\( \mu \text{FX/DLX} \)

A PEDAGOGIC COMPILER

Douglas Grundman
Raymie Stata
James O'Toole

March 1992
This report provides an overview of the µFX/DLX compiler. The source language for the compiler is µFX, a Lisp dialect that is a subset of FX-91. µFX is statically typed, and employs a type reconstruction algorithm to eliminate the need for type declarations. The compiler produces assembly code for the DLX, a simplified RISC architecture introduced by Patterson and Hennessy in their text, Computer Architecture: A Quantitative Approach.

µFX/DLX was written for the primary purpose of conducting experiments concerning basic features of programming language implementation. For example, it is used in the MIT graduate-level programming language course, where students are expected to read, understand, and modify the compiler, in order to investigate the effects of various optimizations.

The organization of the compiler and its intermediate forms are described via examples. The register usage, memory layouts, and calling conventions are explained. Some suggested experiments are presented, and the annotated implementation of µFX/DLX is provided. The report is not entirely self-contained, as it does not completely describe the details of the source and target languages.
Summary

This report provides an overview of the \( \mu \text{FX}/\text{DLX} \) compiler. The source language for the compiler is \( \mu \text{FX} \), a Lisp dialect that is a subset of FX-91. \( \mu \text{FX} \) is statically typed, and employs a type reconstruction algorithm to eliminate the need for type declarations. The compiler produces assembly code for the DLX, a simplified RISC architecture introduced by Patterson and Hennessy in their text, *Computer Architecture: A Quantitative Approach*.

\( \mu \text{FX}/\text{DLX} \) was written for the primary purpose of conducting experiments concerning basic features of programming language implementation. For example, it is used in the MIT graduate-level programming language course, where students are expected to read, understand, and modify the compiler, in order to investigate the effects of various optimizations.

The organization of the compiler and its intermediate forms are described via examples. The register usage, memory layouts, and calling conventions are explained. Some suggested experiments are presented, and the annotated implementation of \( \mu \text{FX}/\text{DLX} \) is provided. The report is not entirely self-contained, as it does not completely describe the details of the source and target languages.

**Keywords:** Programming Languages, Types, Effects, Inference, Polymorphism, Compilation, Optimization

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract number N00014-89-J-1988
Preface

This report represents a combination of several sources of documentation concerning the pedagogical compilers which have been written for MIT EECS course 6.821. The first such compiler was written by Jonathan Rees. That compiler was used for several years between approximately 1987 and 1990. It was modified in minor ways by Franklyn Turbak, Mark Sheldon, and James O'Toole. Students made use of the compiler in the final problem sets of course 6.821.

During early 1991, Doug Grundman rewrote most of the compiler. Doug designed the intermediate code representations and wrote a new backend for the compiler which produced assembly code compatible with the DLX architecture [2]. During the summer of 1991, Raymie Stata joined the compiler project. Raymie and Doug rewrote portions of the compiler to improve performance, implemented tail-recursive call optimization, and corrected the generation of code which used the stack. Raymie also added support for more of the source language (μFX), and modified the garbage collector to avoid copying stack-allocated data.

At the time of this writing, the compiler and associated simulation software are being prepared for student use in the Fall 1992 semester of 6.821. Brian Reistad is improving the typechecking phase of the compiler. This document was compiled from three primary sources: Doug's general overview of the compiler phases, Raymie's description of register usage conventions, and older materials describing Rees's version of the compiler. These documents were merged; some additional text and figures were added.

All of the software and documentation described in this report is available in electronic form. We hope that this report will permit the reader to enjoy the 6.821 pedagogical compiler.

James O'Toole
Editor
Contents

1 Introduction 5
   1.1 The μFX Language ........................................ 5
   1.2 The Target Machine ........................................ 6

2 Compiler structure 8
   2.1 Using the System ........................................... 8
   2.2 The "exp" Intermediate Form .............................. 10
   2.3 The "icode" Intermediate Form ............................ 11
   2.4 The "ocode" Intermediate Form .................................... 12

3 The Runtime Environment 13
   3.1 Memory organization ......................................... 13
      3.1.1 Data Tagging ........................................ 14
      3.1.2 Stack conventions ........................................ 14
      3.1.3 Garbage collection ....................................... 15
   3.2 Calling convention ........................................... 15
      3.2.1 Representation of Environments .......................... 15

