The MIXWare Report and Manual

GEORGE TRYFONAS george.tryfonas@gmail.com

Introduction

THE `MIX` computer is an imaginary computer invented by Don Knuth in the 1960s in order to present his exposition on computer algorithms. As Knuth puts it, the use of an imaginary computer and machine language helps avoid distracting the reader with the technicalities of one particular computer system, and the focus remains on truths that have always been—and will always be—valid, independent of any kind of technological evolution or current trends.

There is no doubt about the truth of this statement. However, another kind of problem presents itself now. The `MIX` computer is, well, . . .imaginary. A reader cannot experiment with it, or even have a go at solving the exercises by sitting in front of a real computer terminal and writing programs. Nor even can one be certain that a particular solution to an exercise is correct, unless one checks the answer to the exercise, and even then, there may be many different solutions to a problem, but the answers present only one. Unless of course the reader is inclined to simulate `MIX` in his or her head, or on a piece of paper. And these may very well have been Knuth's intentions for his readers. After all, there is no doubt that the reader that will get the most out of the books is the one who is patient and dilligent enough to go through this kind of process. But it still is a daunting task, often too frustrating, and for many people in many ways, a distraction of the kind that Knuth wanted for his readers to avoid in the first place.

Apparently, these issues have been prevailing for quite some time, since many people have developed simulators for Knuth's mythical machine over the years. The software you now have in your hands is another such attempt. But why bother making yet another simulator, you may ask? That is a good question. This present attempt in no way claims to solve any problems that previous implementations do not. As a matter of fact, it may have less bells and whistles than most. It might be less friendly, less complete, or even slower in its calculations. But it delivers, correctly. And what has actually been implemented, was done exactly by the book, literally speaking.

The question remains, then. Why another `MIX` simulator? Here is the answer: for the fun of it. And the learning experience. Writing such a piece of software, however trivial it may seem, presents a number of great challenges. It is an exercise in compiler design, input/output, data structures and more. A computer simulator is a platform which, by definition, provides its clients with *everything that the actual computer provides*, imaginary or not. Furthermore, it must be done in the best possible way given the programming tools.

The MIXWare are written in Microsoft Visual C# v3.5, which is a pure object-oriented language.

Athens, Greece —G. T.
June 2009

'Tis the gift to be simple, 'tis the gift to be free.
— JOSEPH BRACKETT, *Simple Gifts* (1848)

Everything should be made as simple as possible,
but no simpler.
— ALBERT EINSTEIN, *On the Method of Theoretical Physics* (1933)

1

MIX—
The MIX Simulator

`MIX`, the `MIX` simulator, incorporates all of the features described in *TAOCP, sec. 1.3.1*, with the exception of floating-point arithmetic (which is not described in that section anyway, except by name). No extensions have been implemented, specifically there is no support for indirect addressing or for interrupts. These extensions are planned for a future version.

Here's a quick overview of a `MIX` computer's features:

- 4000 words of memory;
- two word-sized registers called A and X;
- six signed registers of size two bytes each, called I1 through I6;
- one unsigned register of size two bytes, called J;
- a comparison indicator, which can have the value GREATER, EQUAL or LESS;
- an overflow flag;
- an instruction set of about 150 instructions;
- 21 optional input/output devices: eight tape drives, eight disks, a card reader, a card punch, a line printer, a terminal and paper tape, all of which are asynchronous.

Each `MIX` word is comprised of five bytes and a sign. One byte is equal to six bits, thus being able to have 64 distinct values.

1.1. RUNNING THE SIMULATOR

The `MIX` simulator can either run interactively or non-interactively. In non-interactive mode, it simply loads a compiled `MIX` program and runs it, producing whatever output the program is meant to produce, and then terminates. To run `MIX` non-interactively, type

```
MIX --deck [filename]    or    MIX --binary [filename]
```

where ⟨filename⟩ is optional and if not given the standard input is assumed. The `--deck` and `--binary` switches instruct `MIX` to interpret the input as punched cards (`--deck`) or a binary memory dump (`--binary`).

In interactive mode, one may load and run programs, or execute them step by step, or even set execution breakpoints and inspect the machine's state—the values of the registers, the comparison indicator, the program counter *etc.* To run the simulator interactively, simply type `MIX` at the command prompt. The simulator welcomes you:

```
This is MIX v0, (c) 2009 George Tryfonas
An mplementation of the machine described by Don Knuth.

Type '?' or 'help' at the prompt for instructions.
>
```

The `>` character is the simulator's prompt for your next command. You may quit the simulator by typing '`quit`' or '`bye`'. Commands are case-insensitive and can be categorized as follows:

- execution commands (take care of loading programs to be executed and handle program execution);
- inspection commands (for inspecting the contents of simulated memory and registers);
- state altering commands (that allow you to change the contents of simulated memory and registers);
- device management commands (for inspecting the state of the devices and redirecting input/output).

The following sections provide a detailed overview of the available commands.

1.2. EXECUTION COMMANDS

There are several commands that control the simulator's program loading and execution. Here is a list of them, and what they do.

`load <filename>` Loads a binary memory dump from the file ⟨filename⟩ and sets up the program execution. Firstly, it sets `MIX`'s memory contents to the contents of the memory dump; and then it sets the contents of the program counter to the location of the instruction where execution is to begin.

When the simulator loads the memory dump, control is returned to you after a confirmation message. For example:

```
> load primes.dump
Loaded 40 word(s). PC set to 3000
>
```

at which point you may begin execution, set breakpoints, *etc.*

`loaddeck <filename>` Instructs the simulator to load the first card from the card deck specified in ⟨filename⟩ and begin execution at location 0. This implies that the card reader device (device 16) is redirected to ⟨filename⟩. Since program execution begins immediately, you must set the desired breakpoints, if any, before using '`loaddeck`'.
The reason for this is that '`loaddeck`' behaves in the way described in *TAOCP, sec. 1.3.1, ex.* 26: the first two cards in the deck contain a loading routine, whose purpose is to load the rest of the program into the correct memory locations and jump to the program beginning. Therefore, no part of the program itself is loaded until this loader routine has executed completely.

`go` or `run` Begin execution at the location pointed by the program counter. Execution does not stop unless it reaches a breakpoint.

`step` Execute the instruction pointed by the program counter. Only one instruction is ever executed.

`show time` This command shows how much time has been spent on program execution. Each `MIX` instruction takes a specific amount of simulated time to execute, so for instance the PRIMES program takes $190908u$, where u is an unspecified time unit.

`set breakpoint` Set a breakpoint at a memory location. The command '`set breakpoint <x>`' (or '`set bp <x>`' for short) sets an execution breakpoint at memory location ⟨x⟩. ⟨x⟩ can be a number from 0 to 3999 or, if you loaded a memory dump, a symbol. Each breakpoint that gets `set` is assigned a unique integer identifier.

`show breakpoint` Shows a list of all the breakpoints you have set, along with their unique identifier:

```
> set bp START
> set bp 3006
> show bp
0 @ 3000
1 @ 3006
>
```

In this example, `START` is a symbol for the value 3000. You may also issue `show bp <n>` where ⟨n⟩ is an integer, which will show only the breakpoint whose unique identifier is equal to ⟨n⟩.

`clear breakpoint` Clears a breakpoint that has previously been set. The syntax is `clear breakpoint [n|all]`, where ⟨n⟩ is an integer. If ⟨n⟩ is specified, then the breakpoint whose unique identifier is equal to ⟨n⟩ is removed from the breakpoint list. Otherwise, if ⟨n⟩ is not specified, or if '`all`' is specified instead of ⟨n⟩, all the breakpoints in the breakpoint list are removed.

