-time function identifiers. Furthermore, when an identifier begins with the "@" character, the following identifier is *almost* always a legal HLA compile-time function name (the only exception would be a typographical error, which occurs infrequently); HLA reserved words, however, take the same form as standard program identifiers and identifiers occur very frequently in a source file, therefore, the HLA v2.0 lexer must often *reject* an identifier as a reserved word; this process generally requires the maximum number of comparisons. Although a binary search algorithm is relatively efficient, typically requiring a maximum of five comparisons to accept/reject a given compile-time function ID, we can do a little bit better than this for the standard reserved words by using a hashing function.

The HLA reserved word recognizer uses a two-level hashing scheme. First, like the "CheckFuncs" procedure, it hashes off the length of the identifier. Then it applies a hashing function to each character in the identifier to produce an eight-bit hash value. Then one of up to 256 different code sequences (for each length, producing a theoretical maximum of 2,048 code sequences) compares the suspect string against a small list of reserved words. The end result is that the lexer can accept/reject about have the reserved words with only a single string comparison. About a third of the reserved words require two comparisons to accept/reject. The maximum number of comparisons needed is five (and this occurs only for five reserved words, less than 2% of the total number of reserved words). Therefore, accepting/rejecting identifiers as reserved words is a very efficient process in the HLA v2.x lexer. Of course, this comes at the cost of complexity and space, which is why the HLA lexer doesn't use this technique for compile-time statements and compile-time function names.

The HLA v2.x lexer transfers control to the "CheckRW" procedure whenever it encounters an alphabetic character or an underscore ("_") in the input stream. The CheckRW procedure collects all the symbols that comprise the ID (using the regular expression: ID = [a-zA-Z_][a-zA-Z0-9_]*), it also calculates the length of the identifier while processing these characters.

While collecting the characters that make up the identifier, the lexer also computes the hash function on those characters. The HLA lexer uses the following hash function on identifiers:

```
hash := 0
for each character in the ID:

  hash := hash << 1
  hash := hash XOR current_character
  hash := hash XOR (hash >> 8)

endfor
hash := hash & 0xff
```

This is a surprisingly simple hash function, but provides an excellent distribution of strings in the eight-bit integer space. Although better hash functions exist (that produce fewer collisions), such hash functions are more expensive to compute and the cost of such computation exceeds the cost of the extra comparisons that this hash function produces.

After the "CheckRW" procedure computes the length and hash value for the current identifier/reserved word, it branches to one of 12 different code sequences depending on the length of the string. For strings whose length is 2..12 characters, "CheckRW" code transfers control to some code sequence that loads up the string in the EAX, EBX, ECX, and EDX registers and converts any upper case characters in the string to lower case. These 11 code sequences then transfer control to some other code that we'll discuss in a moment.

For other string lengths the code drops through to a sequence that immediately labels the string an identifier (versus some reserved word) since strings of length one and strings whose lengths are greater than 12 cannot be HLA reserved words. Note that adding reserved words to HLA's set whose length is greater than 12 characters is a major chore. If you decide to add some reserved words to HLA, you should try to limit them to 12 characters or less.

The "CheckRW" procedure essentially uses a switch/case statement to transfer control to the 11 different code sequences that handle strings of length 2..12. As noted above, these 11 code sequences load the registers with the current lexeme (string) and convert any upper case character to lower case in order to do a case insensitive comparison. Finally, these 11 code sequences jump to (possibly) one of 256 different code sequence depending on the eight-bit hash value computed for the string. Note that this is one of 256 different code sequences for *each* string length. In theory, there could be up to 2,816 different code sequences, though in reality there are far fewer because the HLA reserved word set does not completely populate the hash space defined by the hashing function.

