Component Based MMIX Simulator using Multiple Programming Paradigms

A dissertation submitted in partial fulfillment of the requirements for the MSc in Advanced Computing Technologies

by Stephen Edmans

Department of Computer Science and Information Systems

Birkbeck College, University of London

September 2015

Abstract

There are currently over 2,500[1] different programming languages, with more created every year. These programming languages can get grouped together in numerous different ways. This makes the decision of what language to use when starting a new project extremely difficult.

There are several ways in which we can reach this decision; choose the language that your team knows best; choose the language that makes the most sense to implement the critical part of your system; choose a simple general purpose language; choose a language that has got an active community. There is no acknowledged best approach to take.

Another approach would be to split your application up into separate components and using a different programming language for each component. This allows us choose the most appropriate programming language for each component.

The purpose of this project is to examine this approach. The application that we will create will be a simulator for an artificial machine language. The artificial machine language that we will use is called MMIX, it was developed by Donald Knuth as part of his seminal work The Art of Computer Programming[Knu11].

[1]From the language list[Kin]

Contents

List of Figures

Acknowledgments

Chapter 1

Introduction

As software systems get larger and more complex there is a need to handle this complexity. There is a prevailing design paradigm, which addresses these issues, that is to break these systems up into smaller components. This is a sentiment mentioned by Turner [Tur90] He calls these components “collections of modules”, these components will interact with each other to make the complete system.

When you have control over the development of more than one of these components it is a traditional approach to use a single programming paradigm for your components. There is, however, no reason that you cannot use different languages and paradigms for these for each components. The goal of this project is to create a relatively complex system that is made up of multiple components where each component uses the most appropriate programming paradigm for the relevant component.

The system that we have created in this project was inspired by Jeliot [oJ07], which is a tool that is used as an aid in the teaching of Java. The Jeliot system allows a user to give it a piece of Java source code and it will show the user what the underlying java virtual machine is doing when it runs the code.

In his seminal work The Art of Computer Programming [Knu11] Professor Donald Knuth designed an artificial machine language that he called MIX. In a later volume of his work Professor Knuth updated this machine architecture, which he calls MMIX. He later detailed this new version of the architecture in a fascicle [Knu]. This project will create a system that take MMIX assembly code and shows the user, graphically, what the simulated machine is doing. It should be noted that all definitions of an MMIX computer and the assembly language used to program it come directly from either one of these sources.

Chapter 2

Assembler

2.1 Introduction

The first component that we developed takes a text file containing the MMIX assembly language code and translated it into a binary representation of the code. This component it typically called a *compiler*, to quote [ALSU06]

> A compiler is a program that can read a program in one language – the *source* language – and translate it into an equivalent program in another language – the *target* language.

A compiler operates as a series of phases, each of which transforms one representation of the source program into another. A typical decomposition of a compiler into phases, taken from [ALSU06] is shown in Figure 2.1.

A number if these phases are used to convert a higher level language down into a specific machine language. In this project we already start with a machine language, which means that we do not need these phases. A program that takes an assembly language file and translates it into machine language is typically called an *assembler*.

The four phases that we need for our project are Lexical Analysis, Syntax Analysis, Semantic Analysis and Code Generation. There are two of these phases, syntax analysis and semantic analysis, which are usually combined into a single phase, which is typically called a *Parser*.

The first thing that we need to do is decide which programming language is the most appropriate for this component. The component takes a fixed input and always produces the same output. The component does not contain any user interaction and it does not need a user interface. These requirements led us to choose a functional language for this component. The language we chose was Haskell.

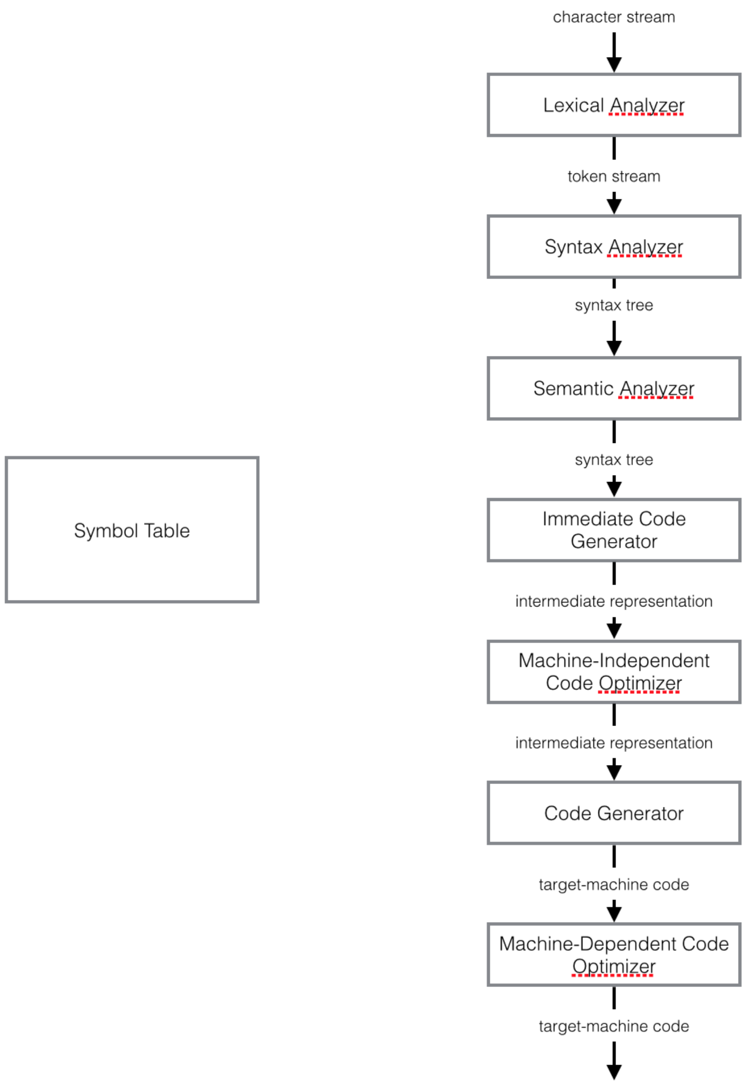


Figure 2.1: Phases of a compiler

We describe how each of these phases are implemented in Haskell in the next few sections.

2.2 Lexer

The initial phase of compilation, lexical analysis, takes a stream of characters and converts them into tokens. Lexical analysis is a well know problem and there are many tools that have been created to make this task simpler. There are several lexers that already exist for Haskell and we have chosen to use a lexer called Alex[Mar].

The first thing that you need to do when creating a lexer with Alex is to determine what set of tokens you will need to create. There are several parts to the MMIX assembly language (Mmixal). The first part is the way you define what operation should be performed by the computer at a specific point in the program. This is acheived with a machine language instrustion, which is typically called an Opcode. Each instruction needs some additional parameters which informs the computer on what the instruction should operate on, these parameters are typically called operands. There are two distinct types of Opcodes in Mmixal. The first type does not vary based on what operands are used with it. The second type does vary based on the operands, and the binary representation of the Opcode is different for the different set of operands. At the end of the assembly process these instructions will be converted into a binary representation that will be stored in the memory of an MMIX computer.

The next type of instruction is used by the assembler specify either the initial state of the computer or assign internal details used as part of the assembly process. The instructions are called, in fascile 1[Knu], pseudo instructions. These pseudo instructions do not necessarily result in anything being stored in memory.

The mmixal language also defines labels, registers, expressions and a few other things.

For this project we are using three basic groupings of tokens. The first group contains a single token, this token is for opcodes that do not vary based on their operands, we have called this token *TOpCodeSimple*. The second group of tokens also contains a single token, this is the token for opcodes that do vary based on their operands, we have called this token *TOpCode*. The third group contains all of the other tokens, see the code listing in Appendix A.1.1 for a complete list of these.

When you have determined what tokens are allowed you need to describe what sequences of characters should be converted to the individual tokens. The way that you describe the tokens in Alex is to create a list of regular expressions, for each regular expression you specify which token should be created if this sequence is found.

The Alex lexer tool allows us to insert code at specific points in the process. The functions that we are using allows us to simplify the process of creating tokens.

The Alex lexer requires us to store all of these definitions in a file with an extension of *x*. When all of the definitions have been completed you run the definition file through the Alex lexer tool and it generates a haskell source file that will perform the lexical analysis for you.

2.3 Parser

Once we have got a stream of tokens from the lexer, we need to perform syntax analysis and semantic analysis to make sure that the supplied program is both syntactically and semantically correct. Both of these steps are usually performed at the same time with a component called a *Parser*. Parsing, like lexical analysis, is a well know problem and there are many tools that have been created to make this task simpler. These tools are generally called parser generators as they take a definition file and produce the actual parser. There are several parser generators that already exist for Haskell and we have chosen to use a parser generator called Happy[GM].

The requirement for a parser is to take a stream of tokens, make sure that the stream is syntactically and semantically correct, and then output an intermediate representation of the code that can be used to generate the final binary representation.

The first thing that we did was to design our intermediate representation, there are four different type of code lines in mmixal, as shown below.

```
BUF   OCTA   0            %Labelled Pseudo Instruction Line
      LOC    #100         %Plain Pseudo Instruction Line
Main  JMP    9F           %Labelled Opcode Line
      STWU   n,ptop,jj    %Plain Opcode Line
```

The way that the Happy parser generator works is that we need to create a definition file that specifies all of the syntactically and semantically correct types of statements in our language. We specify the valid statements using a context free grammar. A context free grammar contains two basic parts, an identifier and list of tokens, or identifiers, that the identifier represents. A cut down version of the parser definition file we have used can be found in Figure 2.2 and a full description of the intermediate representation can be found in Appendix B. The complete definition file can be found in Appendix A.1.2.

The Happy parser generator requires us to store all of these definitions in a file with an extension of *y*. When all of the definitions have been completed you run the definition file through the Happy parser generator tool and it generates a Haskell source file that will perform the syntactic and semantic analysis for you.

```
Program         : AssignmentLines { reverse $1 }

AssignmentLines : {- empty -}       {[]}
                | AssignmentLines AssignmentLine { $2 : $1 }

AssignmentLine :: {Line}
AssingmentLine : OP_CODE OperatorList { defaultPlainOpCodeLine }
               | Identifier PI { defaultLabelledPILine }
               | Identifier OP_CODE OperatorList { defaultLabelledOpCodeLine }
               | PI { defaultPlainPILine }
               | OP_CODE_SIMPLE OperatorList { defaultPlainOpCodeLine }
               | Identifier OP_CODE_SIMPLE OperatorList { defaultLabelledOpCodeLine }
```

Figure 2.2: Sample Context Free Grammar

2.4 Code Generation

Now that we have got our program converted into an intermediate representation, in our case a list of *Lines*, we need to convert this into a binary representation. The mmixal language contains a set of features called *Local Labels* and processing these is the first step we perform when generating the code.

2.4.1 Local Labels

Local labels help compilers and programmers use names temporarily. They create symbols which are guaranteed to be unique over the entire scope of the input source code and which can be referred to by a simple notation. There are two parts to consider when implementing the local labels, the first is the location of the label itself, and the other is references to these labels used elsewhere in the program.

The location of a local label is specified by placing a single digit followed directly by an H, i.e. *0H* as a label at the start of a line. It should be noted that an individual local label can be specified many times in a single program, when they are referenced the closest label in the required direction is used. The way that we handle this is that we rename all of the local labels to system generated labels, these labels are actually illegal for user so we know that we are not creating duplicates. The format that we use is *??LS#H** where # is the local label number and * is a counter. We keep note of a separate counter for each of the possible local label numbers.

The local labels are referenced as either forward or backward references, i.e. *2B* specifies that we should look backwards in the code until we find a *2H* local label and use that local. To achieve this we start off by creating two separate maps, one for forward references and one for backward references. Initially the forward references point to the first possible local label for the mapped digit, the backward references do not contain a reference and any use of it is a semantic error in the program. When we have these maps we iterate through the program replacing any reference to a local label with the appropriate system generated label from the specific map. We then check to see if the line actually contains a local label specification, if it does we update both the forward and backward maps with the appropriate changes.

At the end of this process we have converted all local labels into system generated labels that can be handled as if they were ordinary user specified labels.

2.4.2 Symbol Table

As we can see in Figure 2.1 one of the data structures that we need to create as part of the code generation process is called a *Symbol Table.* This is simply a map that is used to record what labels have been specified and where in the program they actually point to. To create this we simply iterate through each of the lines of the program and if they contain a label we firstly check to see if it is already present and if it does not we add the label with the current location to the symbol table.

The symbol table is used extensively in the later steps of the code generation.

2.4.3 Automatically Assigned Registers

An MMix computer, by definition, contains 256 general purpose registers. The programmer can either specify which register to use directly or they can get the assembler to assign one automatically for them. A new general purpose register is allocated every time the assembler comes across a *GREG* pseudo instruction. The first register that is automatically assigned is $FE (254). Every, subsequent, automatically generated register uses the next lowest register. We have achieved this by iterating over the lines, sending along the value of the next assignable register. If the line is a GREG instruction then we change the command to one that contains the assigned register, and we then decrement the next assignable register before passing it on the the next line.

2.4.4 Handling Operands

Each opcode instructions can be supplied one, two or three operands to specify exactly what we expect to happen. The majority of opcodes can either be supplied with three registers, or it can be supplied with two registers and an immediate value. The registers could, of course, be replaced by labels which represent registers. If the line specifies three registers then the plain opcode is used when we generate the code. In the other case then when we generate the code we increment the opcode by one to let the computer know not to look for this register.

If the opcode is for a branching instruction then it will be supplied with a register and an address. For these instructions we need to determine the number of instructions between the current memory location and the memory location of the required address. We need the number of instructions, not just the difference in memory locations. This is calculated by determining the difference between the memory addresses and dividing this value by four. This address could be either ahead of, or before, the current location. If it is ahead of the current location then we use the plain opcode when generating the code. If the address is before the current location the we generate the code we increment the opcode by one.