1.3. INSPECTION COMMANDS

The simulator gives you the chance to inspect the machine's state using a host of inspection commands, all of which begin with '`show`':

`show symbols` When a memory dump has been loaded (and only a memory dump), you may take a look at the symbol table by issuing the '`show symbols`' command. You may also issue a '`show symbol x`', where ⟨x⟩ is a symbol name, to inspect just the value of that particular

symbol. Example:

```
> show symbols
L      = [+|00|00|00|07|52|] =  500 = '   G@'
OUTDEV = [+|00|00|00|00|18|] =   18 = '    Q'
PRIME  = [-|00|00|00|00|01|] =   -1 = '    A'
BUF0   = [+|00|00|00|31|16|] = 2000 = '   1O'
BUF1   = [+|00|00|00|31|41|] = 2025 = '   1,'
START  = [+|00|00|00|46|56|] = 3000 = '   *?'
|2-1|  = [+|00|00|00|46|59|] = 3003 = '   *?'
|4-1|  = [+|00|00|00|46|62|] = 3006 = '   *?'
|6-1|  = [+|00|00|00|47|00|] = 3008 = '   / '
|2-2|  = [+|00|00|00|47|08|] = 3016 = '   /H'
|2-3|  = [+|00|00|00|47|11|] = 3019 = '   /J'
|4-2|  = [+|00|00|00|47|12|] = 3020 = '   /K'
TITLE  = [+|00|00|00|31|11|] = 1995 = '   1J'
=1-L=  = [+|00|00|00|32|02|] = 2050 = '   2B'
=3=    = [+|00|00|00|32|03|] = 2051 = '   2C'
>
```

`show memory` You may inspect the contents of `MIX`'s memory by issuing the '`show memory`' command (or '`show mem`' for short). This command, if given without any arguments, shows the contents of all of `MIX`'s memory, starting at location 0 and up to 3999. If, however, you specify a range, you will get a portion of the memory contents:

```
> show mem 3000 3010
MEMORY CONTENTS (3000 TO 3010)

@3000: [+|00|00|00|18|35|] =       1187 = '   Q5'
@3001: [+|32|02|00|05|09|] =  537395529 = '2B EI'
@3002: [+|32|03|00|05|10|] =  537657674 = '2C E?'
@3003: [+|00|01|00|00|49|] =     262193 = ' A  $'
@3004: [+|07|51|01|05|26|] =  130814298 = 'G>AEW'
@3005: [+|47|08|00|01|41|] =  790626409 = '/H A,'
@3006: [+|00|02|00|00|50|] =     524338 = ' B  <'
@3007: [+|00|02|00|02|51|] =     524467 = ' B B>'
@3008: [+|00|00|00|02|48|] =        176 = '   B='
@3009: [+|00|00|02|02|55|] =       8375 = '  BB''
@3010: [-|00|01|03|05|04|] =    -274756 = ' ACED'
>
```

Ranges are pairs $\langle x \sqcup y \rangle$, for instance, $3000 \sqcup 3010$ as in the above example. However x or y are not limited to integers: symbols may be specified as well, as in `show mem START 3010`, which produces exactly the same output as above, given that `START` is a symbol whose value equals 3000.

You may also wish to see a disassembly of the memory contents in the range specified. To do this, type '`with disassembly`' (or '`with dasm`' for short) after a range—and only

after a range:

```
> show mem START 3010 with disassembly
MEMORY CONTENTS (3000 TO 3010)

@3000: [+|00|00|00|18|35|] =       1187 = '   Q5'       IOC 0(2:2)
@3001: [+|32|02|00|05|09|] =  537395529 = '2B EI'       LD1 2050
@3002: [+|32|03|00|05|10|] =  537657674 = '2C E?'       LD2 2051
@3003: [+|00|01|00|00|49|] =     262193 = ' A  $'       INC1 1
@3004: [+|07|51|01|05|26|] =  130814298 = 'G>AEW'       ST2 499,1
@3005: [+|47|08|00|01|41|] =  790626409 = '/H A,'       J1Z 3016
@3006: [+|00|02|00|00|50|] =     524338 = ' B  <'       INC2 2
@3007: [+|00|02|00|02|51|] =     524467 = ' B B>'       ENT3 2
@3008: [+|00|00|00|02|48|] =        176 = '   B='       ENTA 0
@3009: [+|00|00|02|02|55|] =       8375 = '  BB''       ENTX 0,2
@3010: [-|00|01|03|05|04|] =    -274756 = ' ACED'       DIV -1,3
>
```

`show state` Shows `MIX`'s internal state. It produces a list with the contents of all registers, rA, rX, rJ, rI⟨1–5⟩, the overflow flag, the comparison indicator and the program counter:

```
> show state
 A: [+|00|00|00|00|00|] = 0 = '     '
 X: [+|00|00|00|00|00|] = 0 = '     '
 J: [+|00|00|00|00|00|] = 0 = '     '
I1: [+|00|00|00|00|00|] = 0 = '     '
I2: [+|00|00|00|00|00|] = 0 = '     '
I3: [+|00|00|00|00|00|] = 0 = '     '
I4: [+|00|00|00|00|00|] = 0 = '     '
I5: [+|00|00|00|00|00|] = 0 = '     '
I6: [+|00|00|00|00|00|] = 0 = '     '
Overflow: False
CI: EQUAL
PC: 3000
>
```

You may also issue the command '`show <reg>`' where ⟨reg⟩ can be any one of the above registers or flags to see the contents of that specific item.

`verbose` Toggles verbosity on and off. When verbose is on, after each '`step`' command, and after program termination following a '`run`' command, a '`show state`' is executed automatically. You may check whether verbose is on or off by issuing the '`show verbose`' command.

1.4. STATE ALTERING COMMANDS

There exist various commands which alter the simulator's state between execution. All of these commands begin with '`set`'. Here is a list of these:

`set memory` Alters the contents of a single memory cell. The syntax is `set memory <loc> <data>`, where ⟨loc⟩ is a memory location and ⟨data⟩ is an integer value. As is the case with `show memory`, the ⟨loc⟩ parameter may be a symbol.

`set <reg> <value>` Alters the contents of one of `MIX`'s registers. The ⟨value⟩ parameter is an integer while the ⟨reg⟩ parameter can be one of the nine `MIX` registers or the program counter, in the form `rA`, `rX`, `rJ`, `rI1` to `rI6` and `PC`.

1.5. DEVICE MANAGEMENT COMMANDS

`MIX` has an array of input/output device units that programs may use. The most common one is the line printer, however `MIX` is also supplied with tape drives, disks, a card reader, a card punch

and a paper tape. All of these devices have been implemented and you may inpspect or alter their status using the following commands:

`show devices` Shows a list of `MIX`'s devices. For example:

```
> show devices
UNIT #0: NOT INSTALLED
UNIT #1: NOT INSTALLED
UNIT #2: NOT INSTALLED
UNIT #3: NOT INSTALLED
UNIT #4: NOT INSTALLED
UNIT #5: NOT INSTALLED
UNIT #6: NOT INSTALLED
UNIT #7: NOT INSTALLED
UNIT #8: NOT INSTALLED
UNIT #9: NOT INSTALLED
UNIT #10: NOT INSTALLED
UNIT #11: NOT INSTALLED
UNIT #12: NOT INSTALLED
UNIT #13: NOT INSTALLED
UNIT #14: NOT INSTALLED
UNIT #15: NOT INSTALLED
UNIT #16: NAME: CARD_READER, BLOCK SIZE: 16, BACKING STORE: CONSOLE
UNIT #17: NOT INSTALLED
UNIT #18: NAME: LINE_PRINTER, BLOCK SIZE: 24, BACKING STORE: CONSOLE
UNIT #19: NAME: TERMINAL, BLOCK SIZE: 14, BACKING STORE: MEMORY
UNIT #20: NOT INSTALLED
>
```

which displays various kinds of information for each device. In this specific example, there are no installed tape or disk units, card punch or paper tape. However the card reader, the line printer and the terminal are present. The first two are connected to the console—so if a program sends output to the line printer it is displayed on the screen, and if it requests input from the card reader it is read from the keyboard. You may also issue the command `show device <d>` where ⟨d⟩ is an integer from 0 to 20, indicating a unit number, to get information on that specific device unit.

`redirect` Redirects a device to a specified file. The syntax is:

`redirect <d> <filename>` or `redirect <d> console`

where ⟨d⟩ is an integer from 0 to 20 representing the device unit. If a filename is specified then input/output is done using that file. Otherwise if you say `console` then input/output is done using the standard input/output. You cannot redirect a device to a file called `console`.

`set device` Alters the synchronicity of a device. The syntax is

`set device <d> sync` or `set device <d> async`

where ⟨d⟩ is the device unit number. `sync` and `async` specify the mode in which device ⟨d⟩ will operate. By default each device is asynchronous, which means that as soon as an input/output operation is executed by a running program, control is returned to that program and the input/output operation continues to execute in the background. Note that this is not in the formal specification of `MIX`.

Japhy, do you think God made the world to amuse himself because he was bored? Because if so he would have to be mean.

— JACK KEROUAC, *The Dharma Bums* (1958)

2

MIXASM—
The MIX Assembler

The assembler implements all the features of the `MIXAL` language as detailed in *TAOCP, sec. 1.3.2*. See that book section for the definitive description of the language. The assembler follows the book faithfully, including the format of the `ALF` pseudo-operation. Appendix A has a detailed treatment of the exact semantics of this particular assembler implementation.

2.1. RUNNING THE ASSEMBLER

The assembler is invoked using the following syntax:

```
MIXASM [arguments] [input-filename]
```

where each ⟨argument⟩ is of the form:

`--long-arg:value` or `-short-arg:value`

where for each `--long-arg` there exist one or more `-short-arg`s that perform the equivalent function but are, well, shorter. The arguments' `value` part is, depending on the argument, either required, optional, or not applicable. Here's a list of the possible arguments and a description of what they do. The applicability of the `value` part is also discussed here.