Like the compile-time function code, generating the many code sequences needed to compare the reserved words via a hash table is an onerous task best left to the machine. There is a program called "rw.exe" in the "rw" subdirectory of the lexer that is responsible for taking a list of reserved words ("rsvd.txt") and generating the necessary files containing the lexer code. Each line in the "rsvd.txt" file contains two items: a token type identifier and a reserved word. The token type identifier must be one of the following words:

```
reg8
reg16
reg32
regmmx
regfpu
instr
stmt
type
rw
```

The *reg8*, *reg16*, and *reg32* token identifiers specify one of the 80x86 integer registers (e.g., al, ax, and eax). The *regmmx* and *regfpu* token types correspond to one of the MMX or FPU registers (e.g., mm0 and st0). The *instr* token type is for reserved words that are actual 80x86 machine instructions (e.g., MOV, ADD, and PUSH). The *stmt* token type is for HLA high level language statement reserved words (e.g., IF, WHILE, UNTIL, and ENDFOR). The *type* token identifier is for the HLA built-in primitive types (e.g., byte, char, int32, and string). The *rw* token type is for all other HLA reserved words (e.g., program, record, macro, procedure, static, and type).

The "rw.exe" program reads the "rsvd.txt" file and emits three text files for use by the HLA compiler: rwtokens.hhf, cmpcode.hla, and jmptbl.hla. The rwtokens.hhf file contains constant declarations for each of the HLA reserved words. The "rw.exe" program generates this file by prepending "tkn_" to each of the reserved words and emitting a sequential integer constant (starting with the value $100) for each symbol. Here's the first few lines of this file:

```
const

tkn_ah          := $000000100;
tkn_al          := $000000101;
tkn_bh          := $000000102;
```

The last line of this file is the following:

```
tkn_endTknList   := $0000002C2;
```

This special identifier, "tkn_endTknList" provides the starting value for the token values that the "fTokens.hhf" file defines (these are the tokens associated with the compile-time function names, described earlier). Note that the above value of *tkn_endTknList* is not guaranteed; this value changes as one adds and removes reserved words from the "rsvd.txt" file.

As for the compile-time function names, the HLA compiler uses a large switch statement to rapidly process identifiers to determine if they are HLA reserved words or simple identifiers. This switch statement uses a two-dimensional jump table with the following declaration:

```
rwJmpTbl: dword[ 13, 256 ]; nostorage;
```

HLA selects the row (the first dimension) by the identifier's length (reserved words in HLA are always between two and twelve characters long); HLA selects the column in this two-dimensional jump table by the eight-bit hash value computed for the identifier. Each entry in this table is the address of some code sequence appearing in the "CmpCode.hla" file. These code sequences have labels like Len2_Hash160_0, Len2_Hash178_0, Len3_Hash88_0, Len5_Hash0_0, and Len10_Hash240_0. The numeric value immediately following "Len" (e.g., the "2" in "Len2" corresponds to the identifiers length. The numeric value immediately following "Hash" (e.g., the "160" in "Hash160") corresponds to the hash value computed for the identifier. The numeric suffix (which is always zero for entries appearing in the jump table) increments for each string collision. That is, if two symbols hash to the same value, then a code sequence with the suffix "_0" will check one of the symbols and a code sequence with the suffix "_1" will check for the other symbol. Here's what a typical code sequence looks like:

```
Len2_Hash135_0:
    cmp( ax, rwstr( "to", 0, 2 ));
    jne IsAnID;
    mov( tkn_to, eax );
    mov( tc_stmt, ebx );
    jmp rwDone;
```

Here's a code sequence that has a collision:

```
Len4_Hash116_0:
    cmp( eax, rwstr( "fsub", 0, 4 ));
    jne Len4_Hash116_1;
    mov( tkn_fsub, eax );
    mov( tc_instr, ebx );
    jmp rwDone;

Len4_Hash116_1:
    cmp( eax, rwstr( "insb", 0, 4 ));
    jne IsAnID;
    mov( tkn_insb, eax );
    mov( tc_instr, ebx );
    jmp rwDone;
```

The *rwstr* macro is very similar to the *funcstr* macro this document describes earlier. It takes the substring of the specified string constant (whose length is in the range 1..4) and translates this to an integer value that the HLA compiler can compare with one of the integer registers. For reserved words whose length is greater than four characters, HLA uses a multiprecision comparison, e.g.,

```
Len6_Hash158_0:
    cmp( eax, rwstr( "fcomip", 0, 4 ));
    jne IsAnID;
    cmp( bx, rwstr( "fcomip", 4, 2 ));
    jne IsAnID;
    mov( tkn_fcomip, eax );
    mov( tc_instr, ebx );
    jmp rwDone;
```