4 Some Suggested Experiments 18

5 Obtaining the Distribution in Electronic Form 18

A The DLX Instruction Set 18
   A.1 Registers and Miscellany .................................... 19
   A.2 Integer Instructions ......................................... 19
   A.3 Pseudo-Integer Instructions .................................. 21
   A.4 Floating Point Instructions .................................. 21

B μFX/DLX Run-Time Library 22

C μFX/DLX Compiler Implementation 26
   C.1 compiler/asm.fx ........................................... 27
   C.2 compiler/bits.fx ........................................... 37
   C.3 compiler/dlxsim.fx ......................................... 39
   C.4 compiler/exp.fx ........................................... 49
   C.5 compiler/exp2ic.fx ......................................... 51
   C.6 compiler/ic2oc.fx ........................................... 60
   C.7 compiler/icode.fx ........................................... 76
   C.8 compiler/lib.fx ............................................ 77
   C.9 compiler/misc.fx ........................................... 91
   C.10 compiler/oc2txt.fx ....................................... 94
   C.11 compiler/ocode.fx ....................................... 96
   C.12 compiler/optimize.fx .................................... 97
   C.13 compiler/parse.fx ....................................... 98
   C.14 compiler/system.fx ....................................... 106
<table>
<thead>
<tr>
<th>CONTENTS CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.15 compiler/table.fx</td>
</tr>
<tr>
<td>C.16 compiler/toplevel.fx</td>
</tr>
<tr>
<td>C.17 compiler/ty_recon.fx</td>
</tr>
<tr>
<td>D  ( \mu )FX/DLX Run-time Implementation</td>
</tr>
<tr>
<td>D.1 runtime/Makefile</td>
</tr>
<tr>
<td>D.2 runtime/alloc.s</td>
</tr>
<tr>
<td>D.3 runtime/epilog.s</td>
</tr>
<tr>
<td>D.4 runtime/frames.s</td>
</tr>
<tr>
<td>D.5 runtime/lib.s</td>
</tr>
<tr>
<td>D.6 runtime/macros.h</td>
</tr>
<tr>
<td>D.7 runtime/printf.s</td>
</tr>
<tr>
<td>D.8 runtime/prolog.s</td>
</tr>
<tr>
<td>References</td>
</tr>
<tr>
<td>Index</td>
</tr>
</tbody>
</table>


1 Introduction

The MIT subject 6.821 is an introduction to programming language semantics and pragmatics for graduate students. In the laboratory portion of 6.821, students read and modify a simple compiler to gain hands-on experience with fundamental concepts in compilation.

We have built a new compiler for the \( \muFX \) programming language for use in the class. The compiler, which generates code for Patterson and Hennessy's DLX architecture, was designed to be especially easy to understand and modify. The purpose of this report is to make the compiler more accessible by explaining its organization and providing an overview of various internal representations. Information for acquiring a copy of the \( \muFX \) software package and a simulator for DLX is in Section 5. Any questions or bug reports concerning the software described in this report should be sent via electronic mail to microfx@brokaw.lcs.mit.edu.

The \( \muFX/\text{DLX} \) compiler was designed with two goals in mind. First, there was the pedagogic goal: the compiler needed to be useful for teaching compilation techniques to students. The second goal was that the compiler had to afford easy experimentation by being easy to modify. That way, the compiler would be useful not only in the classroom, but in a research setting as well.

These two goals — readability and writability — overshadowed all others in the design of the compiler. \( \muFX/\text{DLX} \) makes no pretense of being a production compiler. Code quality and space efficiency have been largely ignored in favor of intelligibility. For example, the compiler's code generator contains only a few special case code improvements, and these were added only to improve the readability of the emitted code.

These two primary goals determine that the compiler be as simple as possible, and as modular as possible. The former aspect says that the compiler contains no hidden intricacies to improve run-time performance or compilation speed, while the latter decomposes the compiler into several simple passes that interact only through well-defined and well-documented interfaces.

It follows that adding a new pass (such as an optimizer) is relatively easy, as is making modifications to any of the pre-existing passes. Experience has indeed shown this to be the case.

The remaining sections of this report discuss the features of the \( \muFX \) language, introduce the reader to the DLX target machine, give an overview of the compiler, and tell where to obtain a copy of the software.

