

UNDERGRADUATE THESIS  
School of Engineering and Applied Science  
University of Virginia

**A Programming Language for  
the Leta Distributed Shared Memory Platform**

Zach Buckner  
Electrical Engineering  
TCC402  
July 27, 2002

On my honor as a University student, on this assignment I have neither given nor received  
unauthorized aid as defined by the Honor Guidelines for Papers in TCC Courses.

---

Approved \_\_\_\_\_ Date \_\_\_\_\_  
TCC Advisor: Bernard Carlson

Approved \_\_\_\_\_ Date \_\_\_\_\_  
Technical Advisor: Chris Milner

# Table of Contents

|   |    |
|---|----|
| Table of Contents.....                              | 2  |
| Glossary of Terms.....                              | 3  |
| Abstract.....                                       | 5  |
| Introduction .....                                  | 6  |
| Review of Relevant Literature and Technologies..... | 8  |
| Web.....  | 8  |
| Distributed Shared Memory.....                      | 9  |
| Single Address Space Operating Systems.....         | 9  |
| Rationale for Creating a Compiler.....              | 11 |
| Wigwams Grammar .....                               | 11 |
| Numbers .....                                       | 13 |
| Functions .....                                     | 13 |
| Compiler Implementation.....                        | 15 |
| Marker Preservation During Function Calls.....      | 15 |
| Hardware Stack .....                                | 16 |
| Testing.....  | 17 |
| Bibliography .....                                  | 20 |
| Appendix A – Source Code.....                       | 23 |
| wigwamsParser.g.....                                | 23 |
| Compiler.java.....                                  | 30 |
| VirtualMachine.java.....                            | 49 |
| assemblerParser.g.....                              | 61 |
| Assembler.java.....                                 | 64 |

## Glossary of Terms

**Arbitrary Length Addressing (ALA)** – A design feature that enables a processor to handle memory addresses of any length. While most processors only handle fixed-size address sizes (usually either 32-bit or 64-bit), instructions issued to an ALA processor can be any length.

**Arbitrary Length/Precision Math (ALPM)** – A design feature that enables a processor to handle mathematical operands of any length and scale. Non-ALPM processors have fixed-sized registers that hold integer and floating point operand (usually either 32-bit or 64-bit), confining numbers to a strict range of possible values. ALPM processors have no such restrictions; any finite floating point number can be encoded as an instruction operand.

**Distributed Shared Memory (DSM)** – A research area for distributed and parallel computing; each computer is presented with the abstraction of reading and writing from a single, large memory pool. If one computer writes a value to an address, another computer could then read the value from that address.

**Lleta** - A downloadable DSM toolkit that enables personal computers to share data in an Internet-wide shared memory.

**Lleta Memory Management Layer** – Lleta’s first layer. This software layer provides access to the shared memory pool. The Memory Management Layer is responsible for all network communication between computers running Lleta.

**Lleta Instruction Set Architecture (ISA)** – The bit format for all programs published in the Lleta memory pool. This is standardized so that different computers can share programs. The ISA defines allowable instruction types (like ADD, DIVIDE, CALL) and number formats. The ISA also prescribes exact behavior for a processor executing Lleta instructions.

**Lleta Machine Code** – A block of instructions that follow the Lleta ISA. Machine code is raw binary data read by a processor. It is therefore not directly human-readable.

**Lleta Processor / Virtual Machine** – Lleta’s second layer. A processor is a hardware or software component that is capable of executing instructions according to the Lleta Instruction Set Architecture specifications. The current Lleta prototype uses a software processor called a *virtual machine*. The virtual machine translates Lleta instructions into instructions that can be executed by a Java microprocessor (although the Java microprocessor is itself usually a virtual machine).

**Lleta Assembler** – Lleta’s third layer. This program creates raw binary machine code that can be executed by Lleta processor. Instructions inputted into the assembler using a minimalist text-based language.

**Lleta Marker** – Abstract pointers to main memory used by the Lleta Virtual Machine. Since Lleta’s processor is ALA and ALPM, it can’t use registers to house operands and addresses. Instead, it uses as markers as “pointers” to operands and addresses in the Lleta memory pool. Each marker has two fields: an identifier and a position. The virtual machine uses the identifier field, an ALPM nonnegative integer, as a unique label to identify a particular marker. The position field refers to the marker’s offset within the Lleta memory pool. This, too, is an ALPM nonnegative integer.

**Stack** – A section of memory, managed by a processor, that holds an ordered list of data elements. For example, it could hold the numbers 3,5,7,9 in their correct. Stacks can be *popped* and *pushed*; Elements are always popped (removed from the stack) in the reverse order that they were pushed (added to the stack). This is analogous to stack of plates at a cafeteria: the last plates added to the stack are always the first ones removed.

**Wigwams Language** – Lleta’s fourth layer, the high level language described in this report.

## Abstract

Distributed web software is difficult to develop, as current technology requires programmers to spend too much time writing *messaging code* to establish point-to-point connections, make requests, send data and handle errors. Distributed Shared Memory (DSM) technology could make programming simpler by hiding the details of messaging. Behind the scenes, a DSM system still sends messaging code, but the user is presented with the abstraction of reading and writing from a single, large memory pool. Many researchers have explored the application of DSM to parallel programming (usually a dedicated cluster of workstations working together to solve a single task), but few people have explored the use of DSM for consumer software development. To explore this frontier, I developed Lleta, a downloadable toolkit that enables personal computers to share data in an Internet-wide distributed shared memory. But to truly benefit web software developers, Lleta needs its own programming language and compiler.

During this study, I built and tested a language and compiler for Lleta, called Wigwams. I chose a C-like grammar as a foundation because of its simplicity and widespread use. Because Lleta's processor is registerless and uses arbitrary length/precision arithmetic, new features were incorporated into the language grammar to describe arbitrary length/precision variables. Additional features like dynamic memory allocation and synchronization variables were considered, but because of the difficult design decisions they introduced, I decided that these features should be omitted. I wrote the compiler in Java using the Antlr compiler-generator. A hardware stack and several new instructions were added to the processor specification so that it could better support arithmetic and functions. The language and compiler were tested and meet the design goals.

## Introduction

Personal Computing is dead, but its replacement, Network Computing, is still embryonic. A few good network applications exist now, but they are difficult to engineer and difficult to use. In a few years, however, all software will be delivered via the Internet, freeing users of the chore of managing an operating system and software. The network will control even hardware. A user will plug in a new digital camera, for instance, and the network will take care of the rest.

How can we reach this vision? Proponents of *Distributed Shared Memory* systems (DSM) suggest that computers on a network share their memory. This gives each computer the illusion of owning a very large memory pool (possibly a billion times larger than the modest 128 megabytes standard on today's computers). *Every* morsel of data on *every* computer will be stitched together into this large virtual memory pool, including hard drive space, main memory, microprocessor state, and even the low level registers that control devices like video cards. This makes data sharing truly effortless; programs will access local and foreign data of any origin.

The DSM vision is promising, but today's DSM systems fall short of it. They are designed for scientists, for users that own all of the computers in the memory pool. The Internet, however, is a much harsher climate. For a DSM system to survive here, it must be rugged. It must require no special hardware and support a rapidly growing number of users. It must work without centralized 'government' and be simple and elegant – delivering instant success to new users.

Lleta is DSM software that fulfils the needs outlined above. Lleta could be a large step forward in the evolution of computing, making Internet services more powerful and making data easier to manage. It may open entirely new doors, as well. Since all software and algorithms will be at the fingertips of each computer, the network will quickly become the software equivalent to the Library of Congress. Software designers will be able to reuse each other's components with unprecedented ease.

Part of Lleta's strength and flexibility comes from its processor. While most of today's computers have fixed-sized memory blocks called *registers* that house both memory addresses and arithmetic operands, the Lleta processor steers around this design. The Lleta processor and its machine code use *arbitrary length addresses* and *arbitrary length/precision operands*, freeing the processor from constraints on (A) the total amount of memory the processor can work with and (B) the types of data that the processor can manipulate. These two factors are critical in fulfilling Lleta's goals. Arbitrary length addresses make Lleta resilient to rapid data growth, and arbitrary length/precision operands make Lleta useful for doing math in diverse application areas such as signal processing and cryptography.

Much of the first Lleta prototype is complete; it lacks only a good programming language and compiler. Existing languages like C and Java are strong foundations, but they don't fully harness Lleta's power and flexibility. To compensate, new ideas meld with old to form the *Wigwams Language*. This language has tomorrow's features and a comfortable, familiar grammar. This report highlights the development of a language and compiler for the Lleta system.

## Review of Relevant Literature and Technologies

Lleta is a new technology, best explained as the intersection of the World Wide Web, Distributed Shared Memory (DSM) computing, and Single Address Space Operating Systems. Like the World Wide Web, Lleta delivers documents, forms, and other files. It is a Distributed Shared Memory (DSM) system, allowing authors and programmers to publish their works in a shared memory pool. Since concurrent processes share Lleta's memory pool, Lleta must provide protection and resource allocation mechanisms, elements of the blossoming Single Address Space Operating System research area.

### Web

The World Wide Web (WWW) is a simple standard that lets end users retrieve documents from publishers. The public has widely adopted the WWW and documentation abounds. The problem with the WWW is its inflexibility; while it is convenient for browsing hypertext, it is difficult to build network software that rides atop the WWW. For example: although bank customers can easily access their account balance from a web browser, it is difficult to incorporate this account balance in a larger piece of software. If a programmer had accounts at both the Bank Of Alaska and Anchorage Bank, it would be difficult to write a program to retrieve both balances and display the sum. The programmer he would spend a lot of time writing *messaging code*.

WWW browsers and servers communicate by sending brief request/reply messages like the following:

*Browser:* "Server, please give me The Declaration of Independence"

*Server:* "Okay, here it is: When in the Course of human events..."

These messages are tedious for programmers to generate [1]. Programmers need a better technology for software design: something that supports web-like document exchange and is as simple to program as a personal computer.

## **Distributed Shared Memory**

Distributed Shared Memory (DSM) systems [1] provide a single virtual memory for multiple workstations. Kühn describes shared memory systems as offering:

“a conceptually higher level of abstraction than message-passing systems. It makes robust and distributed application development easy. . . They allow the design of symmetrical application architectures, thus avoiding the client/server bottleneck.”

[17]

Unfortunately, DSM research focuses primarily on performance improvements and convenience for parallel programming [9]. There is no published researcher of its use as a information management tool, or of its scalability to the Internet.

## **Single Address Space Operating Systems**

Modern 64-bit microprocessors can address a stunning 16 million terabytes of data, much more than the storage requirements for today's users [2,3]. It is now possible to map all of the workstation's memory resources into a single linear address space. This design is called a Single Address Space Operating System (SASOS). Several SASOSs have been developed, including the popular Mungi System[5] and Opal [6]. Although these operating systems are designed for personal computers, they reveal the problems and design considerations of sharing an address space among several programs.

SASOS designs have fueled controversy about the relative size of a 64-bit address space against an average computer's memory requirements [4]. This disputed area is crucial both for single-computer and distributed address spaces. Address regions will need to be allocated in advance to allow processes and computers to grow into their allocated space. Although Chase et al. claim that a 64-bit address space is large enough to sustain user requirements for a "long time" [13], a technology often outgrows the original designer's expectations. For example, the Internet has nearly exhausted the IPv4 addresses that identify individual hosts [10].

Because of the ultimate limits of a fixed-size address space, there has been very little research in public, mutually untrusting shared address spaces [10]. Kotz and Crow[2] argue that, even for single-user SASOSs, the operating system must accommodate expansion to the address space. The Lleta design [11] appears to be the only architecture meeting this demand.