2.4.5 Assembler Directives

TODO

2.4.6 Generating the Output

The final stage of the assembler is the actual outputting of the binary representation of the program. The way that we have achieved this is a two stage process. In the first stage we convert the intermediate representation of the code into a new representation of the code that makes creating the output file simpler.

```
CodeLine {cl_address = 256, cl_size = 4, cl_code = "\240\NUL\NUL\ETB"}
```

This representation includes the start address for this line of code, the size of the code and the binary representation of that line of code.

The final stage is to output these code lines to a file. The structure of the file contains two separate parts, the first part contains the data the needs to be placed in memory, and the second part contains the initial values that the used registers need to be set to.

To create the first part we group the code lines together into contiguous blocks. The first four bytes of this section contains the number of blocks that we have got, we then include the details for each block. The first four bytes of each block contains the start address of the block, next next four bytes of the block contains the size of the block, after this we include the actual code for the block.

The first four bytes of the second part contains the number of registers we are defining. We then include the details for each register. The first byte of the register is register number, the next eight bytes are the initial value of that register.

2.5 Executable

TODO —– HOW WE RUN THE ASSEMBLER, WHAT THE PARAMETERS ARE & WHAT THE OUTPUT IS

2.6 Component Testing

When it comes to testing this component we tested it these levels.

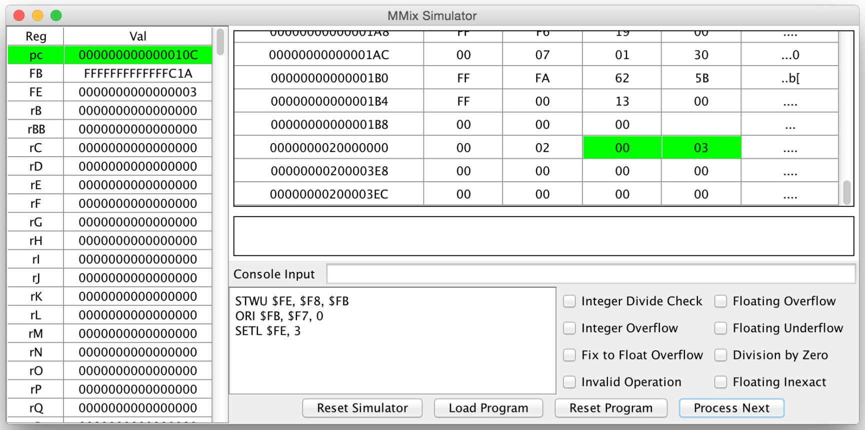
- We tested the lexer on its own.
- We tested the lexer and parser together.
- We tested the Local Label generation on its own.
- We tested the Automatically Assigned Registers on their own.
- We tested the Code Generation on its own.
- We tested the component as a whole.

There are several sample programs that are written in mmixal, we used several of these when testing the assembler. The main mmixal program we used to test this component is one that determines the first 500 prime numbers. A fuller description of this program can be found at Chapter 5.2.1.

Chapter 3

Graphical User Interface

3.1 Introduction



Reg	Val
pc	000000000000010C
FB	FFFFFFFFFFFFFC1A
FE	0000000000000003
rB	0000000000000000
rBB	0000000000000000
rC	0000000000000000
rD	0000000000000000
rE	0000000000000000
rF	0000000000000000
rG	0000000000000000
rH	0000000000000000
rI	0000000000000000
rJ	0000000000000000
rK	0000000000000000
rL	0000000000000000
rM	0000000000000000
rN	0000000000000000
rO	0000000000000000
rP	0000000000000000
rQ	0000000000000000

00000000000001A8	FF	F6	19	00	....
00000000000001AC	00	07	01	30	...0
00000000000001B0	FF	FA	62	5B	..b[
00000000000001B4	FF	00	13	00	....
00000000000001B8	00	00	00		...
0000000020000000	00	02	00	03	....
00000000200003E8	00	00	00	00	....
00000000200003EC	00	00	00	00	....

```
STWU $FE, $F8, $FB
ORI $FB, $F7, 0
SETL $FE, 3
```

Figure 3.1: GUI Sample Screen shot

3.2 User Interface Design

3.2.1 Console Panel

3.2.2 Controls Panel

3.2.3 Main State Panel

3.2.4 Memory Panel

3.2.5 Registers Panel

3.3 Asynchronous User Interface Programming with Actors

3.4 Communication

3.5 Component Testing

Chapter 4

Virtual Machine

4.1 Introduction

All definition of an MMIX computer come directly from either [Knu11] or [Knu]

Architecture of a computer CPU, ALU, Memory, Secondary Storage, IO Devices

The virtual machine we are developing will only consist of a CPU and memory.

The way that memory is organized can be considered a hierarchy, to quote Aho et al[ALSU06]

> A memory hierarchy consists of several levels of storage with different speeds and sizes, with the levels closest to the processor being the fastest but smallest... Memory hierarchies are found in all machines. A processor usually has a small number of registers consisting of hundreds of bytes, several levels of caches containing kilobytes to megabytes, physical memory containing megabytes to gigabytes, and finally secondary storage that contains gigabytes and beyond.

For this project we will only be considering the physical memory and the registers.

4.2 Memory

Wyde

$M_2[0] = M_2[1] = M[0]M[1]$

Tetra

$M_4[4k] = M_4[4k+1] = ... = M_4[4k+3] = M[4k]M[4k+1]...M[4k+3]$

Octa

$M_8[8k] = M_8[8k+1] = ... = M_8[8k+7] = M[8k]M[8k+1]...M[8k+7]$

4.3 Registers

An MMIX computer contains two distinct types of registers, 256 general purpose registers and 32 special purpose registers. A complete list of the special registers can be found in Figure 4.1

rA Arithmetic Status Register

least significant byte contains eight event bits. DVWIOUZX

Register	**Description**
D	Integer Divide Check
V	Integer Overflow
W	Float-to-Fix Overflow
I	Invalid Operation
O	Floating Overflow
U	Floating Underflow
Z	Division by Zero
X	Floating Inexact

The next least significant byte contains eight “enable” bits with the same name DVWIOUZX and the same meanings.

When an exceptional condition occurs, there are two cases: If the corresponding enable bit is 0, the corresponding event bit is set to 1; but if the corresponding enable bit is 1, MMIX interrupts its current instruction stream and execute a special “exception handler”. Thus, the event bits record exceptions that have not been “tripped”.

This leaves six high order bytes. At present, only two of those 48 bits are defined. The two bits corresponding to 2^{17} and 2^{16} in rA specify a rounding mode, as follows: -

00	Round to the nearest
01	Round off
10	Round up
11	Round down

Identifier	**Description**
rA	Arithmetic Status Register
rB	Bootstrap Register
rC	Continuation Register
rD	Dividend Register
rE	Epsilon Register
rF	Failure Location Register
rG	Global Threshold Register
rH	Himult Register
rI	Interval Counter
rJ	Return-Jump Register
rK	Interrupt Mask Register
rL	Local Threshold Register
rM	Multiplex Mask Register
rN	Serial Number
rO	Register Stack Offset
rP	Prediction Register
rQ	Interrupt Request Register
rR	Remainder Register
rS	Register Stack Pointer
rT	Trap Address Register
rU	Usage Counter
rV	Virtual Translation Register
rW	Where Interrupted Register
rX	Execution Register
rY	Y Operand
rZ	Z Operand
rBB	Bootstrap Register
rTT	Dynamic Trap Address Register
rWW	Where Interrupted Register
rXX	Execution Register
rYY	Y Operand
rZZ	Z Operand

Figure 4.1: Special Registers

4.4 Central Processing Unit

4.5 Calling the Operating System

4.6 Communication

4.7 Component Testing

Chapter 5

Simulator Application

5.1 Introduction

5.2 Integration Testing

5.2.1 Generate Prime Numbers Sample Application

Conclusion

References

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[GM] Andy Gill and Simon Marlow. Happy: The parser generator for haskell. <`https://www.haskell.org/happy/` >[Access 7 September 2015].

[Inc] Typesafe Inc. Akka toolkit. <`http://akka.io/` >[Access 24 August 2015].

[Kin] Bill Kinnersley. The language list. <`http://people.ku.edu/~nkinners/LangList/Extras/langlist.htm` >[Access 7 September 2015].

[Knu] D.E. Knuth. The art of computer programming fascicle 1 mmix [e-book]. Stanford University: Addison Wesley Available through: Stanford University <`http://www-cs-faculty.stanford.edu/~uno/fasc1.ps.gz`>[Access 7 April 2013].

[Knu90] D.E. Knuth. *MMIXware A RISC Computer for the Third Millennium*. Springer, 1990.

[Knu11] D.E. Knuth. *The Art of Computer Programming*, volume 1-4a. 1st ed. Addison Wesley, 2011.

[Mar] Simon Marlow. Alex: A lexical analyser generator for haskell. <`https://www.haskell.org/alex/` >[Access 7 September 2015].

[oJ07] University of Joensuu. Jeliot 3. Available at: <`http://cs.joensuu.fi/jeliot/`, 2007.

[Ruc12] Martin Ruckert. Mmix quick reference card. <`http://mmix.cs.hm.edu/doc/mmix-refcard-a4.pdf` >[Access 24 August 2015], 2012.

[Tur90] D. Turner. Research topics in functional programming, addison-wesley. Available through <`http://www.cs.utexas.edu/~shmat/courses/cs345/whyfp.pdf`, 1990.