1.1 The \( \muFX \) Language

\( \muFX \) is a subset of the FX-91 programming language [1], and may be thought of as a cross between Scheme and ML. The \( \muFX \) syntax is shown below.

\( \muFX \) is lexically scoped, with all parameters passed by value. Like FX-91, \( \muFX \) is strongly typed, incorporating an ML-style type reconstructor. The language has first-class procedures, tail-recursion, and garbage collection. Its primitive data types include integers, characters, symbols, strings, references, and procedures.
1.2 The Target Machine

The language supported by the compiler uses the S-expression style in order to simplify parsing and permit easy experimentation with new language features. A number of primitive procedures are supported by the compiler as part of the runtime library (see Appendix B).

1.2 The Target Machine

The DLX architecture was introduced by Patterson and Hennessy in their book, Computer Architecture: A Quantitative Approach [2]. DLX has a generic RISC instruction set very similar to that of the MIPS architecture. It has 32 32-bit general-purpose registers, 32 32-bit floating-point registers, and no condition codes. There is one data addressing mode: register indirect with (signed) 16-bit offset. All memory accesses must be aligned according to the size of the referenced datum, otherwise a trap occurs. DLX has a delay slot following each branch or trap instruction, but differs from the MIPS architecture in that it has load interlocks.

There is a publicly available simulator for DLX, called dlzsim. The µFX/DLX compiler emits code which runs directly on dlzsim, but also contains a built-in DLX emulator that suffices for running small test cases. The simulator for DLX may be obtained via anonymous ftp from max.stanford.edu in pub/hennessy-patterson.software.
1 INTRODUCTION

1.2 The Target Machine

Figure 1: The DLX Processor
2 Compiler structure

μFX/DLX is organized into seven passes that make use of three intermediate forms: exp, icode, and ocode. For modularity, the compiler passes communicate only through these intermediate forms, which have been designed to be easy to understand and to work with so that students need not deal with unnecessary obstacles.

The seven compiler passes are:

- **parser**: converts s-expressions to ezps.
- **type reconstructor**: annotates exp nodes with type information.
- **translator**: converts ezps to icode.
- **optimizer**: icode to (improved) icode.
- **code generator**: converts icode to ocode.
- **peephole optimizer**: ocode to (improved) ocode.
- **output stage**: ocode to assembly-code text.

The parser, type reconstructor, translator, and code generator have been mentioned above. The local optimizer currently only implements tail calls. Although a peephole optimizer is not currently implemented, the compiler provides for one so that delay slot filling can be done. The final output stage takes care of formatting the instructions into an ASCII text representation so that dlxsim will accept the output file. It also supplements the compiled program with a pre-written run-time support system (see Appendix D), thus functioning as a simple linker.

The software comes with a built-in interpreter which can execute programs at the ocode level. This interpreter, though slow, provides an easy way to test new compiler features. It is outfitted with a run-time environment similar to that supplied for running under dlxsim. Figure 2 shows the organization of the various stages of the compiler and its associated ocode simulator.

2.1 Using the System

The system is written in mini-FX, which is a slightly larger subset of FX-91 than is μFX. Several commands have been implemented that allow the user to display the output of any stage of the compiler. All of these procedures take a single argument which is a μFX expression represented as an S-expression. This set of commands includes:

- **(test-parse sexpr)**: prints a freshly-parsed representation of the μFX expression sexpr.
- **(check sexpr)**: type-checks sexpr.
- **(show-type-check sexpr)**: type-checks sexpr and displays the parse tree annotated with the reconstructed type information.
- **(itest-compile sexpr)**: compiles sexpr and prints (unoptimized) icode.
- **(otest-compile sexpr)**: compiles sexpr and prints optimized icode.
Figure 2: Organization of μFX/DLX
2.2 The “exp” Intermediate Form

The highest-level intermediate form — the “exp” form — is a parsed representation of the source language. It encodes primitives (booleans, characters, integers, strings, symbols and variables), abstractions, applications, conditionals, let, and letrec. The parser desugars all other language features into these forms.

> (test-parse '(lambda (x) x))
(ABSTRACTION->EXP TY (X) (VARIABLE->EXP TY X))

In the above example, the micro-FX program is quoted to make it a scheme s-expression. The parse tree which is returned indicates that this s-expression represents an abstraction. Inside the parse tree, TY stands for type information which will be filled in by the type-reconstructor.