Long Form	*Short Form*	*Description*
`--output:filename`	`-o:filename`	redirects output to `filename`.
`--symtab[:filename]`	`-s[:filename]`	produces the symbol table. If `filename` is specified, the symbol table is written to that file, otherwise it is sent to standard output.
`--list-file:filename`	`-lf:filename`	produces a listing and writes it to `filename` (see below).
`--format:...`	`-f:...`	defines the output format. Can be either `Card` or `Binary` (default is `Binary`, see below).
`--append-deck:filename`	`-a:filename`	appends the `filename` specified to the output card deck (applicable only when the output format is `Card`).
`--pretty-print`	`-pp`	pretty prints the symbol table and/or the list file (if requested) for use with TEX.
`--version`	`-v`	display version information and exit.
`--help`	`-h` `-?`	display a short help message and exit.

If `input-filename` is not specified then input is done from the standard input. If the `--output` parameter is omitted then the compiler emits output to the standard output. The arguments are *not* case-sensitive. Here's a bunch of examples:

*Command**	*Comments*
`MIXASM -f:card primes.mixal`	compiles the file `primes.mixal` to a card deck and sends the output to the display
`MIXASM -f:card primes.mixal -lf:primes.tex -pp`	as above, but also pretty-prints the program listing for use with TEX, and saves it to the file `primes.tex`.
`MIXASM --output:primes.dump < primes.mixal`	compiles the file `primes.mixal` and writes the (binary) output to file `primes.dump`
`MIXASM -f:card primes.mixal | MIX --deck`	compiles the file `primes.mixal` as a card deck and starts non-interactive simulation

*These commands are, of course, all one-liners.

The last couple of examples are worth particular mention. They bring to the light the joy of pipelining, which so many UNIX users have come to worship. Here's why the third example works: `MIXASM`, unless told otherwise, expects that the file to be assembled comes from the standard input. But here we redirect the standard input to the file `primes.mixal`. In the fourth example, we redirect `MIXASM`'s output to the input of `MIX`, the simulator. Remember that the `--deck` argument informs the simulator to interpret the input as a card deck.

The following sections provide a description of the rest of `MIXASM`'s features (well, except for the dead obvious ones).

2.2. OUTPUT FORMATS

`MIXASM` can produce two kinds of output, namely a binary memory dump or a card deck. Memory dumps are not human-readable and are meant only to be loaded and executed by `MIX`, the simulator (*cf.* Chapter 1 and Appendix A for techincal details). Card deck output, however, is in ASCII format and can be read by humans too—well, at least those that can make sense of it.

A card deck is a text file containing 80-column lines. Each line represents a punched card and is read by `MIX`'s card reader. Each `MIX` program that has been compiled to a card deck adheres to the specification given in *TAOCP, sec. 1.3.1, ex.* 26: The first two cards contain a loading routine which serves to load the rest of the program into the correct locations in memory and start execution. Note that programs that are compiled to cards must start at a memory location greater than 100, a restriction that does not apply when a program has been compiled to binary. This is due to the fact that these locations are occupied by the loading routine itself.

When a `MIXAL` program is compiled to a card deck, one may specify the `--append-deck` argument, which in effect appends extra cards to the output deck. For example, typing

```
MIXASM perms.mixal -o:perms.deck -f:binary -a:perms-input.deck
```

will produce the card deck `primes.deck` which will contain the compiled program plus the cards found in the `primes-input.deck` card deck (all these files can be found in the *Examples* directory of the MIXWare distribution). This latter deck contains sample input data for the permutations program.

There is actually a (good) reason behind this. If you use the simulator's `loaddeck` command to load the compiled program, the loader routine, program and all, will execute immediately, giving you no chance to set up the card reader with any input data (if there is any, of course). Therefore, the `--append-deck` argument gives you a chance to set things up beforehand.

2.3. SYMBOL TABLES

The argument `--symtab` instructs `MIXASM` to produce a symbol table. If the `filename` parameter is given to the argument, then the symbol table is written to that file, otherwise it is sent to the standard output. The symbols in `MIX`'s assembly language are divided into three categories and so is the symbol table: (1) the main symbols, which are `EQU` constants and instruction labels; (2) the local symbols, i.e., labels like `1H` and `9H` in the assembly source, and; (3) literals, i.e., the address parts of instructions with the format `=23//1=`.

Each segment of the symbol table is divided into three columns. The first column is the symbol name. The second one is the symbol's value as it gets stored in a `MIX` word. The third column is the symbol's decimal value. Main symbols are copied verbatim as described. Local symbols and literals, however, receive special treatment.

Local symbols get renamed according to the following rule: when a label of the form n`H` is encountered, where n is a digit, then that label is renamed to `|`n-x_n`|`, where x_n is initially equal to 1. If the assembler encounters the label n`H` again, then it increases x_n by 1 and repeats the renaming process. For example: when the assembler first encounters a label `2H`, it sets x_2 equal to 1 and renames the label to `|2-1|`. If the assembler encounters the label again further down the source code, it sets x_2 to 2 and renames the label to `|2-2|`.

Literals do not appear as symbols in the `MIXAL` source. They appear as the address part of `MIXAL` instructions, for example in the instruction `LD2 =3=`, the address `=3=` is a literal constant. This has the effect of creating a constant with the value 3 and creating an internal

symbol whose value is equal to the address that constant is stored. `MIXASM` names that symbol as the literal—in the given example that symbol will be called `=3=`. Note that this implies that only one constant is created, regardless of how many times the literal `=3=` appears in the source code. There is, however, one exception to this rule, and that is when the literal contains the location counter, *e.g.* `=*+3=`. See Appendix A for more details.

2.4. LIST FILES

`MIXASM` can produce list files similar to the program listings in *The Art of Computer Programming*: for each line of source, the listing contains the location in memory where that line is assembled, the assembled word, the line number and the line itself. If the instruction is a `MIXAL` pseudo-instruction, the first two parts are omitted. The program listing is followed by the symbol table as described in the previous section. A typical `MIX` instruction is stored in a `MIX` word as follows:

±		AA	I	F	C

where `AA` is the address, ranging from 0 to 3999, `I` is the index, `F` is the field and `C` is the instruction's opcode. For more information on how these fields are used see *TAOCP, sec. 1.3.1.*

The list file that `MIXASM` produces is stored in a text file that's specified as an argument to the `--list-file` parameter. If the `--pretty-print` option is given to the assembler, the list file is written in TEX format instead. You may refer to Appendix A for a full listing of a short example program, to examine what such listings look like when they are typeset by TEX.

When someone says: 'I want a programming language in which I need only say what I wish done', give him a lollipop.

— ALAN PERLIS, *Software Metrics* (1981)

Trying to outsmart a compiler defeats much of the purpose of using one.

— KERNIGHAN & PLAUGER, *The Elements of Programming Style* (1978)

If you can't do it in Fortran, do it in assembly language. If you can't do it in assembly language, it isn't worth doing.

— ED POST, *Real Programmers Don't Use Pascal* (1983)

3

Simulator Implementation

We shall inspect, in this and the next chapter, the various details behind the simulator and the assembler. These chapters are concerned a bit more with the hairy implementation details and a good working knowledge of object oriented programming is assumed. The actual implementation language is C#, but we shall avoid getting into actual programming details. However, will will talk about classes, structures, methods and inheritance, and the reader is assumed to know what each of these constructs means.

Note that all the information we shall provide is "back-end" information. All the implementation details related in the following sections do not deal with the user interface, and with good reason. These back-end classes are stored in a class library called `MIXLib`. The two principal programs, `MIX` and `MIXASM`, use these library classes to perform their function, in addition to providing a console user interface. But they may just as well have been implemented as GUI programs, since nothing in the `MIXLib` library is bound to a particular user interface type.

We start our discussion with the simulator, which is probably the simplest of the two programs, therefore easier to grasp.

The simulator closely follows the specification given in *TAOCP sec. 1.3.1.* Everything described in that section has been implemented in its entirety. Perhaps the most challenging task of the implementation process is the fact that `MIX` is an archaic design. Many of its components do not exist today. Certainly there are no card readers, card punches or tape drives. Printers have stopped being line-oriented. Byte size has been standardized and it certainly isn't six bits. Computers do generally distinguish between $+0$ and -0 however (see the IEEE 754 standard for floating point numbers).

Object oriented design helps hide such minor setbacks. We recognize the abstractions behind the computer specification and create classes to encapsulate them. For example, we know that a word represents a number, of any radix, whose minimum and maximum limits are known, and on which we may perform simple arithmetic. We are not concerned with how many bits represent one word. This is a minor implementation detail which is kept hidden within the encapsulating class.

The same goes with the other important parts of the `MIX` design. Breaking a `MIX` computer to its abstract components, we are left with the following important classes:

`MIX` Abstraction	Encapsulating Class	Remarks
Word	`MIXWord`	A word of memory: five bytes and a sign
Instruction	`MIXInstruction`	An abstract `MIX` instruction, like `LDA`. Consists of an `Execute()` method, which when overridden performs the actual instruction.
Device	`MIXDevice`	An abstract `MIX` device. Descendants are the concrete `MIX` devices, such as disk drives, the line printer, *etc.*
`MIX`	`MIXMachine`	The actual `MIX` computer. Contains properties such as `Memory`, a set of register properties, the device array *etc.*

Table 3.1 `MIX` Description

Figure 3.1 shows the overall structure of the `MIX` simulator design without many details. The diagram drives home the fact that the `MIXMachine` class is the one class that brings together the various parts that make up `MIX`.