HLA uses the following registers to compare strings of length 2, 3, 4, ..., 12 characters:

Table 1: Registers HLA Uses to Check for Reserved Words

String Length	Registers
2	ax
3	ax, bl
4	eax
5	eax, bl
6	eax, bx
7	eax, bx, cl
8	eax, ebx
9	eax, ebx, cl
10	eax, ebx, cx
11	eax, ebx, cx, dl
12	eax, ebx, ecx

Note that the reserved word comparisons use different register sets than the compile-time function comparisons. This is for efficiency reasons (i.e., loading the registers above with 2..12 characters is more efficient than packing the strings in as few registers as possible; the compiler couldn't do this for the function names because their lengths vary between one and sixteen characters).

The Lex Function

The *lex* function, appearing in the "lex.hla" source file, is the main procedure that implements the HLA lexical analyzer. Each call to this function returns a set of token values in the EAX and EBX registers. This procedure also returns EDI pointing at the start of the associated lexeme and ESI pointing one character beyond the lexeme.

The caller must pass the lex function a pointer, in ESI, to the next available character in the source file that HLA is compiling. The lex function begins by checking to see if the pointer in ESI points beyond the end of the file in memory. If so, the lex procedure closes any open include file (and resets the pointer back to the file that did the file inclusion) or the lexer reports an "unexpected end of file" error if there is no open include file.

If ESI does not point beyond the end of the file, the lexer fetches this (eight-bit) character and uses it as an index into a 256-element jump table. This transfers control to some code sequence that handles lexemes consisting of, or beginning with, the specified character. Note that many characters are illegal in the source file (e.g., all character values whose high-order bit is set, most of the control characters, and certain punctuation characters are not legal in an HLA source file); if the lexer encounters one of these characters, it prints an appropriate error message and skips the character.

Like a traditional lexical analyzer, the lex procedure skips all whitespace (including newlines) in the source file. HLA updates a source file line count variable upon encountering the new line sequence, but the lex procedure does not otherwise return a value indicating the presence of white space. Immediately after skipping any white space, the lex procedure processes the next character in the source file.

HLA treats the following symbols as single-character lexemes:

```
~  -  +  *  ^  (  ;  [  ]  {  ,
```

If the lex procedure encounters one of these symbols, it returns the ASCII code of the symbol in EAX and a token type (tc_notOp for “~”, tc_addOp for “+” or “-”, tc_mulOp for “*”, tc_orOp for “^”, and tc_punct for the other characters) in EBX.

If the lexer encounters the “/” character, it checks the character immediately following the slash to see if it’s a “*” or a second “/” character. If so, the lexer skips over the comment (to the matching “*/” for “/*” or to the end of the current source line for “//”). Like whitespace, the lexer ignores any comments and continues processing the text after the comment ends. If the “/” symbol does not begin a comment, then HLA assumes that this is the division operator and returns the ASCII code for “/” in EAX and the tc_mulOp constant in EBX.

Certain lexemes consist of two or more characters and the first character is often a lexeme all on its own. Examples include “|” and “||”, “&” and “&&”, and “!” and “!=”. Whenever the lex procedure encounters one of these leading characters, it must check the character immediately following to see if the longer lexeme is present. The lex procedure always matches the longest possible lexeme. The following table lists these multi-character lexemes:

Table 2: Multi-Character Lexemes

Leading Character	Possible Lexemes
|	| ||
&	& &&
:	: :: :=
.	. ..
=	= ==
!	! !=
>	> >= >>
<	< <= << <>
}	} }#
)	))#
#	See discussion below

The “#” symbol, as a lead-off character, is fairly complex. HLA allows the “#” by itself; it allows the “#” as a lead-off character for the lexemes "#{", "##", "#(", and "#:"; HLA lets you specify character constants using the syntax #dd, #%bb, or #$hh (where dd, bb, and hh represent strings of decimal, binary, and hexadecimal digits, respectively); and, finally HLA also prefixes certain compile-time statements with the “#” (e.g., “#IF”). Fortunately, there is never any ambiguity; HLA can determine the token type by looking at the first character (if any) beyond the “#”. Note that the lex procedure uses a simple linear search when checking for the compile-time statement identifiers (e.g., “#IF” and “#WHILE”). There is no need for anything fancy (like lex uses for compile-time function names and reserved words) because (1) there are so few compile-time statements, and (2) they don’t appear very often in the source file. Note that for the #dd, #%bb, and #$hh forms, HLA also returns the value of the character constant in the *attribute.v_char* variable and the constant *t_char* in the *attribute.attrType* variable.