Appendix A

Source Code

A.1 Assembler

A.1.1 Lexer

```
{
module MMix_Lexer where

import Data.Char (chr)
import Numeric (readDec)
import Numeric (readHex)

import Debug.Trace
}

%wrapper "monadUserState"

$digit    = 0-9             -- digits
$alpha    = [a-zA-Z]        -- alphabetic characters
$hexdigit = [0-9a-eA-E]     -- hexadecimal digits

tokens :-

<0>$white+                                      ;
<0>[Ii][Ss]                          { mkT TIS }
<0>[Gg][Rr][Ee][Gg]                  { mkT TGREG }
<0>[Ll][Oo][Cc]                      { mkT TLOC }
<0>[Bb][Yy][Tt][Ee]                  { mkT TByte }
<0>[Ww][Yy][Dd][Ee]                  { mkT TWyde }
<0>[Tt][Ee][Tt][Rr][Aa]              { mkT TTetra }
<0>[Oo][Cc][Tt][Aa]                  { mkT TOcta }
<0>[Ss][Ee][Tt]                      { mkT TSet }
<0>($digit)H                         { mkLocalLabel }
<0>($digit)F                         { mkLocalForwardOperand }
<0>($digit)B                         { mkLocalBackwardOperand }
<0>Data_Segment                      { mkT TDataSegment }
<0>\@                                { mkT TAtSign }
<0>\+                                { mkT TPlus }
<0>\-                                { mkT TMinus }
<0>\*                                { mkT TMult }
<0>\/                                { mkT TDivide }
<0>\(                                { mkT TOpenParen }
<0>\)                                { mkT TCloseParen }
<0>\#$hexdigit+                      { mkHex }
<0>\$$digit+                         { mkRegister }
<0>[Tt][Rr][Aa][Pp]                  { mkT $ TOpCodeSimple 0x00 }
<0>[Ff][Cc][Mm][Pp]                  { mkT $ TOpCodeSimple 0x01 }
<0>[Ff][Uu][Nn]                      { mkT $ TOpCodeSimple 0x02 }
<0>[Ff][Ee][Qq][Ll]                  { mkT $ TOpCodeSimple 0x03 }
<0>[Ff][Aa][Dd][Dd]                  { mkT $ TOpCodeSimple 0x04 }
<0>[Ff][Ii][Xx]                      { mkT $ TOpCodeSimple 0x05 }
<0>[Ff][Ss][Uu][Bb]                  { mkT $ TOpCodeSimple 0x06 }
<0>[Ff][Ii][Xx][Uu]                  { mkT $ TOpCodeSimple 0x07 }
```

```
<0>[Ff][Ll][Oo][Tt]                     { mkT $ TOpCode 0x08 }
<0>[Ff][Ll][Oo][Tt][Uu]                 { mkT $ TOpCode 0x0A }
<0>[Ss][Ff][Ll][Oo][Tt]                 { mkT $ TOpCode 0x0C }
<0>[Ss][Ff][Ll][Oo][Tt][Uu]             { mkT $ TOpCode 0x0E }
<0>[Ff][Mm][Uu][Ll]                     { mkT $ TOpCodeSimple 0x10 }
<0>[Ff][Cc][Mm][Pp][Ee]                 { mkT $ TOpCodeSimple 0x11 }
<0>[Ff][Uu][Nn][Ee]                     { mkT $ TOpCodeSimple 0x12 }
<0>[Ff][Ee][Qq][Ll][Ee]                 { mkT $ TOpCodeSimple 0x13 }
<0>[Ff][Dd][Ii][Vv]                     { mkT $ TOpCodeSimple 0x14 }
<0>[Ff][Ss][Qq][Rr][Tt]                 { mkT $ TOpCodeSimple 0x15 }
<0>[Ff][Rr][Ee][Mm]                     { mkT $ TOpCodeSimple 0x16 }
<0>[Ff][Ii][Nn][Tt]                     { mkT $ TOpCodeSimple 0x17 }
<0>[Mm][Uu][Ll]                         { mkT $ TOpCode 0x18 }
<0>[Mm][Uu][Ll][Uu]                     { mkT $ TOpCode 0x1A }
<0>[Dd][Ii][Vv]                         { mkT $ TOpCode 0x1C }
<0>[Dd][Ii][Vv][Uu]                     { mkT $ TOpCode 0x1E }
<0>[Aa][Dd][Dd]                         { mkT $ TOpCode 0x20 }
<0>[Ll][Dd][Aa]                         { mkT $ TOpCode 0x22 }
<0>[Aa][Dd][Dd][Uu]                     { mkT $ TOpCode 0x22 }
<0>[Ss][Uu][Bb]                         { mkT $ TOpCode 0x24 }
<0>[Ss][Uu][Bb][Uu]                     { mkT $ TOpCode 0x26 }
<0>2[Aa][Dd][Dd][Uu]                    { mkT $ TOpCode 0x28 }
<0>4[Aa][Dd][Dd][Uu]                    { mkT $ TOpCode 0x2A }
<0>8[Aa][Dd][Dd][Uu]                    { mkT $ TOpCode 0x2C }
<0>16[Aa][Dd][Dd][Uu]                   { mkT $ TOpCode 0x2E }
<0>[Cc][Mm][Pp]                         { mkT $ TOpCode 0x30 }
<0>[Cc][Mm][Pp][Uu]                     { mkT $ TOpCode 0x32 }
<0>[Nn][Ee][Gg]                         { mkT $ TOpCode 0x34 }
<0>[Nn][Ee][Gg][Uu]                     { mkT $ TOpCode 0x36 }
<0>[Ss][Ll]                             { mkT $ TOpCode 0x38 }
<0>[Ss][Ll][Uu]                         { mkT $ TOpCode 0x3A }
<0>[Ss][Rr]                             { mkT $ TOpCode 0x3C }
<0>[Ss][Rr][Uu]                         { mkT $ TOpCode 0x3E }
<0>[Bb][Nn]                             { mkT $ TOpCode 0x40 }
<0>[Bb][Zz]                             { mkT $ TOpCode 0x42 }
<0>[Bb][Pp]                             { mkT $ TOpCode 0x44 }
<0>[Bb][Oo][Dd]                         { mkT $ TOpCode 0x46 }
<0>[Bb][Nn][Nn]                         { mkT $ TOpCode 0x48 }
<0>[Bb][Nn][Zz]                         { mkT $ TOpCode 0x4A }
<0>[Bb][Nn][Pp]                         { mkT $ TOpCode 0x4C }
<0>[Bb][Ee][Vv]                         { mkT $ TOpCode 0x4E }
<0>[Pp][Bb][Nn]                         { mkT $ TOpCode 0x50 }
<0>[Pp][Bb][Zz]                         { mkT $ TOpCode 0x52 }
<0>[Pp][Bb][Pp]                         { mkT $ TOpCode 0x54 }
<0>[Pp][Bb][Oo][Dd]                     { mkT $ TOpCode 0x56 }
<0>[Pp][Bb][Nn][Nn]                     { mkT $ TOpCode 0x58 }
<0>[Pp][Bb][Nn][Zz]                     { mkT $ TOpCode 0x5A }
<0>[Pp][Bb][Nn][Pp]                     { mkT $ TOpCode 0x5C }
<0>[Pp][Bb][Ee][Vv]                     { mkT $ TOpCode 0x5E }
<0>[Cc][Ss][Nn]                         { mkT $ TOpCode 0x60 }
<0>[Cc][Ss][Zz]                         { mkT $ TOpCode 0x62 }
<0>[Cc][Ss][Pp]                         { mkT $ TOpCode 0x64 }
<0>[Cc][Ss][Oo][Dd]                     { mkT $ TOpCode 0x66 }
<0>[Cc][Ss][Nn][Nn]                     { mkT $ TOpCode 0x68 }
<0>[Cc][Ss][Nn][Zz]                     { mkT $ TOpCode 0x6A }
<0>[Cc][Ss][Nn][Pp]                     { mkT $ TOpCode 0x6C }
<0>[Cc][Ss][Ee][Vv]                     { mkT $ TOpCode 0x6E }
<0>[Zz][Ss][Nn]                         { mkT $ TOpCode 0x70 }
<0>[Zz][Ss][Zz]                         { mkT $ TOpCode 0x72 }
<0>[Zz][Ss][Pp]                         { mkT $ TOpCode 0x74 }
<0>[Zz][Ss][Oo][Dd]                     { mkT $ TOpCode 0x76 }
<0>[Zz][Ss][Nn][Nn]                     { mkT $ TOpCode 0x78 }
<0>[Zz][Ss][Nn][Zz]                     { mkT $ TOpCode 0x7A }
<0>[Zz][Ss][Nn][Pp]                     { mkT $ TOpCode 0x7C }
<0>[Zz][Ss][Ee][Vv]                     { mkT $ TOpCode 0x7E }
<0>[Ll][Dd][Bb]                         { mkT $ TOpCode 0x80 }
<0>[Ll][Dd][Bb][Uu]                     { mkT $ TOpCode 0x82 }
<0>[Ll][Dd][Ww]                         { mkT $ TOpCode 0x84 }
<0>[Ll][Dd][Ww][Uu]                     { mkT $ TOpCode 0x86 }
<0>[Ll][Dd][Tt]                         { mkT $ TOpCode 0x88 }
<0>[Ll][Dd][Tt][Uu]                     { mkT $ TOpCode 0x8A }
<0>[Ll][Dd][Oo]                         { mkT $ TOpCode 0x8C }
<0>[Ll][Dd][Oo][Uu]                     { mkT $ TOpCode 0x8E }
<0>[Ll][Dd][Ss][Ff]                     { mkT $ TOpCode 0x90 }
<0>[Ll][Dd][Hh][Tt]                     { mkT $ TOpCode 0x92 }
<0>[Cc][Ss][Ww][Aa][Pp]                 { mkT $ TOpCode 0x94 }
<0>[Ll][Dd][Uu][Nn][Cc]                 { mkT $ TOpCode 0x96 }
<0>[Ll][Dd][Vv][Tt][Ss]                 { mkT $ TOpCode 0x98 }
<0>[Pp][Rr][Ee][Ll][Dd]                 { mkT $ TOpCode 0x9A }
<0>[Pp][Rr][Ee][Gg][Oo]                 { mkT $ TOpCode 0x9C }
<0>[Gg][Oo]                             { mkT $ TOpCode 0x9E }
<0>[Ss][Tt][Bb]                         { mkT $ TOpCode 0xA0 }
<0>[Ss][Tt][Bb][Uu]                     { mkT $ TOpCode 0xA2 }
<0>[Ss][Tt][Ww]                         { mkT $ TOpCode 0xA4 }
```

```
<0>[Ss][Tt][Ww][Uu]                     { mkT $ TOpCode 0xA6 }
<0>[Ss][Tt][Tt]                         { mkT $ TOpCode 0xA8 }
<0>[Ss][Tt][Tt][Uu]                     { mkT $ TOpCode 0xAA }
<0>[Ss][Tt][Oo]                         { mkT $ TOpCode 0xAC }
<0>[Ss][Tt][Oo][Uu]                     { mkT $ TOpCode 0xAE }
<0>[Ss][Tt][Ss][Ff]                     { mkT $ TOpCode 0xB0 }
<0>[Ss][Tt][Hh][Tt]                     { mkT $ TOpCode 0xB2 }
<0>[Ss][Tt][Cc][Oo]                     { mkT $ TOpCode 0xB4 }
<0>[Ss][Tt][Uu][Nn][Cc]                 { mkT $ TOpCode 0xB6 }
<0>[Ss][Yy][Nn][Cc][Dd]                 { mkT $ TOpCode 0xB8 }
<0>[Pp][Rr][Ee][Ss][Tt]                 { mkT $ TOpCode 0xBA }
<0>[Ss][Yy][Nn][Cc][Ii][Dd]             { mkT $ TOpCode 0xBC }
<0>[Pp][Uu][Ss][Hh][Gg][Oo]             { mkT $ TOpCode 0xBE }
<0>[Oo][Rr]                             { mkT $ TOpCode 0xC0 }
<0>[Oo][Rr][Nn]                         { mkT $ TOpCode 0xC2 }
<0>[Nn][Oo][Rr]                         { mkT $ TOpCode 0xC4 }
<0>[Xx][Oo][Rr]                         { mkT $ TOpCode 0xC6 }
<0>[Aa][Nn][Dd]                         { mkT $ TOpCode 0xC8 }
<0>[Aa][Nn][Dd][Nn]                     { mkT $ TOpCode 0xCA }
<0>[Nn][Aa][Nn][Dd]                     { mkT $ TOpCode 0xCC }
<0>[Nn][Xx][Oo][Rr]                     { mkT $ TOpCode 0xCE }
<0>[Bb][Dd][Ii][Ff]                     { mkT $ TOpCode 0xD0 }
<0>[Ww][Dd][Ii][Ff]                     { mkT $ TOpCode 0xD2 }
<0>[Tt][Dd][Ii][Ff]                     { mkT $ TOpCode 0xD4 }
<0>[Oo][Dd][Ii][Ff]                     { mkT $ TOpCode 0xD6 }
<0>[Mm][Uu][Xx]                         { mkT $ TOpCode 0xD8 }
<0>[Ss][Aa][Dd][Dd]                     { mkT $ TOpCode 0xDA }
<0>[Mm][Oo][Rr]                         { mkT $ TOpCode 0xDC }
<0>[Mm][Xx][Oo][Rr]                     { mkT $ TOpCode 0xDE }
<0>[Ss][Ee][Tt][Hh]                     { mkT $ TOpCodeSimple 0xE0 }
<0>[Ss][Ee][Tt][Mm][Hh]                 { mkT $ TOpCodeSimple 0xE1 }
<0>[Ss][Ee][Tt][Mm][Ll]                 { mkT $ TOpCodeSimple 0xE2 }
<0>[Ss][Ee][Tt][Ll]                     { mkT $ TOpCodeSimple 0xE3 }
<0>[Ii][Nn][Cc][Hh]                     { mkT $ TOpCodeSimple 0xE4 }
<0>[Ii][Nn][Cc][Mm][Hh]                 { mkT $ TOpCodeSimple 0xE5 }
<0>[Ii][Nn][Cc][Mm][Ll]                 { mkT $ TOpCodeSimple 0xE6 }
<0>[Ii][Nn][Cc][Ll]                     { mkT $ TOpCodeSimple 0xE7 }
<0>[Oo][Rr][Hh]                         { mkT $ TOpCodeSimple 0xE8 }
<0>[Oo][Rr][Mm][Hh]                     { mkT $ TOpCodeSimple 0xE9 }
<0>[Oo][Rr][Mm][Ll]                     { mkT $ TOpCodeSimple 0xEA }
<0>[Oo][Rr][Ll]                         { mkT $ TOpCodeSimple 0xEB }
<0>[Aa][Nn][Dd][Nn][Hh]                 { mkT $ TOpCodeSimple 0xEC }
<0>[Aa][Nn][Dd][Nn][Mm][Hh]             { mkT $ TOpCodeSimple 0xED }
<0>[Aa][Nn][Dd][Nn][Mm][Ll]             { mkT $ TOpCodeSimple 0xEE }
<0>[Aa][Nn][Dd][Nn][Ll]                 { mkT $ TOpCodeSimple 0xEF }
<0>[Jj][Mm][Pp]                         { mkT $ TOpCode 0xF0 }
<0>[Pp][Uu][Ss][Hh][Jj]                 { mkT $ TOpCodeSimple 0xF2 }
<0>[Gg][Ee][Tt][Aa]                     { mkT $ TOpCodeSimple 0xF4 }
<0>[Pp][Uu][Tt]                         { mkT $ TOpCode 0xF6 }
<0>[Pp][Oo][Pp]                         { mkT $ TOpCodeSimple 0xF8 }
<0>[Rr][Ee][Ss][Uu][Mm][Ee]             { mkT $ TOpCodeSimple 0xF9 }
<0>[Ss][Aa][Vv][Ee]                     { mkT $ TOpCodeSimple 0xFA }
<0>[Ss][Yy][Nn][Cc]                     { mkT $ TOpCodeSimple 0xFC }
<0>[Ss][Ww][Yy][Mm]                     { mkT $ TOpCodeSimple 0xFD }
<0>[Gg][Ee][Tt]                         { mkT $ TOpCodeSimple 0xFE }
<0>[Tt][Rr][Ii][Pp]                     { mkT $ TOpCodeSimple 0xFF }
<0>Fputs                                { mkT TFputS }
<0>StdOut                               { mkT TStdOut }
<0>Halt                                 { mkT THalt }
<0>$digit+                              { mkInteger }
<0>\,                                   { mkT TComma }
<0>\"                                   { startString `andBegin` string }
<string>\\\"                            { addCharToString '\"' }
<string>\\\\                            { addCharToString '\\' }
<string>\"                              { endString `andBegin` state_initial }
<string>.                               { addCurrentToString }
<0>\'.\'                                { mkChar }
<0>$alpha [$alpha $digit \_ \']*        { mkIdentifier }
<0>.                                    { mkError }

{
data Token = LEOF
            | TIdentifier { tid_name :: String }
            | TError { terr_text :: String }
            | TInteger { tint_value :: Int }
            | THexLiteral { thex_value :: Int }
            | TRegister { treg_value :: Int }
            | TStringLiteral { tsl_text :: String }
            | TLocalForwardOperand { tlfo :: Int }
            | TLocalBackwardOperand { tlbo :: Int }
            | TLocalLabel { tll :: Int }
            | TIS
            | TByte
            | TGREG
            | TLOC
            | TWyde
            | TTetra
            | TOcta
            | TSet
            | TFputS
            | TStdOut
            | THalt
            | TOpCode { toc_value :: Int }
            | TOpCodeSimple { soc_value :: Int }
            | TDataSegment
            | TAtSign
            | TComma
            | TPlus
            | TMult
            | TMinus
            | TDivide
            | TOpenParen
            | TCloseParen
            | TByteLiteral Char
            | W String
            | CommentStart
            | CommentEnd
            | CommentBody String
            deriving (Show,Eq)

state_initial :: Int
state_initial = 0

data AlexUserState = AlexUserState
                   {
                     lexerStringState    :: Bool
                     , lexerStringValue  :: String
                   }

alexInitUserState :: AlexUserState
alexInitUserState = AlexUserState
                   {
                     lexerStringState    = False
                     , lexerStringValue  = ""
                   }

setLexerStringState :: Bool -> Alex ()
setLexerStringState ss = Alex $ \s -> Right (s{alex_ust=(alex_ust s){lexerStringState=ss}}, ()
     )

setLexerStringValue :: String -> Alex ()
setLexerStringValue ss = Alex $ \s -> Right (s{alex_ust=(alex_ust s){lexerStringValue=ss}}, ()
     )

getLexerStringValue :: Alex String
getLexerStringValue = Alex $ \s@AlexState{alex_ust=ust} -> Right (s, lexerStringValue ust)

addCharToLexerStringValue :: Char -> Alex ()
addCharToLexerStringValue c = Alex $ \s -> Right (s{alex_ust=(alex_ust s){lexerStringValue=c:
     lexerStringValue (alex_ust s)}}, ())

addCurrentToString :: (t, t1, t2, String) -> Int -> Alex Token
addCurrentToString input@(_, _, _, remaining) length =
    addCharToString c input length
    where
        c = if (length == 1)
            then head remaining
            else error "Invalid call to addCurrentString"

addCharToString :: Char -> t -> t1 -> Alex Token
addCharToString c _     _   =
    do
        addCharToLexerStringValue c
        alexMonadScan

word a@(_,c,_,inp) len = mkT (W (take len inp)) a len

extractValue :: Num a => [(a, String)] -> a
extractValue ((value, ""):_) = value
extractValue _ = error "Invalid Hex Value"

mkHex :: Monad m => (t, t1, t2, String) -> Int -> m Token
mkHex input length =
    mkT (THexLiteral decValue) input length
    where
        str = getStr input length
        hexPart = tail str
        decValue = extractValue $ readHex hexPart
```

```
mkLocalLabel input length = mkT (TLocalLabel val) input length
    where val = read (getStr input 1) :: Int

mkChar input length = mkT (TByteLiteral val) input length
    where val = (getStr input 2) !! 1 :: Char

mkLocalForwardOperand input length = mkT (TLocalForwardOperand val) input length
    where val = read (getStr input 1) :: Int

mkLocalBackwardOperand input length = mkT (TLocalBackwardOperand val) input length
    where val = read (getStr input 1) :: Int

mkInteger input length
    | val >= 0 && val < 256 = mkT (TByteLiteral (chr val)) input length
    | otherwise = mkT (TInteger val) input length
    where val = read (getStr input length) :: Int

mkRegister input length
    | val >=0 && val < 256 = mkT (TRegister val) input length
    | otherwise = mkT (TError ("Invalid Register " ++ registerText)) input length
    where
        registerText = getStr input length
        val = read (tail registerText) :: Int

mkIdentifier :: Monad m => (t, t1, t2, String) -> Int -> m Token
mkIdentifier input length =
    mkT (TIdentifier label) input length
    where label = getStr input length

mkError :: Monad m => (t, t1, t2, String) -> Int -> m Token
mkError input length =
    mkT (TError label) input length
    where label = getStr input length

getStr (_, _, _, remaining) length = take length remaining

mkT :: (Monad m) => Token -> t -> t1 -> m Token
mkT token _ _ = return $ token

alexEOF = return LEOF

startString _ _ =
    do
        setLexerStringValue ""
        setLexerStringState True
        alexMonadScan

endString input length =
    do
        s <- getLexerStringValue
        setLexerStringState False
        mkT (TStringLiteral (reverse s)) input length

tokens str = runAlex str $ do
               let loop = do tok <- alexMonadScan
                             if tok == LEOF
                               then return [ LEOF ]
                               else do toks <- loop
                                       return $ tok : toks
               loop
}
```

