1 HLA Overview

HLA, the High Level Assembler, is a vast improvement over traditional assembly languages. With HLA, programmers can learn assembly language faster than ever before and they can write assembly code faster than ever before. John Levine, comp.compilers moderator, makes the case for HLA when describing the PL/360 machine specific language:

1999/07/11 19:36:51, the moderator wrote:

"There's no reason that assemblers have to have awful syntax. About 30 years ago I used Niklaus Wirth's PL360, which was basically a S/360 assembler with Algol syntax and a a little syntactic sugar like while loops that turned into the obvious branches. It really was an assembler, e.g., you had to write out your expressions with explicit assignments of values to registers, but it was nice. Wirth used it to write Algol W, a small fast Algol subset, which was a predecessor to Pascal. ... -John"

PL/360, and variants that followed like PL/M, PL/M-86, and PL/68K, were true "mid-level languages" that let you work down at the machine level while using more modern control structures (i.e., those loosely based on the PL/I language). Although many refer to "C" as a "medium-level language", C truly is high level when compared with languages like PL/*. The PL/* languages were very popular with those who needed the power of assembly language in the early days of the microcomputer revolution. While it's stretching the point to say that PL/M is "really an assembler," the basic idea is sound. There really is no reason that assemblers have to have an awful syntax.

HLA bridges the gap between very low level languages and very high level languages. Unlike the PL/* languages, HLA really is an assembly language. You can do just about anything with HLA that you can do with a traditional assembler like MASM, TASM, NASM, or Gas. If you want to write low-level assembly code using x86 machine instructions, HLA does not get in your way; if you want to use compares and conditional branches rather than structured control statements, you can. On the other hand, if you prefer to use more readable high-level control structures, HLA allows this, as well. HLA lets you work at the level you are most comfortable with and at the level that is most appropriate for the task at hand.

Beyond supplying a "non-awful" syntax, HLA has one other important feature -- it's extensible. HLA provides special features that let you add new statements to the language. So if HLA is not "high level" (or "low level") enough for your tastes, you can extend it. This document will expend considerable effort describing exactly how to do this in a later section.

In addition to the HLA language itself, the HLA system provides one other very important component - the HLA Standard Library. This is a collection of hundreds of functions that you can use to write assembly language programs as quickly and easily as you would write C programs.

1.1 What is a "High Level Assembler"?

The name "High Level Assembler" and its abbreviation "HLA" is certainly not new[1]. Nor is the concept of a *high level assembler*. David Salomon in his 1992 text "Assemblers and Loaders" (Ellis Horwood, ISBN 0-13-052564-2) uses these terms to describe various assembly languages dating back to 1966. Furthermore, both IBM and Motorola have assembler products with very similar names (e.g., IBM's HLAsm, though it's somewhat debatable whether HLAsm is truly a high level assembler).

Salomon offers the following definitions for a High Level Assembler (or HLA):

*A high-level assembler language (HLA) is a programming language where each instruction is translated into a few machine instructions. The translator is somewhat more complex than an assembler, but much simpler than a compiler. Such a language should not have features like the **if**, **for**, and case control structures,*

1. This section will use the term "HLA/86" when specifically taking about the High Level Assembler product this documentation describes and use "HLA" as a generic term. After this section, this documentation will use the term "HLA" to specifically describe the "HLA/86" product.

complex arithmetic, logical expressions, and multi-dimensional arrays. It should consist of simple instructions, closely resembling traditional assembler instructions, and of a few simple data types.

Since Salomon describes a couple of high level assemblers that exceed this definition, he offers a second definition for high level assemblers that is a bit higher-level:

A high-level assembler language (HLA) is a language that combines most of the features of higher-level languages (easy to use control structures, variables, scope, data types, block structure) with one important feature of assembler languages namely, machine dependence.

Neither definition is particularly useful for describing HLA/86 and other HLAs like Terse, MASM and TASM. Of course the term "High Level Assembler" is very nebulous and offers a fair amount of latitude. Almost any macro assembler could pass as an HLA on the basis that a macro-instruction expands into a few machine instructions.