If the lex procedure encounters the “@” character, it passes control to the code that checks for a compile-time function name.

If the lex procedure encounters an alphabetic character, it passes control to the function that checks the following identifier against one of the reserved words. If an identifier begins with an underscore, HLA simply collects the iden-

tifier and returns it (without checking to see if it is a reserved word) since no reserved words begin with an underscore.

If the lex procedure encounters an apostrophe, it processes a character constant. A character constant takes the form 'x' where *x* is a graphic character that is not an apostrophe; or a character constant takes the form '''' (four apostrophes in a row), which represents the apostrophe character. Any other form involving apostrophes is illegal. In particular, you may not embed control characters in a character constant of this form (use the #$hh form, instead). The lex function returns the value of the character constant in the *attribute.v_char* variable. It also returns the constant *t_char* in the *attribute.attrType* variable.

If the lex function encounters a quote character, it will process the string constant that immediately follows. Note that the lex procedure must handle the special cases of the empty string and strings containing two adjacent quotes (signifying a single quote in the string). The lex procedure allocates storage for the final string constant on the heap and returns a pointer to this string in the *attribute.v_string* variable. It also returns the constant *t_string* in the *attribute.attrType* variable. It is the caller's responsibility to free the storage associated with this string when the caller is done with the string value.

If the lex procedure encounters a "$", "%", or a numeric digit, then it processes the string of hexadecimal, binary, or decimal digits (which could be an integer or floating point constant). Processing hexadecimal and binary constants is rather straight-forward other than the fact that the lexer uses extended precision input routines (128 bits) for all integer numeric formats. There are two functions, *getHexConst* and *getBinConst* that handle the task of converting strings of hexadecimal or binary digits into a 128-bit integer format. If the lexer encounters a decimal digit, things are a bit more complicated because both integer and floating point constants begin with a decimal digit. So the lexer has to first scan through the lexeme to determine whether it has a floating point or (unsigned) integer constant. Floating point constants contain a period, an "e", or an "E" character within the string of digits (of course, the regular expression for a floating point constant is a little bit more demanding than this, but the presence of a period, "e", or "E" character definitely suggests that the value is *not* a unsigned integer constant). If the value appears to be a floating point constant, then the lexer calls the *conv.atof* function in the HLA Standard Library to do the conversion. If the lexeme corresponds to a unsigned integer constant, then the lexer calls the *getDecConst* function to translate the string to a 128-bit unsigned integer. The HLA lexer always treats numeric lexemes as unsigned because the leading "-" character for negative values is handled as a separate token.

As briefly noted earlier, the lex function returns several values. It returns the token value in the EAX register, the token classification in the EBX register, a pointer to the beginning and end of the lexeme in EDI/ESI, and the current lexemes attributes (if any) in the global variable *Attr*. Whomever calls the lex procedure can obtain any necessary information about the current lexeme that lex has processed by looking at these values.

The lexer returns the current token's value in the EAX register. This is an ASCII character code for all single character lexemes and it is a value greater than or equal to $100 for all other lexemes. See the rwtokens.hhf, fTokens.hhf, and hlaCompiler.hhf header files for the symbol definitions of these token constants. The convention is to use the "tkn_" prefix for all constant values that the lexer returns in the EAX register.