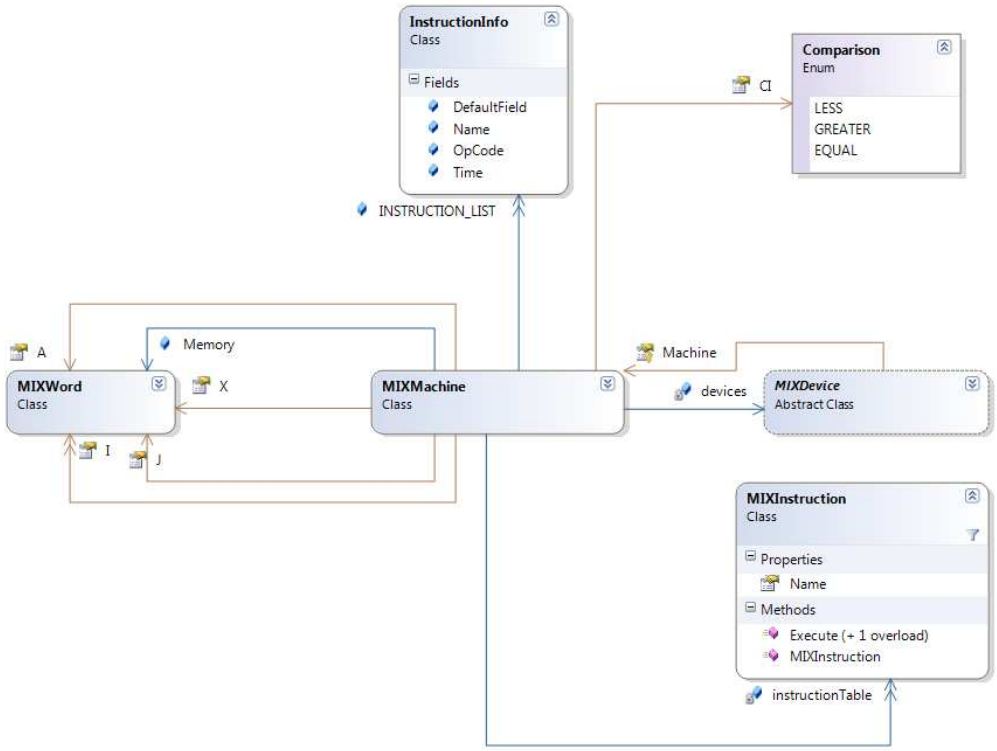


Figure 3.1 `MIX` Simulator class diagram

We now shall discuss the various parts of the `MIX` architecture as evidenced in the class diagram in a bit more detail.

3.1. WORDS

`MIX` words are used throughout the specification of the `MIX` machine: `MIX`'s memory is an array of `MIX` words; the `MIX` registers hold values that fit into `MIX` words (in fact, the index registers, rI1–rI6, as well as rJ, are shorter, but regarding them as such is a good simplification); each `MIX` instruction is encoded as a word; device I/O is done by sending and receiving words to and from the devices. It is a ubiquitous abstraction.

It follows that such an ever-present abstraction will make very frequent appearances in any kind of implementation. To this end, we go at great length to define the `MIXWord` class in such a way that it may be used as naturally as the standard integral types of the implementation language. The `MIXWord` class can be constructed from the language's existing integral values (`int` or `byte`) and can be cast back to them. In the case of `int`, an explicit cast is needed because the range of a 32-bit integer is greater than that of the 30-bit `MIX` word. In addition, we overload the standard C# arithmetic and relational operators in such a way that we may write

```
if (w1 > w2)
    w1 += 100;
```

Note that because of our type-coercion operators we are allowed to add a `byte` (such as 100) to a `MIXWord`.

An important characteristic of a `MIX` word is that we can examine the value of, and assign to, byte slices of it. For example, we can assign the value '1000' to bytes 2–3 of a word, leaving the rest of the word the same. Or, we can read the value of bytes 2–4, ignoring the other bytes. These byte slices are called *fields*.

Because fields are contiguous, it is simple to encapsulate them in the `MIXWord` class by overloading the `this[]` property. We provide two overloads:

1. `public byte this[byte b]` assigns to or reads the value of a single byte of the `MIXWord`; and

2. `public int this[byte l, byte r]` assigns to or reads the value of a field of the `MIXWord`, defined by the range l–r, where l is the most significant byte of the field and r is the least significant byte.

therefore providing the means to write code such as:

```
MIXWord w = new MIXWord(5000);
w[2,3] = 1000;
int a = w[1,4];
```

In this way we make the implementation of the `MIX` instruction set significantly easier, because the effective address of most instructions takes a field specification into account.

Finally, we override the `ToString()` method to return a convenient textual representation of the word, useful in program listings and in memory dumps.

3.2. INSTRUCTIONS

A `MIX` instruction is encoded in a word according to the conventions specified in *TAOCP sec. 1.3.1.* The `MIX` central processing unit decodes the word and, according to the C- and F-fields of the instruction word (see also the section on list files in the previous chapter), performs a specified operation using `CONTENTS(M)`, the latter defined as

$$\text{CONTENTS}(M) \equiv A(F) + \text{rI}(I)$$

where A, F and I are the A-, F- and I- fields of the instruction word respectively.

To encapsulate this abstraction, we create two separate classes: the `MIXInstruction` class represents the execution-related aspects of an instruction, while the `InstructionInfo` class represents its informational aspects.

There are two distinct paths we may follow in order to represent the execution of an instruction in the `MIXInstruction` class. We can either: (1) create an abstract superclass and inherit it 157 times, once for each instruction; or (2) create a concrete class and instantiate it with a different method delegate 157 times, again once for each instruction. The first method encapsulates the abstraction at the class level while the second one at the instance level.

There are no significant advantages if we choos one method over the other, apart from the fact that the first one leads to an explosion of subclasses, each one of which must be named differently, while the second method can be implemented using anonymous delegates (λ-expressions). We chose the second method for this simple reason.

Each `MIX` instruction takes a specified amount of time to complete. In addition, if the F-field of an instruction word is left unspecified by the programmer, it takes a default value. Finally, each instruction stored in a word is identified by the value of its C-field, and sometimes the F-field discriminates it further. This kind of identification information is stored in instances of the `InstructionInfo` class.

3.3. DEVICES

As required by the specification, the `MIX` computer features asynchronous I/O. All devices communicate with the computer by sending and receiving blocks of data, either as full `MIX` words (in the case of disks and magnetic tape) or in character mode (for the other devices). In character mode the sign of the `MIX` word is ignored and bytes 1–4 are interpreted as a characters from the `MIX` character set. Block size varies from device type to device type.

This abstraction is encapsulated in the `MIXDevice` abstract class and its subclasses, which are illustrated in Figure 3.2 below:

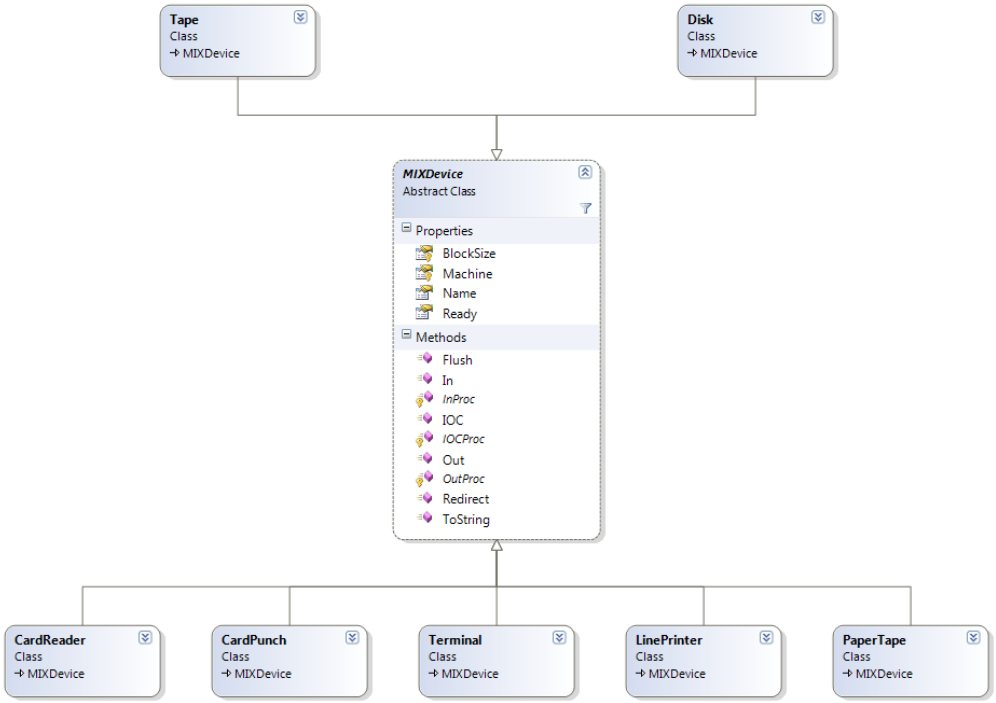


Figure 3.2 `MIX` Devices class diagram

The `MIXDevice` abstract class contains the core interface to a `MIX` device. Its important methods are `In()`, `Out()` and `IOC()`. These methods simply call the `InProc()`, `OutProc()` and `IOCProc()` methods respectively in a new thread of execution—this provides the asynchronous device operation required by the specification.