David Salomon describes several different high level assemblers in his text. The examples he describes are PL/360, NEAT/3, PL516, and BABBAGE.

PL/360 and PL516 are products that conform to the second definition above. They allow simple arithmetic expressions and assignment statements, the use of high level control structures (**if, for, while**, etc.), high level data declarations, and block structure (among other things). These languages expose the underlying machine's registers and allow the use of machine instructions using a "functional" syntax.

The NEAT/3 language is a much lower-level language; basically it is an assembly language for the NCR Century computers that provide COBOL-style data declarations. Most of its "instructions" translate one-for-one into Century machine instructions, though it does automatically insert code to convert data types from one format two another if the data types of an instruction's operands are incompatible.

The BABBAGE assembly language is an expression-based assembly language (very similar to Terse). It allows simplified high level control structures like **if** and **while**. The interesting thing about this assembler is that it was the only assembler for the GEC4000 family of computers.

In addition to the HLAs that Salomon describes, there have been several other high level assemblers created over the years. PL/M and PL/M-86 was designed by Intel for their 8080 and 8086 CPU families. This was an obvious adaptation of the PL/360 style HLA for Intel's CPUs. PL/68 was also available for the Motorola 680x0 family. SL/65 was a similar adaptation of PL/360 for the 6502 family. At one point there was a product named "High Level Assembler" for the Atari ST system (68K based). Jim Neil has also created an expression-based high level assembler (similar in principle to Babbage) for Intel's x86 family. MASM and TASM (for the x86) also fall into the category of a high level assembler due to their inclusion of high level control structures and logical expressions.

So where does HLA/86 fit into these definitions? In truth, the definition of HLA/86 falls somewhere between these two definitions. So the following paragraphs will define the term "High Level Assembler" as it should apply to HLA/86 and similar high level assemblers.

The first definition above is overly restrictive. It implies that any language that exceeds these limits is a high level language, not a high level assembly or traditional assembly language. Obviously, this definition is too restrictive in the sense that by this definition many traditional assemblers would have to be considered as high level languages (even beyond a high level assembler). Furthermore, it elevates many traditional assemblers to the status of an HLA even though we wouldn't normally think of them as high level assemblers; i.e., most macro assemblers provide the ability to create instructions that translate into a few machine instructions. Macro facilities, however, are something we expect out of a modern assembly language; their presence doesn't make the language a "high level" assembly language in most people's mind. Furthermore, most modern assemblers provide a mechanism for declaring multi-dimensional arrays (even though you still have to use some sequence of instructions to index into said arrays).

The second definition David Salomon provides hits the other extreme. Arguably, languages like C could be called HLAs under this definition (yes, there are some machine dependent features in C, though probably not enough to satisfy David Salomon's original intent).

The definition of high level assemblers like Terse, MASM, TASM, and HLA/86 fall somewhere between these extremes. Therefore, this document will define a high level assembler as follows:

A "high level assembly language" (HLAL) is a language that provides a set of statements or instructions that practically map one-to-one to machine instructions of the underlying architecture. The HLAL exposes the underlying machine architecture *including access to machine registers, flags, memory, I/O, and addressing modes. Any operation that is possible with a traditional assembler should be possible within the HLAL. In addition to providing access to the underlying architecture, the HLAL must provide some abstractions that are not normally found in traditional assemblers and that are typically found in traditional high level languages; this could include structured control statements (e.g.,* ***if, for,*** *and* ***while****), high level data types and data structuring facilities, extensive compile-time language facilities, run-time expression evaluation, and standard library support. A "High Level Assembler" is a translator that converts a high level assembly language to machine code.*

There is a very important difference between this definition and the ones that David Salomon provides. Specifically, a high-level assembly language must provide access to the underlying machine architecture. Within the HLAL you must be able to specify any (reasonable) machine instruction that is available on the CPU. The HLAL may provide other statements that do not directly map to machine instructions (e.g., an **if** statement), but it must, at least, provide a set of statements that *practically* map one-to-one with the machine instructions. The "practically" modifier appears here for two reasons. First of all, some assembly source statements may map to two or more different, but equivalent, machine instructions. A good example is the x86 "mov reg, reg" which can map to two different (though equivalent) opcodes depending on the setting of the direction bit in the opcode. Most assemblers will map the source statement to only one of these opcodes, hence there is not truly a one-to-one mapping (since there exist some opcodes that do not map back to some source instruction). Another allowable restriction is that the HLAL may limit the programmer to a subset of the complete machine instruction set if it makes sense to do so (e.g., many modern x86 assemblers do not support 16-bit mode on the 80x86).