The lex procedure returns a token classification in the EBX register. This value is useful for quickly determining the type of a lexeme without have to compare the value in EAX against a large set of individual token values. For example, a single comparison with EBX will tell you whether the current lexeme is a reserved word, an identifier, a certain type of constant, a certain type of punctuation, etc. This single comparison is usually far more efficient than checking the current token value against a set of token values or even doing a range comparison to see if a token value lies between two other token values. This value will help reduce the size of various routines in the parser by a fair amount. HLA uses a set of enumerated constants for these token class constants. The *tokenClass_t* type definition (in hlaCompiler.hhf) defines these constant values. At the time this was written the following token class constants were defined (though this is subject to change, please see the hlacompiler.hhf header files for the accurate definition):

```
tc_EOF,
tc_rawID,
tc_undefID,
tc_localID,
tc_globalID,
tc_instr,
tc_rw,
```

```
        tc_func,
        tc_stmt,
        tc_type,
        tc_notOp,
        tc_mulOp,
        tc_addOp,
        tc_relOp,
        tc_andOp,
        tc_orOp,
        tc_punct,
        tc_const,
        tc_regmmx,
        tc_regfpu,

        // Note: tc_reg8, tc_reg16, and tc_reg32 must be last
        // in this list (to make checking for an integer register
        // easier).

        tc_reg8,
        tc_reg16,
        tc_reg32
```

The *tc_EOF* constant is a special value reserved for the end of file token. The lexer returns *tc_rawID, tc_undefID, tc_localID*, or *tc_globalID* whenever it encounters an identifier in the input stream. For these lexeme classes, the lex procedure also returns some addition information about the identifier in the global *Attr* variable (see below).

The lexer returns *tc_instr* for 80x86 machine instruction reserved words, it returns *tc_func* for compile-time function names, and *tc_stmt* for HLA HLL reserved words. The lexer returns *tc_regmmx, tc_regfpu, tc_reg8, tc_reg16,* and *tc_reg32* for one of the 80x86 register names. Note that the *tc_reg8, tc_reg16*, and *tc_reg32* token classifications must be last. This is because the parser compares for greater than or equal against *tc_reg8* to determine if the current lexeme is any integer register. This allows the use of a single comparison to check for an arbitrary integer register. The *tc_type* classification value corresponds to the HLA reserved words for the built-in data types (e.g., "int32" and "char"). The HLA lexer returns *tc_rw* for other reserved words in the HLA language. The exact token classification value returned for each of these reserved words is set in the "rsvd.txt" file as mentioned earlier. Each entry in this file consists of two words: the token classification type and the reserved word. For example, the entry for the AH token contains the following line:

```
reg8  ah
```

This tells the rw.exe program that "ah" is a reserved word whose token class value is *tc_reg8*. See the rsvd.txt file and the rw.hla source file for additional details.

The HLA lex procedure returns the *tc_const* class value for any literal constant value. This includes integer, character, string, floating point, and other constant types. For this particular lexeme class, the lexer also returns some additional data in the global *Attr* variable (see below).

The EDI and ESI registers point at the start of the lexeme (EDI) and one character beyond the end of the lexeme (ESI). Note that these pointers contain addresses within the memory-mapped source file. The lexer does not copy the lexeme into some temporary string storage space. Note that HLA opens the source files in a read-only mode, so you cannot store any data into the address range specified by EDI..ESI. If you need to compute the length of the lexeme, this is easily accomplished by subtracting the value in EDI from the value in ESI.

The *tc_notOp, tc_mulOp, tc_addOp, tc_relOp, tc_andOp,* and *tc_orOp* token classes are for various arithmetic expression lexemes. This allows the parser to quickly test for classes of operators that have the same precedence level. The *tc_punct* handles all punctation symbols other than the operators handled by these token classes.

For certain tokens, the lexer must return additional information to the caller. The global *Attr* variable provides the return location for such values. At the time this was written, the Attr variable had the following type:

```
attr_t:
      record
            attrType    :attr_variant_t;
            align( 4 );
            union

                  v_byte      :byte;
                  v_word      :word;
                  v_dword     :dword;
                  v_qword     :qword;
                  v_tbyte     :tbyte;
                  v_lword     :dword[4];

                  v_int8      :int8;
                  v_int16     :int16;
                  v_int32     :int32;
                  v_int64     :qword;
                  v_int128    :dword[4];

                  v_real32    :real32;
                  v_real64    :real64;
                  v_real80    :real80;

                  v_boolean   :boolean;
                  v_char      :char;
                  v_string    :string;
                  v_cset      :cset;
                  v_xcset     :cset[2];

            endunion;

      endrecord;
```

This is, effectively, a variant data type that can hold any arbitrary literal value possible in an HLA source file. At this time this was written there were a few fields that had not been defined in this record type; notably absent are the fields to hold information about identifiers. See the hlacompiler.hhf header files for a complete definition of this record type.