## Rationale for Creating a Compiler

To develop software, Lleta programmers needed a high level language, and the choice between a compiled and interpreted language was difficult. An environment for an interpreted language would be easier to debug and faster to develop as flow of control elements and arithmetic could be directly delegated to the interpreter's language. For example, an interpreter written in Java might offer the 'plus' operator, while Java might actually do the adding. Interpreted languages also execute human-readable source code, which make it easier for programmers to share software and algorithms.

Building a compiler is a more natural choice for Lleta, since Lleta has a full-featured processor instruction set. The Lleta Instruction Set Architecture remains the only strict publishing standard, and improvements to a compiled language could still compile down to meet this standard. A hardware processor may ultimately replace the software virtual machine, making compiled programs much faster. A compiled language was clearly the best choice. This language was codenamed *Wigwams*.

## Wigwams Grammar

In decreasing order of importance, the requirements for the grammar of Wigwams were:

- 1) The language must be easy to learn, by reusing an existing popular language
- 2) The language must support the Lleta's eccentric handling of arithmetic and addresses conveniently.
- 3) As far as possible consistent with requirements 1) and 2), the language must parallel the grammar of the assembler.

Since Java and C are widely used and share much of the same grammar, these languages provided a good foundation. Although the Wigwams language can reuse many of the arithmetic operators and flow-of-control syntax that are common to both languages, Lleta's *markers* and *variables* required more than either Java or C could provide.

One possible solution was to adopt an object-like syntax for creating variables and markers. For example, the line

```
Marker a = new Data(-2324, 12);
```

allocates 12 bits of space for the variable *a*, initially valued at  $-2324$ . Although this syntax is easy to learn since most programmers are familiar with object operations, it is awkward for creating simple constants and generally inconsistent with the rest of the language.

Instead of reusing the object-like syntax, three new, terse declaration statements were created: *data*, *bits*, and *marker*. The *bits* statement allocates a raw bitstring without a Lleta variable header and sets a named marker at the start of the bitstring. The *data* statement statically allocates a Lleta variable, and sets a named marker at the start of the variable. The *marker* statement sets a named marker to an absolute address. These statements are illustrated by the following examples:

```
bits b = #(0 length 1024);  
data d = #(2.4 scale 3 length 32);  
marker m = #(3234);
```

After declaration, each of the symbols created by the three statements refer to markers. These markers can be used directly for arithmetic operation or – like C pointers – can be referenced and dereferenced using the *\** and *&* operators. For example:

```

// add the values c and b together and store the result in a
a = b + c;

// set marker d point to the address 5
&d =5;

// make the value of variable a point to the value
// at memory address 5.
a = *d;

```

## Numbers

Numbers appear in source code in two places: in variable declarations, where the number is statically allocated into the memory map, and in arithmetic expressions, where the numbers go to the processor's stack. In both cases the number grammar is the same. For example:

```

// the number 33 is statically allocated
data d = #(33 length 16);

// the operand 12 is pushed on the stack during calculation
b = 12 + a;

```

Numbers can appear without any other type information, like 432.24 or 1, or they can appear with other “length” and/or options, with a special expression `#(..)` expression. The length option determines how wide the variable is (in bits) and scale determines the number of decimal digits should be on the right side of the equal sign. This scale and length format parallels Java `BigDecimal` format (see `java.math.BigDecimal` class, included with the Java Development Kit). If length or scale option is omitted, the compiler creates a number with the minimum size and scale necessary to encode the number.

## Functions

Function calls remain similar to their counterparts in C. Functions are named, take parameters, and enclose their body in curly braces, `{` and `}`. The following is an example of functions:

```

average(a, b) {
    data x #(2.4 scale 3 length 32);
    x = (a + b) / 2;
    return x;
}

main() {
    data x #(2.4 scale 3 length 32);
    data y #(50 scale 3 length 32)\n;
    data j #(0 scale 3 length 32)\n;
}

```

```
    while (x < 1000) {
        data q #(4 scale 3 length 32);
        x = x * q;
    }
    x = average(x, y);
}
```

The example contains two functions, `main` and `average`. The `main` function declares several variables, performs some arithmetic operations and calls the `average` function, which returns the 32-bit average of the two operands  $a$  and  $b$ . The `main` function is special; it represents the start of execution.

## Compiler Implementation

The compiler was built in Java using the Antlr Translator Generator [18]. Antlr creates a parser from a grammar file (the Wigwams grammar is listed in the Appendix) and offers several options for interfacing a code generator. These include:

- 1) Directly calling code generation functions as production rules are matched.
- 2) Constructing an abstract syntax tree (AST) for the parsed code, enabling the user to manually traverse the AST to generate code.
- 3) Generate an AST, then use the Antlr-supplied “tree parser” to recognize language elements and call user code. The tree parser visits AST nodes in depth-first order.
- 4) Generate an AST, but instead of using generic AST node types, replace nodes with user-supplied nodes, according to the matched production rule. For example, The root node to a for loop might be an object of class `lleta.virtualmachine.compiler.ForLoop`.

I used a combination of options 3) and 4). Option 3) is the most automated and convenient, since code generation is embedded directly in the grammar file. Its depth-first traversal is only appropriate for generating numbers data, however, as other language elements require more control over the traversal order. For example, block nodes (those that indicate groups of source code between curly brackets) need to emit code both before and after its children emit code. Hence, all other language elements beside use option 4, Wigwams nodes replace AST nodes, typed according the matched production rule.

### Marker Preservation During Function Calls

Programs use numbered markers to point to important offsets in memory, including variable positions and jump addresses. These markers constitute the execution state, so the compiler must preserved these markers position during function calls and restore them after the function is complete. For example, if `functionA()` has Markers 3,4,5 and 6 set to important

data elements and it calls functionB(), it should not be possible for functionB() to irreparably reposition functionA()'s markers.

One option to is to allocate random marker identifiers for functions' local variables. Since the Lleta marker identifier are APLM numbers, the range of possible identifiers is infinite. Even if the range was restricted to 128-bit numbers, the chance of collision would be remote. Unfortunately, this scheme doesn't give users the option to pick specific numbers and the algorithms to generated identifiers may be complex and nonportable. It also does not eliminate the risk of collision.

The remaining options involve saving a copy of the marker positions before transferring to a new function, so that they can be restored when the function call is complete. Marker states can be serialized and written directly to memory, but this technique requires dynamic memory allocation, since the size of the serialized data will vary depending on the number of allocated markers. An individual layer of the Lleta design (memory management, virtual machine, etc) must only use layers beneath it. Unfortunately, dynamic memory allocation is above the compiler, ruling out this option.

The final option, the one taken during implementation, is to serialize marker state and push it onto a hardware stack. This technique avoids *enforcing* dynamic allocation, as designers are free to implement the hardware stack on the processor in any way deemed convenient. The processor must, however, push the current marker state onto the stack at the start of the CALL instruction, and restore the state after a HALT instruction.

## **Hardware Stack**

A hardware stack provides protection for functions' local state, their markers, from corruption once another function is called. It also allows functions to return their result to the expression that called the function. For example, if functionB() is called in the expression  $a = 2 * (\text{functionB}() + 1)$ , then functionB can place its result value back onto the stack, ready to be used in the arithmetic expression. Finally, a hardware stack makes arithmetic simpler. Since Lleta's processor is ALPM, it is difficult to allocate scratch space for calculations. For example, if a,b,c, and d are markers, the expression  $(a * b) + (c * d)$

produces intermediate values ( $a*b$ ) and ( $c * d$ ) before the addition operation. The compiler can't allocate space in memory for these intermediary values, since operands' lengths and scales may not be known at compile time. A stack, however, could store these temporary values..

Because of these advantages, a stack was added to the Lleta Instruction Set Architecture (ISA) specification and to the prototype Lleta Virtual Machine. From the ISA's perspective, this was simply a "black box" First In First Out (FIFO) structure with instructions to add and remove (*push* and *pop*) Lleta data variables. The stack shifts responsibility to the processor, where intermediate values could be stored on dedicated, high-speed on-processor memory. For the first Lleta prototype, a hardware stack was added directly to the VirtualMachine and the entire VM state was encapsulated into a single *state* class that could be added and removed from the stack. See the VirtualMachine.State class and the CALL and HALT sections of VirtualMachine.cycle() function in the Appendix.

## Testing

The compiler was built in stages. After new features were added to the compiler, a test program was written and executed to assure that the feature had been implemented correctly. This repetitious *spiral development* process insured against major, expected problems at completion. All translation features of the compiler were tested: data declarations, arithmetic, conditionals, loops and functions. Correctness isn't sufficient, however. From the programmer's perspective, convenience is the most important metric. The following example demonstrates how the Wigwams language makes programming tasks easier. First, the assembly program:

```
counter:      arith (65 length 8);
diff:        arith (0 length 8);

programStart:

// marker 1 goes to an offset 0, IP address
bumpA    0 1 0 (0x0C0A800970000000 length 65);

// marker 2 goes to the counter
bumpR    0 2 0 0 0 counter;

// marker 3 goes to the data portion of the counter.
bumpR    0 3 0 2 0 0;
bumpP    0 3;
```

```

// marker 4 is the difference
bumpR    0 4 0 0 0 diff;

loopTop:// [4] = [2] - (ASCII Z)
subtract 2 0 (91 length 8) 0 4;

// if [4] is negative, halt
cond     4;
halt;

// copy data to client area (8 bits from [3] to [1])
copy     0 3 0 1 0 8;

// update counter, data marker, then loop
add      2 0 1 0 2;           // [2] 1 -> [2]
bumpR    0 1 0 1 0 8;
bumpR    0 0 0 0 0 loopTop;

```

And now, the Wigwams source code:

```

marker destination = #(0x0C0A8009700000000 length 65);
for (data i = # (65 length 8); i <= 91; i++) {
    memcpy(&destination, bits(i), length(i));
    &destination += length(i);
}

```

These two programs do the same thing: they write the ASCII letters A through Z to successive memory locations. The assembly language version is much longer and more difficult to understand. Notice the functions *memcpy()*, *bits()*, and *length()*. These are functions provided via a standard Wigwams library, created directly in assembly language.

## Conclusion

Lleta's immediate needs were satisfied by the Wigwams language and compiler built during this design project. The C language served as a foundation for the grammar, requiring only minor modifications. The compiler was made more difficult by several obstacles, the most significant involving Lleta's ALPA and ALM design features. I overcame these obstacles by adding a stack to Virtual Machine and by adding new instructions to the ISA. The language and compiler were tested by writing several small programs. This demonstrated correct behavior for all key aspects of the language.

The Wigwams language brings Lleta closer to its debut on the Internet. It enables programmers to write large software projects and is a good platform for future research on Internet-scale distributed shared memory systems. However, Lleta still needs dynamic memory allocation and process synchronization features – challenging features that were postponed during this project. Since the test programs written in this project were not extensive, I recommend performing case studies on larger programs. This will could better validate the design decisions and may identify performance bottlenecks.

## Bibliography

1. J. Silcock, Distributed Shared Memory: A Survey, School of Computing and Mathematics, Deakin University, Technical Report TR C95/22, June 1995.
2. David Kotz and Preston Crow. The expected lifetime of single-address-space operating systems. In Proceedings of SIGMETRICS'94, Nashville, Tennessee, (USA), May 1994. ACM Press.
3. J.S. Chase, H.M. Levy, M. Barker-Harvey, and E.D. Lazowska. How to use a 64-bit virtual address space. Technical Report 92-03-02, Department of Computer Science and Engineering, University of Washington, March 1992.
4. Alberto Bartoli, Sape J. Mullender, and Martijn van der Valk. Wide-Address Spaces -- - Exploring the Design Space. ACM Operating Systems Review 27, 1 (1993) 11-17
5. Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. The Mungi Single-Address-Space Operating System. Software – Practice and Experience, 28, 9 (1998), 901-928.
6. Jeffrey S. Chase, Miche Baker-Harvey, Henry M. Levy, Edward D. Lazowska. Opal: A Single Address Space System for 64-Bit Architectures. Operating Systems Review, 26, 2 (1992) 9.
7. Dearle, A., Bona, R., Farrow, J., Henskens, F., Lindstrom, A., Rosenberg, J., and Vaughn, F., “ Grasshopper: An Orthogonally Persistent Operating System”, Computing Systems, 7(3), pp. 289-312, Summer 1994.
8. J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: A Fast Capability System. In Proc. 17th ACM Symposium on Operating Systems Principles, pages 170-185, Kiawah Island Resort, near Charleston, SC, USA, Dec. 1999. ACM.