In addition to supporting the underlying machine architecture (which almost any traditional assembler will do), the HLAL must also provide support for some features normally found in a high level language. The definition does not require that a HLAL support all the features listed above, nor is it restricted to just the features listed, but a HLAL must support some of the features traditionally found in a high level language. The number and type of features the HLAL supports determines how "high level" the assembly language is. Like HLLs, we can have "low-level" HLALs, "medium-level" HLALs, "high-level" HLALs, and even "very high-level" HLALs. NEAT/3, for example, would be a low-level HLAL since it provides higher-level data types, conversions, and not much else.

MASM and TASM are probably best considered medium-to-high-level HLALs since they provide high level data structuring facilities, structured control statements, high level procedure definitions and invocations, a limited block structure, powerful compile-time language (macro) facilities, standard library support (e.g., the UCR Standard Library and many other available library modules), and other high level language features. In actual use, the programmer is expected to normally use standard machine instructions and rise up to the high level statements only as necessary.

The Terse language is a good example of a medium level HLAL since it uses an expression syntax but otherwise maps statements fairly closely to the assembly counterparts. It does provide some higher-level data structuring capabilities, though this is inherited from the underlying assembler(s) on which Terse is based.

PL/360 and PL516 are definitely high-level HLALs because they fully support simplified arithmetic expressions, control structures, high-level data types, and other features. These languages provide access to the underlying architecture, but the emphasis is to use these languages as a high level language and drop down to the machine instructions only as necessary.

HLA/86 probably falls in the high-level-to-very-high-level range because it provides high level data types and data structuring abilities, high level and very high level control structures, extensive parameter passing facilities (more than most high level languages), a very extensive compile time language, a very extensive standard library, built-in parsing facilities for language extension, and many other features. Generally, HLA/86 has a larger feature set than the other HLALs described above. There are a few design goals that limit the "high-levelness" of HLA/86:

(1) With one exception, HLA never emits any code behind the programmer’s back that modifies registers or flags (the one exception is object method invocation, and this is well documented), and

(2) HLA doesn’t support arithmetic expressions (it does support a limited form of logical/ boolean expressions).

One interesting aspect of HLA/86 is that it is extensible. Using features built into the language, you can extend HLA/86’s syntax by adding new statements and other features. This feature gives you the ability to make HLA/86 as high level as you desire (though it may take some effort to achieve certain language features). The bottom line is this: in some ways, HLA/86 is lower level than languages like PL/360 and PL516; in other ways, it's higher level than these HLALs. However, as the definition requires, almost anything you can do with a traditional assembler is possible in HLA/86.

1.2 What is an "Assembler"

Because high-level assemblers are clearly different that traditional assemblers, one might question whether a high level assembly language is truly an assembly language and whether translators for high-level assembly languages can be properly called an assembler. Unfortunately, there is a considerable range of opinions as to exactly what constitutes an "assembler" versus other translators. This document will not attempt to get involved in this debate. Instead, this section provides a set of definitions that are useful for describing assemblers at various levels of abstraction.

Pure Assembler:

A "pure assembler" is a program that processes an assembly language source file and translates the source code using a direct mapping from source code instructions to individual machine instructions (each source instruction is mapped to exactly one machine instruction). The assembler only provides machine-primitive data types like bytes, words, double words, etc. A pure assembler does not provide macro facilities. A pure assembler always produces machine code as output.

Traditional Assembler:

A "traditional assembler" is a pure assembler plus macro facilities. The assembler may provide some "built-in macros" and instruction synonyms, but in general, the built-in statements should still map to individual machine instructions (note that the programmer may extend this by writing macros). There is no support by the assembler for run-time arithmetic or boolean expressions. A traditional assembler may also provide some simple data typing facilities (such as the ability to rename primitive data types as something else, e.g., byte->char). A traditional assembler always emits machine code as output.