Each exp tree node contains a field to hold the type of an expression. The type of each primitive constant is filled in at parse time; all others are filled in later by the type reconstructor.

> (test-parse '((lambda (x) x) 19))
(COMBINATION->EXP TY (ABSTRACTION->EXP TY (X) (VARIABLE->EXP TY X)) ((INT->EXP TY 19)))

In the above example, the closure is created and called with a list of arguments (19). The type reconstructor implements generic polymorphism [1].

> (check '(lambda (x) x))
(-> (?X-1 ?X-1)

In this example, the type computed by the reconstruction phase is a function taking an argument of type ?X-1 and returning a value of type ?X-1. The notation 'X-1 represents an unbound type variable, and indicates that the type-checking problem is under-constrained. In other words, this function is polymorphic in the type of its one argument.

The show-type-check procedure shows the parse tree for an expression with full type information:
In this example, the type of the argument and result of the lambda expression have been determined to be the same as the type of the integer constant 19.

2.3 The "icode" Intermediate Form

The middle-most intermediate form is termed "icode". Icode is designed to represent programs at the lowest possible level of abstraction without explicit register references. Type information present in the parse tree is also stored in the icode, although it is not normally printed when icode is displayed. The absence of explicit register references allows the issues of instruction selection and register allocation to be deferred to a subsequent pass. This goal was achieved with one exception: icode knows of the existence of an environment register so that the creation of closures can be represented.

```lisp
> (itest-compile '(+ 1 2))

Type: int

Icode:

START_1:
  (return (+ 1 2))
```

Here, START_1 is a label, and the "return" line is a tree printed (roughly) in preorder. START_1 always appears, and it is where the execution of the compiled program begins.

```lisp
> (itest-compile '(((lambda (x) (+ x x)) 7))

Type: int

Icode:

START_1:
  (return (call (alloc 2 ("LAMBDA_2" (r 2))) 7))
LAMBDA_2:
  (body (1) (return (+ (var 0 1) (var 0 1)))))
```

In this example, the lambda body has been separated from the call. (alloc 2 ( . . . )) allocates 2 locations, initializing them with the values shown as arguments (the location LAMBDA_2 and register 2 (the previously-mentioned environment register)).
2.4 The "ocode" Intermediate Form

The instance of CALL shown here takes two "parameters": the first is the function (closure), and the second is the function's parameter. Finally, (body (1) ...) accepts one parameter, an expression, and runs it within a (newly-constructed) environment accepting 1 parameter. The icode form (var 0 1) is a reference to the variable x. For further explanation of environments and variable access, see section 3.2.1.

> (otest-compile '(((lambda (x) (+ x x)) 7))

Type: int

Icode:

START_1:
  (jump (alloc 2 ("LAMBDA_2" (r 2))) 7)
LAMBDA_2:
  (body (1) (return (+ (var 0 1) (var 0 1))))

In this example, the call—return sequence has been replaced with a jump, thus implementing a tail-call. (JUMP has the same syntax as CALL.)

Any program expressed in icode takes the form of a list of labelled trees. The job of the compiler pass converting exps to icode is primarily to separate lambda bodies from the creation of corresponding closures. This explains why icode takes the form of a list of trees rather than a single tree: any expression containing a lambda sub-expression is split into two trees. One of these trees describes the creation of the closure, while the other describes the function's code.

2.4 The "ocode" Intermediate Form

The lowest-level intermediate form is termed "ocode". It has a 1-1 correspondence with DLX assembly code, and is converted to textual assembly code by the compiler's final pass. The code generator could have been designed to emit assembly-code directly, but the use of an intermediate form like ocode facilitates the construction of peep-hole optimizers, delay-slot fillers and the like.
3 THE RUNTIME ENVIRONMENT

This example shows the tiny program from above, fully compiled. The "1" and "2" show up as "2" and "4" in the object code because they are tagged values (see section 3.1.1). There is an assumption here that this code has been called with the standard calling convention. Both tagging and calling conventions will be explained more fully in section 3.2.

3 The Runtime Environment

The compiler comes with its own runtime library. Included in the library are a memory allocator (including a simple stop-and-copy garbage-collector), routines for saving and restoring registers to and from frames, and miscellaneous built-in primitives and I/O functions. The runtime library handles program invocation and termination, and prints the result of each computation and memory usage statistics after each run.