9. J. B. Carter, A. L. Cox, D. B. Johnson, and W. Zwanepoel. Distributed operating systems based on a protected global virtual address space. In 3rd IEEE Workshop on Workstation Operating Systems, pages 75--79, Key Biscayne, FL, April 1992.
10. Salus, Peter. Penguin's Progress: A Look at IPv6. Linux Journal 2000, 69es (January 2000) Article 23.
11. Buckner, Zach. Lleta Design Specification. (July 2001).
12. Koch, Povl T. Distributed Wide-Address Operating Systems. Actes des Journees des Jeunes Chercheurs en Systemes R'epartis (1993), 43-49.
13. Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and Protection in a Single-Address-Space Operating System. ACM Transactions on Computer Systems 12, 4 (1994) 271-307.
14. Ravi Patnayakuni and Nainika Seth. Why license when you can rent? Risks and rewards of the application service provider model. In Proceedings of the 2001 ACM SIGCPR conference on Computer personnel research , pages 182—188, New York, NY, 2001.
15. J Iivari. A hierarchical spiral model for the software process. ACM SIGSOFT Software Engineering Notes 12,1 (January 1987), 35-37.
16. Lawrence G. Roberts. Beyond Moore's Law: Internet Growth Trends. IEEE Computer, January 2000.
17. Eva Kühn. How to Approach the Virtual Shared Memory Paradigm. Parallel and Distributed Computing Practices, 1, 3 (1998).
18. Antlr Translator Generator, <http://www.antlr.org/>



## Appendix A – Source Code

### wigwamsParser.g

```
header {
package lleta.virtualmachine.compiler;
import Compiler.*;
import lleta.virtualmachine.assembler.*;
import lleta.memory.*;
import lleta.virtualmachine.*;
import lleta.utilities.*;
import java.util.*;
import java.math.*;
import antlr.*;
import antlr.collections.AST;
}
class WigwamsParser extends Parser;
options {
    k = 3; // three token lookahead
    buildAST = true;
}

tokens {
    NEGATIVE;
    PROGRAM;
    NUMBER;
    SYMBOLREF;
    IMMEDIATE;
    FUNCTION;
    PARAMETERS;
    FUNCTIONCALL;
}

{
    private static org.apache.log4j.Category log
        = org.apache.log4j.Category.getInstance(WigwamsParser.class);
    public lleta.virtualmachine.compiler.Compiler compiler;
}

program: (function)* EOF! {
    #program = #( #[PROGRAM, "program"], #program);
};
```

```

// -----
// Number
// -----

number: ( numberData |
          (POUND! LPAREN! numberData ( ("scale" tScale:INT) |
                                       ("length" tLength:INT) ) * RPAREN! )) {
          #number = #( #[NUMBER, "number"], #number);
};

numberData: (minus:MINUS!)? (integer | decimal | hex)
{
  if (minus_AST != null)
    #numberData = #( #[NEGATIVE, "negative"], #numberData);
};

integer:      tInt:INT^ ;

decimal :     tFloatBig:INT DOT^ tFloatSmall:INT ;

hex:         tHex:HEX^ ;

// -----
// Function
// -----

function: functionName:ID! LPAREN! parameterList RPAREN! compoundStatement {
  #function = #( #[FUNCTION, functionName.getText()], #function);
};

parameterList: (ID)? (COMMA! ID)* {
  #parameterList = #( #[PARAMETERS, "parameters"], #parameterList);
};

// -----
// Statement
// -----

statement:
  declarationStatement |
  compoundStatement |
  exposedExpression SEMI^ |
  "if"^ LPAREN! expression RPAREN! statement (

```

```

        // CONFLICT: the old "dangling-else" problem... ANTLR generates
        // proper code matching as soon as possible.  Hush warning.
        options { warnWhenFollowAmbig = false; }
:
    "else"! statement
)? |
"while"^ LPAREN! expression RPAREN! statement |
forStatement |
"return"^ primary SEMI!;

exposedExpression: root:expression {
    ((Expression) #root).exposed = true;
};

compoundStatement:
    lc:LCURLY^ (statement)* RCURLY!;

forStatement:
"for"^ LPAREN! exposedExpression SEMI! expression SEMI! exposedExpression RPAREN!
statement;

// -----
// Declarations
// -----

declarationStatement : (dataDeclaration | bitsDeclaration | markerDeclaration);
dataDeclaration : "data"^ name:ID value:number SEMI!;
bitsDeclaration : "bits"^ name:ID value:number SEMI!;
markerDeclaration : "marker"^ name:ID number SEMI!;

// -----
// Expressions
// -----

//   lowest  (13)  = *= /= %= += -= <<= >>= >>>= &= ^= |=
//           (12)  ?:
//           (11)  ||
//           (10)  &&
//           ( 9)  |
//           ( 8)  ^
//           ( 7)  &
//           ( 6)  == !=
//           ( 5)  < <= > >=
//           ( 4)  << >>

```

```

//          ( 3) +(binary) -(binary)
//          ( 2) * / %
//          ( 1) ++ -- +(unary) -(unary) ~ ! (type)

expression:  assignmentExpression;

assignmentExpression
:          relationalExpression
  ( ASSIGN^ assignmentExpression)?
;

relationalExpression:
additiveExpression ( ( LT^ | GT^ | LE^ | GE^ ) additiveExpression)* ;

additiveExpression:
left:multiplicativeExpression ( (PLUS^ | MINUS^ ) right:multiplicativeExpression)*;

multiplicativeExpression:
left:primary ( (STAR^ | DIV^ ) right:primary)*;

primary:
primarySymbolReference | primaryImmediate | primaryExpression | functionCall;

primarySymbolReference: name:ID^ {
#primarySymbolReference = #( #[SYMBOLREF, "symbolref"], #primarySymbolReference);
};

primaryImmediate: value:number {
primaryImmediate = #( #[IMMEDIATE, "immediate"], #primaryImmediate);
};

primaryExpression: LPAREN!  expression RPAREN!;

functionCall: functionName:ID! LPAREN! parameterList RPAREN! {
#functionCall = #( #[FUNCTIONCALL, functionName.getText()], #functionCall);
};

//-----
// Scanner
//-----

class WigwamsLexer extends Lexer;

options {

```

```

    testLiterals=true; // don't automatically test for literals
    k=4; // four characters of lookahead
}

```

```

QUESTION      :      '?'      ;
LPAREN        :      '('      ;
RPAREN        :      ')'      ;
LBRACK        :      '['      ;
RBRACK        :      ']'      ;
LCURLY        :      '{'      ;
RCURLY        :      '}'      ;
COLON         :      ':'      ;
COMMA         :      ','      ;
DOT           :      '.'      ;
ASSIGN        :      '='      ;
EQUAL         :      '=='     ;
LNOT          :      '!'      ;
BNOT          :      '~'      ;
NOT_EQUAL     :      '!='     ;
DIV           :      '/'      ;
DIV_ASSIGN    :      '/='     ;
PLUS          :      '+'      ;
PLUS_ASSIGN   :      '+='     ;
INC           :      '++'     ;
MINUS         :      '-'      ;
MINUS_ASSIGN  :      '-='     ;
DEC           :      '--'     ;
STAR          :      '*'      ;
STAR_ASSIGN   :      '*='     ;
MOD           :      '%'      ;
MOD_ASSIGN    :      '%='     ;
SR            :      '>>'     ;
SR_ASSIGN     :      '>>='    ;
BSR           :      '>>>'    ;
BSR_ASSIGN    :      '>>>='   ;
GE            :      '>='     ;
GT            :      '>'      ;
SL            :      '<<'     ;
SL_ASSIGN     :      '<<='    ;
LE            :      '<='     ;
LT            :      '<'      ;
BXOR          :      '^'      ;
BXOR_ASSIGN   :      '^='     ;
BOR           :      '|'      ;

```

```

BOR_ASSIGN      :      "|=" ;
LOR             :      "||" ;
BAND           :      "&" ;
BAND_ASSIGN    :      "&=" ;
LAND           :      "&&" ;
SEMI           :      ";" ;
POUND          :      "#" ;

// Whitespace -- ignored
WS      :      (
    |      '\t'
    |      '\f'
    // handle newlines
    |      (
        |      "\r\n" // Evil DOS
        |      '\r'   // Macintosh
        |      '\n'   // Unix (the right way)
        )
        { newline(); }
    )
    { _ttype = Token.SKIP; }
;

// Single-line comments
SL_COMMENT
:      "//"
    ( ~('\n'|\r')* ('\n'|\r'('\n'))? )
    { $setType(Token.SKIP); newline(); }
;

// multiple-line comments
ML_COMMENT
:      "/*"
    ( /* '\r' '\n' can be matched in one alternative or by matching
        '\r' in one iteration and '\n' in another. I am trying to
        handle any flavor of newline that comes in, but the language
        that allows both "\r\n" and "\r" and "\n" to all be valid
        newline is ambiguous. Consequently, the resulting grammar
        must be ambiguous. I'm shutting this warning off.
        */
        options {
            generateAmbigWarnings=false;
        }
    :
    { LA(2)!='/' }? '*'

```

```

        | '\r' '\n'           {newline();}
        | '\r'               {newline();}
        | '\n'               {newline();}
        | ~('*'|\n|\r)
    )*
    "*"
    {$setType(Token.SKIP);}
;

protected
UNDERSCORE:    '_';

protected
DIGIT:         '0'..'9';

protected
LETTER:        ('a'..'z' | 'A'..'Z');

protected
HEXLETTER:     ('a'..'f' | 'A'..'F');

INT:           (DIGIT)+;

ID
options {testLiterals=true;}
:      ('a'..'z'|'A'..'Z'|'_'|'$') ('a'..'z'|'A'..'Z'|'_'|'0'..'9'|'$')*
;

HEX:          "0x" (HEXLETTER | DIGIT)+;

//-----
// Tree Walker
//-----

class WigwamsTreeWalker extends TreeParser;
{
    private static org.apache.log4j.Category log =
        org.apache.log4j.Category.getInstance(WigwamsTreeWalker.class);
}

number returns [Assembler.Data.Encoded d] {
    d = null;

```

```

        BigDecimal n;
    }:
    #( NUMBER ( n=numberData | #(negative:NEGATIVE n=numberData))
    ("scale" tScale:INT) | ("length" tLength:INT))* {
        if (tScale != null) {
            int scale = Integer.parseInt(tScale.getText());
            n = n.setScale(scale);
        }
        int length = ( tLength != null ?
            Integer.parseInt(tLength.getText()) :
            -1 ); //minimal

        if (negative != null) n = n.negate();
        d = new Assembler.Data.Encoded(n, length);
    };

    numberData returns [java.math.BigDecimal n] {n = null;}:
    (n=integer | n=decimal | n=hex);

    integer returns [java.math.BigDecimal n] {n = null;}:
    tInt:INT {
        n = new BigDecimal(tInt.getText());
    };

    decimal returns [java.math.BigDecimal n] {n = null;}:
    #( DOT tFloatBig:INT tFloatSmall:INT ) {
        if (log.isDebugEnabled()) log.debug("FLOAT");
        n = new BigDecimal(tFloatBig.getText() + "." + tFloatSmall.getText());
    };

    hex returns [java.math.BigDecimal n] {n = null;}:
    tHex:HEX {
        String sHex = tHex.getText();
        BitVector bvHex = new BitVector(sHex.substring(2, sHex.length()));
        n = new BigDecimal(bvHex.toBigInteger());
    };