High Level Assembler:

A high-level assembler is a macro assembler plus some additional high-level language-like facilities, such as high-level control constructs or high-level-like procedure calls. If a programmer elects to ignore these additional facilities, they still have all the capabilities of a macro assembler at their disposal.

1.3 Is HLA a True Assembly Language?

Some people are confused by HLA. On the one hand, it looks like a High Level Language, employing syntax similar to Pascal and C/C++. On the other hand, it does support the machine instructions found in a typical assembly language. Many people accuse HLA of being a compiler rather than an assembler. What's the truth?

The truth is, assembly languages have evolved, just as high-level languages have evolved, and we can no longer use a definition for an assembler that made sense in the 1950s when describing modern assemblers such as MASM, TASM, and HLA. Today, the best definition we can use is that an assembler is a compiler for an assembly language. An assembler accepts a source file written in some sort of assembly language and produces an object file as its output.

The real question, then, is not whether HLA is an assembler, but whether the HLA language is an assembly language. Some people argue that any compiler that includes any sort of statement that compiles into more than one machine instruction cannot be called an "assembler." However, such an argument immediately eliminates macro assemblers. Eliminating macro assemblers is unsatisfactory because almost every modern assembler provides, at the very least, some simple macro facilities. Whether you implement an "IF" statement with a macro (generally supplied by the assembler's author, as is the case, for example, with FASM) that you have to include into your source file, or via a 'macro' that the assembler's author has provided as part of the assembler is really a matter of implementation. To the end user of the assembler, the "IF" statement is just as much a part of the language that they can use regardless of the implementation. The fact that assemblers such as MASM, TASM, and HLA provide these high-level-like control structures in assembly language does not imply that the languages these products implement are not assembly languages.

Some people argue that "high-level assemblers" such as MASM, TASM, and HLA are not assemblers any more than C/C++ compilers could be considered assemblers if those C/C++ compilers support an in-line assembly capability. However, their arguments strengthen the case for calling a product like HLA an "assembler." After all, if we're going to continue to call C/C++ a high-level language even though it provides support for machine instructions, then there is no reason we cannot call a product like MASM, TASM, or HLA "assemblers" even though they provide a modicum of support for high-level-like control structures. Ultimately, language's focus determines its type. C/C++'s focus is on writing high-level language programs, with a few machine instructions thrown in now and then when the high-level language doesn't quite handle everything. High-level assemblers, such as HLA, MASM, and TASM are focused on writing assembly language modules. They have some high-level control structures thrown in to simplify some tasks (e.g., in the case of HLA, the high-level control structures exists as a bridge between HLLs and assembly during the learning process), but the focus is mainly on writing assembly language code.

Some people feel that if you learn HLA (or some other high level assembler), then you're not really learning "assembly language." This is utter nonsense. If you thoroughly learn HLA, you'll know assembly language programming inside and out. Switching to a different assembler from HLA would be no different, say, than switching from Gnu's Gas assembler to MASM (or vice versa). One might bemoan the features lost in such a translation, but when going from HLA to some other assembler you're typically *giving up* features rather than gaining anything.

Still there is a pervasive argument that high-level control structures like IF/WHILE/FOR/etc. don't belong in a true assembler. Well, HLA, MASM, and TASM users can elect to ignore these statements (as many old-time MASM programmers do; with HLA you can even disable these statements). As long as the rest of the assembler supports a language that allows one to write "pure" assembly language code, why would anyone question the validity of the title "assembly language" for the code? (Unless, of course, they have an ax to grind.) For those who are diametrically opposed to allowing any language that contains IF/WHILE/FOR/etc. statements to be called assembly language, well, that's why we call these things *high level assembly languages:* to note the fact that they are a little more powerful than traditional assembly languages.

The bottom line is this: if you learn HLA, you will learn assembly language programming. As long as you understand how to write the low-level code (within HLA) and don't rely exclusively on the high-level control statements in your programs, no one can truthfully question your assembly language programming knowledge.