3.1 Memory organization

All heap and stack memory is organized into blocks. Each block is a sequence of four-byte slots whose address is a multiple of four. The main reason for this is that in DLX, as in most other modern RISC processors, all memory references to pointers or integers work only at four-byte boundaries. One particular slot in each block is called the "size" slot and the remaining slots are called the "data slots." The "size" slot is only accessible by the run-time system (which includes the memory allocator), while the data slots are accessible to the user's program. This use of explicit size information, though somewhat wasteful of memory, makes the system's garbage collector easier to read and understand.

The slots of a block are numbered starting from \(-1\). Slot \(-1\) is the size slot. Slot 0 is the first data slot; slot 1 is the second data slot; etc. Objects start at the lower address: the address of slot \(i + 1\) is always four bytes past the address of slot \(i\).

The system supports two memory allocation primitives: \_SALLOC, which allocates and zeroes a memory block from the stack, and \_ALLOC, which allocates and zeroes a
3.1 Memory organization

memory block from the heap. If the stack memory allocator discovers that there is insufficient stack memory available with which to satisfy the current _SALLOC request, execution halts with an error. On the other hand, if the heap memory allocator discovers that there is insufficient heap memory available with which to satisfy the current _ALLOC request, it calls the run-time system's garbage collector in an attempt to discover memory that may be reused. If this attempt is successful, _ALLOC proceeds to allocate from the recycled memory, otherwise execution halts with an error.

3.1.1 Data Tagging

Any garbage collector needs to be able to distinguish pointers into the heap from miscellaneous integer values in the machine's registers. This is a run-time determination, and there are many ways that the compiler can help the garbage collector do this. The way used in the μFX/DLX compiler is that of tagging values at run-time. This relies on the convention that all data objects be aligned on even-byte boundaries (ours are aligned on four-byte boundaries).

The tag of a (4-byte) data item is its lowest bit. Our tagging convention is that integers have a low bit of zero, while a pointer into the heap or stack has a low bit of one. This means the compiler has to adhere to two more conventions so that things all work. First, every integer n (and every atom representable in a single word) is represented by \(2 \times n\). This makes the low bit of every integer a zero at the cost of decreasing the range of the integers we can represent and of complicating the code for multiplication and division (addition and subtraction work unchanged). Second, every pointer is represented by a word value that is one (byte) greater than the address of the object pointed-to. Since all objects are even-byte aligned, this makes all pointers (which did have a low bit of zero) have a low bit of one. The cost is that every memory reference through a pointer \(p\) must be adjusted at run-time to be a reference through \(p - 1\).

Executable code in our model resides neither in the heap nor in the stack, so pointers into the code (such as return addresses) are never tagged. This agrees nicely with the semantics of DLX's jump-and-link instruction.

In the compiler code, the function otag (offset tag) provides a convenient way to name a slot; if \(p\) is a tagged pointer, \(p + \text{otag}(-1)\) is the machine address of the size slot, \(p + \text{otag}(0)\) is the machine address of the first data slot, etc.

3.1.2 Stack conventions

The stack is organized as a stack of four byte slots. Thus, the stack pointer is always moved in increments of four bytes. The stack register always contains the untagged address of the first free slot on the stack. The stack grows downward in the DLX address space.

Stack allocation of a block is done by subtracting the size of the object to be pushed in bytes, including the size slot from the stack pointer. A properly tagged pointer to the resulting block is the new stack pointer plus five. The runtime routine _SALLOC allocates and zeroes a block on the stack and accumulates statistics about stack space usage.
3.1.3 Garbage collection

Storage management is based upon a stop-and-copy garbage collector. In this scheme, the heap is divided into two equal-sized semispaces. At any time, one is considered "active" and the other "empty". Every heap allocation is made from the active semispace. When the memory allocator finds that the active semispace has filled up, it calls the garbage collector. The garbage collector swaps the active and empty semispaces, then scans all blocks pointed-to by any register and any blocks transitively accessible from any scanned block. It copies all of these accessible blocks into the (newly) active semispace. The garbage collector does not copy any memory block that is inaccessible, or any memory block on the stack (although any accessible blocks on the stack are scanned to discover new pointers into the heap).