Concrete children of the `MIXDevice` class override these last three methods to provide the functionality of the implemented device. Some devices are meant only for input, others only for output, in which case the `InProc()` or the `OutProc()` methods respectively have null implementations.

The `Ready` property is the device's "status indicator". It is cleared by the `InProc()`, `OutProc()` and `IOCProc()` methods whenever they begin execution, and reset when they end. It is important that the body of these methods is written inside a critical region to provide the appropriate synchronization. C#'s `lock` primitive is used for this purpose.

Devices are actually implemented using streams as backing store. Upon instantiation of a `MIXDevice` subclass, we specify a `Stream` instance for such use, but it's possible to redirect a device by simply changing the underlying stream, to a `MemoryStream`; the standard input/output streams; a `FileStream`; `null` if the device is unbacked, making the device behave as if it did not exist at all. The `Redirect()` method of the `MIXDevice` class is provided exactly for this purpose.

3.4. THE MACHINE

As evidenced on Figure 3.1, the `MIXMachine` class ties all the helper classes. Details such as the `MIX` character map, the register set (a set of `MIXWord` properties), the overflow/comparison indicators, the memory (an array of `MIXWords`), the device array and the instruction set are all implemented as fields/properties in this class.

3.4.1. Initialization

`MakeCharTable()`, `MakeInstructionTable()` and `MakeInstructionList()` set up respectively the `MIX` character set as a `Dictionary<char, byte>`, the execution procedures of each instruction as an array of `MIXInstruction` instances, and the `MIX` instruction set as an array of `InstructionInfo` instances. These methods are called in order by the `InitMachine()` method, which also clears the memory array and the register values, effectively performing a machine

"reboot". Each `MIX` device is stored in the appropriate location in the `devices` array. The `InitDevices()` method resets this array to its default values.

3.4.2. **Loading and Execution**

A `MIXMachine` instance can load a binary memory dump or a card deck as they are output from the assembler by means of the `LoadImage()` and `LoadDeck()` methods. The former simply populates the `Memory` array at appropriate locations with a list of instruction words (as `MIXWord` instances). The latter reads the first card from the deck in the card reader and begins execution immediately. It is implied that this card should contain a loading routine, otherwise the outcome is undefined.

Execution is performed by the `Step()` and `Run()` methods. The first one executes the instruction in `Memory` at the location of `PC` (a special property which stands for Program Counter) and advances the clock by the time required by that instruction.

There exists a special boolean property called `Running` which governs execution. The `Run()` method calls `Step()` repeatedly until this property becomes false, or until the `PC` points to a location beyond memory bounds. The `Running` property is set to false by the `HLT` instruction, or by the `Step()` method if it has reached a breakpoint location. Breakpoints are stored in a `List<int>`, whose contents are memory indices at which execution should stop.

One Ring to rule them all, One Ring to find them,
One Ring to bring them all and in the darkness bind them
In the Land of Mordor where the Shadows lie.

— J. R. R. TOLKIEN, *The Fellowship of the Ring* (1954)

4

Assembler Implementation

As with most compilers, `MIXASM` is divided into three important units:

1. The scanner;
2. The parser; and
3. The code generator.

We shall consider each unit in turn, in this order. But first, let us restate some important language rules concerning the symbols that may appear in the source code, in order to justify some of the decisions that have been taken. For a full treatment of the language rules, albeit somewhat informal, see *TAOCP, sec. 1.3.1, pp.* 153–156.

Format Each line in a `MIXAL` source code file may be empty, in which case it is ignored, or nonempty in which case it represents a single instruction and it is divided in three fields: (1) the `LOC` field, which may be blank; (2) the `OP` field, which is never blank; and (3) the `OPERAND` field, which may also be blank. Each field is separated by one or more spaces or tabs (with one exception that we shall discuss later). Anything on a line that follows the `OPERAND` field is ignored. Lines that begin with the character '`*`' are ignored completely.

The `OPERAND` field, when not blank, is divided further into: (1) the `ADDRESS` part; (2) the `INDEX` part; and (3) the `FIELD` part.

Numbers A *number* is a sequence of one to ten digits. Examples are `42` and `0005345`.

Symbols A *symbol* is a sequence of one to ten digits or letters, containing at least one letter. For the purposes of this discussion, a symbol plays the rôle of a C identifier. Note however that, unlike most modern computer languages, the requirement that a symbol start with a letter is relaxed. Examples are '`HELLO`', '`GR8`' and '`10FOUR`'. Neither '`TEN_FOUR`' nor '`VERYLONGSYMBOLOF28CHARACTERS`' are valid symbols.

Keywords A *keyword* is a symbol that appears in the `OP` field of an instruction line and matches an item from `MIX`'s instruction set. Examples are '`LDA`', '`STZ`' *etc.*

Pseudos A *pseudo-operation* is a symbol that appears in the `OP` field of an instruction line but does not generate a `MIX` instruction. Instead it governs the assembly process in important ways. There are five `MIXAL` pseudo-operations, namely '`ORIG`', '`EQU`', '`CON`', '`ALF`' and '`END`'.

Strings A symbol of five characters that appears next to an '`ALF`' pseudo-operation is called a *string*. Strings form an exception to the rule on dividing an instruction line, because they must be separated from the '`ALF`' pseudo-operation by *exactly* one or two spaces. If they are separated by one space then the next character must be non-blank. Otherwise any five characters from `MIX`'s character set may follow. For example: '`ALF␣FIRST`' defines the string '`FIRST`'; '`ALF␣␣␣FIVE`' defines the string '`␣FIVE`'.

Locals A symbol that appears in the `LOC` field and has the form n`H`, where $0 \leq n \leq 9$ is a digit, has special meaning and is called a *local symbol*. Similarly, a symbol that appears in the `ADDRESS` field and has the form n`F` or n`B`, where $0 \leq n \leq 9$, also has special meaning and is called a *local reference*. We shall discuss local symbols and references in a second.

Symbols on each source code line are either *defined symbols* or *future references*. Defined symbols are the symbols that have already made an appearance in the `LOC` field of a previous line, and when they do we shall call then *labels* unless they appear before an `EQU` pseudo-operation.

A defined symbol has a value, called its *equivalent*, which is equal to the value of $\circledast$ for the line on which this defined symbol appears as a label. However, if the defined symbol appears in the `LOC` field of an `EQU` pseudo-operation, its equivalent value is defined to be equal to the value of the `ADDRESS` field of the instruction. Symbols that are not defined symbols are called *future references*. It is illegal to define a symbol more than once in the source code.

Each local symbol may appear multiple times in the source code, however this does not violate the multiple definitions rule, because each local symbol is internally given a different, unique name, and that internal name becomes the defined symbol. For $0 \leq n \leq 9$, the local reference n`B` refers to the most recently defined local symbol n`H`, while n`F` refers to the next local symbol n`H` that will appear later on and therefore is always a future reference.

4.1. THE SCANNER

We now turn our attention to the `MIXASM` scanner, which is the part of the assembler that reads an entire source code (input) file and tries to make sense of it. To this end, it scans the input on a line-by-line, character-by-character basis, ignoring parts of it that should be ignored, and forms strings according to specific rules. Instead of returing these strings *per se*, it categorizes them from a finite set of categories, called *tokens*, and returns a *token stream*, which is a stream of 'categorized strings'—plus some other book-keeping information, like the token's position in the source file *etc.*

The most important task in designing the scanner is choosing an appropriate token set. Possibly the most naïve strategy would be to go with what the language rules state in plain English. For a line of `MIXAL` code, according to the above informal description, and to Don Knuth's description in *TAOCP*, this would be the set

$$T_0 = \langle \textbf{symbol}, \textbf{number}, \textbf{string}, \texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{//}, \texttt{:}, \texttt{=}, \texttt{,}, \texttt{(}, \texttt{)}, \textbf{whitespace} \rangle$$

where **symbol**, **number** and **string** are as defined previously. However, this puts all the burden on the parser to make sense of what the scanner has recognized—and it is well known that parsers are tougher beasts than scanners to get to work correctly. For instance, is the symbol a label? Or is it a reference? Or maybe it is an instruction?