1.4 HLA Design Goals

HLA was originally conceived as a tool to teach assembly language programming. In early 1996 I decided to do a Windows version of my electronic text "the Art of Assembly Language Programming" (AoA). After an attempt to develop a new version of the " UCR Standard Library for 80x86 Programmers" (a mainstay of AoA), I came to the conclusion that MASM just wasn't powerful enough to make learning assembly language really easy. I decided to develop an assembler with sufficient power, providing the tools for a good standard library as well as satisfy some other requirements. Therefore, HLA has two important goals: provide a system that is powerful enough to develop code and macros to make learning assembly language, which simultaneously providing a system that is easy for beginners to learn.

The principle goal of HLA was to leverage student's existing programming knowledge. For example, a good Pascal programmer can get their first C/C++ program operational in a few minutes. All they have to do is note the similarities between the two programming languages, make the appropriate syntactical changes, and they're up and running. Take that same Pascal programmer and expect them to learn LISP or Prolog the same way, and you'll not meet with the same success. LISP and Prolog are completely different, they use a different "programming paradigm," so the student has to "start over from scratch" when learning these languages. Although assembly language is an imperative language (like Pascal and C/C++), there is a considerable "paradigm shift" when moving from one of these high level languages to assembly. In HLA, I wanted to create a language with high level control structures and declarations that made it possible for someone familiar with an imperative language like Pascal or C/C++ to get their first HLA program running in a matter of minutes (or, at worst, a matter of hours). Of course, to achieve this goal, I needed to add high-level data declarations and high-level control constructs to the HLA language.

The astute reader will quickly point out that high level control structures are not assembly language and letting the students use these types of statements is not really teaching them assembly language. This is quite true; since the purpose of teaching an assembly language course is to teach the students *assembly* language programming it is quite clear that HLA would fail if it *only* provided these high level control structures (e.g., like the PL/M language does). Fortunately, this is not the case. HLA supports all standard assembly language instructions including CMP and Jcc instructions, so you can still write "pure" assembly language programs without using those high-level language control structures. However, it does take time to learn the several hundred different machine instructions. Traditionally, it's taken my students (using only MASM) about five weeks before they could really write any meaningful programs in assembly language (you have to cover things like numeric representation, basic CPU architecture, addressing modes, data types, and introduce the instruction set before any real programs can be written).

HLA lets students write *meaningful* programs within about a week of its introduction (e.g., the first assignment I give in a typical quarter is to write an "addition table" program that computes the outer product [addition table] of the two vectors 0..15 and 0..15, printing the table formatted nicely). They achieve this by using statements they already know (like IF and WHILE) with the injection of just a few assembly language concepts (registers, and the MOV and ADD instructions) plus an introduction to the HLA Standard Library. Over the next several weeks, these students write increasingly complex programs as they are introduced to new assembly language and HLA concepts (e.g., data representation, basic architecture, addressing modes, data types, and additional instructions). At about the sixth week, I begin "weaning" these students off the high-level language statements and force them to use the low-level machine instructions. It turns out that they learn how to simulate an IF statement at roughly the same point in the quarter as they did when they used only MASM, but the big difference is that they've written a lot more code up to that point proving out other concepts in machine organization and assembly language programming. In my limited experience with classroom testing, I've found that students spend less time on the class, cover more material, and retain the knowledge better (by the time of the final exam) than they did when I only used MASM.

The general goal of reducing the learning curve for students is achieved several ways.

(1) As noted above, HLA allows a gradual transition from high-level languages into pure assembly language. My favorite analogy here is the Nicoderm CQ smoking cessation system ("gradual steps are better."). Like the Nicoderm system, HLA allows students learn assembly language in gradual steps rather than throwing them into the water and shouting "sink or swim!"