The registers which start the copying process are called the root set. The root set includes all user data registers, the frame pointer, the environment pointer, and the temporary VAL register. These registers are discussed further in section 3.2. Of course, any register or block slot that does not contain a tagged pointer is not considered to be a pointer by the garbage collector.

Note that if a block in the stack is not reachable from the root set then it will not be scanned by the garbage collector. It is therefore safe to put data on the stack that is not in the standard memory block format, as long as this data does not include pointers which must be examined by the garbage collector. This is useful when writing the assembly code routines that interface to the operating system.

3.2 Calling convention

The caller is responsible for preserving the values of all registers except for VAL, ATEMP, and RETADR. Registers are saved in activation frames, which are linked dynamically and form the "dynamic chain." The head of the dynamic chain is pointed to by the frame pointer register FP (a tagged pointer).

Before each procedure call, an activation frame is allocated and linked into the dynamic chain. All registers (except VAL, ATEMP, and RETADR) are saved in this activation frame. Arguments to the procedure are evaluated and placed in registers ARG0, ARG1, etc.

By convention, at the time of a procedure call, register ARG0 holds a pointer to the procedure being called and ARGn holds the nth formal. When the procedure is called, the return address is stored into slot 2 of the activation record. The layout of the activation record is indicated in Figure 3. The live registers are restored from the current activation frame and computation continues.

At the end of a procedure body, the callee places the result in the VAL register and jumps to the return address stored in the activation frame. When the callee returns, the caller restores the stack pointer and frame pointer registers. Figure 4 shows the register usage of the μFX/DLX runtime system.

3.2.1 Representation of Environments

μFX/DLX environments are represented in memory by chains of blocks (called ribs). Indices 1 through n of each rib contain n values of a lexical environment's variables,
### 3.2 Calling convention

#### The runtime environment

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>frame size</td>
<td>Caller saves these registers. Put into frame when it is first alloc'd</td>
</tr>
<tr>
<td>0</td>
<td>FP (dynamic chain)</td>
<td>before a procedure invocation.</td>
</tr>
<tr>
<td>1</td>
<td>SP (before frame was pushed!)</td>
<td>before a procedure invocation.</td>
</tr>
<tr>
<td>2</td>
<td>return address</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>ENV</td>
<td>Caller saves these registers by invoking</td>
</tr>
<tr>
<td>4</td>
<td>r6</td>
<td>SAVE just before arguments to callee are evaluated. Restored after called procedure returns.</td>
</tr>
<tr>
<td>5</td>
<td>r7</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>r29</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 3: The Frame Layout**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Machine register number and Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZERO</td>
<td>0</td>
</tr>
<tr>
<td>VAL</td>
<td>1</td>
</tr>
<tr>
<td>ENV</td>
<td>2</td>
</tr>
<tr>
<td>FP</td>
<td>3</td>
</tr>
<tr>
<td>SP</td>
<td>4</td>
</tr>
<tr>
<td>HP</td>
<td>5</td>
</tr>
<tr>
<td>ARG0</td>
<td>6</td>
</tr>
<tr>
<td>ARG1</td>
<td>7</td>
</tr>
<tr>
<td>ARG2</td>
<td>7</td>
</tr>
<tr>
<td>ARG3</td>
<td>8</td>
</tr>
<tr>
<td>ATEMP</td>
<td>30</td>
</tr>
<tr>
<td>RETADR</td>
<td>31</td>
</tr>
</tbody>
</table>

**Figure 4: DLX Register Usage**
and index 0 contains a pointer to the parent environment. For example, consider a set of three nested \texttt{let} expressions:

\begin{verbatim}
(let ((a 1)
     (b 2))
(let ((c 3)
     (d 4)
     (e 5))
(let ((f 6))
  \texttt{Ibody}))
\end{verbatim}

When \texttt{Ibody} is evaluated, the environment structure is represented by the chain of blocks shown in Figure 5. The variable names shown in Figure 5 are not explicitly represented in the environment structure, only their values are. The names are shown in the figure for documentation purposes only.