```

## Compiler.java

```

package org.lleta.virtualmachine.compiler;
import org.lleta.virtualmachine.assembler.*;
import org.lleta.virtualmachine.*;

```

```

import java.io.*;
import java.util.*;
import java.math.*;
import antlr.*;
import antlr.collections.*;
import antlr.debug.misc.*;

public class Compiler implements WigwamsParserTokenTypes {

    private static org.apache.log4j.Category log =
        org.apache.log4j.Category.getInstance(Compiler.class);

    /** The root of the syntax tree that is currently being compiler */
    Program program = null;

    /** The assembler that is created after compilation*/
    public Assembler assembler = null;

    /**
     * text -> assembler
     * Use the java.io.Reader (which is a character 'frontend' for
     * any input stream) to Construct the assembler.
     */
    public void parse(String s) { parse(new StringReader(s)); }
    public void parse(Reader reader) {
        try {
            // initialize the local, compiletime variables
            assembler = new Assembler();

            // create new lexer / parser
            WigwamsLexer lexer = new WigwamsLexer(reader);
            WigwamsParser parser = new WigwamsParser(lexer);

            // creat the parser
            parser.setASTFactory(this.new ASTFactory());
            parser.compiler = this;

            // parse, and log the AST
            parser.program();
            WigwamsAST t = (WigwamsAST)parser.getAST();
            if (log.isDebugEnabled()) {
                log.debug(t.toStringTree());

                // display the parse tree in a nice little window

```

```

        ASTFrame frame = new ASTFrame("", t);
        frame.setVisible(true);
    }

    // produce code, add
    t.generate();

    if (log.isDebugEnabled()) {
        log.debug("Assembler is null? " + (assembler == null));
        log.debug(assembler);
    }

} catch (Exception e) {
    e.printStackTrace();
}
}

/**
 * The parser builds an abstract syntax tree based on the text it sees.
 * This factory method tells the parser which type of node to
 * create, depending on the token it encounters. For example, the root of
 * a for loop (which is identified by the token 'for') should be
 * represented in the abstract syntax tree as a Statement.For node.
 */
public class ASTFactory extends antlr.ASTFactory {

    public AST create() { return new WigwamsAST( Compiler.this ); }

    public AST create(int type) {
        AST ast = null;
        switch (type) {
            case PROGRAM:
                ast = new Program(Compiler.this);
                break;

            case FUNCTION:
                ast = new Function(Compiler.this);
                break;

            case PARAMETERS:
                ast = new Function.Parameters(Compiler.this);
                break;

            case LITERAL_return:

```

```

        ast = new Statement.Return(Compiler.this);
        break;

    case LITERAL_data:
        ast = new Statement.Declaration.Data(Compiler.this);
        break;

    case LITERAL_bits:
        ast = new Statement.Declaration.Bits(Compiler.this);
        break;

    case LITERAL_marker:
        ast = new Statement.Declaration.Marker(Compiler.this);
        break;

    case NUMBER:
        ast = new DataAST(Compiler.this);
        break;

    case LITERAL_if:
        ast = new Statement.If(Compiler.this);
        break;

    case LITERAL_for:
        ast = new Statement.For(Compiler.this);
        break;

    case LITERAL_while:
        ast = new Statement.While(Compiler.this);
        break;

    case LCURLY:
        ast = new Statement.Compound(Compiler.this);
        // if (currentBlock == null)
        //   currentBlock = (Statement.Compound) ast;
        break;

    case ASSIGN:
        ast = new Expression.Binary.Assignment(Compiler.this);
        break;

    case PLUS:
        ast = new Expression.Binary.Add(Compiler.this);
        break;

```

```
case MINUS:
    ast = new Expression.Binary.Subtract(Compiler.this);
    break;

case STAR:
    ast = new Expression.Binary.Multiply(Compiler.this);
    break;

case DIV:
    ast = new Expression.Binary.Divide(Compiler.this);
    break;

case LT:
    ast = new Expression.Binary.LessThan(Compiler.this);
    break;

case GT:
    ast = new Expression.Binary.GreaterThan(Compiler.this);
    break;

case LE:
    ast = new Expression.Binary.LessThanOrEqualTo(Compiler.this);
    break;

case GE:
    ast = new Expression.Binary.GreaterThanOrEqualTo(Compiler.this);
    break;

case SYMBOLREF:
    ast = new Expression.Primary.SymbolReference(Compiler.this);
    break;

case IMMEDIATE:
    ast = new Expression.Primary.Immediate(Compiler.this);
    break;

case FUNCTIONCALL:
    ast = new Expression.Primary.FunctionCall(Compiler.this);
    break;

case SEMI:
    ast = new Statement.ExposedExpression(Compiler.this);
    break;
```

```

        default:
            ast = new WigwamsAST(Compiler.this);
            break;
        }

        ast.setType(type);
        return ast;
    }
    public AST create(int type, String txt) {
        AST s = create(type);
        s.setText(txt);
        return s;
    }
    public AST create(AST tr) {
        if ( tr==null ) return null;           // create(null) == null
        AST t = create(tr.getType());
        t.setText(tr.getText());
        return t;
    }
    public AST create(Token tok) {
        if ( tok==null ) return null;         // create(null) == null
        AST t = create(tok.getType());
        t.setText(tok.getText());
        return t;
    }
}

public static class WigwamsAST extends antlr.CommonAST {
    public Compiler compiler;

    public WigwamsAST(Compiler compiler) { this.compiler = compiler; }
    public void generate() throws Exception {
        WigwamsAST t = (WigwamsAST) this.getFirstChild();
        while (t != null) {
            t.generate();
            t = (WigwamsAST) t.getNextSibling();
        }
    }
}

/**
 * This is the top level of the source file.
 */

```

```

public static class Program extends WigwamsAST {

    /**
     * This keeps track of the function that is currently being compiled.
     */
    public Function currentFunction;

    public Program(Compiler compiler) { super(compiler); }

    public void generate() throws Exception {
        compiler.program = this;
        super.generate();
    }
}

/**
 * These are huge branches off the Program. Each function gets the
 * virtually its own machine state (so nothing is forbidden) except that
 * function arguments are each allocated a marker.
 */
public static class Function extends WigwamsAST {

    /**
     * This counter keeps track of which is the next marker that can be
     * allocated. This starts at 1 to prevent the IP, 0, from being used.
     */
    int markerCounter = 1;

    /**
     * This is regularly updated to reflect the currently compiling block.
     * This should only be valid during the generate() step.
     */
    Statement.Compound currentBlock = null;

    public Function(Compiler compiler) { super(compiler); }
    public Parameters parameters() { return (Parameters) getFirstChild(); }
    public Statement.Compound block() {
        return (Statement.Compound) parameters().getNextSibling(); }

    public void generate() throws Exception {
        compiler.program.currentFunction = this;
        currentBlock = block();

        // add a label for the start of this function

```

```

        compiler.add(new Assembler.Placeholder.Label(getText()));

        parameters().generate();
        block().generate();
        compiler.simpleInstruction(VirtualMachine.OP_HALT);
    }

    /**
     * The parameter block for the function.  These parameters are each
     * allocated a marker, right at the beginning.  They are within scope
     * for the remainder of the function body.
     */
    public static class Parameters extends WigwamsAST {
        public Parameters(Compiler compiler) { super(compiler); }

        public Vector unroll() {
            Vector v = new Vector();
            AST cP /*currentParameter*/ = getFirstChild();
            while (cP != null) {
                v.add(cP);
                cP = cP.getNextSibling();
            }
            return v;
        }

        public void generate() {
            AST cP /*currentParameter*/ = getFirstChild();

            Statement.Compound cB /*currentBlock*/ =
                compiler.program.currentFunction.currentBlock;

            Function cF /*currentFunction*/ =
                compiler.program.currentFunction;

            while (cP != null) {
                BigDecimal marker = new BigDecimal(cF.markerCounter++);
                cB.put(cP.getText(), marker);
                cP = cP.getNextSibling();
            }
        }
    }
}

```

```

public static class DataAST extends WigwamsAST {
    public DataAST(Compiler compiler) { super(compiler); }
    public Assembler.Data.Encoded data() throws Exception {
        /** An Antlr treewalker creates immediate data elements*/
        WigwamsTreeWalker walker = new WigwamsTreeWalker();
        return (Assembler.Data.Encoded) walker.number(this);
    }
}

public static class Statement extends WigwamsAST {
    public Statement(Compiler compiler) { super(compiler); }

    public abstract static class Declaration extends Statement {
        public abstract Assembler.Data data() throws Exception;
        public Declaration(Compiler compiler) { super(compiler); }
        public String name() { return getFirstChild().getText(); }

        public void generate() throws Exception {

            // the linkables
            Assembler.Placeholder.Linkable startOfDataBlock =
                new Assembler.Placeholder.Linkable();
            Assembler.Placeholder.Linkable endOfDataBlock =
                new Assembler.Placeholder.Linkable();

            Function cF /*currentFunction*/ =
                compiler.program.currentFunction;

            // Jump past the data block
            // link: start of data block
            // label: functionName,
            // data block
            // link: end of data block
            compiler.jump(endOfDataBlock);
            compiler.add(startOfDataBlock);

            String labelText = cF.getText() + "." + name();
            compiler.add(new Assembler.Placeholder.Label(labelText));

            compiler.add(data());
            compiler.add(endOfDataBlock);

            compiler.allocateMarker(name(),

```

```

        new Assembler.Argument.Reference.Link(startOfDataBlock));
    }

/**
 * Defines a data element that can be referenced by name until
 * the closure of this scope.
 */
public static class Data extends Declaration {
    public Data(Compiler compiler) { super(compiler); }
    public Assembler.Data data() throws Exception {
        DataAST d = (DataAST) getFirstChild().getNextSibling();
        return d.data();
    }
}

/**
 * Defines a raw data region that can be referenced by name until
 * the closure of this scope.
 */
public static class Bits extends Declaration {
    public Bits(Compiler compiler) { super(compiler); }
    public Assembler.Data data() throws Exception {
        DataAST d = (DataAST) getFirstChild().getNextSibling();
        return d.data().bits();
    }
}

/**
 * Sets a marker that can be referenced by name until the closure
 * of this scope.
 */
public static class Marker extends Declaration {
    public Marker(Compiler compiler) { super(compiler); }
    public Assembler.Data data() throws Exception {
        return ((DataAST) getFirstChild().getNextSibling()).data(); }
    public void generate() throws Exception {
        Assembler.Argument.Immediate i =
            new Assembler.Argument.Immediate(
                (Assembler.Data.Encoded) data
            );
        compiler.allocateMarker(name(), i);
    }
}

```

```

}

/**
 * Nested block of statements which form a context for data
 * declarations. A symbol that is declared in a compound statement
 * is only visible to children of this Node.
 */
public static class Compound extends Statement {
    /**
     * Compound blocks are nested. This points to the single parent
     * of the current block. This parenting is unknown until the
     * generate() step. The root node is parented with itself, which
     * is the assumed value of
     * compiler.program.currentFunction.currentBlock at the first
     * call to a Statement.Compound.generate().
     * see Function.generate() and Compound.generate()
     */
    Compound parent = null;

    HashMap symbols = new HashMap();

    public Compound(Compiler compiler) { super(compiler); }

    public void put(String symbol, BigDecimal markerIndex) {
        symbols.put(symbol, markerIndex);
    }

    public BigDecimal get(String symbol) {
        Compound cC /*currentCompund*/ = this;

        while (true) {
            if (cC.symbols.containsKey(symbol))
                return (BigDecimal) cC.symbols.get(symbol);

            // see definition of parent
            if (cC.parent == cC) break;
            cC = cC.parent;
        }
        return null /*notFound*/;
    }

    public boolean has(String symbol) {
        return symbols.containsKey(symbol); }
}