A.1.2 Parser

```
{
module MMix_Parser where

import Data.Char
import MMix_Lexer
}

%name parseFile
%tokentype { Token }
%error { parseError }
%monad { Alex }
%lexer { lexwrap } { LEOF }

%token
    OP_CODE        { TOpCode $$ }
    OP_CODE_SIMPLE { TOpCodeSimple $$ }
    SET            { TSet }
    COMMA          { TComma }
    HALT           { THalt }
    FPUTS          { TFputS }
    STDOUT         { TStdOut }
    BYTE_LIT       { TByteLiteral $$ }
    ID             { TIdentifier $$ }
    REG            { TRegister $$ }
    INT            { TInteger $$ }
    LOCAL_LABEL    { TLocalLabel $$ }
    FORWARD        { TLocalForwardOperand $$ }
    BACKWARD       { TLocalBackwardOperand $$ }
    LOC            { TLOC }
    IS             { TIS }
    WYDE           { TWyde }
    TETRA          { TTetra }
    OCTA           { TOcta }
    GREG           { TGREG }
    PLUS           { TPlus }
    MINUS          { TMinus }
    MULTIPLY       { TMult }
    DIVIDE         { TDivide }
    AT             { TAtSign }
    DS             { TDataSegment }
    BYTE           { TByte }
    STR            { TStringLiteral $$ }
    HEX            { THexLiteral $$ }
    OPEN           { TOpenParen }
    CLOSE          { TCloseParen }
%%

Program         : AssignmentLines { reverse $1 }

AssignmentLines : {- empty -}        {[]}
                | AssignmentLines AssignmentLine { $2 : $1 }

AssignmentLine :: {Line}
AssingmentLine : OP_CODE OperatorList { defaultPlainOpCodeLine { pocl_code = $1, pocl_ops = (
     reverse $2) } }
               | Identifier PI { defaultLabelledPILine { lppl_id = $2, lppl_ident = $1 } }
               | Identifier OP_CODE OperatorList { defaultLabelledOpCodeLine { lpocl_code = $2
                   , lpocl_ops = (reverse $3), lpocl_ident = $1 }  }
               | PI { defaultPlainPILine { ppl_id = $1 } }
               | OP_CODE_SIMPLE OperatorList { defaultPlainOpCodeLine { pocl_code = $1,
                   pocl_ops = (reverse $2), pocl_sim = True } }
               | Identifier OP_CODE_SIMPLE OperatorList { defaultLabelledOpCodeLine {
                   lpocl_code = $2, lpocl_ops = (reverse $3), lpocl_ident = $1, lpocl_sim =
                   True }  }

OperatorList : OperatorElement { $1 : [] }
    | OperatorList COMMA OperatorElement { $3 : $1 }

OperatorElement : HALT       { PseudoCode 0 }
                | FPUTS      { fputs }
                | STDOUT     { PseudoCode 1 }
                | REG        { Register (chr $1) }
                | FORWARD    { LocalForward $1 }
                | BACKWARD   { LocalBackward $1 }
                | Expression { Expr $1 }

Identifier : ID { Id $1 }
           | LOCAL_LABEL { LocalLabel $1 }

PI : LOC Expression      { LocEx $2 }
   | GREG Expression     { GregEx $2 }
   | SET OperatorElement COMMA OperatorElement { Set ($2, $4) }
   | BYTE Byte_Array     { ByteArray (reverse $2) }
   | WYDE Byte_Array     { WydeArray (reverse $2) }
   | TETRA Byte_Array    { TetraArray (reverse $2) }
   | OCTA Byte_Array     { OctaArray (reverse $2) }
   | IS INT              { IsNumber $2 }
   | IS BYTE_LIT         { IsNumber (ord $2) }
   | IS REG              { IsRegister $2 }
   | IS Identifier       { IsIdentifier $2 }

Byte_Array : STR { reverse $1 }
           | HEX { (chr $1) : [] }
           | BYTE_LIT { $1 : [] }
           | Byte_Array COMMA STR { (reverse $3) ++ $1 }
           | Byte_Array COMMA BYTE_LIT { $3 : $1 }
           | Byte_Array COMMA HEX { (chr $3) : $1 }

GlobalVariables : DS { 0x20000000 }
```

```
--                    | OPEN Expression CLOSE { [ExpressionClose] ++ $2 ++ [ExpressionOpen] }

Expression : Term { $1 }
           | Expression PLUS Term { ExpressionPlus $1 $3 }
           | Expression MINUS Term { ExpressionMinus $1 $3 }

Term : Primary_Expression { $1 }
     | Term MULTIPLY Primary_Expression { ExpressionMultiply $1 $3 }
     | Term DIVIDE Primary_Expression { ExpressionDivide $1 $3 }

Primary_Expression : INT                   { ExpressionNumber $1 }
                   | Identifier            { ExpressionIdentifier $1 }
                   | BYTE_LIT              { ExpressionNumber (ord $1) }
                   | AT                    { ExpressionAT }
                   | HEX                   { ExpressionNumber $1 }
                   | GlobalVariables       { ExpressionNumber $1 }
                   | OPEN Expression CLOSE { $2 }


{
data Line = PlainOpCodeLine { pocl_code :: Int, pocl_ops :: [OperatorElement], pocl_loc :: Int
     , pocl_sim :: Bool }
          | LabelledOpCodeLine { lpocl_code :: Int, lpocl_ops :: [OperatorElement],
              lpocl_ident :: Identifier, lpocl_loc :: Int, lpocl_sim :: Bool }
          | PlainPILine { ppl_id :: PseudoInstruction, ppl_loc :: Int }
          | LabelledPILine { lppl_id :: PseudoInstruction, lppl_ident :: Identifier, lppl_loc
              :: Int }
          deriving (Eq, Show)

data Identifier = Id String
                | LocalLabel Int
                deriving (Eq, Show, Ord)

data OperatorElement = ByteLiteral Char
               | PseudoCode Int
               | Register Char
               | Ident Identifier
               | LocalForward Int
               | LocalBackward Int
               | Expr ExpressionEntry
               deriving (Eq, Show)

data ExpressionEntry = ExpressionNumber Int
                       | ExpressionRegister Char ExpressionEntry
                       | ExpressionIdentifier Identifier
                       | ExpressionGV Int
                       | Expression
                       | ExpressionAT
                       | ExpressionPlus ExpressionEntry ExpressionEntry
                       | ExpressionMinus ExpressionEntry ExpressionEntry
                       | ExpressionMultiply ExpressionEntry ExpressionEntry
                       | ExpressionDivide ExpressionEntry ExpressionEntry
                       | ExpressionOpen
                       | ExpressionClose
                       deriving (Eq, Show)

data PseudoInstruction = LOC Int
                       | LocEx ExpressionEntry
                       | GregAuto
                       | GregSpecific Char
                       | GregEx ExpressionEntry
                       | ByteArray [Char]
                       | WydeArray [Char]
                       | TetraArray [Char]
                       | OctaArray [Char]
                       | IsRegister Int
                       | IsNumber Int
                       | IsIdentifier Identifier
                       | Set (OperatorElement, OperatorElement)
                       deriving (Eq, Show)

-- fullParse "/home/steveedmans/test.mms"
-- fullParse "/home/steveedmans/hail.mms"

defaultPlainOpCodeLine = PlainOpCodeLine { pocl_loc = -1, pocl_sim = False }
defaultLabelledOpCodeLine = LabelledOpCodeLine { lpocl_loc = -1, lpocl_sim = False }
defaultPlainPILine = PlainPILine { ppl_loc = -1 }
defaultLabelledPILine = LabelledPILine { lppl_loc = -1 }

parseError m = alexError $ "WHY! " ++ show m

lexwrap :: (Token -> Alex a) -> Alex a
lexwrap cont = do
    token <- alexMonadScan
    cont token

fputs = PseudoCode 7

fullParse path = do
    contents <- readFile path
    print $ parseStr contents

parseStr str = runAlex str parseFile

}
```