(2) In addition to letting the students employ high level language statements in their assembly language programs, HLA contains several other familiar concepts and syntactical items that ease the transition from high-level language programming to assembly language. For example, HLA uses the familiar (to C/C++ programmers) "/*" and "*/" comment delimiters (as well as the "//" comment delimiter). Statements generally end with a semicolon (just as in high level languages). Machine instructions use a functional notation rather than "mnemonic-operand" notation. Constant, type, and variable declarations should look very familiar to Pascal programmers. HLA's standard library should look comfortable to anyone who has used the C/C++ standard library.

In addition to syntactical similarities, well-written HLA programs share a similar programming style with modern high-level languages. Therefore, a student who has learned how to write readable Pascal, C/C++, or Java programs will be able to write readable HLA programs with almost no additional study. Contrast this with the style guide I've written for (MASM) assembly language programmers that is quite a bit different than high level languages and takes a while to master.

Another factor many people don't consider is the evaluation of a programming project. At UC Riverside instructors are given about 1.5-2 hours per student per quarter of reader (student grader) time to grade projects. Experienced readers who can grade (or want to grade) assembly language projects are few and far in-between. Most readers are "stuck" with grading the assembly class rather than volunteer for the job. The fact that most student assembly language projects have a horrible programming style and are hard to read only exacerbates this situation. HLA helps solve this problem. Since good HLA programming style is very similar to good C/C++ style, UC Riverside's readers have a much easier time reading the projects and evaluating their programming style. In addition, since the students have (presumably) learned good programming style in the prerequisite course(s), they tend to write easier to read HLA programs than MASM programs. This lets the instructor assign more projects without fear of exceeding my reader budget each quarter.

HLA's advantages are easily summed up by a complaint I had from a student once. She said "HLA drives me nuts. It's so similar to C++ that I often get confused and try out something that would work in C++ only to have the HLA compiler reject it." I agreed with this student that this was a bit of a problem, but I also mentioned, "what about all the times you've tried something from C++ and it HAS worked?" She thought about it for a moment and walked away agreeing with my assessment of her complaint. Had this student been learning assembly the traditional way, she wouldn't have bothered to try anything. She would had to have spent extra time learning how to achieve what she wanted by reading an assembly text or she would have missed out on the opportunity to actually learn something new. HLA's similarity to C++ encouraged her to try something out on her own. The experiments weren't always successful, but in those cases where they were, she benefited greatly from this. This anecdote, more than any other, sums up what my goals with HLA were and describes the success I believe I have achieved with it.

1.5 How to Learn Assembly Programming Using HLA

Of course, a compiler without a language reference manual and tutorial is useless. This document will provide a reference to the HLA programming language. It is not, however, appropriate pedagogy for beginners (it's more suitable for those who already know assembly language programming and wish to learn HLA's syntax). A better text for beginners is "The Art of Assembly Language Programming, Second Edition" available from No Starch Press. This provides a complete college level textbook that teaches assembly language programming from the ground up using HLA. You can also find an electronic copy of "AoA" on Webster at http://webster.cs.ucr.edu. Webster also contains the latest version of HLA as well as tons of HLA sample source code. That's the first place you should go for information on learning HLA.

1.6 Legal Notice

The HLA v2.x implementation is a prototype intended to test language design and implementation features. I (Randall Hyde) have placed this code and language design in the public domain so others may benefit from this work. However, keep in mind that, as a prototype, HLA is not up to contemporary commercial standards for software quality. It is your responsibility to evaluate whether HLA is suitable for whatever purpose you have.

At any given time, there are several known and unknown defects in this software. Some may be corrected in later releases of HLA v2.x; some may never be corrected in the v2.x series. I (Randall Hyde) do not warrant or guarantee this software in any way. In particular, you cannot expect corrections of any given defect in the system. Obviously, I try to fix known problems (if possible), but I refuse to be held legally responsible for such defects in the software.

The purpose of developing a prototype implementation of the HLA language was to try out language design and implementation ideas. The prototype phase of HLA development is rapidly coming to an end and an "official" HLA language design will be forthcoming. HLA v3.0 will implement this new language. The only guarantees I make about compatibility between HLA v2.x and HLA v3.0 is that there *will* be some incompatibilities. The exact nature and magnitude of those incompatibilities is unknown at this point, but it is safe to assume that no HLA v2.x program will compile under HLA v3.0 without at least some minor source code changes. So please don't get the idea that any investment you make in HLA source code will be protected in v3.0 (note: after the release of v3.0 this is a relatively safe assumption to make, though there will still be no guarantees).