```

```

public void generate() throws Exception {

    // my parent is the guy who just called me
    parent = compiler.program.currentFunction.currentBlock;

    // I'm the current block now
    compiler.program.currentFunction.currentBlock = this;

    // keep track of where the marker counter is so that, after
    // this block is finished, we can reuse all of the markers
    // that were allocated here.
    int mC = compiler.program.currentFunction.markerCounter;
    super.generate();
    compiler.program.currentFunction.markerCounter = mC;

    // go back to the older context
    compiler.program.currentFunction.currentBlock = parent;
}
public String toString() {
    boolean hasParent = (parent != this) & (parent != null);
    return symbols.toString() +
        (hasParent ? parent.toString() : "");
}
}

public static class If extends Statement {
    public If(Compiler compiler) { super(compiler); }
    public Expression condition() { return (Expression) getFirstChild(); }
    public Statement block() { return (Statement) condition().getNextSibling(); }
    public Statement elseBlock() { return (Statement) block().getNextSibling(); }

    public void generate() throws Exception {

        Assembler.Placeholder.Linkable elseLink = new Assembler.Placeholder.Linkable();
        Assembler.Placeholder.Linkable elseFinishedLink = new Assembler.Placeholder.Linkable();

        // evaluate conditional expression, which should leave the result on stack
        // if the condition is not met, the jump to the else label
        condition().generate();
        compiler.simpleInstruction(VirtualMachine.OP_NOT);
        compiler.simpleInstruction(VirtualMachine.OP_CONDITIONAL);
        compiler.jump(elseLink);

        // The main clause
    }
}

```

```

        // the else block immediately follows, which we want to skip
        block().generate();
        compiler.jump(elseFinishedLink);

        // The else block
        compiler.add(elseLink);
        if ((elseBlock() != null)) elseBlock().generate();
        compiler.add(elseFinishedLink);
    }
}

public static class While extends For {
    public While(Compiler compiler) { super(compiler); }
    public Expression initializer() { return new Expression.Empty(compiler); }
    public Expression condition() { return (Expression) getFirstChild(); }
    public Expression finisher() { return new Expression.Empty(compiler); }
    public Statement block() { return (Statement) condition().getNextSibling(); }
}

public static class For extends Statement {
    public For(Compiler compiler) { super(compiler); }

    public Expression initializer() { return (Expression) getFirstChild(); }
    public Expression condition() { return (Expression) initializer().getNextSibling(); }
    public Expression finisher() { return (Expression) condition().getNextSibling(); }
    public Statement block() { return (Statement) finisher().getNextSibling(); }

    public void generate() throws Exception {

        // determine jump target (top and bottom)
        Assembler.Placeholder.Linkable jumpTargetTop = new Assembler.Placeholder.Linkable();
        Assembler.Placeholder.Linkable jumpTargetBottom = new Assembler.Placeholder.Linkable();
        initializer().generate();

        // TOP:
        compiler.add(jumpTargetTop);

        // evaluate conditional expression, which should leave the result on stack
        // negate the stack element
        // insert a conditional
        condition().generate();
        compiler.simpleInstruction(VirtualMachine.OP_NOT);
        compiler.simpleInstruction(VirtualMachine.OP_CONDITIONAL);
    }
}

```

```

        // if the condition was NOT met, the jump to the bottom
        compiler.jump(jumpTargetBottom);

        // execute the body of the for loop
        // perform the finisher (usually just update a loop counter or whatever)
        // jump back to the top to reevaluate the conditional
        block().generate();
        finisher().generate();
        compiler.jump(jumpTargetTop);

        // BOTTOM:
        compiler.add(jumpTargetBottom);
    }
}

public static class ExposedExpression extends Statement {
    public ExposedExpression(Compiler compiler) { super(compiler); }
    public Expression expression() { return (Expression) getFirstChild(); }
    public void generate() throws Exception {
        expression().generate();
    }
}

public static class Return extends Statement {
    public Return(Compiler compiler) { super(compiler); }
    public Expression.Primary primary() { return (Expression.Primary) getFirstChild(); }
    public void generate() throws Exception {
        primary().generate();
    }
}
}

public abstract static class Expression extends WigwamsAST {
    /**
     * This gets set to true when the expression is exposed, in other words,
     * this is an outermost expression that is followed by a semicolon. This
     * value is set directly by the parser
     */
    boolean exposed = false;

    public Expression(Compiler compiler) { super(compiler); }

    /**

```

```

    * An empty expression is just completely vacant, basically. See
    * Statement.While for its use.
    */
public static class Empty extends Expression {
    public Empty(Compiler compiler) { super(compiler); }
}

public abstract static class Primary extends Expression {
    public Primary(Compiler compiler) { super(compiler); }

    public static class FunctionCall extends Primary {
        public FunctionCall(Compiler compiler) { super(compiler); }
        public Function.Parameters parameters() { return (Function.Parameters) getFirstChild(); }
        public String name() { return getText(); }

        public void generate() throws Exception {
            Assembler.Instruction cI /*callInstruction*/ = new Assembler.Instruction(VirtualMachine.OP_CALL);
            Vector parameters = parameters().unroll();
            cI.arguments.add(new Assembler.Argument.Immediate(0));
            cI.arguments.add(new Assembler.Argument.Immediate(parameters.size() + 1));

            // if there's not already a marker with this name, allocate a new one
            if (! compiler.program.currentFunction.currentBlock.has(name()))
                compiler.allocateMarker(name());

            // Marker 0 (the instruction pointer) of the new state should point to
            // the function's starting offset
            BigDecimal startOffset = compiler.program.currentFunction.currentBlock.get(name());
            cI.arguments.add(new Assembler.Argument.Immediate(0));
            cI.arguments.add(new Assembler.Argument.Immediate(new Assembler.Data.Encoded(startOffset)));

            for (int i=0; i < parameters.size(); i++) {
                String cP /*currentParameter*/ = ((AST) parameters.get(i)).getText();
                BigDecimal m /*markerNumberForCurrentParameter*/ =
compiler.program.currentFunction.currentBlock.get(cP);

                // the marker number
                cI.arguments.add(new Assembler.Argument.Immediate(0));
                cI.arguments.add(new Assembler.Argument.Immediate(new Assembler.Data.Encoded(m)));
            }
            compiler.add(cI);
        }
    }
}

```

```

public static class SymbolReference extends Primary {
    public SymbolReference(Compiler compiler) { super(compiler); }
    public String symbol() { return getFirstChild().getText(); }

    /**
     * Root through the compiler's symbol table to figure out which marker
     * this symbol refers to.
     */
    public Assembler.Data.Encoded resolve() throws Exception {
        BigDecimal markerNumber = compiler.program.currentFunction.currentBlock.get(symbol());
        if (markerNumber == null) {
            log.debug("SymbolReference dump of symbol table:\n" +
                compiler.program.currentFunction.currentBlock.toString());
            throw new Exception("Can't find symbol: " + symbol());
        }
        return new Assembler.Data.Encoded(markerNumber);
    }

    /**
     * This refers to the generation with respect to an expression tree (it leave referenced
     * data element on the stack. Note that SymbolReferences are also used in assignment
     * operations, where the code generation behavior is different.
     */
    public void generate() throws Exception {
        if (log.isDebugEnabled()) log.debug("primary - symbol, " + symbol());
        if (! exposed) {
            Assembler.Instruction p = new Assembler.Instruction(VirtualMachine.OP_PUSH);
            p.arguments.add(new Assembler.Argument.Immediate(resolve()));
            compiler.add(p);
        }
    }
}

public static class Immediate extends Primary {
    public Immediate(Compiler compiler) { super(compiler); }
    public Assembler.Data.Encoded data() throws Exception { return ((DataAST) getFirstChild()).data(); }
    public void generate() throws Exception {
        if (log.isDebugEnabled()) log.debug("primary - immediate, " + data());
        if (! exposed) {
            Assembler.Instruction p = new Assembler.Instruction(VirtualMachine.OP_PUSH);
            p.arguments.add(new Assembler.Argument.Immediate(new Assembler.Data.Encoded(new BigDecimal(0))));
            p.arguments.add(new Assembler.Argument.Immediate(data()));
            compiler.add(p);
        }
    }
}

```

```

    }
}

public static abstract class Binary extends Expression {
    public Binary(Compiler compiler) { super(compiler); }
    public Expression left() { return (Expression) getFirstChild(); }
    public Expression right() { return (Expression) getFirstChild().getNextSibling(); }

    /**
     * For most of the binary expressions (all except assignment), if the expression is exposed
     * (for example, consider the code line "1+1;"), then nothing should actually be generated
     * on the stack. In this case, both the left and right operands are considered exposed.
     * Consider this example: 1 + (a = 5);
     * The outermost + operation is definitely exposed, so no code is generated and both the primary
     * expression '1' and the assignmentExpression (a = 5) are marked as exposed. No code will
     * be generated for the exposed primaryExpression, and the exposed assignment update the variable
     * without leaving anything on the stack.
     */
    protected void propagateExposure() {
        if (exposed) {
            left().exposed = true;
            right().exposed = true;
        }
    }

    /** Assignment is a wierd kind of binary expression */
    public static class Assignment extends Binary {
        public Assignment(Compiler compiler) { super(compiler); }
        public Expression.Primary.SymbolReference assignTo() { return (Expression.Primary.SymbolReference)
getFirstChild(); }

        public Expression expression() { return (Expression) getFirstChild().getNextSibling(); }

        public void generate() throws Exception {

            if (log.isDebugEnabled()) log.debug("assignmentExpression, to variable " + assignTo().symbol());

            expression().generate();

            // Peek or pop this value, depending on whether this is exposed or not
            Assembler.Instruction i = new Assembler.Instruction( (exposed ? VirtualMachine.OP_POP :
VirtualMachine.OP_PEEK));
            i.arguments.add(new Assembler.Argument.Immediate(new Assembler.Data.Encoded(new BigDecimal(0))));
            i.arguments.add(new Assembler.Argument.Immediate(assignTo().resolve()));

```

```

        compiler.add(i);
    }
}

public static class Add extends Binary {
    public Add(Compiler compiler) { super(compiler); }
    public void generate() throws Exception {
        if (log.isDebugEnabled()) log.debug("add");

        this.propagateExposure();
        left().generate();
        right().generate();
        if (! exposed) compiler.simpleInstruction(VirtualMachine.OP_ADD);
    }
}

public static class Subtract extends Binary {
    public Subtract(Compiler compiler) { super(compiler); }
    public void generate() throws Exception {
        if (log.isDebugEnabled()) log.debug("subtract");
        propagateExposure();
        left().generate();
        right().generate();
        if (! exposed) compiler.simpleInstruction(VirtualMachine.OP_SUBTRACT);
    }
}

public static class Multiply extends Binary {
    public Multiply(Compiler compiler) { super(compiler); }
    public void generate() throws Exception {
        if (log.isDebugEnabled()) log.debug("multiply");
        propagateExposure();
        left().generate();
        right().generate();
        if (! exposed) compiler.simpleInstruction(VirtualMachine.OP_MULTIPLY);
    }
}

public static class Divide extends Binary {
    public Divide(Compiler compiler) { super(compiler); }
    public void generate() throws Exception {
        if (log.isDebugEnabled()) log.debug("divide");
        propagateExposure();
        left().generate();
    }
}