A better approach would be to let the scanner make such decisions as a first step, which is not a very difficult task anyway. It can be easily implemented by letting the scanner be in one of four states $\{\text{LOC}, \text{OP}, \text{OPERAND}, \text{STRING}\}$ and deciding the meaning of the symbol it just recognized. The scanner changes state as soon as it encounters whitespace, which is ignored. State changes happen in the order the states were listed, with the two last states sharing an equal position, *i.e.*, LOC→OP; OP→OPERAND, or LOC→OP; OP→STRING. Whitespace itself is ignored completely, unless it is part of a string, in which case it becomes part of the token anyway.

We make an extra distinction in the OP state: the token value is examined to ascertain whether it is a `MIX` instruction or a `MIXAL` pseudo-operation. The reason is that pseudo-operations are syntactically different than instructions, and recognizing them as such will better aid the parser to make sense of the `OPERAND` field. The token set then becomes:

$$T = \langle \textbf{label}, \textbf{keyword}, \textbf{equ}, \textbf{orig}, \textbf{con}, \textbf{alf}, \textbf{end}, \textbf{symbol}, \textbf{number}, \textbf{string}, \texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{//}, \texttt{:}, \texttt{=}, \texttt{,}, \texttt{(}, \texttt{)} \rangle$$

Note that the token set has been expanded with what amounts to 'imaginary' tokens—tokens that are lexically defined in exactly the same way as other tokens that already existed in our previous attempt: **label**, **keyword** and **equ**...**end** are lexically equivalent to the **symbol** token. These tokens, however, have a semantic meaning which generally is left for the parser to discover. We have 'cheated' here and let semantics creep into the scanner, but in effect things are simplified this way, and this is common to languages where whitespace or indentation is important, because the scanner gets to 'see' the input directly, a benefit parsers don't enjoy.

Taking all the above into consideration, we create the `Scanner` class, which is initialized with a string and recognizes `MIXAL` tokens. The available tokens are encapsulated in the `Token` structure whose most important properties are: (1) `Type`, of the `TokenType` enumeration type; and (2) `Text`, which is the actual token text. For example, the string '`42`' would be recognized as a token with `Type` ≡ `TokenType.Number` and with `Text` ≡ `"42"`.

Tokens are recognized from the input string and placed in the `Tokens` property of the `Scanner` class, which is of type `IEnumerable<Token>`. This gives us serval advantages. Firstly, by `using System.LINQ`, we may exploit LINQ's extened syntax to query the enumeration—which turns out to be unnecessary after all, but it is there should one need it. Secondly, and more importantly, the token enumeration is constructed using `yield return` and iterators, which gives us the power of coroutines, a feature that our implementation language does not possess out of the box, but which turns up to be extremely useful in our case, because in this way we only read the next token when we actually need it and no sooner.

4.2. THE PARSER

In general, the parser is the focal point in compiler writing, however important the other phases of the compiler may be. Getting the parser right is in many cases the key to efficiency as well as ease of further development. There exist a great variety of methods to develop parsers, as well as the associated literature. Methods range from top-down; bottom-up; no lookahead; n-token lookahead; table-driven; recursive; and all sorts of lovely combinations of these. Tools exist to help the compiler writer create his or her parser from an extended BNF grammar—most of these tools use the table-driven approach, but see `ANTLR` for an interesting tool that generates recursive-descent parsers. Other people prefer to roll their own.

The most important choice to make here is whether to write our own parser or use a compiler-compiler to create it. The choice made was to write our own, but there are several factors that influenced this decision:

- `MIXAL` is a truly simple language and using a general-purpose tool would be overkill;
- There are no adequate tools known to the author that are both simple to use and generate efficient code for our implementation language (except, perhaps, for `ANTLR`, which again has its own issues);
- Corollary of the above: it is usually the case that one has to take a break from the design/development of the compiler in order to learn how to use a particular tool, and, even more importantly, one may have to adapt one's design process to the peculiarities of that tool; (this remark, unfortunately, is relevant in other areas of application as well.)
- Compiler-compilers generate table-driven parsers which are usually $LALR(1)$. This is too much for `MIXAL`, for which a simple $LL(1)$ parser is sufficient; (again, see `ANTLR` for a compiler-compiler which generates recursive-descent $LL(k)$ parsers.)

So in view of the above, we create a recursive-descent parser which is tailored to `MIXAL`. The techniques are quite well-known, so we shall not go into much depth here. It is very important to note, however, that the `MIXAL` grammar, as stated informally in *TAOCP*, is not $LL(1)$ because it is left-recursive. We shall restate it here in extended BNF form, transformed to $LL(1)$, and the interested reader may refer to Appendix B for the language's syntax diagrams:

$$\langle\text{atom}\rangle ::= \textbf{definedsymbol} \mid \textbf{number} \mid \text{``*''} \tag{1}$$

$$\langle\text{binary op}\rangle ::= \text{``+''} \mid \text{``-''} \mid \text{``*''} \mid \text{``/''} \mid \text{``//''} \mid \text{``:''} \tag{2}$$

$$\langle\text{expression}\rangle ::= [\text{``+''} \mid \text{``-''}]\ \langle\text{atom}\rangle\ \{\langle\text{binary op}\rangle\ \langle\text{atom}\rangle\} \tag{3}$$

$$\langle\text{a-part}\rangle ::= [\langle\text{expression}\rangle \mid \langle\text{literal}\rangle \mid \textbf{futurereference}] \tag{4}$$

$$\langle\text{i-part}\rangle ::= [\text{``,''}\ \langle\text{expression}\rangle] \tag{5}$$

$$\langle\text{f-part}\rangle ::= [\text{``(''}\ \langle\text{expression}\rangle\ \text{``)''}] \tag{6}$$

$$\langle\text{w-value}\rangle ::= \langle\text{expression}\rangle\ \langle\text{f-part}\rangle\ \{\text{``,''}\ \langle\text{expression}\rangle\ \langle\text{f-part}\rangle\} \tag{7}$$

$$\langle\text{literal}\rangle ::= \text{``=''}\ \langle\text{w-value}\rangle\ \text{``=''} \tag{8}$$

$$\langle\text{keyword-part}\rangle ::= \langle\text{keyword}\rangle\ \langle\text{a-part}\rangle\ \langle\text{i-part}\rangle\ \langle\text{f-part}\rangle \tag{10}$$

$$\begin{aligned}\langle\text{pseudo-part}\rangle ::= {} & \text{``ORIG''}\ \langle\text{w-value}\rangle \\ \mid {} & \text{``EQU''}\ \langle\text{w-value}\rangle \\ \mid {} & \text{``END''}\ \langle\text{w-value}\rangle \\ \mid {} & \text{``CON''}\ \langle\text{w-value}\rangle \\ \mid {} & \text{``ALF''}\ \langle\text{string}\rangle\end{aligned} \tag{11}$$

$$\langle\text{opt-label}\rangle ::= [\langle\text{label}\rangle] \tag{12}$$

$$\langle\text{instr-part}\rangle ::= \langle\text{keyword-part}\rangle \mid \langle\text{pseudo-part}\rangle \tag{13}$$

$$\langle\text{instruction}\rangle ::= \langle\text{opt-label}\rangle\ \langle\text{instr-part}\rangle \tag{14}$$

$$\langle\text{program}\rangle ::= \langle\text{instruction}\rangle\ \{\langle\text{instruction}\rangle\} \tag{14}$$

Note that in eqs. (1) and (4) the terminal symbols **definedsymbol** and **futurereference** are used. However, these are not tokens that the scanner can recognize, the reason being that the

scanner has no knowledge of the symbol tables. The scanner only recognizes **symbol**s, and it is up to the parser to determine whether a **symbol** is a **definedsymbol** or a **futurereference**.

Let us now look at the grammar more rigorously. We use Noam Chomsky's definition of a formal grammar:

Definition 1. *Formal Grammars.*

A formal grammar is an ordered quad-tuple (N, Σ, P, S) where:

- N is a finite set of *nonterminal symbols*;
- $S \in N$ is a distinguished symbol that denotes the start symbol.
- Σ is a finite set of *terminal symbols* such that $\Sigma \cap S = \emptyset$;
- P is a finite set of production rules, each rule having the form

$$(\Sigma \cup N)^* N (\Sigma \cup N)^* \longrightarrow (\Sigma \cup N)^*$$

where * denotes the Kleene closure operator;

The special sequence $(\Sigma \cup N)^*$ is called a *sentential form*, and has two degenerate cases:

- It contains no elements: in this case it is denoted by the symbol ϵ; or
- It is of the form Σ^*: in this case it called a string.

It follows then that a grammar is simply a set of rules that govern how one may rewrite a string as the start symbol using productions and intermediate sentential forms, or vice-versa. The set of all strings that may be produced from the start symbol is called the *language* of the grammar.