Because HLA is constantly changing (typical of a prototype), it is very difficult to keep the documentation in phase with the language. You can expect this documentation (and all HLA documentation) to contain omissions (e.g., of new features that have yet to be documented), discussion of features removed from HLA, and incorrect descriptions of HLA features. Every attempt will be made to keep the documentation in phase with the software, but like so many free software projects, lack of time and motivation prevents perfection[1].

This software is not fit for use in mission-critical or life-support software systems. This software is principally intended for evaluation and educational (i.e., learning assembly language) purposes only. It has been successfully used to develop commercial and industrial applications (including a nuclear reactor control system) and it has been successfully used in educational environments, but again, you are personally responsible for determining the fitness of this software and documentation for your particular application and you must take responsibility for that choice.

1. You must admit, though, HLA's documentation is better than that of most free software.

HLA’s current design makes use of other software tools that I (Randall Hyde) did not write. These tools include the Microsoft Linker, the Microsoft Librarian, the Pelles C linker, the Pelles C librarian, and the Free Software Foundations *ld* and *as* programs. It can optionally make use of programs such as MASM, FASM, TASM, and NASM. Because some of these tools are commercial products and are covered by various license agreements, not all of these tools come with the HLA distribution. For example, if you want to use the Microsoft or Borland tools, you’ll have to obtain copies of them from some other source. Note that using HLA does not require the Microsoft or Borland tools; HLA is simply compatible with these tools if you already own them and would prefer to use them. HLA does ship with all the tools you need to effectively use HLA; the use of these non-free tools is optional.

1.7 Teaching Assembly Language using HLA

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 provided 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 when 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 also 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.) However, 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 over. Therefore, 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. I was writing some 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. This would cause 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. 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 1, 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 straightforward, you would be surprised how many students cannot visualize this code. 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 them 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. Therefore, 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.

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 typical 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. Codeview, which is practically useless under Windows, was the most advanced tool Microsoft provided 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 was 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. Therefore, 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 and 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 middle of 2002, The Art of Assembly Language was on-line and receiving rave reviews across the internet. In 2003, No Starch Press published an edited and revised edition in "treeware" form.

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 (under Windows) 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. HLA v2.0 also supports Mac OS X, Linux, and FreeBSD. Applications written in HLA that use the HLA Standard Library can run under all four operating systems with nothing more than a recompile. This allows a student, for example, to work under Windows at home and submit projects under Linux (or any of the other OSes) at school.

When designing the HLA language, I chose a syntax that is very similar to common imperative high-level languages such as 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 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 occured",nl


                         except  $Overflow

                            cout    "Overflow error occured",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 allow me to 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 clever macros (e.g., enum, cout, and cin in the MASM code), it is quite possible to do some 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 rapidly gets ugly.

The "die-hard" assembly fan will probably observe 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" )


static
```

```
                dword 37, 37;

    TcHWpStr:   dword; @nostorage;

                byte  "The classical 'Hello World' program: ",0,0,0;


                dword 11, 11;

    HWstr:      dword; @nostorage;

                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 STATIC 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 very easy to write some unreadable code if you abuse instruction composition. E.g., consider:

```
mov( add( mov( 0, eax ), sub( ebx, ecx)), edx ), mov( i, esi ));
```

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( @c ) then

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

endif;
```

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 is a good reason 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 or parameter values it must compute.

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 many 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 ))

             // Load address of string constant into EAX.

            lea( eax, SomeString );


        #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. Invocations 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 some sample MASM code, similar to what the HLA compiler emits (when using the -masm and -source command-line options) for the sequence above:

```
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, although the HLA language provides a SWITCH/CASE statement, you can create a new one with slightly different semantics. I implemented the SWITCH .. CASE .. DEFAULT .. ENDCASE statement using HLA's macro facilities (as a demonstration of HLA's power). An HLA SWITCH statement (using this macro) 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.