A.1.3 Symbol Table

```
module SymbolTable where

import MMix_Parser
import Registers
import qualified Data.Map.Lazy as M
import Data.Char (chr, ord)
import Text.Regex.Posix
import DataTypes

type Table = M.Map String Int
type BaseTable = M.Map Char Int
type RegisterOffset = (Char, Int)
type CounterMap = M.Map Int Int

createSymbolTable :: Either String [Line] -> Either String SymbolTable
createSymbolTable (Left msg) = Left msg
createSymbolTable (Right lines) =
        let symbols = foldl getSymbol (Right M.empty) lines
            regs = createRegisterTable $ Right lines
        in symbols

getSymbol :: Either String SymbolTable -> Line -> Either String SymbolTable
getSymbol (Left errorMsg) _ = Left errorMsg
getSymbol (Right table) (LabelledPILine pi@(GregAuto) ident address)
          | M.member ident table = Left $ "Identifier already present " ++ (show ident)
          | otherwise = Right $ M.insert ident (address, Just pi) table
getSymbol (Right table) (LabelledPILine pi@(GregSpecific _) ident address)
          | M.member ident table = Left $ "Identifier already present " ++ (show ident)
          | otherwise = Right $ M.insert ident (address, Just pi) table
getSymbol (Right table) (LabelledPILine pi@(GregEx (ExpressionRegister reg _)) ident address)
          | M.member ident table = Left $ "Identifier already present " ++ (show ident)
          | otherwise = Right $ M.insert ident (address, Just (IsRegister (ord reg))) table
getSymbol (Right table) (LabelledPILine val@(IsNumber _) ident address)
          | M.member ident table = Left $ "Identifier already present " ++ (show ident)
          | otherwise = Right $ M.insert ident (address, Just val) table
getSymbol (Right table) (LabelledPILine val@(IsRegister _) ident address)
          | M.member ident table = Left $ "Identifier already present " ++ (show ident)
          | otherwise = Right $ M.insert ident (address, Just val) table
getSymbol (Right table) (LabelledPILine val@(IsIdentifier _) ident address)
          | M.member ident table = Left $ "Identifier already present " ++ (show ident)
          | otherwise = Right $ M.insert ident (address, Just val) table
getSymbol (Right table) (LabelledPILine val ident address)
          | M.member ident table = Left $ "Identifier already present " ++ (show ident)
          | otherwise = Right $ M.insert ident (address, Just(val)) table
getSymbol (Right table) (LabelledOpCodeLine _ _ ident address _)
          | M.member ident table = Left $ "Identifier already present " ++ (show ident)
          | otherwise = Right $ M.insert ident (address, Nothing) table
getSymbol (Right table) _ = Right $ table

getRegisterFromSymbol :: SymbolTable -> Identifier -> Int
getRegisterFromSymbol st id = reg
    where reg = case M.lookup id st of
                     Just(_, Just (IsRegister r)) -> r
                     _                            -> -1

determineBaseAddressAndOffset :: (M.Map ExpressionEntry Char) -> RegisterAddress -> Maybe(
     RegisterOffset)
determineBaseAddressAndOffset rfa (required_address, _) =
  case (M.lookupLE (ExpressionNumber required_address) rfa) of
    Just((ExpressionNumber address), register) -> Just(register, offset)
      where offset = required_address - address
    _ -> Nothing

mapSymbolToAddress :: SymbolTable -> RegisterTable -> Identifier -> Maybe(RegisterOffset)
mapSymbolToAddress symbols registers identifier@(Id _)
    | M.member identifier symbols = result
    | otherwise = Just('b', 2)
      where registersByAddress = registersFromAddresses registers
```

```
            requiredAddress = symbols M.! identifier
            exactRegister   = extractRegister requiredAddress
            result          = case exactRegister of
                                Just(reg) -> Just(reg, 0)
                                _         -> determineBaseAddressAndOffset registersByAddress
                                      requiredAddress
mapSymbolToAddress _ _ _ = Nothing

extractRegister :: RegisterAddress -> Maybe(Char)
extractRegister (_, Just(IsRegister reg)) = Just(chr reg)
extractRegister _ = Nothing

getSymbolAddress :: SymbolTable -> Identifier -> Int
getSymbolAddress symbols identifier = add
    where Just(add, _) = M.lookup identifier symbols

update_counter :: Int -> Maybe Int -> Int -> CounterMap -> CounterMap
update_counter label (Just old_counter) adjustment counters = M.insert label new_counter
     counters
    where new_counter = old_counter + adjustment
update_counter _ _ _ counters = counters

updated_label :: Int -> Maybe Int -> Identifier
updated_label label (Just current_counter)  = Id $ system_symbol label current_counter
updated_label label _  = Id $ "??" ++ (show label) ++ "HMissing"

transformLocalSymbolLabel :: CounterMap -> Line -> (CounterMap, Line)
transformLocalSymbolLabel counters ln@(LabelledOpCodeLine _ _ (LocalLabel label) _ _) = (
     new_counters, ln{lpocl_ident=new_label})
    where current_counter = M.lookup label counters
          new_label = updated_label label current_counter
          new_counters = update_counter label current_counter 1 counters
transformLocalSymbolLabel counters ln@(LabelledPILine _ (LocalLabel label) _) = (new_counters,
      ln{lppl_ident=new_label})
    where current_counter = M.lookup label counters
          new_label = updated_label label current_counter
          new_counters = update_counter label current_counter 1 counters
transformLocalSymbolLabel counter line = (counter, line)

setLocalSymbolLabel :: CounterMap -> [Line] -> [Line] -> [Line]
setLocalSymbolLabel _ acc [] = reverse acc
setLocalSymbolLabel current_counters acc (x:xs) =
    let (new_counters, new_line) = transformLocalSymbolLabel current_counters x
        new_acc = new_line : acc
    in setLocalSymbolLabel new_counters new_acc xs

setLocalSymbolLabelAuto :: Either String [Line] -> Either String [Line]
setLocalSymbolLabelAuto (Right lns) = Right $ operands_set
    where labels_set = setLocalSymbolLabel localSymbolCounterMap [] lns
          operands_set = transformLocalSymbolLines initialForwardSymbolMap
               initialBackwardSymbolMap labels_set []
setLocalSymbolLabelAuto msg = msg

localSymbolCounterMap :: CounterMap
localSymbolCounterMap = M.fromList $ map (\x -> (x, 0)) [0..9]

initialForwardSymbolMap :: M.Map Int Identifier
initialForwardSymbolMap = M.fromList $ map (\x -> (x, (Id (system_symbol x 0)))) [0..9]

initialBackwardSymbolMap :: M.Map Int (Maybe Identifier)
initialBackwardSymbolMap = M.fromList $ map (\x -> (x, Nothing)) [0..9]

transformLocalSymbolLines :: M.Map Int Identifier -> M.Map Int (Maybe Identifier) -> [Line] ->
      [Line] -> [Line]
transformLocalSymbolLines _ _ [] acc = reverse acc
transformLocalSymbolLines f b (x:xs) acc = transformLocalSymbolLines f' b' xs new_acc
    where (f', b', new_line) = transformLocalSymbol f b x
          new_acc = new_line : acc

transformLocalSymbol :: M.Map Int Identifier -> M.Map Int (Maybe Identifier) -> Line -> (M.Map
      Int Identifier, M.Map Int (Maybe Identifier), Line)
transformLocalSymbol f b l = (f', b', l')
    where f' = transformForward f l
          b' = transformBackward b l
          l' = transformLocalSymbolLine f b l

transformLocalSymbolLine :: M.Map Int Identifier -> M.Map Int (Maybe Identifier) -> Line ->
     Line
transformLocalSymbolLine f b ln@(PlainOpCodeLine _ elements _ _) = ln{pocl_ops = new_elements}
    where new_elements = transformLocalSymbolElements f b elements []
transformLocalSymbolLine f b ln@(LabelledOpCodeLine _ elements _ _ _) = ln{lpocl_ops =
     new_elements}
    where new_elements = transformLocalSymbolElements f b elements []
transformLocalSymbolLine _ _ ln = ln
```

```
transformForward :: M.Map Int Identifier -> Line -> M.Map Int Identifier
transformForward f ln@(LabelledOpCodeLine _ _ (Id label) _ _)
    | is_system_id label = M.insert l new_id f
    | otherwise          = f
       where Just(l, c) = system_id label
             new_label  = system_symbol l (c + 1)
             new_id     = Id new_label
transformForward f ln@(LabelledPILine _ (Id label) _)
    | is_system_id label = M.insert l new_id f
    | otherwise          = f
       where Just(l, c) = system_id label
             new_label  = system_symbol l (c + 1)
             new_id     = Id new_label
transformForward f _ = f


transformBackward :: M.Map Int (Maybe Identifier) -> Line -> M.Map Int (Maybe Identifier)
transformBackward b ln@(LabelledOpCodeLine _ _ (Id label) _ _)
    | is_system_id label = M.insert l new_id b
    | otherwise          = b
       where Just(l, _) = system_id label
             new_id     = Just(Id label)
transformBackward b ln@(LabelledPILine _ (Id label) _)
    | is_system_id label = M.insert l new_id b
    | otherwise          = b
       where Just(l, _) = system_id label
             new_id     = Just(Id label)
transformBackward b l = b

transformLocalSymbolElements :: M.Map Int Identifier -> M.Map Int (Maybe Identifier) -> [
     OperatorElement] -> [OperatorElement] -> [OperatorElement]
transformLocalSymbolElements _ _ [] acc = reverse acc
transformLocalSymbolElements f b (x:xs) acc = transformLocalSymbolElements f b xs new_acc
    where new_value = transformLocalSymbolElement f b x
          new_acc = new_value : acc

transformLocalSymbolElement :: M.Map Int Identifier -> M.Map Int (Maybe Identifier) ->
     OperatorElement -> OperatorElement
transformLocalSymbolElement forwards _ (LocalForward label) = Ident (identifier)
    where Just identifier = M.lookup label forwards
transformLocalSymbolElement _ backwards elem@(LocalBackward label) = v
    where i = M.lookup label backwards
          v = extractWithDefault i elem
transformLocalSymbolElement _ _ element = element

extractWithDefault :: Maybe (Maybe Identifier) -> OperatorElement -> OperatorElement
extractWithDefault (Just (Just v)) _ = Ident v
extractWithDefault _ d = d

system_id :: String -> Maybe (Int, Int)
system_id label
    | is_system_id label = Just(l, c)
    | otherwise = Nothing
        where l = read $ drop 2 $ take 3 label
              c = read $ drop 4 label

system_symbol_pattern = "^\\?\\?[0-9]H[0-9]+$"

is_system_id :: String -> Bool
is_system_id symbol = symbol =~ system_symbol_pattern

system_symbol :: Int -> Int -> String
system_symbol label counter = "??" ++ (show label) ++ "H" ++ (show counter)
```

A.1.4 Common Data Types

```
module DataTypes where
import qualified Data.Map.Lazy as M
import MMix_Parser

type SymbolTable = M.Map Identifier RegisterAddress
type RegisterAddress = (Int, Maybe PseudoInstruction)

instance Ord ExpressionEntry where
    (ExpressionNumber num1) `compare` (ExpressionNumber num2) = num1 `compare` num2
```

A.1.5 Expressions

```
module Expressions where

import MMix_Parser
import DataTypes
import qualified Data.Map.Lazy as M

isSingleExprNumber :: ExpressionEntry -> Maybe Int
isSingleExprNumber (ExpressionNumber val) = Just val
isSingleExprNumber _ = Nothing

evaluateAllLocExpressions :: Either String [Line] -> Either String SymbolTable -> Either
     String [Line]
evaluateAllLocExpressions (Left msg) _ = Left msg
evaluateAllLocExpressions _ (Left msg) = Left msg
evaluateAllLocExpressions (Right lines) (Right st) = Right $ evaluateAllLocLines st lines []

evaluateAllLocLines :: SymbolTable -> [Line] -> [Line] -> [Line]
evaluateAllLocLines _ [] acc = reverse acc
evaluateAllLocLines st (ln:lns) acc = evaluateAllLocLines st lns (new_line : acc)
        where new_line = evaluateLocLine st ln

evaluateLocLine :: SymbolTable -> Line -> Line
evaluateLocLine st ln@(LabelledPILine (LocEx expr) _ address) = ln{lppl_id = (LocEx (
     ExpressionNumber v))}
   where v = evaluate expr address st
evaluateLocLine st ln@(PlainPILine (LocEx expr) address) = ln{ppl_id = (LocEx (
     ExpressionNumber v))}
   where v = evaluate expr address st
evaluateLocLine _ ln = ln

evaluateAllExpressions :: Either String [Line] -> Either String SymbolTable -> Either String [
     Line]
evaluateAllExpressions (Left msg) _ = Left msg
evaluateAllExpressions _ (Left msg) = Left msg
evaluateAllExpressions (Right lines) (Right st) = Right $ evaluateAllLines st lines []

evaluateAllLines :: SymbolTable -> [Line] -> [Line] -> [Line]
evaluateAllLines _ [] acc = reverse acc
evaluateAllLines st (ln:lns) acc = evaluateAllLines st lns (new_line : acc)
    where new_line = evaluateLine st ln

evaluateLine :: SymbolTable -> Line -> Line
evaluateLine st ln@(LabelledPILine (GregEx (ExpressionRegister reg expr)) _ address) =
     new_line
   where v = evaluate expr address st
         new_reg = ExpressionRegister reg (ExpressionNumber v)
         new_line = ln{lppl_id = (GregEx new_reg)}
evaluateLine st ln@(LabelledPILine (LocEx expr) _ address) = ln{lppl_id = (LocEx (
     ExpressionNumber v))}
   where v = evaluate expr address st
evaluateLine st ln@(PlainPILine (LocEx expr) address) = ln{ppl_id = (LocEx (ExpressionNumber v
     ))}
   where v = evaluate expr address st
evaluateLine st ln@(PlainOpCodeLine _ ops _ _) = ln{pocl_ops = updated_operands}
   where updated_operands = evaluateOperands st [] ops
evaluateLine st ln@(LabelledOpCodeLine _ ops _ _ _) = ln{lpocl_ops = updated_operands}
   where updated_operands = evaluateOperands st [] ops
evaluateLine _ ln = ln

evaluateOperands :: SymbolTable -> [OperatorElement] -> [OperatorElement] -> [OperatorElement]
evaluateOperands st acc [] = reverse acc
evaluateOperands st acc (op:ops) = evaluateOperands st (new_op : acc) ops
    where new_op = evaluateOperand st op

evaluateOperand :: SymbolTable -> OperatorElement -> OperatorElement
evaluateOperand _ op@(Expr (ExpressionNumber _)) = op
evaluateOperand _ op@(Expr (ExpressionRegister _ _)) = op
evaluateOperand _ op@(Expr (ExpressionIdentifier _)) = op
evaluateOperand _ op@(Expr (ExpressionGV _)) = op
evaluateOperand _ op@(Expr ExpressionAT) = op
evaluateOperand st (Expr expr) = Expr (ExpressionNumber val)
    where val = evaluate expr 0 st
evaluateOperand _ op = op

evaluate :: ExpressionEntry -> Int -> SymbolTable -> Int
evaluate (ExpressionNumber val) _ _ = val
evaluate ExpressionAT loc _ = loc
evaluate (ExpressionMinus expr1 expr2) loc st = v1 - v2
    where v1 = evaluate expr1 loc st
          v2 = evaluate expr2 loc st
evaluate (ExpressionPlus expr1 expr2) loc st = v1 + v2
    where v1 = evaluate expr1 loc st
          v2 = evaluate expr2 loc st
evaluate (ExpressionMultiply expr1 expr2) loc st = v1 * v2
```

```
    where v1 = evaluate expr1 loc st
          v2 = evaluate expr2 loc st
evaluate (ExpressionDivide expr1 expr2) loc st = quot v1 v2
    where v1 = evaluate expr1 loc st
          v2 = evaluate expr2 loc st
evaluate (ExpressionIdentifier id) _ st
    | M.member id st = v
        where Just(val, lv) = M.lookup id st
              v = evaluatePI lv val
evaluate _ _ _ = -999999

evaluatePI :: Maybe PseudoInstruction -> Int -> Int
evaluatePI (Just (IsNumber val)) _ = val
evaluatePI _ val = val
```