```

```

        right().generate();
        if (! exposed) compiler.simpleInstruction(VirtualMachine.OP_DIVIDE);
    }
}

public static class LessThan extends Binary {
    public LessThan(Compiler compiler) { super(compiler); }
    /** A < B becomes A - B */
    public void generate() throws Exception {
        propagateExposure();
        left().generate();
        right().generate();
        if (! exposed) compiler.simpleInstruction(VirtualMachine.OP_SUBTRACT);
    }
}

public static class GreaterThan extends Binary {
    public GreaterThan(Compiler compiler) { super(compiler); }
    /** A < B becomes B - A */
    public void generate() throws Exception {
        propagateExposure();
        right().generate();
        left().generate();
        if (! exposed) compiler.simpleInstruction(VirtualMachine.OP_SUBTRACT);
    }
}

public static class LessThanOrEqualTo extends Binary {
    public LessThanOrEqualTo(Compiler compiler) { super(compiler); }
    /** A <= B becomes not( B - A ) */
    public void generate() throws Exception {
        propagateExposure();
        right().generate();
        left().generate();
        if (! exposed) {
            compiler.simpleInstruction(VirtualMachine.OP_SUBTRACT);
            compiler.simpleInstruction(VirtualMachine.OP_NOT);
        }
    }
}

public static class GreaterThanOrEqualTo extends Binary {
    public GreaterThanOrEqualTo(Compiler compiler) { super(compiler); }
    /** A >= B becomes not( A - B ) */

```

```

        public void generate() throws Exception {
            propagateExposure();
            left().generate();
            right().generate();
            if (!exposed) {
                compiler.simpleInstruction(VirtualMachine.OP_SUBTRACT);
                compiler.simpleInstruction(VirtualMachine.OP_NOT);
            }
        }
    }
}

void add(Assembler.MemoryElement o) { assembler.elements.add(o); }

void simpleInstruction(int opcode) { assembler.elements.add(new Assembler.Instruction(opcode)); }

void jump(String s) { assembler.elements.add(new Assembler.Instruction.Jump(new Assembler.Argument.Reference.Label(s))); }

void jump(Assembler.Placeholder.Linkable l) { assembler.elements.add(new Assembler.Instruction.Jump(new
Assembler.Argument.Reference.Link(l))); }

/**
 * Allocates a named marker in the current context.
 * @see Assembler.Instruction.MarkerSet for an explanation of the reference argument
 */
void allocateMarker(String s, Assembler.Argument reference) {
    // get a new marker
    // create a marker set instruction (create a reference to the specified linkable)
    // update the symbol table so that this marker label can be identified
    BigDecimal markerNumber = new BigDecimal(program.currentFunction.markerCounter);
    add(new Assembler.Instruction.MarkerSet(markerNumber, reference));
    program.currentFunction.currentBlock.put(s, markerNumber);
    program.currentFunction.markerCounter++;
}

void allocateMarker(String s) { allocateMarker(s, new Assembler.Argument.Reference.Label(s)); }
}

```

## VirtualMachine.java

```

package org.lleta.virtualmachine;
import org.lleta.utilities.*;

```

```

import java.util.*;
import java.math.*;
import org.lleta.virtualmachine.Assembler.*;
import org.lleta.memory.Memory;
import org.lleta.memory.Marker;

/**
 * A software prototype of the Lleta processor. Connect it to the Lleta memory
 * and it'll execute instructions generated with the Assembler or Compiler.
 * It's a Von Neumann computer, both instructions and data are stored in the
 * distributed memory.
 */
public class VirtualMachine {

    /**
     * The memory which this virtual machine attaches to. Note that the
     * markerpool contains pointers to this memory directly, so it can't really
     * be changed after the virtual machine starts.
     */
    protected Memory memory = null;

    /** This represents the current operating state of the virtual machine */
    public State state = null;

    /**
     * A sparse collection of markers. NOTE: This must be recreated EVERY TIME
     * a new memory is swapped in.
     */
    public class State {

        /** The 'hardware' stack, used for arithmetic */
        public Stack stack = new Stack();

        /**
         * Set high after executing the 'conditional' instruction, indicating
         * that the next instruction will be skipped. The flag is reset
         * during the cycle where the instruction should have been processed.
         */
        public boolean conditionalSkipFlag = false;

        /** Fast convenient pointer to Marker0, the instruction pointer */
        public Marker IP;

        /** The pool of markers, keyed by BigInteger marker identifier */

```

```

private HashMap hash = new HashMap();

/**
 * A new state is created every time the virtual machine encounters a
 * OP_CALL instruction. This is a reference to the state that will be
 * made current after the function completes (when a halt instruction
 * is encountered). This should remain null for the main state.
 */
public State returnState = null;

public State() { IP = get(BigInteger.ZERO); }
public void clear() { hash.clear(); }

public Marker get(BigInteger markerNumber) {
    if (hash.containsKey(markerNumber))
        return (Marker) hash.get(markerNumber);

    // else create the new marker
    Marker m = new Marker(memory);
    hash.put(markerNumber, m);
    return m;
}

}

public static final int OP_BUMP_RELATIVE = 0;
public static final int OP_BUMP_ABSOLUTE = 1;
public static final int OP_BUMP_DATA_PREFIX = 2;
public static final int OP_CONDITIONAL = 3;
public static final int OP_COPY = 4;
public static final int OP_DATA_PREFIX_READ = 5;
public static final int OP_DATA_PREFIX_WRITE = 6;
public static final int OP_CALL = 7;

public static final int OP_PUSH = 8;
public static final int OP_POP = 9;
public static final int OP_PEEK = 10;
public static final int OP_PUSH_MARKER_OFFSET = 11;
public static final int OP_POP_MARKER_OFFSET = 12;
public static final int OP_PEEK_MARKER_OFFSET = 13;

public static final int OP_NAND = 14;
public static final int OP_AND = 15;
public static final int OP_OR = 16;
public static final int OP_XOR = 17;

```

```

public static final int OP_NOT = 18;
public static final int OP_SHIFT = 19;
public static final int OP_ADD = 20;
public static final int OP_SUBTRACT = 21;
public static final int OP_MULTIPLY = 22;
public static final int OP_DIVIDE = 23;
public static final int OP_HALT = 24;

public static HashMap nameToCodeMap = new HashMap();
public static HashMap codeToNameMap = new HashMap();
static {
    String codes[] = new String[] {
        "bumpR", "bumpA", "bumpP",
        "cond",
        "copy", "prefixR", "prefixW",
        "call",
        "push", "pop", "peek", "pushM", "popM", "peekM",
        "nand", "and", "or", "xor", "not", "shift",
        "add", "subtract", "multiply", "divide",
        "halt"
    };
    for (int i=0; i < codes.length; i++) {
        nameToCodeMap.put(codes[i], new Integer(i));
        codeToNameMap.put(new Integer(i), codes[i]);
    }
}

private static org.apache.log4j.Category log =
    org.apache.log4j.Category.getInstance(VirtualMachine.class);

/** Constructor that attaches the virtual machine to the bus as a whole */
public VirtualMachine() { this(org.lleta.Lleta.app.memoryManagement); }

/** Constructor that establishes a connection to the provided memory */
public VirtualMachine(Memory memory) {
    this.memory = memory;
    this.state = new State();
}

public void cycleUntilHalt() {
    try {
        while(true) { cycle(); }
    } catch (HaltException h) {
        return;
    }
}

```

```

    }
}

/**
 * Process the instruction currently queued at state[IP]
 */
public void cycle() throws HaltException {

    // Keep track of the IP's starting value, as all marker movement
    // instructions that involve the IP are based on the starting value.
    BigInteger IPstartOffset = state.IP.offset;

    if (log.isDebugEnabled())
        log.debug("cycle(): offset? " + state.IP.offset);
    int opCode = readIntArgument().intValue();
    if (log.isDebugEnabled()) log.debug("cycle(): op? " +
        codeToNameMap.get(new Integer(opCode)));
    BigInteger argumentsLength = readIntArgument();

    if (state.conditionalSkipFlag) {
        // reset, skip the next instruction, finish
        state.conditionalSkipFlag = false;
        state.IP.bump(argumentsLength);
        return;
    }

    switch (opCode) {

        // -----
        // Marker Bump Instructions
        // -----

        case OP_BUMP_ABSOLUTE: {
            BigInteger bumpMarker = readIntArgument();
            state.get(bumpMarker).offset = readIntArgument();
            break;
        }

        case OP_BUMP_RELATIVE: {
            BigInteger bumpMarker = readIntArgument();
            BigInteger relativeMarker = readIntArgument();

            // if we're referencing the IP, use the value offset at the
            // beginning of the instruction

```

```

        BigInteger ROffset =
            (relativeMarker.equals(BigInteger.ZERO) ?
             IPstartOffset :
             state.get(relativeMarker).offset );

        state.get(bumpMarker).offset =
            ROffset.add(readIntArgument());
        break;
    }
    case OP_BUMP_DATA_PREFIX: {
        BigInteger bumpMarker = readIntArgument();
        state.get(bumpMarker).readAL();
        state.get(bumpMarker).readAL();
        break;
    }

    case OP_CALL: {
        State newState = new State();
        newState.returnState = state;
        int numMarkers = readIntArgument().intValue();
        log.debug("call: read " + numMarkers);
        for (int i=0; i < numMarkers; i++) {
            Marker m = newState.get(BigInteger.valueOf(i));
            m.offset = state.get(readIntArgument()).offset;
            log.debug("call: read " + i + ", " + m);
        }
        state = newState;
        break;
    }

    // -----
    // Conditional
    // -----

    case OP_CONDITIONAL:
        // If the operand is negative (true), then perform the next
        // instruction, else (false) skip it
        BigDecimal op = (BigDecimal) state.stack.pop();
        if (op.compareTo(new BigDecimal(BigInteger.ZERO)) >= 0 )
            state.conditionalSkipFlag = true;
        break;

    // -----
    // Stitch: Reflection

```

```

// -----
case OP_COPY: {
    BigInteger mF /*markerFrom*/    = readIntArgument();
    BigInteger mT /*markerTo*/      = readIntArgument();
    int length                       = readIntArgument().intValue();

    if (log.isDebugEnabled()) {
        log.debug("COPY: from marker " + mF + " from offset " +
            state.get(mF).offset + " length " + length);
        log.debug("COPY: to marker " + mT + " from offset " +
            state.get(mT).offset + " length " + length);
    }

    BitVector data = state.get(mF).read(length, false);
    state.get(mT).write(data, false);
    break;
}

case OP_DATA_PREFIX_READ: {
    Marker mD /*markerData*/    = state.get(readIntArgument());
    Marker mS /*markerScale*/    = state.get(readIntArgument());
    Marker mL /*markerLength*/   = state.get(readIntArgument());

    BigInteger startOffset = mD.offset;
    mS.writeDataExisting(new BigDecimal(mD.readAL()), false);
    mL.writeDataExisting(new BigDecimal(mD.readAL()), false);
    mD.offset = startOffset;
    break;
}

case OP_DATA_PREFIX_WRITE: {
    Marker mD /*markerData*/    = state.get(readIntArgument());
    BigInteger scale = readIntArgument();
    BigInteger length = readIntArgument();

    BigInteger startOffset = mD.offset;
    mD.writeAL( scale );
    mD.writeAL( length );
    mD.offset = startOffset;
    break;
}

// -----

```

```

// Stack operations
// -----

case OP_PUSH: {
    state.stack.push(readFloatArgument());
    break;
}

case OP_POP: {
    // write it, without bumping
    BigDecimal op = (BigDecimal) state.stack.pop();
    state.get(readIntArgument()).writeDataExisting(op, false);
    break;
}

case OP_PEEK: {
    // write it, without bumping
    BigDecimal op = (BigDecimal) state.stack.peek();
    state.get(readIntArgument()).writeDataExisting(op, false);
    break;
}

case OP_PUSH_MARKER_OFFSET: {
    Marker m = state.get(readIntArgument());
    state.stack.push(new BigDecimal(m.offset));
    break;
}

case OP_POP_MARKER_OFFSET: {
    BigDecimal op = (BigDecimal) state.stack.pop();
    state.get(readIntArgument()).offset = op.unscaledValue();
    break;
}

case OP_PEEK_MARKER_OFFSET: {
    BigDecimal op = (BigDecimal) state.stack.peek();
    state.get(readIntArgument()).offset = op.unscaledValue();
    break;
}

// -----
// Stitch: 2-arg Integer Instructions
// -----