Variables are accessed by traversing ribs "backward" until the proper rib is found, then by indexing "over" to set or retrieve the proper value. Thus, each accessible variable may be referenced from a code location by knowing only its "back" and "over" numbers. The compiler automatically maintains such numbers and does this for the user. In \texttt{ic}ode, a variable reference is encoded by the low-level primitive \texttt{(var back over)}. In figure 5, \texttt{f} is \texttt{(var 0 1)}, \texttt{c} is \texttt{(var 1 1)}, \texttt{d} is \texttt{(var 1 2)}, \texttt{e} is \texttt{(var 1 3)}, \texttt{a} is \texttt{(var 2 1)}, and \texttt{b} is \texttt{(var 2 2)}. In compiled code, \texttt{(var x y)}
translates to \( x \) indirect references through the environment pointer followed by an indexed access to the \( y \)-th component of that block.

4 Some Suggested Experiments

We have come up with a set of possible projects that illustrate various problems in compilation, some of which are in use here at MIT. Our list includes:

- Implementing stack allocation of frames and of environments when analysis shows this to be possible.
- The (re-)implementation of any one of the existing compiler phases (type reconstruction and code generation are good candidates).
- Replacing static links with displays.
- Allocating closures statically, or eliminating their construction altogether when analysis shows this to be possible.
- Modify activation record allocation to save only the registers which are live at the time of the call.
- Modify heap allocation to be performed inline instead of always calling _ALLOC.
- Adding new language features.

Further ideas for easy yet illustrative experiments for students would be valued.

5 Obtaining the Distribution in Electronic Form

Our system may be obtained via anonymous ftp from host brokaw.lcs.mit.edu in the directory pub/microfx. The package includes documentation, the compiler, runtime library, ocode interpreter, and a compiler test suite. Mini-FX is available via anonymous ftp from the same host.

Appendices C and D contain the source code of the compiler and the run-time library. This material is indexed by procedures, types, and variable names at the end of this report. This appendix represents a snapshot as of February 12, 1992 of the 25 Mini-FX source files which are available via FTP.

The simulator for DLX may be obtained via anonymous ftp from the host max.stanford.edu, in pub/hennessy-patterson.software, and is included in the pub/microfx distribution.

A The DLX Instruction Set

This appendix contains a brief description of the DLX instruction set, as used in \( \mu \)FX/DLX. This description is based on the assembler and interpreter supplied with dlxsim.
A.1 Registers and Miscellany

32 32-bit integer registers (r0..r31); r0 always reads as 0. 32 32-bit floating-pt registers (f0..f31) (doubles use even-odd pairs). There are no condition code bits. DLX has load interlocks. All loads and stores trap on unaligned access.

A.2 Integer Instructions

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operands</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>d,s1,s2</td>
<td>[traps on overflow]</td>
</tr>
<tr>
<td>addi</td>
<td>d,s,i16</td>
<td>[traps on overflow] (sign-extended i16 DATA)</td>
</tr>
<tr>
<td>addu</td>
<td>d,s1,s2</td>
<td></td>
</tr>
<tr>
<td>addui</td>
<td>d,s,i16</td>
<td></td>
</tr>
<tr>
<td>and</td>
<td>d,s1,s2</td>
<td></td>
</tr>
<tr>
<td>andi</td>
<td>d,s,i16</td>
<td></td>
</tr>
<tr>
<td>beqz</td>
<td>r,label</td>
<td>branch if reg is zero.</td>
</tr>
<tr>
<td>bnez</td>
<td>r,label</td>
<td>branch if reg is non-zero.</td>
</tr>
<tr>
<td>j</td>
<td>label</td>
<td></td>
</tr>
<tr>
<td>jal</td>
<td>label</td>
<td>leaves address of instr after delayed instr in ...</td>
</tr>
<tr>
<td>jr</td>
<td>r</td>
<td>s&amp;3=0 else trap.</td>
</tr>
<tr>
<td>jalr</td>
<td>r</td>
<td>s&amp;3=0 else trap. rct address like jsr.</td>
</tr>
<tr>
<td>lb</td>
<td>d,i16(s)</td>
<td>(sign-extended i16 OFFSET) sxt data byte to word</td>
</tr>
<tr>
<td>lbu</td>
<td>d,i16(s)</td>
<td>(sign-extended i16 OFFSET) zeroes high bits</td>
</tr>
<tr>
<td>lh</td>
<td>d,i16(s)</td>
<td>(sign-extended i16 OFFSET) sxt data halfword to word</td>
</tr>
<tr>
<td>lhi</td>
<td>d,i16</td>
<td>load immediate halfword &lt;&lt; 16. zeroes low 16 bits</td>
</tr>
<tr>
<td>lhu</td>
<td>d,i16(s)</td>
<td>(sign-extended i16 OFFSET) zeroes high bits</td>
</tr>
<tr>
<td>lw</td>
<td>d,i16(s)</td>
<td>(sign-extended i16 OFFSET)</td>
</tr>
<tr>
<td>nop</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### A.2 Integer Instructions