A.1.6 Locations

```
module Locations where

import MMix_Parser
import Expressions

setInnerLocation nextLoc acc [] = reverse acc
setInnerLocation nextLoc acc (ln:lns) = setInnerLocation newLoc newAcc lns
    where (newLoc, newLine) = setLocation nextLoc ln
          newAcc = newLine : acc

setLocation :: Int -> Line -> (Int, Line)
setLocation nextLoc ln@(PlainPILine (LocEx loc) _) =
    case isSingleExprNumber loc of
       Just val -> (val, ln { ppl_loc = val })
       _ -> (nextLoc, ln)
setLocation nextLoc ln@(LabelledPILine (LocEx loc) _ _) =
    case isSingleExprNumber loc of
       Just val -> (val, ln { lppl_loc = val })
       _ -> (nextLoc, ln)
setLocation nextLoc ln@(LabelledPILine (ByteArray arr) _ _) = (newLoc, ln { lppl_loc = nextLoc
     })
    where size = length arr
          adjustment = case (rem size 4) of
                         0 -> 0
                         x -> 4 - x
          newLoc = nextLoc + size
setLocation nextLoc ln@(PlainPILine (ByteArray arr) _) = (newLoc, ln { ppl_loc = nextLoc })
    where size = length arr
          adjustment = case (rem size 4) of
                         0 -> 0
                         x -> 4 - x
          newLoc = nextLoc + size
setLocation nextLoc ln@(LabelledPILine (WydeArray arr) _ _) = (newLoc, ln { lppl_loc =
     addjusted_loc })
    where addjusted_loc = case (rem nextLoc 2) of
                            0 -> nextLoc
                            x -> nextLoc + x
          newLoc = addjusted_loc + ((length arr) * 2)
setLocation nextLoc ln@(PlainPILine (WydeArray arr) _) = (newLoc, ln { ppl_loc = addjusted_loc
     })
    where addjusted_loc = case (rem nextLoc 2) of
                            0 -> nextLoc
                            x -> nextLoc + x
          newLoc = addjusted_loc + ((length arr) * 2)
setLocation nextLoc ln@(LabelledPILine (TetraArray arr) _ _) = (newLoc, ln { lppl_loc =
     addjusted_loc })
    where addjusted_loc = case (rem nextLoc 4) of
                            0 -> nextLoc
                            x -> nextLoc + (4 - x)
          newLoc = addjusted_loc + ((length arr) * 4)
setLocation nextLoc ln@(PlainPILine (TetraArray arr) _) = (newLoc, ln { ppl_loc =
     addjusted_loc })
    where addjusted_loc = case (rem nextLoc 4) of
                            0 -> nextLoc
                            x -> nextLoc + (4 - x)
          newLoc = addjusted_loc + ((length arr) * 4)
setLocation nextLoc ln@(LabelledPILine (OctaArray arr) _ _) = (newLoc, ln { lppl_loc =
     addjusted_loc })
    where addjusted_loc = case (rem nextLoc 8) of
                            0 -> nextLoc
                            x -> nextLoc + (8 - x)
          newLoc = addjusted_loc + ((length arr) * 8)
setLocation nextLoc ln@(PlainPILine (OctaArray arr) _) = (newLoc, ln { ppl_loc = addjusted_loc
     })
    where addjusted_loc = case (rem nextLoc 8) of
```

```
                              0 -> nextLoc
                              x -> nextLoc + (8 - x)
          newLoc = addjusted_loc + ((length arr) * 8)
setLocation nextLoc ln@(LabelledPILine (Set _) _ _) = (newLoc, ln { lppl_loc = nextLoc })
    where newLoc = nextLoc + 4
setLocation nextLoc ln@(PlainPILine (Set _) _) = (newLoc, ln { ppl_loc = nextLoc })
    where newLoc = nextLoc + 4
setLocation nextLoc ln@(PlainPILine _ _) = (nextLoc, ln { ppl_loc = nextLoc })
setLocation nextLoc ln@(LabelledPILine _ _ _) = (nextLoc, ln { lppl_loc = nextLoc })
setLocation nextLoc ln@(PlainOpCodeLine _ _ _ _) = (newLoc, ln { pocl_loc = adjusted_loc })
    where adjusted_loc = case (rem nextLoc 4) of
                             0 -> nextLoc
                             x -> nextLoc + (4 - x)
          newLoc = adjusted_loc + 4
setLocation nextLoc ln@(LabelledOpCodeLine _ _ _ _ _) = (newLoc, ln { lpocl_loc = adjusted_loc
      })
    where adjusted_loc = case (rem nextLoc 4) of
                             0 -> nextLoc
                             x -> nextLoc + (4 - x)
          newLoc = adjusted_loc + 4
```

A.1.7 Registers

```
module Registers
--(
--    RegisterAddress,
--    RegisterTable,
--    createRegisterTable
--)
where

import MMix_Parser
import qualified Data.Map.Lazy as M
import Data.Char (chr, ord)
import Expressions
import DataTypes

type RegisterTable = M.Map Char ExpressionEntry
type AlternativeRegisterTable = M.Map ExpressionEntry Char

setAlexGregAuto :: Either String [Line] -> Either String [Line]
setAlexGregAuto (Right lns) = Right $ setGregAuto 254 [] lns
setAlexGregAuto msg = msg

setGregAuto :: Int -> [Line] -> [Line] -> [Line]
setGregAuto _ acc [] = reverse acc
setGregAuto currentRegister acc (x:xs) =
    let (newLine, nextRegister) = specifyGregAuto x currentRegister
        newAcc = newLine : acc
    in setGregAuto nextRegister newAcc xs

specifyGregAuto :: Line -> Int -> (Line, Int)
specifyGregAuto ln@(LabelledPILine (GregEx val) _ loc) nxt = (new_line, new_counter)
   where new_line = ln{lppl_id = GregEx (ExpressionRegister (chr nxt) val)}
         new_counter = nxt - 1
specifyGregAuto ln@(PlainPILine (GregEx val) loc) nxt =  (new_line, new_counter)
   where new_line = ln{ppl_id = GregEx (ExpressionRegister (chr nxt) val)}
         new_counter = nxt - 1
specifyGregAuto line nxt = (line, nxt)


createRegisterTable :: Either String [Line] -> Either String RegisterTable
createRegisterTable (Left msg) = Left msg
createRegisterTable (Right lines) = foldl getRegister (Right M.empty) lines

getRegister :: Either String RegisterTable -> Line -> Either String RegisterTable
getRegister (Left msg) _ = Left msg
getRegister (Right table) (LabelledOpCodeLine _ _ (Id "Main") address _)
        | M.member (chr 255) table = Left $ "Duplicate Main section definition"
        | otherwise = Right $ M.insert (chr 255) (ExpressionNumber address) table
getRegister (Right table) (LabelledPILine _ (Id "Main") address)
        | M.member (chr 255) table = Left $ "Duplicate Main section definition"
        | otherwise = Right $ M.insert (chr 255) (ExpressionNumber address) table
getRegister (Right table) (PlainPILine pi@(GregEx (ExpressionRegister r ExpressionAT)) address
      ) =
        addRegister table r (ExpressionNumber address)
getRegister (Right table) (LabelledPILine pi@(GregEx (ExpressionRegister r ExpressionAT)) _
      address) =
        addRegister table r (ExpressionNumber address)
getRegister (Right table) (LabelledPILine pi@(GregEx (ExpressionRegister r (ExpressionNumber v
      ))) _ address) =
        addRegister table r (ExpressionNumber v)
```

```
getRegister (Right table) _ = Right $ table

addRegister table register address
    | M.member register table = Left $ "Duplicate register definition " ++ (show register)
    | otherwise = Right $ M.insert register address table

registersFromAddresses :: RegisterTable -> AlternativeRegisterTable
registersFromAddresses orig = M.foldrWithKey addNextRegister M.empty without_main
   where without_main = M.filterWithKey remove_main orig

remove_main :: Char -> ExpressionEntry -> Bool
remove_main k _
  | k == (chr 255) = False
  | otherwise      = True

addNextRegister :: Char -> ExpressionEntry -> AlternativeRegisterTable ->
      AlternativeRegisterTable
addNextRegister k v orig = M.insert v k orig

getRegisterDetails :: [ExpressionEntry] -> Maybe(Char, Int)
getRegisterDetails _ = Nothing
```