```

```

case OP_NAND:
case OP_AND:
case OP_OR:
case OP_XOR: {

    // pop the second argument first
    BigInteger b = popIntArgument();
    BigInteger a = popIntArgument();
    BigInteger out = null;
    switch (opCode) {
        case OP_NAND:
            out = a.and(b);
            out = out.not();
            break;
        case OP_AND:
            out = a.and(b);
            break;
        case OP_OR:
            out = a.or(b);
            break;
        case OP_XOR:
            out = a.xor(b);
            break;
    }

    // next argument says where to write the thing
    state.stack.push(new BigDecimal(out));
    break;
}

// -----
// Stitch: 1-arg Integer instructions
// -----

case OP_SHIFT: {
    BigInteger data = popIntArgument();
    data = data.shiftLeft(readIntArgument().intValue());
    state.stack.push(new BigDecimal(data));
    break;
}

case OP_NOT: {
    BigInteger data = popIntArgument().not();
    state.stack.push(new BigDecimal(data));
}

```

```

        break;
    }

    // -----
    // Stitch: 2-arg Float Arithmetic
    // -----

    case OP_ADD:
    case OP_SUBTRACT:
    case OP_MULTIPLY:
    case OP_DIVIDE: {

        // read b first
        BigDecimal b = (BigDecimal) state.stack.pop();
        BigDecimal a = (BigDecimal) state.stack.pop();
        BigDecimal answer = null;

        switch (opCode) {
            case OP_ADD:
                answer = a.add(b);
                break;
            case OP_SUBTRACT:
                answer = a.subtract(b);
                break;
            case OP_MULTIPLY:
                answer = a.multiply(b);
                break;
            case OP_DIVIDE:
                answer = a.divide(b, BigDecimal.ROUND_HALF_UP);
                break;
        }

        // output
        state.stack.push(answer);
        break;
    }

    case OP_HALT:
        // if the return stack is empty, the show ends
        // else this signifies the end of a function call
        if (state.returnState == null)
            throw new HaltException();

        // if there is a value on my arithmetic stack, then push it

```

```

        // onto the return state's stack
        if (! state.stack.isEmpty())
            state.returnState.stack.push(state.stack.peek());

        // go back to the older days;
        state = state.returnState;
        break;

    default:
        throw new RuntimeException("Unknown opcode: " + opCode);
}
}

// -----
// cycle() helpers
// -----

/**
 * All arguments (usually embedded within the instruction) are read
 * using this method. Any argument can come from any marker, so it
 * first reads the IP to determine which marker should be read, then
 * reads it.
 *
 * The IP will get bumped after reading the marker number, no matter
 * what. If the marker number refers to the IP, then it will be
 * bumped for the immediate data. If it refers to any other marker,
 * then it won't be bumped.
 */
public BigDecimal readFloatArgument() {
    // figure out where to read (mN ~= markerNumber)
    BigInteger mN = state.IP.readData().unscaledValue();
    return state.get(mN).readData( mN.equals(BigInteger.ZERO) );
}

public BigInteger readIntArgument() {
    BigDecimal argument = readFloatArgument();
    if (argument.scale() != 0) log.debug("readIntArgument() Read " +
        argument + ", not an integer!");
    return argument.unscaledValue();
}

public BigInteger popIntArgument() {
    BigDecimal argument = (BigDecimal) state.stack.pop();
    if (argument.scale() != 0) log.debug("popIntArgument() Popped " +

```

```

        argument + ", not an integer!");
    return argument.unscaledValue();
}

// -----
// Debugging stuff
// -----

/**
 * Display an overview of the machine's state.
 */
public void debug() {
    System.out.println("IP: " + state.IP.offset + ", skip: " +
        this.state.conditionalSkipFlag);
    System.out.println(memory);
}

/**
 * Display some nice information about the instruction currently queued at
 * state.get(IP).
 */
public String debugCycle() {
    String s = "Offset: " + state.IP.offset + "\n" +
        "Op: " + codeToNameMap.get(
            new Integer( readIntArgument().intValue() ) ) + "\n";

    BigInteger argsLength = readIntArgument();
    s += "ArgsLen: " + argsLength + "\n";

    // print the arguments. Use the 'arguments length' field to
    // figure out how many arguments there are.
    BigInteger stopAt = argsLength.add(state.IP.offset);
    while (state.IP.offset.compareTo(stopAt) < 0) {
        // Read the argument value only if it's coming from 0, the IP
        // (the other markers probably aren't correctly queued)
        if (state.IP.readData(false).equals(BigDecimal.valueOf(0))) {
            s += "Arg:" + readFloatArgument() + "\n";
        } else {
            s += "Arg:(read from marker " + state.IP.readData() + ")\n";
        }
    }
    return s;
}
}

```

```

/** Thrown when the virtual machine encounters a halt instruction */
public static class HaltException extends Exception { }
}

```

## assemblerParser.g

```

header {
package lleta.virtualmachine.assembler;
import Assembler.*;
import lleta.memory.*;
import lleta.virtualmachine.*;
import lleta.utilities.*;
import java.util.*;
import java.math.*;
import antlr.*;
import antlr.collections.AST;
}

class AssemblerParser extends Parser;
options { buildAST = true; k=2; }
{
    public Assembler.Block elements = new Assembler.Block();
    private LinkedList constructionStack = new LinkedList();
}

program :
        (instruction | label | dataArithmetic | dataBits)* EOF!;

instruction :
        opcode:ID^ (argument)* SEMI! {
        // opcode
        if (! VirtualMachine.nameToCodeMap.containsKey(opcode.getText()))
            throw new RuntimeException("Unknown opcode: " + opcode.getText());
        int op = ( (Integer) VirtualMachine.nameToCodeMap.get(opcode.getText()) ).intValue();

        Instruction i = new Instruction(op);

        // children are arguments
        while (constructionStack.size() > 0) {
            Argument a = (Argument) constructionStack.removeFirst();
            a.instruction = i;
            i.arguments.add(a);
        }
}

```

```

        // create the instruction
        elements.add(i);
};

argument :          (argumentImmediate | argumentLabelRef);
argumentImmediate:  number {
    // working backwards, so I don't know my instruction yet (null)
    // should be a Data.Encoded already on the stack
    constructionStack.add(new Argument.Immediate((Data.Encoded) constructionStack.removeLast()));
};

argumentLabelRef:  reference:ID {
    // working backwards, so I don't know my instruction yet (null)
    constructionStack.add(new Argument.Reference.Label(reference.getText()));
};

label :            labelText:ID^ COLON! {
    elements.add(new Placeholder.Label(labelText.getText()));
};

dataArithmetic :  "arith"!  number SEMI! {
    // there should be a Data.Encoded already on the stack
    elements.add((Assembler.Data.Encoded) constructionStack.removeLast());
};

dataBits :        "bits"  number SEMI! {
    // there should be a Data.Encoded already on the stack
    elements.add( ((Assembler.Data.Encoded) constructionStack.removeLast()).bits() );
};

number: (( "(" numberData ( ("scale" tScale:INT) | ("length" tLength:INT) ) * ")" ) | numberData ) {
    BigDecimal data = (BigDecimal) constructionStack.removeLast();
    if (tScale != null) data.setScale(Integer.parseInt(tScale.getText()));

    int length = ( tLength != null ?
        Integer.parseInt(tLength.getText()) :
        -1 ); //minimal

    constructionStack.add(new Data.Encoded(data, length));
};

numberData:  (integer | decimal | hex);

integer:     tInt:INT {

```

```

        constructionStack.add(new BigDecimal(tInt.getText()));
    };

    decimal :      tFloatBig:INT "." tFloatSmall:INT {
        constructionStack.add(new BigDecimal(tFloatBig.getText() + "." + tFloatSmall.getText()));
    };

    hex:          tHex:HEX {
        String sHex = tHex.getText();
        BitVector bvHex = new BitVector(sHex.substring(2, sHex.length()));
        constructionStack.add(new BigDecimal(bvHex.toBigInteger()));
    };

class AssemblerScanner extends Lexer;
options { k=2; }
tokens {
    INSTRUCTION;
    LABEL;
    ARG_IMMEDIATE;
    ARG_LABELREF;
    DATA_ARITH;
    DATA_BITS;
}

WS:      ( ' '
| '\t'
| '\n'
| '\r' )
        { _ttype = Token.SKIP; }
;

LPAREN:  '(';
RPAREN:  ')';
STAR:    '*';
PLUS:    '+';
SEMI:    ';';
COLON:   ':';

protected
UNDERSCORE:  '_';

protected
DIGIT:       '0'..'9';

```

```

protected
LETTER:      ('a'..'z' | 'A'..'Z');

protected
HEXLETTER:   ('a'..'f' | 'A'..'F');

INT:         ('-')? (DIGIT)+;
ID:          (LETTER | UNDERSCORE) (LETTER | DIGIT | UNDERSCORE)*;
HEX:         "0x" (HEXLETTER | DIGIT)+;

```

## Assembler.java

```

package org.lleta.virtualmachine.assembler;
import org.lleta.virtualmachine.*;
import org.lleta.utilities.*;
import java.util.*;
import java.io.*;
import java.math.*;
import cern.jet.math.Arithmetic;
import org.lleta.memory.Memory;
import org.lleta.memory.Marker;
import org.lleta.memory.TransientMemory;
import org.lleta.memory.LocalMemory;

/**
 * This is an assembler for the Computer2 architecture
 */
public class Assembler {

    /**
     * This contains the list of all the memory elements (data, instructions, etc)
     * which will be stitched together during the assemble() method.
     */
    public Block elements = new Block();

    /**
     * Amount of address room to reserve for resolving label, during the first pass
     * This assembler uses two passes to resolve labels. It's 'dumb' in the
     * sense that... on the first pass, it leaves a fixed amount of room for the

```

```

    * addresses, which it fills in on the second pass.
    */
public static final int addressPlaceholderSize = 16;

/**
 * This is the complete memory, which is stitched together from all the
 * individual MemoryElements. It is available after the first assembly
 * pass
 */
public TransientMemory memoryMap = null;

/**
 * This keeps track of the current memory offset. It is valid only during
 * the first pass.
 */
public Marker firstPassMarker = null;

/**
 * This keeps track of all of the Labels encountered during an assembly
 * session
 * Key: the label string
 * Value: the LabeledElement object
 */
public HashMap labels = null;

/**
 * Constructor, given the specified memory elements
 */
public Assembler(MemoryElement[] elements) {
    for (int i=0; i < elements.length; i++) { this.elements.add(elements[i]); }
}
public Assembler(Block elements) { this.elements = elements; }
public Assembler() { }

/** Text -> elements */
public void parse(String s) { parse(new StringReader(s)); }
public void parse(Reader reader) {
    try {
        // create new lexer / parser
        AssemblerScanner lexer = new AssemblerScanner(reader);
        AssemblerParser parser = new AssemblerParser(lexer);

        // parse

```

```

        parser.program();
        this.elements = parser.elements;

    } catch (Exception e) {
        e.printStackTrace();
    }
}

/**
 * elements -> 0101010100
 * Dump the entire memory map which is described by this assembler
 */
public BitVector assemble() {

    // prepare
    memoryMap = new TransientMemory(new BitVector(1024), true);
    firstPassMarker = new Marker(memoryMap);
    labels = new HashMap();

    // first pass
    for (int i=0; i < elements.size(); i++) { ((MemoryElement) elements.get(i)).firstPass(this); }

    // second pass
    for (int i=0; i<elements.size(); i++) { ((MemoryElement) elements.get(i)).secondPass(); }

    return memoryMap.bits;
}

/**
 * Return a formatted list of all of the MemoryElements
 */
public String toString() {
    String s = "";
    for (int i=0; i<elements.size(); i++) {
        s += (elements.get(i) + "\n");
    }
    return s;
}

public Placeholder.Label getLabel(String labelText) {
    if (! labels.containsKey(labelText) ) throw new RuntimeException("specified label not found (" + labelText + ")");
    return (Placeholder.Label) labels.get(labelText);
}
}