For a grammar to be $LL(1)$, we must be able to determine the next production to use by looking at the next terminal symbol of the string being parsed and no further. This amounts to the requirement that given a production

$$P \longrightarrow \xi_1 \mid \xi_2 \mid \cdots \mid \xi_\nu$$

the following identity must hold:

$$\text{first}(\xi_1) \cap \text{first}(\xi_2) \cap \ldots \cap \text{first}(\xi_\nu) = \emptyset \tag{15}$$

where each ξ_i is a sentential form and $\text{first}(\xi)$ is a recursive function defined as

$$\text{first}(\xi) = \begin{cases} \{\alpha\}, & \text{if } \xi \text{ is of the form } \alpha\xi' \text{ where } \alpha \in \Sigma \\ \text{first}(\alpha_1) \cap \text{first}(\alpha_2) \cap \ldots \cap \text{first}(\alpha_\nu), & \text{if } \xi \text{ is of the form } S\xi' \text{ where } S \in N \\ & \text{and } S \longrightarrow \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_\nu \end{cases}$$

Unfortunately, this turns out to be inadequate, for we must also consider productions of the form

$$S \longrightarrow \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_\nu \mid \epsilon$$

where ϵ is the empty string. These pose problems in cases like

$$S \longrightarrow Ax$$
$$A \longrightarrow x \mid \epsilon$$

whereby it is not clear whether to reduce a string 'x' using production $x \longleftarrow A$, or the sequence of productions $x \longleftarrow \epsilon\, x \longleftarrow Ax \longleftarrow S$. This class of grammars, where one may be able to arrive at a terminal string from a start symbol S using two or more distinct sequences of productions, are called *ambiguous grammars*.

Therefore we pose one more restriction for a grammar to be $LL(1)$: For every symbol $A \in N$ of our grammar, which may after a sequence of productions generate the empty string ϵ, we require that

$$\text{fist}(A) \cap \text{follow}(A) = \emptyset$$

where $\text{follow}(A)$ is defined for every production of the form $S_i \longrightarrow \alpha_i A \eta_i$ as

$$\text{first}(\eta_1) \cup \text{first}(\eta_2) \cup \ldots \cup \text{first}(\eta_k)$$

where $1 \leq k \leq i$. If some η_i may generate the empty string then follow(η_i) must be included in the above union as well.

This is where the syntax diagrams of Appendix B pay off. The first rule amounts to the requirement that given a syntax diagram, at every fork the distinct branches to be pursued must start with a different symbol. The second rule states that for every graph that contains an empty branch, the first symbols that may follow such a graph must be disjoint than the first symbols of the graph itself. It is easy to verify that both rules apply in our grammar. Table 3.2 contains the 'first' and 'follow' symbols for all the graphs of the Appendix–except for "Program", which is identical to "Instruction".

Graph G	first(G)	follow(G)
Atom	**number defined symbol** `*`	`+ - * / // :`
Expression	`+ - *` **defined symbol number**	`( ) , =`
A-Part	`+ - * =` **defined symbol number future reference**	`( ,`
I-Part	`,`	`(`
F-Part	`(`	`,`
W-Value	`+ - *` **defined symbol number**	`=`
Instruction	**label keyword equ orig con alf end**	

Table 3.2 `MIXAL` Syntax Analysis

Taking into account the results of the above discussion, we create a `Parser` class which contains methods for each nonterminal. The method that parses a whole program is called `ParseProgram()`. If it fails to parse correctly, the `Errors` and/or `Warnings` collections are populated with information as to why things went wrong. The error messages consist of context information, *i.e.* which parsing method was the one that failed to complete, the cartesian location of the error in the source program, found text *etc.*

4.3. THE CODE GENERATOR

After the parsing process has completed successfully, we should somehow collect the parsed program, evaluate its meaning, and write it out to an appropriate form. There exist several options to do this. Perhaps the most popular is to generate an abstract syntax tree of the parsed program and visit each node of the tree to generate the appropriate instruction words. This method has its strengths. It lends itself to further syntax checking, code analysis, even tree transformation; a careful compiler writer may implement optimization algorithms based on tree structure; language translation is possible, between languages that share a common abstract syntax tree; and many other options.

However, our goals are far from being this complex. Furthermore, a process utilizing abstract syntax trees is a two-pass process by definition: one pass to construct the tree and a second pass to visit the nodes. In its author's own words, `MIXAL` has been designed to be a simple language to compile and a one-pass compiler should do the trick.

All the semantic evaluation and code generation is done in the `Parser` class. Each one of the parsing methods returns a value, which denotes the semantic value of the corresponding nonterminal. For example, `Atom()` returns the numeric value of the matched symbol or number; the `ParseLine()` method returns a `MIX` instruction word after it has finished parsing one line.

An *assembly item* is the tuple (l, w) where l is the location in memory of the assembled instruction and w is the instruction word. This is encapsulated in the `MemoryCell` class (which is definitely a bad case of misnaming). A *program assembly* is an unordered collection of assembly items. Its type is `List<MemoryCell>`.

One important issue here is that of future references, literals and forward local symbols (all of which can be broadly categorized as future references). When the address part parser is

asked to deal with a future reference, instead of returning the actual value of the address part, it returns `null` and stores the future reference. When the line parser attempts to assemble the instruction word, it either: (1) sees a valid address part, in which case it proceeds with the assembly as normal; or (2) sees a `null` address part, in which case it generates a "promise" to assemble the instruction as soon as it knows the equivalent value of the future reference—this is encapsulated in the `FutureReference` class. Future references can only be evaluated safely after all of the program has been parsed completely, because the `MIXAL` specification states that a future reference may never become a defined symbol, in which case it is assumed to be the constant 0.

After the assembly of the rest of the program has completed, all the instructions containing future references are evaluated and appended at the end of the program assembly, taking into account the aforementioned rule about future references that are never defined.

The "`END`" pseudo-instruction is special, because when the parser encounters it, it has to remember both the value of $\circledast$ and the value of its operand. The former is the location at which any remaining undefined future references, as well as the literal constants, finally become defined, while the latter is required by the loading routine that will load the program for execution within a simulator.

Following the creation of the program assembly, the compiler writes it out to either: (1) a binary file, using the binary serialization utilities of the *.NET* framework; or (2) a card deck (text) file, prepended with the loading routine from *TAOCP, sec. 1.3.1, ex. 26.*

Knowledge and insight on [the subject of compilers] will both enhance the general understanding of the art of programming in terms of high-level languages and will make it easier for a programmer to develop his own systems appropriate for specific purposes and areas of application.

— NIKLAUS WIRTH, *Algorithms+Data Structures=Programs* (1976)

Woe to the author who always wants to teach!
The secret of being a bore is to tell everything.

— VOLTAIRE, *De la Nature de l'Homme* (1737)

A

Example Program Listing

The following is the listing of the program `PRIMES.MIXAL`, as generated by `MIXASM`. End-of-instruction comments have been stripped in order for it to fit the page.

```
                                    1 * Table of primes (TAOCP p. 148, Program P)
                                    2 *
                                    3 L       EQU   500
                                    4 OUTDEV  EQU   18
                                    5 PRIME   EQU   -1
                                    6 BUF0    EQU   2000
                                    7 BUF1    EQU   BUF0+25
                                    8         ORIG  3000
3000:  +     0  0  18 35           9 START   IOC   0(OUTDEV)
3001:  +  2050  0   5  9          10         LD1   =1-L=
3002:  +  2051  0   5 10          11         LD2   =3=
3003:  +     1  0   0 49          12 2H      INC1  1
3004:  +   499  1   5 26          13         ST2   PRIME+L,1
3005:  +  3016  0   1 41          14         J1Z   2F
3006:  +     2  0   0 50          15 4H      INC2  2
3007:  +     2  0   2 51          16         ENT3  2
3008:  +     0  0   2 48          17 6H      ENTA  0
3009:  +     0  2   2 55          18         ENTX  0,2
3010:  -     1  3   5  4          19         DIV   PRIME,3
3011:  +  3006  0   1 47          20         JXZ   4B
3012:  -     1  3   5 56          21         CMPA  PRIME,3
3013:  +     1  0   0 51          22         INC3  1
3014:  +  3008  0   6 39          23         JG    6B
3015:  +  3003  0   0 39          24         JMP   2B
3016:  +  1995  0  18 37          25 2H      OUT   TITLE(OUTDEV)
3017:  +  2035  0   2 52          26         ENT4  BUF1+10
3018:  -    50  0   2 53          27         ENT5  -50
3019:  +   501  0   0 53          28 2H      INC5  L+1
3020:  -     1  5   5  8          29 4H      LDA   PRIME,5
3021:  +     0  0   1  5          30         CHAR
3022:  +     0  4  12 31          31         STX   0,4(1:4)
3023:  +     1  0   1 52          32         DEC4  1
3024:  +    50  0   1 53          33         DEC5  50
3025:  +  3020  0   2 45          34         J5P   4B
3026:  +     0  4  18 37          35         OUT   0,4(OUTDEV)
3027:  +    24  4   5 12          36         LD4   24,4
3028:  +  3019  0   0 45          37         J5N   2B
```

```
3029:  +     0  0  2  5     38          HLT
                            39 * Initial contents.
                            40          ORIG   PRIME+1
0000:  +     0  0  0  2     41          CON    2
                            42          ORIG   BUF0-5
1995:  +   393 19 22 23     43 TITLE    ALF  FIRST
1996:  +     6  9 25  5     44          ALF   FIVE
1997:  +     8 24 15  4     45          ALF   HUND
1998:  +  1221  4  0 17     46          ALF  RED P
1999:  +  1225 14  5 22     47          ALF  RIMES
                            48          ORIG   BUF0+24
2024:  +     0  0 31 51     49          CON    BUF1+10
                            50          ORIG   BUF1+24
2049:  +     0  0 31 26     51          CON    BUF0+10
                            52          END    START
```