A.1.8 Code Generation

```
module CodeGen where

import SymbolTable
import qualified Data.Map.Lazy as M
import qualified Data.List.Ordered as O
import qualified Data.ByteString.Lazy as B
import Data.Binary
import MMix_Parser
import Data.Char (chr, ord)
import Registers
import Expressions as E
import Numeric (showHex)
import DataTypes

type AdjustedOperands = (Int, String)

data CodeLine = CodeLine { cl_address :: Int, cl_size :: Int, cl_code :: [Char] }
                deriving(Show)

instance Eq CodeLine where
    (CodeLine address1 _ _) == (CodeLine address2 _ _) = address1 == address2
instance Ord CodeLine where
    (CodeLine address1 _ _) `compare` (CodeLine address2 _ _) = address1 `compare` address2

encodeProgram :: Either String [CodeLine] -> Either String RegisterTable -> Either String
      String
encodeProgram (Left code_error) _ = Left code_error
encodeProgram _ (Left register_error) = Left register_error
encodeProgram (Right code) (Right regs) = Right $ map chr $ encodeProgramInt code regs

genCodeForLine :: SymbolTable -> RegisterTable -> Line -> Maybe(CodeLine)
genCodeForLine symbols registers (LabelledOpCodeLine opcode operands _ address simple_code) =
      genOpCodeOutput symbols registers opcode operands address simple_code
genCodeForLine symbols registers (PlainOpCodeLine opcode operands address simple_code) =
      genOpCodeOutput symbols registers opcode operands address simple_code
genCodeForLine _ _ (LabelledPILine (ByteArray arr) _ address) = Just(CodeLine {cl_address =
      address, cl_size = s, cl_code = arr})
    where s = length arr
genCodeForLine _ _ (LabelledPILine (WydeArray arr) _ address) = Just(CodeLine {cl_address =
      address, cl_size = s, cl_code = wyde_array})
    where wyde_array = make_bytes arr 2
          s = length wyde_array
genCodeForLine _ _ (LabelledPILine (TetraArray arr) _ address) = Just(CodeLine {cl_address =
      address, cl_size = s, cl_code = wyde_array})
    where wyde_array = make_bytes arr 4
          s = length wyde_array
genCodeForLine _ _ (LabelledPILine (OctaArray arr) _ address) = Just(CodeLine {cl_address =
      address, cl_size = s, cl_code = wyde_array})
    where wyde_array = make_bytes arr 8
          s = length wyde_array
genCodeForLine symbols registers (LabelledPILine (Set (e1, e2)) _ address) =
    genPICodeOutput symbols registers address e1 e2
genCodeForLine symbols registers (PlainPILine (Set (e1, e2)) address) =
    genPICodeOutput symbols registers address e1 e2
genCodeForLine _ _ _ = Nothing

genPICodeOutput :: SymbolTable -> RegisterTable -> Int -> OperatorElement -> OperatorElement
      -> Maybe(CodeLine)
```

```
genPICodeOutput symbols registers address i1@(Expr (ExpressionIdentifier _)) i2@(Expr (
     ExpressionIdentifier _)) = genOpCodeOutput symbols registers 193 operands address True
    where operands = i1 : i2 : (Expr (ExpressionNumber 0)) : []
genPICodeOutput symbols registers address i1@(Expr (ExpressionIdentifier _)) i2@(Expr (
     ExpressionNumber _)) = genOpCodeOutput symbols registers 227 operands address True
    where operands = i1 : (Expr (ExpressionNumber 0)) : i2 : []
genPICodeOutput symbols registers address r1@(Register _) r2@(Register _) = genOpCodeOutput
     symbols registers 192 operands address True
    where operands = r1 : r2 : (Expr (ExpressionNumber 0)) : []
genPICodeOutput symbols registers address r1@(Register _) r2@(Expr (ExpressionNumber _)) =
     genOpCodeOutput symbols registers 227 operands address True
    where operands = r1 : r2 : []
genPICodeOuput _ _ _ _ _ = Nothing

genOpCodeOutput :: SymbolTable -> RegisterTable -> Int -> [OperatorElement] -> Int -> Bool ->
     Maybe CodeLine
genOpCodeOutput symbols registers 254 operands address _ = Just(CodeLine {cl_address = address
     , cl_size = 4, cl_code = code})
    where code = splitSpecialRegisters symbols operands
genOpCodeOutput symbols registers opcode operands address False
    | opcode >= 60 && opcode <= 95 = Just(CodeLine {cl_address = address, cl_size = 4, cl_code
          = (chr (opcode + local_adjustment)) : code})
    | opcode == 240 = Just(CodeLine {cl_address = address, cl_size = 4, cl_code = (chr (opcode
          + jump_adjustment)) : jump_code})
    | opcode == 34 = case splitOperandsAddress symbols registers operands of
                        Just(address_adjustment, address_code) -> Just(CodeLine {cl_address =
                              address, cl_size = 4, cl_code = (chr (opcode +
                              address_adjustment)) : address_code})
                        _ -> Nothing
    | otherwise = case splitOperands symbols registers operands of
        Just((adjustment,params)) -> Just(CodeLine {cl_address = address, cl_size = 4, cl_code
              = (chr (opcode + adjustment)) : params})
        _ -> Nothing
      where (local_adjustment, code) = splitLocalOperands symbols operands address
            (jump_adjustment, jump_code) = jumpOperands symbols operands address
genOpCodeOutput symbols registers opcode operands address True =
    case splitOperands symbols registers operands of
        Just((adjustment,params)) -> Just(CodeLine {cl_address = address, cl_size = 4, cl_code
              = (chr opcode) : params})
        _ -> Nothing

localLabelOffset :: Int -> Int -> (Int, Int)
localLabelOffset current required
    | required < current = (1, (quot (current - required) 4))
    | otherwise          = (0, (quot (required - current) 4))

formatElement :: SymbolTable -> OperatorElement -> Char
formatElement _ (ByteLiteral b) = b
formatElement _ (PseudoCode pc) = chr pc
formatElement _ (Register r) = r
formatElement st (Expr x@(ExpressionIdentifier id)) =
    case M.lookup id st of
        (Just (_, Just (IsRegister r))) -> chr r
        (Just (_, Just (IsIdentifier r))) -> formatElement st (Expr (ExpressionIdentifier r))
        otherwise -> evaluateByteToChar st x
formatElement st (Expr x) = evaluateByteToChar st x

evaluateByteToChar :: SymbolTable -> ExpressionEntry -> Char
evaluateByteToChar st x = chr digit
    where plain_digit = (E.evaluate x 0 st)
          digit = if plain_digit < 0
                  then 256 + plain_digit
                  else plain_digit

jumpOperands :: SymbolTable -> [OperatorElement] -> Int -> (Int, String)
jumpOperands symbols ((Ident id):[]) address = (adjustment, code)
    where ro = getSymbolAddress symbols id
          (adjustment, offset) = localLabelOffset address ro
          b2 = rem offset 256
          q2 = quot offset 256
          b1 = rem q2 256
          b0 = quot q2 256
          code = (chr b0) : (chr b1) : (chr b2) : []

splitOperandsAddress :: SymbolTable -> RegisterTable -> [OperatorElement] -> Maybe(
     AdjustedOperands)
splitOperandsAddress symbols registers (x:(Expr (ExpressionNumber y)):[]) = Just(1, code)
    where registersByAddress = registersFromAddresses registers
          Just(reg, offset) = determineBaseAddressAndOffset registersByAddress (y, Nothing)
          formatted_x = formatElement symbols x
          code = formatted_x : reg : (chr offset) : []
splitOperandsAddress symbols registers (x:Expr (ExpressionIdentifier y):[]) =
    case mapSymbolToAddress symbols registers y of
       Just(y_reg, y_offset) -> Just(1, code)
```

```
          where formatted_x = formatElement symbols x
                code = formatted_x : y_reg : (chr y_offset) : []
       otherwise -> Nothing
splitOperandsAddress _ _ _ = Nothing

splitLocalOperands :: SymbolTable -> [OperatorElement] -> Int -> (Int, String)
splitLocalOperands symbols (x:(Ident id):[]) address = (adjustment, code)
    where ro = getSymbolAddress symbols id
          formatted_x = formatElement symbols x
          (adjustment, offset) = localLabelOffset address ro
          b1 = chr (quot offset 256)
          b2 = chr (rem  offset 256)
          code = formatted_x : b1 : b2 : []

splitOperands :: SymbolTable -> RegisterTable -> [OperatorElement] -> Maybe(AdjustedOperands)
splitOperands symbols registers ((Ident id):[]) = Just(1, code)
    where ro = mapSymbolToAddress symbols registers id
          code = case ro of
                     Just((base,offset)) -> (chr 0) : base : (chr offset) : []
splitOperands symbols registers ((Ident id1):(Expr (ExpressionIdentifier id2)):[]) = Just(1,
     code)
    where ro1 = mapSymbolToAddress symbols registers id1
          ro2 = mapSymbolToAddress symbols registers id2
          code = case (ro1, ro2) of
                     (Just((base1,_)), Just((base2,offset2))) -> base1 : base2 : (chr offset2)
                           : []
                     otherwise -> []
splitOperands symbols registers (x:(Expr (ExpressionNumber y)):[]) = Just(1, code)
    where ops = map chr $ drop 2 $ char4 y
          formatted_x = formatElement symbols x
          code = formatted_x : ops
splitOperands symbols registers (x:(Ident id):[]) = Just(1, code)
    where ro = mapSymbolToAddress symbols registers id
          formatted_x = formatElement symbols x
          code = case ro of
                     Just((base,offset)) -> formatted_x : base : (chr offset) : []
                     otherwise -> []
splitOperands symbols registers (x:(Expr (ExpressionIdentifier id)):[]) = Just(1, code)
    where ro = mapSymbolToAddress symbols registers id
          formatted_x = formatElement symbols x
          code = case ro of
                     Just((base,offset)) -> formatted_x : base : (chr offset) : []
                     otherwise -> []
splitOperands symbols registers (x : y : z@(Expr (ExpressionNumber _)) : []) = Just(1, code)
    where formatted_x = formatElement symbols x
          formatted_y = formatElement symbols y
          formatted_z = formatElement symbols z
          code = formatted_x : formatted_y : formatted_z : []
splitOperands symbols registers (x : y : z : []) = Just(0, code)
    where formatted_x = formatElement symbols x
          formatted_y = formatElement symbols y
          formatted_z = formatElement symbols z
          code = formatted_x : formatted_y : formatted_z : []
splitOperands _ _ _ = Nothing

splitSpecialRegisters :: SymbolTable -> [OperatorElement] -> String
splitSpecialRegisters st (x:(Expr (ExpressionIdentifier (Id special))):[]) = (chr 254) : reg :
      special_register_to_operand special
    where reg = formatElement st x

special_register_to_operand :: String -> String
special_register_to_operand "rA"  = (chr 0) : (chr 21) : []
special_register_to_operand "rB"  = (chr 0) : (chr 0)  : []
special_register_to_operand "rC"  = (chr 0) : (chr 8)  : []
special_register_to_operand "rD"  = (chr 0) : (chr 1)  : []
special_register_to_operand "rE"  = (chr 0) : (chr 2)  : []
special_register_to_operand "rF"  = (chr 0) : (chr 22) : []
special_register_to_operand "rG"  = (chr 0) : (chr 19) : []
special_register_to_operand "rH"  = (chr 0) : (chr 3)  : []
special_register_to_operand "rI"  = (chr 0) : (chr 12) : []
special_register_to_operand "rJ"  = (chr 0) : (chr 4)  : []
special_register_to_operand "rK"  = (chr 0) : (chr 15) : []
special_register_to_operand "rL"  = (chr 0) : (chr 20) : []
special_register_to_operand "rM"  = (chr 0) : (chr 5)  : []
special_register_to_operand "rN"  = (chr 0) : (chr 9)  : []
special_register_to_operand "rO"  = (chr 0) : (chr 10) : []
special_register_to_operand "rP"  = (chr 0) : (chr 23) : []
special_register_to_operand "rQ"  = (chr 0) : (chr 16) : []
special_register_to_operand "rR"  = (chr 0) : (chr 6)  : []
special_register_to_operand "rS"  = (chr 0) : (chr 11) : []
special_register_to_operand "rT"  = (chr 0) : (chr 13) : []
special_register_to_operand "rU"  = (chr 0) : (chr 17) : []
special_register_to_operand "rV"  = (chr 0) : (chr 18) : []
special_register_to_operand "rW"  = (chr 0) : (chr 24) : []
special_register_to_operand "rX"  = (chr 0) : (chr 25) : []
special_register_to_operand "rY"  = (chr 0) : (chr 26) : []
special_register_to_operand "rZ"  = (chr 0) : (chr 27) : []
special_register_to_operand "rBB" = (chr 0) : (chr 7)  : []
special_register_to_operand "rTT" = (chr 0) : (chr 14) : []
special_register_to_operand "rWW" = (chr 0) : (chr 28) : []
special_register_to_operand "rXX" = (chr 0) : (chr 29) : []
special_register_to_operand "rYY" = (chr 0) : (chr 30) : []
special_register_to_operand "rZZ" = (chr 0) : (chr 31) : []

make_bytes :: [Char] -> Int -> [Char]
make_bytes arr size = make_inner_bytes size arr []

make_inner_bytes _ [] acc = acc
make_inner_bytes size (x:xs) acc = make_inner_bytes size xs new_acc
    where extended_byte = make_byte x size []
          new_acc = acc ++ extended_byte

make_byte :: Char -> Int -> [Char] -> [Char]
make_byte _ 0 acc = reverse acc
make_byte b 1 acc = make_byte b 0 (b : acc)
make_byte b n acc = make_byte b (n-1) ((chr 0):acc)

type BlockSummary = (Int, Int, [Int]) -- Starting Address, Size, Data in block

blocks :: [CodeLine] -> [BlockSummary]
blocks [] = []
blocks lines = nextBlock [] (O.sort lines)

nextBlock :: [BlockSummary] -> [CodeLine] -> [BlockSummary]
nextBlock [] (currentLine:rest) = nextBlock [((cl_address currentLine), (cl_size currentLine),
      (map ord (cl_code currentLine)))] rest
nextBlock (currentBlock:blocks) (currentLine:rest)
    | (start + size) == (cl_address currentLine) = nextBlock (updateBlock:blocks) rest
    | otherwise = nextBlock (newBlock:currentBlock:blocks) rest
        where (start, size, code) = currentBlock
              extraCode = map ord (cl_code currentLine)
              updatedCode = code ++ extraCode
              updateBlock = (start, size + (cl_size currentLine), updatedCode)
              newBlock = ((cl_address currentLine), (cl_size currentLine), extraCode)
nextBlock result [] = O.sort result

encodeProgramInt :: [CodeLine] -> RegisterTable -> [Int]
encodeProgramInt prog regs = hdr ++ tbl
    where hdr = header prog
          tbl = encodeRegisterTable regs

header :: [CodeLine] -> [Int]
header program = details
    where bs = blocks program
          num_bs = char4 . length $ bs
          bh = blockDetails (O.sort bs) []
          details = num_bs ++ bh

encodeRegisterTable :: RegisterTable -> [Int]
encodeRegisterTable regs = size ++ vals
    where vals = M.foldrWithKey encodeRegister [] regs
          size = char4 $ M.size regs

encodeRegister :: Char -> ExpressionEntry -> [Int] -> [Int]
encodeRegister r (ExpressionNumber v) a = nextPart ++ a
    where nextPart = (ord r) : char8 v

blockDetails :: [BlockSummary] -> [Int] -> [Int]
blockDetails [] final = final
blockDetails (currentBlock:rest) acc = blockDetails rest newAcc
    where newAcc = acc ++ (blockDetail currentBlock)

blockDetail :: BlockSummary -> [Int]
blockDetail (start, size, code) = startc ++ sizec ++ code
    where startc = char4 start
          sizec  = char4 size

char4 :: Int -> [Int]
char4 val = char4tail [] val

char4tail :: [Int] -> Int -> [Int]
char4tail acc val
    | (val == -1) && ((length acc) == 4) = acc
    | (val < 0) = case (divMod val 256) of
                    (m, r) -> char4tail (r : acc) m
    | (val == 0) && ((length acc) == 4) = acc
    | (val == 0) = char4tail (0 : acc) val
    | otherwise = case (divMod val 256) of
```

```
        (m, r) -> char4tail (r : acc) m

char8 :: Int -> [Int]
char8 val = char8tail [] val

char8tail :: [Int] -> Int -> [Int]
char8tail acc val
    | (val == -1) && ((length acc) == 8) = acc
    | (val < 0) = case (divMod val 256) of
                     (m, r) -> char8tail (r : acc) m
    | (val == 0) && ((length acc) == 8) = acc
    | (val == 0) = char8tail (0 : acc) val
    | otherwise = case (divMod val 256) of
        (m, r) -> char8tail (r : acc) m
```

A.1.9 External Interface

```
module Main where

import MMix_Lexer
import MMix_Parser
import Text.Printf
import qualified Data.Map.Lazy as M
import qualified Data.List.Ordered as O
import Data.Char
import SymbolTable
import CodeGen
import Locations
import Registers
import DataTypes
import Expressions as E

main :: IO()
main = undefined

contents ifs ofs = do
    x <- readFile ifs
    printf "%s\n" x
    let s0 = parseStr x
    let s1 =  setLocalSymbolLabelAuto s0
    let s2 = setAlexLoc s1
    let initial_st = createSymbolTable s2
    let s3 = evaluateAllLocExpressions s2 initial_st
    let s4 = setAlexLoc s3
    let s5 = setAlexGregAuto s4
    let st = createSymbolTable s5
    let s6 = evaluateAllExpressions s5 st
    let regs = createRegisterTable s6
    let st2 = createSymbolTable s6
    let code = acg st2 regs s6
    print code
    let pg = encodeProgram code regs
    case pg of
        Right encoded_program -> writeFile ofs encoded_program
        Left error            -> print error
    print pg
    return s6

setAlexLoc :: Either String [Line] -> Either String [Line]
setAlexLoc (Right lns) = Right $ setLoc 0 lns
setAlexLoc m = m

setLoc :: Int -> [Line] -> [Line]
setLoc startLoc lns = setInnerLocation startLoc [] lns

showAlexLocs :: Either String [Line] -> Either String [Int]
showAlexLocs (Right lns) = Right $ showLocs lns
showAlexLocs (Left msg) = Left msg

showLocs :: [Line] -> [Int]
showLocs lns = foldr showLoc [] lns

showLoc :: Line -> [Int] -> [Int]
showLoc (PlainPILine _ loc) acc =  loc : acc
showLoc (LabelledPILine _ _ loc) acc =  loc : acc
showLoc (PlainOpCodeLine _ _ loc _) acc =  loc : acc
showLoc (LabelledOpCodeLine _ _ _ loc _) acc =  loc : acc

acg (Right sym) (Right regs) (Right lns) = Right $ cg sym regs [] lns
acg _ _ _ = Left "Something is missing!!!"

--cg :: (M.Map String RegisterAddress) -> (M.Map Char Int) -> [Line] -> [Line]
```

```
cg _ _ acc [] = acc
cg s r acc (ln:lns) = cg s r newAcc lns
    where cgl = genCodeForLine s r ln
          newAcc = case cgl of
              Just(codeline) -> codeline : acc
              Nothing -> acc
                  where newline = CodeLine {cl_address = 0, cl_size = 0, cl_code = (show ln)}
```