```

```

public MemoryElement getElementAtLabel(String labelText) {
    return elements.getNextRealElement(getLabel(labelText));
}

/**
 * Represents a region of memory (be it data or an instructions)
 */
public static abstract class MemoryElement {

    /**
     * All memory elements keep track of where they're inserted into the
     * memory map. This means they can be used to 'watch' a running application,
     * etc. This is created during firstPass()
     */
    BigInteger memOffset = BigInteger.ONE.negate();

    /**
     * All memory elements keep a reference to the 'root' which is the assembly
     * data. This is created during firstPass()
     */
    Assembler assembler = null;

    /**
     * Perform the first pass of assembly. When this method is called from
     * assembler (during its assembler() method) it should know where this
     * memory element will be placed, already. Individual instances of MemoryElement
     * are responsible for advancing the memOffset pointer during this method
     */
    public void firstPass(Assembler assembler) {
        this.assembler = assembler;
        memOffset = assembler.firstPassMarker.offset;
    }

    public void secondPass() {}

    public BigInteger getMemOffset() { return memOffset; }

    /**
     * Given a memory object, most memory element types can look and check
     * out the memory contents.
     */
    public String inspect(Memory memory) { return "don't know how to inspect this memory element type"; }
}

```

```

/**
 * This represents a linear group of MemoryElements.  These blocks can be nested.
 */
public static class Block extends MemoryElement {
    private Vector vector = new Vector();
    public boolean add(MemoryElement e) { return vector.add(e); }
    public MemoryElement get(int index) { return (MemoryElement) vector.get(index); }
    public int size() { return vector.size(); }

    public void firstPass(Assembler assembler) {
        super.firstPass(assembler);
        for (int i = 0; i < vector.size(); i++) { get(i).firstPass(assembler); }
    }

    public void secondPass() {
        super.secondPass();
        for (int i=0; i < vector.size(); i++) { get(i).secondPass(); }
    }
}

/**
 * For the given memory element, find the next real (non label) element
 * that follows it.  Returns null on all questionable circumstances.
 * NOTE: This should be used for label resolution only.  It uses a miserable
 * linear search that would be torture as a traversal.
 */
public MemoryElement getNextRealElement(MemoryElement e) {
    for (int i=0; i < size(); i++) {

        // if we see the element, then check to see if there's a next element
        if (get(i) == e) {
            int j = i;
            while (++j < size()) {
                if ( (get(j) instanceof Data) ||
                    (get(j) instanceof Instruction) )
                )
                    return get(j);
            }
        }

        // if we see a block, test it recursively
        if (get(i) instanceof Block ) {
            MemoryElement mE = ((Block) get(i)).getNextRealElement(e);
            if (mE != null) return mE;
        }
    }
}

```

```

        }
        return null;
    }

    public String toString() {
        String s = "Block:\n";
        for (int i=0; i<size(); i++) {
            s += (org.lleta.utilities.Utilities.indent(get(i).toString(), 1) + "\n");
        }
        return s;
    }
}

/**
 * A 0-length MemoryElement that can be placed in the memory map, so that
 * other instructions can reference it symbolically. These are often used
 * as targets for jumps, etc.
 */
public abstract static class Placeholder extends MemoryElement {

    /**
     * This is a placeholder that can be generated programatically and
     * linked to from an Argument.Reference.Link
     */
    public static class Linkable extends Placeholder {}

    /**
     * This is a placeholder with a name
     */
    public static class Label extends Placeholder {
        public String labelText;

        public Label(String labelText) { this.labelText = labelText; }
        public String toString() { return "Label (" + labelText + ")"; }

        public void firstPass(Assembler assembler) {
            super.firstPass(assembler);

            // make sure the assembler hasn't seen my label before, then add it
            if(assembler.labels.containsKey(labelText))
                throw new RuntimeException("duplicate label: " + labelText);
            assembler.labels.put(labelText, this);
        }
    }
}

```

```

}

/**
 * This represents some piece of data that gets placed into the memory map.
 */
public static abstract class Data extends MemoryElement {
    public Data() { super(); }

    /**
     * Bits doesn't have any of the special fields that ArithmeticData does, it
     * isn't ArbitraryLength prefixed, either. IT IS REVERSED before it gets
     * placed, however.
     */
    public static class Bits extends Data {
        public BitVector data;
        public Bits(BitVector data) {this.data = data;}
        public void firstPass(Assembler assembler) {
            super.firstPass(assembler);
            assembler.firstPassMarker.write(data);
        }
        public String toString() { return data.toString(); }
    }

    /**
     * This is the most common data element. It is used both for holding
     * data to use for arithmetic instructions, and for instruction arguments
     * @see Argument.Immediate
     */
    public static class Encoded extends Data {

        /** The data */
        public BigDecimal data;

        /** The number of bits that the data should occupy. Set this this
         * -1 to use the fewest bits possible */
        public int length = -1;

        public Encoded(BigDecimal data) {this.data = data; }
        public Encoded(BigDecimal data, int length) {this.data = data; this.length = length;}
        public String toString() {
            if (data == null) return "Encoded Data (null!)";
            return "Encoded Data (int: " + data + ", bits: " + bits() + ", scale: " + data.scale() + ")"; }

        public String inspect(Memory memory) {

```

```

        Marker marker = new Marker(memory);
        marker.offset = this.memOffset;
        return marker.readData().toString();
    }

    public void firstPass(Assembler assembler) {
        super.firstPass(assembler);
        assembler.firstPassMarker.writeDataNew(data, length, true);
    }

    public Data.Bits bits() {
        BitVector bits = new BitVector(data.unscaledValue());
        bits.arithmeticResize((length == -1 ? data.unscaledValue().bitLength() + 1 : length));
        return new Data.Bits(bits);
    }
}

public static class Instruction extends MemoryElement {
    public int opcode;

    // Arguments need to have parenting information. This modified version automatically
    // performs parenting.
    public ArgumentVector arguments = new ArgumentVector();
    public class ArgumentVector extends Vector {
        public boolean add(Object o) {
            if (!(o instanceof Argument)) throw new RuntimeException("Can't add non-Arguments to an argument vector");
            ((Argument) o).instruction = Instruction.this;
            return super.add(o);
        }
    }

    public Instruction(int opcode) { this.opcode = opcode; }

    public String toString() {
        String s = "Instruction (" + VirtualMachine.codeToNameMap.get(new Integer(opcode)) + ")\n";
        for (int i=0; i < arguments.size(); i++) {
            s += Utilities.indent(arguments.get(i).toString(), 1) + "\n";
        }
        return s;
    }

    public void firstPass(Assembler assembler) {

```

```

super.firstPass(assembler);

// -----
// Opcode
// -----
assembler.firstPassMarker.writeDataNew(BigDecimal.valueOf(0));
assembler.firstPassMarker.writeDataNew(BigDecimal.valueOf(opcode));

// -----
// Arguments length
// -----

// figure out how big the arguments are by writing them once (scratch)
// and seeing how far the marker moves
BigInteger argLengthOffset = assembler.firstPassMarker.offset;
for (int i=0; i<arguments.size(); i++) { ((Argument) arguments.get(i)).firstPass(assembler); }
BigInteger argLength = assembler.firstPassMarker.offset.subtract(argLengthOffset);

// reset the marker and write the length of the arguments
assembler.firstPassMarker.offset = argLengthOffset;
assembler.firstPassMarker.writeDataNew(BigDecimal.valueOf(0));
assembler.firstPassMarker.writeDataNew(new BigDecimal(argLength));

// -----
// Arguments (now write them for real)
// -----
for (int i=0; i < arguments.size(); i++) { ((Argument) arguments.get(i)).firstPass(assembler); }
}

/**
 * Do nothing, just call this for all of the arguments
 */
public void secondPass() {
    for (int i=0; i < arguments.size(); i++) { ((Argument) arguments.get(i)).secondPass(); }
}

/**
 * Sets the instruction pointer
 * @see MarkerSet
 */
public static class Jump extends MarkerSet {
    public Jump(Argument destination) { super(new BigDecimal(0), destination); }
}

```

```

/**
 * An instruction to move the specified marker to the specified
 * destination. This destination can be an Argument.Reference (a
 * symbolic reference link to another part of the assembled code) or
 * an Argument.Immediate (an absolute reference).
 */
public static class MarkerSet extends Instruction {
    public MarkerSet(BigDecimal markerNumber, Argument destination) {
        super(VirtualMachine.OP_BUMP_RELATIVE);

        // first argument (with 'zero' prefix)... the marker number
        arguments.add( new Assembler.Argument.Immediate(new Assembler.Data.Encoded(new BigDecimal(0))) );
        arguments.add( new Assembler.Argument.Immediate(new Assembler.Data.Encoded(markerNumber));

        // second argument (with 'zero' prefix)... what's it relative to? 0, the IP
        arguments.add( new Assembler.Argument.Immediate(new Assembler.Data.Encoded(new BigDecimal(0))) );
        arguments.add( new Assembler.Argument.Immediate(new Assembler.Data.Encoded(new BigDecimal(0))) );

        // third argument (with 'zero' prefix)... the a reference to the label
        arguments.add( new Assembler.Argument.Immediate(new Assembler.Data.Encoded(new BigDecimal(0))) );
        arguments.add( destination );
    }
}

public static abstract class Argument extends MemoryElement {
    /**
     * This very important field is automatically handled by ArgumentVector.add(),
     * which is how these things get attached to instructions.
     */
    public Instruction instruction;

    public void firstPass(Assembler assembler) { super.firstPass(assembler); }

    public static class Immediate extends Argument {
        public Data.Encoded data;
        public Immediate(Data.Encoded data) {
            super();
            this.data = data;
        }
        public Immediate(int value) { this(new Data.Encoded(new BigDecimal(value))); }
    }
}

```

```

    public String toString() { return "Immediate Arg: " + data.toString(); }
    public void firstPass(Assembler assembler) {
        super.firstPass(assembler);
        data.firstPass(assembler);
    }
}

/**
 * A Reference is an argument that points to a symbolic placeholder, within
 * the block of code that is currently being assembled.
 */
public abstract static class Reference extends Argument {
    public abstract Placeholder resolveReference();

    public void firstPass(Assembler assembler) {
        super.firstPass(assembler);
        assembler.firstPassMarker.writeDataNew(BigDecimal.valueOf(0), Assembler.addressPlaceholderSize, true);
    }

    public void secondPass() {

        Placeholder p = resolveReference();
        if ( (instruction.opcode != VirtualMachine.OP_BUMP_RELATIVE) &&
            (instruction.opcode != VirtualMachine.OP_BUMP_ABSOLUTE) )
            throw new RuntimeException("Can't reference label '" + p + "' from instruction '" +
VirtualMachine.codeToNameMap.get(new Integer(instruction.opcode)) + "'");

        // Our final value will depend on what type of instruction this is...
        BigInteger ref = (instruction.opcode == VirtualMachine.OP_BUMP_ABSOLUTE) ?
            p.memOffset :
            p.memOffset.subtract(instruction.memOffset);
        // Utilities.log("Referencing label '" + label + "' from instruction '" + Computer2.codeToNameMap.get(new
Integer(instruction.opcode)) + "' yields value " + ref);

        Marker m = new Marker(assembler.memoryMap, this.memOffset);
        m.writeDataExisting(new BigDecimal(ref));
    }

    /**
     * This is the one that's generated from parsed user assembler code. It is
     * a reference to a Placeholder.Label using the string
     */
    public static class Label extends Reference {
        public String label;
    }
}

```