MAIN SYMBOLS

```
L:       +  0  0  0  7 52   = 500
OUTDEV:  +  0  0  0  0 18   = 18
PRIME:   -  0  0  0  0  1   = -1
BUF0:    +  0  0  0 31 16   = 2000
BUF1:    +  0  0  0 31 41   = 2025
START:   +  0  0  0 46 56   = 3000
TITLE:   +  0  0  0 31 11   = 1995
```

LOCAL SYMBOLS

```
|2-1|:   +  0  0  0 46 59   = 3003
|4-1|:   +  0  0  0 46 62   = 3006
|6-1|:   +  0  0  0 47  0   = 3008
|2-2|:   +  0  0  0 47  8   = 3016
|2-3|:   +  0  0  0 47 11   = 3019
|4-2|:   +  0  0  0 47 12   = 3020
```

LITERALS

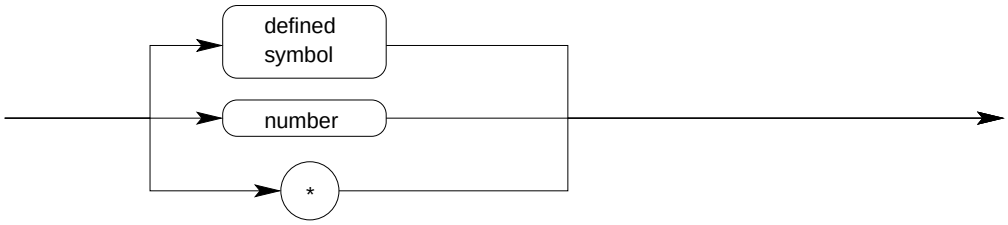
```
=-499=:  +  0  0  0 32  2   = 2050
=3=:     +  0  0  0 32  3   = 2051
```

B

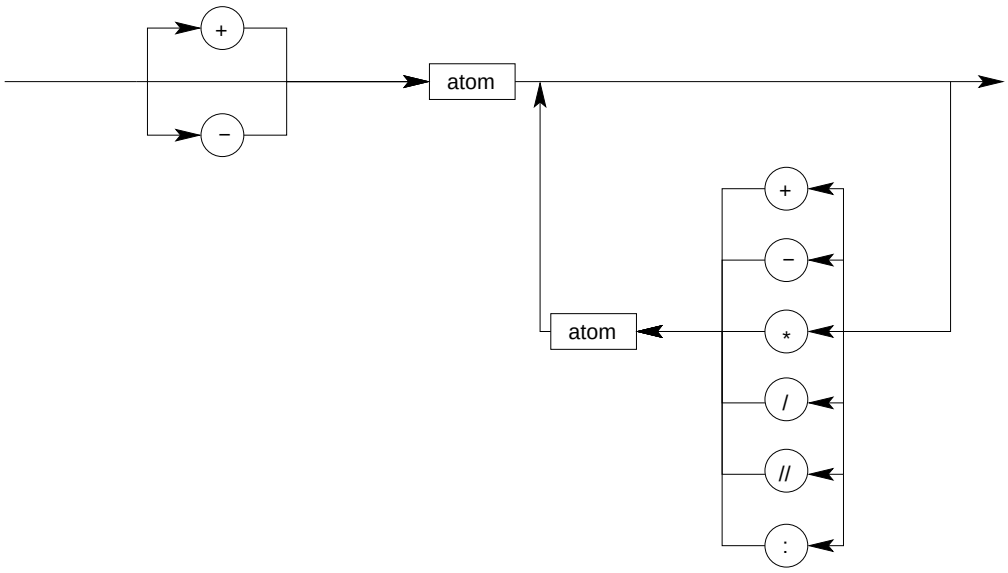
MIXAL Syntax Diagrams

Included here for future reference (no pun intended) are the syntax diagrams for the `MIXAL` grammar. By inspection, it can be proven that the grammar we have distilled from the specification is in fact $LL(1)$.

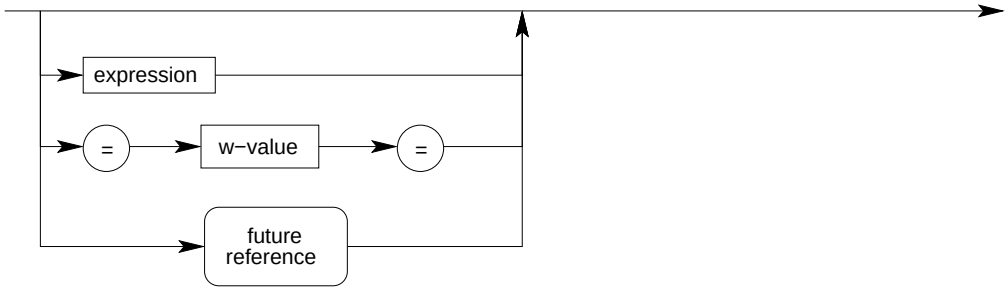
Atom



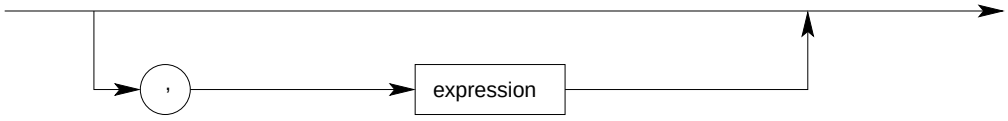
Expression



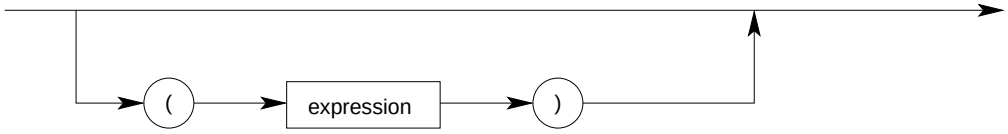
A−Part



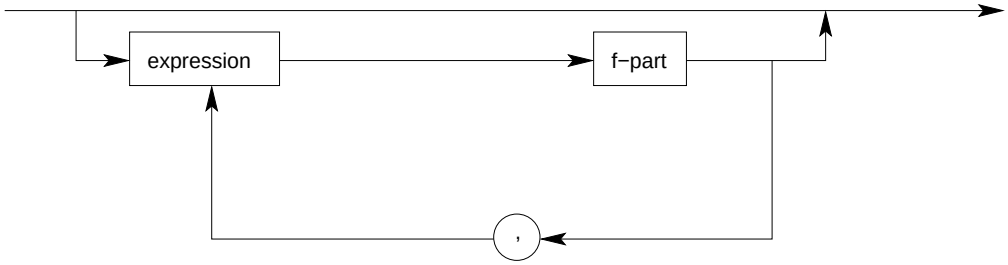
I−Part



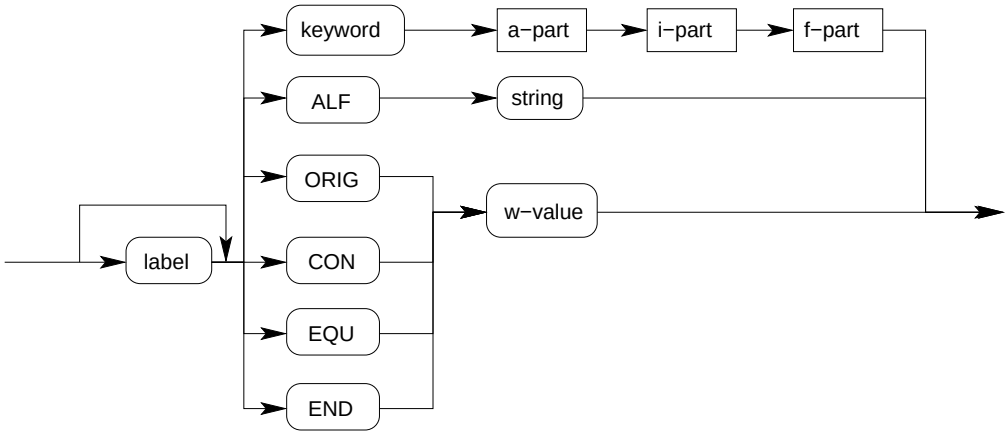
F−Part



W−Value



Instruction



Program