A.2 Graphical User Interface

A.3 Virtual Machine

Appendix B

Intermediate Assembler Representations

B.1 Definitions

B.2 Test Application

B.2.1 Sample Test MMIXAL Code

The sample mmixal application I am using to test the system is taken from Fascile 1[Knu]. The complete code listing is: -

```
L       IS     500
t       IS     $255
n       GREG   0
q       GREG   0
r       GREG   0
jj      GREG   0
kk      GREG   0
pk      GREG   0
mm      IS     kk
        LOC    Data_Segment
PRIME1  WYDE   2
        LOC    PRIME1+2*L
ptop    GREG   @
j0      GREG   PRIME1+2-@
BUF     OCTA   0
        LOC    #100
Main    GREG   @
        SET    n,3
        SET    jj,j0
2H      STWU   n,ptop,jj
        INCL   jj,2
3H      BZ     jj,2F
4H      INCL   n,2
5H      SET    kk,j0
6H      LDWU   pk,ptop,kk
        DIV    q,n,pk
        GET    r,rR
        BZ     r,4B
7H      CMP    t,q,pk
        BNP    t,2B
8H      INCL   kk,2
        JMP    6B
        GREG   @
Title   BYTE   "First Five Hundred Primes"
NewLn   BYTE   #a,0
Blanks  BYTE   "   ",0
2H      LDA    t,Title
        TRAP   0,Fputs,StdOut
        NEG    mm,2
3H      ADD    mm,mm,j0
        LDA    t,Blanks
        TRAP   0,Fputs,StdOut
2H      LDWU   pk,ptop,mm
0H      GREG   #2030303030000000
        STOU   0B,BUF
        LDA    t,BUF+4
1H      DIV    pk,pk,10
        GET    r,rR
        INCL   r,'0'
        STBU   r,t,0
        SUB    t,t,1
        PBNZ   pk,1B
        LDA    t,BUF
        TRAP   0,Fputs,StdOut
        INCL   mm,2*L/10
        PBN    mm,2B
        LDA    t,NewLn
        TRAP   0,Fputs,StdOut
        CMP    t,mm,2*(L/10-1)
        PBNZ   t,3B
        TRAP   0,Halt,0
```

B.2.2 Parsed Sample File

The final intermediate representation of the parsed source code for the test application is: -

```
LabelledPILine {
        lppl_id = IsNumber 500, lppl_ident = Id "L", lppl_loc = 0
}
LabelledPILine {
        lppl_id = IsRegister 255, lppl_ident = Id "t", lppl_loc = 0
}
LabelledPILine {
        lppl_id = GregEx (ExpressionRegister '\254' (ExpressionNumber 0)), lppl_ident = Id "n"
             , lppl_loc = 0
}
LabelledPILine {
        lppl_id = GregEx (ExpressionRegister '\253' (ExpressionNumber 0)), lppl_ident = Id "q"
             , lppl_loc = 0
}
LabelledPILine {
        lppl_id = GregEx (ExpressionRegister '\252' (ExpressionNumber 0)), lppl_ident = Id "r"
             , lppl_loc = 0
}
LabelledPILine {
        lppl_id = GregEx (ExpressionRegister '\251' (ExpressionNumber 0)), lppl_ident = Id "jj
             ", lppl_loc = 0
}
LabelledPILine {
        lppl_id = GregEx (ExpressionRegister '\250' (ExpressionNumber 0)), lppl_ident = Id "kk
             ", lppl_loc = 0
}
LabelledPILine {
        lppl_id = GregEx (ExpressionRegister '\249' (ExpressionNumber 0)), lppl_ident = Id "pk
             ", lppl_loc = 0
}
LabelledPILine {
        lppl_id = GregEx (ExpressionRegister '\248' (ExpressionNumber 0)), lppl_ident = Id "mm
             ", lppl_loc = 0
}
PlainPILine {
        ppl_id = LocEx (ExpressionNumber 536870912), ppl_loc = 536870912
}
LabelledPILine {
        lppl_id = WydeArray "\STX", lppl_ident = Id "PRIME1", lppl_loc = 536870912
}
PlainPILine {
        ppl_id = LocEx (ExpressionNumber 536871912), ppl_loc = 536871912
}
LabelledPILine {
        lppl_id = GregEx (ExpressionRegister '\247' (ExpressionNumber 536871912)), lppl_ident
             = Id "ptop", lppl_loc = 536871912
}
LabelledPILine {
        lppl_id = GregEx (ExpressionRegister '\246' (ExpressionNumber (-998))), lppl_ident =
             Id "j0", lppl_loc = 536871912
}
LabelledPILine {
        lppl_id = OctaArray "\NUL", lppl_ident = Id "BUF", lppl_loc = 536871912
}
PlainPILine {
        ppl_id = LocEx (ExpressionNumber 256), ppl_loc = 256
}
LabelledPILine {
        lppl_id = Set (Expr (ExpressionIdentifier (Id "n")),Expr (ExpressionNumber 3)),
             lppl_ident = Id "Main", lppl_loc = 256
}
PlainPILine {
        ppl_id = Set (Expr (ExpressionIdentifier (Id "jj")),Expr (ExpressionIdentifier (Id "j0
             "))), ppl_loc = 260
}
LabelledOpCodeLine {
        lpocl_code = 166, lpocl_ops = [Expr (ExpressionIdentifier (Id "n")),Expr (
             ExpressionIdentifier (Id "ptop")),Expr (ExpressionIdentifier (Id "jj"))],
             lpocl_ident = Id "??2H0", lpocl_loc = 264
}
PlainOpCodeLine {
        pocl_code = 231, pocl_ops = [Expr (ExpressionIdentifier (Id "jj")),Expr (
             ExpressionNumber 2)], pocl_loc = 268
}
LabelledOpCodeLine {
        lpocl_code = 66, lpocl_ops = [Expr (ExpressionIdentifier (Id "jj")),Ident (Id "??2H1")
             ], lpocl_ident = Id "??3H0", lpocl_loc = 272
}
LabelledOpCodeLine {
        lpocl_code = 231, lpocl_ops = [Expr (ExpressionIdentifier (Id "n")),Expr (
             ExpressionNumber 2)], lpocl_ident = Id "??4H0", lpocl_loc = 276
}
LabelledPILine {
        lppl_id = Set (Expr (ExpressionIdentifier (Id "kk")),Expr (ExpressionIdentifier (Id "
             j0"))), lppl_ident = Id "??5H0", lppl_loc = 280
}
LabelledOpCodeLine {
        lpocl_code = 134, lpocl_ops = [Expr (ExpressionIdentifier (Id "pk")),Expr (
             ExpressionIdentifier (Id "ptop")),Expr (ExpressionIdentifier (Id "kk"))],
             lpocl_ident = Id "??6H0", lpocl_loc = 284
}
PlainOpCodeLine {
        pocl_code = 28, pocl_ops = [Expr (ExpressionIdentifier (Id "q")),Expr (
             ExpressionIdentifier (Id "n")),Expr (ExpressionIdentifier (Id "pk"))], pocl_loc
             = 288
```

```
}
PlainOpCodeLine {
        pocl_code = 254, pocl_ops = [Expr (ExpressionIdentifier (Id "r")),Expr (
             ExpressionIdentifier (Id "rR"))], pocl_loc = 292
}
PlainOpCodeLine {
        pocl_code = 66, pocl_ops = [Expr (ExpressionIdentifier (Id "r")),Ident (Id "??4H0")],
             pocl_loc = 296
}
LabelledOpCodeLine {
        lpocl_code = 48, lpocl_ops = [Expr (ExpressionIdentifier (Id "t")),Expr (
             ExpressionIdentifier (Id "q")),Expr (ExpressionIdentifier (Id "pk"))],
             lpocl_ident = Id "??7H0", lpocl_loc = 300
}
PlainOpCodeLine {
        pocl_code = 76, pocl_ops = [Expr (ExpressionIdentifier (Id "t")),Ident (Id "??2H0")],
             pocl_loc = 304
}
LabelledOpCodeLine {
        lpocl_code = 231, lpocl_ops = [Expr (ExpressionIdentifier (Id "kk")),Expr (
             ExpressionNumber 2)], lpocl_ident = Id "??8H0", lpocl_loc = 308
}
PlainOpCodeLine {
        pocl_code = 240, pocl_ops = [Ident (Id "??6H0")], pocl_loc = 312
}
PlainPILine {
        ppl_id = GregEx (ExpressionRegister '\245' ExpressionAT), ppl_loc = 316
}
LabelledPILine {
        lppl_id = ByteArray "First Five Hundred Primes", lppl_ident = Id "Title", lppl_loc =
             316
}
LabelledPILine {
        lppl_id = ByteArray "\n\NUL", lppl_ident = Id "NewLn", lppl_loc = 341
}
LabelledPILine {
        lppl_id = ByteArray "   \NUL", lppl_ident = Id "Blanks", lppl_loc = 343
}
LabelledOpCodeLine {
        lpocl_code = 34, lpocl_ops = [Expr (ExpressionIdentifier (Id "t")),Expr (
             ExpressionIdentifier (Id "Title"))], lpocl_ident = Id "??2H1", lpocl_loc = 347
}
PlainOpCodeLine {
        pocl_code = 0, pocl_ops = [Expr (ExpressionNumber 0),PseudoCode 7,PseudoCode 1],
             pocl_loc = 351
}
PlainOpCodeLine {
        pocl_code = 52, pocl_ops = [Expr (ExpressionIdentifier (Id "mm")),Expr (
             ExpressionNumber 2)], pocl_loc = 355
}
LabelledOpCodeLine {
        lpocl_code = 32, lpocl_ops = [Expr (ExpressionIdentifier (Id "mm")),Expr (
             ExpressionIdentifier (Id "mm")),Expr (ExpressionIdentifier (Id "j0"))],
             lpocl_ident = Id "??3H1", lpocl_loc = 359
}
PlainOpCodeLine {
        pocl_code = 34, pocl_ops = [Expr (ExpressionIdentifier (Id "t")),Expr (
             ExpressionIdentifier (Id "Blanks"))], pocl_loc = 363
}
PlainOpCodeLine {
        pocl_code = 0, pocl_ops = [Expr (ExpressionNumber 0),PseudoCode 7,PseudoCode 1],
             pocl_loc = 367
}
LabelledOpCodeLine {
        lpocl_code = 134, lpocl_ops = [Expr (ExpressionIdentifier (Id "pk")),Expr (
             ExpressionIdentifier (Id "ptop")),Expr (ExpressionIdentifier (Id "mm"))],
             lpocl_ident = Id "??2H2", lpocl_loc = 371
}
LabelledPILine {
        lppl_id = GregEx (ExpressionRegister '\244' (ExpressionNumber 2319406791617675264)),
             lppl_ident = Id "??0H0", lppl_loc = 375
}
PlainOpCodeLine {
        pocl_code = 174, pocl_ops = [Ident (Id "??0H0"),Expr (ExpressionIdentifier (Id "BUF"))
             ], pocl_loc = 375
}
PlainOpCodeLine {
        pocl_code = 34, pocl_ops = [Expr (ExpressionIdentifier (Id "t")),Expr (
             ExpressionNumber 536870924)], pocl_loc = 379
}
LabelledOpCodeLine {
        lpocl_code = 28, lpocl_ops = [Expr (ExpressionIdentifier (Id "pk")),Expr (
             ExpressionIdentifier (Id "pk")),Expr (ExpressionNumber 10)], lpocl_ident = Id "
             ??1H0", lpocl_loc = 383
```

```
}
PlainOpCodeLine {
        pocl_code = 254, pocl_ops = [Expr (ExpressionIdentifier (Id "r")),Expr (
             ExpressionIdentifier (Id "rR"))], pocl_loc = 387
}
PlainOpCodeLine {
        pocl_code = 231, pocl_ops = [Expr (ExpressionIdentifier (Id "r")),Expr (
             ExpressionNumber 48)], pocl_loc = 391
}
PlainOpCodeLine {
        pocl_code = 162, pocl_ops = [Expr (ExpressionIdentifier (Id "r")),Expr (
             ExpressionIdentifier (Id "t")),Expr (ExpressionNumber 0)], pocl_loc = 395
}
PlainOpCodeLine {
        pocl_code = 36, pocl_ops = [Expr (ExpressionIdentifier (Id "t")),Expr (
             ExpressionIdentifier (Id "t")),Expr (ExpressionNumber 1)], pocl_loc = 399
}
PlainOpCodeLine {
        pocl_code = 90, pocl_ops = [Expr (ExpressionIdentifier (Id "pk")),Ident (Id "??1H0")],
              pocl_loc = 403
}
PlainOpCodeLine {
        pocl_code = 34, pocl_ops = [Expr (ExpressionIdentifier (Id "t")),Expr (
             ExpressionIdentifier (Id "BUF"))], pocl_loc = 407
}
PlainOpCodeLine {
        pocl_code = 0, pocl_ops = [Expr (ExpressionNumber 0),PseudoCode 7,PseudoCode 1],
             pocl_loc = 411
}
PlainOpCodeLine {
        pocl_code = 231, pocl_ops = [Expr (ExpressionIdentifier (Id "mm")),Expr (
             ExpressionNumber 100)], pocl_loc = 415
}
PlainOpCodeLine {
        pocl_code = 80, pocl_ops = [Expr (ExpressionIdentifier (Id "mm")),Ident (Id "??2H2")],
              pocl_loc = 419
}
PlainOpCodeLine {
        pocl_code = 34, pocl_ops = [Expr (ExpressionIdentifier (Id "t")),Expr (
             ExpressionIdentifier (Id "NewLn"))], pocl_loc = 423
}
PlainOpCodeLine {
        pocl_code = 0, pocl_ops = [Expr (ExpressionNumber 0),PseudoCode 7,PseudoCode 1],
             pocl_loc = 427
}
PlainOpCodeLine {
        pocl_code = 48, pocl_ops = [Expr (ExpressionIdentifier (Id "t")),Expr (
             ExpressionIdentifier (Id "mm")),Expr (ExpressionNumber 98)], pocl_loc = 431
}
PlainOpCodeLine {
        pocl_code = 90, pocl_ops = [Expr (ExpressionIdentifier (Id "t")),Ident (Id "??3H1")],
             pocl_loc = 435
}
PlainOpCodeLine {
        pocl_code = 0, pocl_ops = [Expr (ExpressionNumber 0),PseudoCode 0,Expr (
             ExpressionNumber 0)], pocl_loc = 439
}
```