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operands</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>or</td>
<td>d,s1,s2</td>
<td></td>
</tr>
<tr>
<td>ori</td>
<td>d,s,i16</td>
<td></td>
</tr>
<tr>
<td>sb</td>
<td>i16(s),d</td>
<td>(sign-extended i16 OFFSET)</td>
</tr>
<tr>
<td>seq</td>
<td>d,s1,s2</td>
<td></td>
</tr>
<tr>
<td>seqi</td>
<td>d,s,i16</td>
<td>(sign-extended i16 DATA)</td>
</tr>
<tr>
<td>sequ</td>
<td>d,s1,s2</td>
<td>same as seq</td>
</tr>
<tr>
<td>sequi</td>
<td>d,s,i16</td>
<td>different from seqi (no sign-ext of imm data here)</td>
</tr>
<tr>
<td>sge</td>
<td>d,s1,s2</td>
<td></td>
</tr>
<tr>
<td>sgei</td>
<td>d,s,i16</td>
<td>(sign-extended i16 DATA)</td>
</tr>
<tr>
<td>sgeu</td>
<td>d,s1,s2</td>
<td></td>
</tr>
<tr>
<td>sgeui</td>
<td>d,s,i16</td>
<td></td>
</tr>
<tr>
<td>sgt</td>
<td>d,s1,s2</td>
<td></td>
</tr>
<tr>
<td>sgti</td>
<td>d,s,i16</td>
<td>(sign-extended i16 DATA)</td>
</tr>
<tr>
<td>sgtu</td>
<td>d,s1,s2</td>
<td></td>
</tr>
<tr>
<td>sgtui</td>
<td>d,s,i16</td>
<td></td>
</tr>
<tr>
<td>sh</td>
<td>i16(s),d</td>
<td>(sign-extended i16 OFFSET)</td>
</tr>
<tr>
<td>sle</td>
<td>d,s1,s2</td>
<td></td>
</tr>
<tr>
<td>slei</td>
<td>d,s,i16</td>
<td>(sign-extended i16 DATA)</td>
</tr>
<tr>
<td>sleu</td>
<td>d,s1,s2</td>
<td></td>
</tr>
<tr>
<td>sleui</td>
<td>d,s,i16</td>
<td></td>
</tr>
<tr>
<td>sll</td>
<td>d,s1,s2</td>
<td></td>
</tr>
<tr>
<td>sl</td>
<td>d,s1,s2</td>
<td></td>
</tr>
<tr>
<td>sls</td>
<td>d,s,i16</td>
<td></td>
</tr>
<tr>
<td>sls1i</td>
<td>d,s,i16</td>
<td></td>
</tr>
<tr>
<td>slt</td>
<td>d,s1,s2</td>
<td></td>
</tr>
<tr>
<td>slti</td>
<td>d,s,i16</td>
<td>(sign-extended i16 DATA)</td>
</tr>
<tr>
<td>sltu</td>
<td>d,s1,s2</td>
<td></td>
</tr>
<tr>
<td>sltui</td>
<td>d,s,i16</td>
<td></td>
</tr>
<tr>
<td>sne</td>
<td>d,s1,s2</td>
<td></td>
</tr>
<tr>
<td>snei</td>
<td>d,s,i16</td>
<td>(sign-extended i16 DATA)</td>
</tr>
<tr>
<td>sneu</td>
<td>d,s1,s2</td>
<td>same as sne</td>
</tr>
<tr>
<td>sneui</td>
<td>d,s,i16</td>
<td>different from snei (no sign-ext of imm data here)</td>
</tr>
<tr>
<td>sra</td>
<td>d,s1,s2</td>
<td></td>
</tr>
<tr>
<td>srai</td>
<td>d,s,i16</td>
<td></td>
</tr>
<tr>
<td>srl</td>
<td>d,s1,s2</td>
<td></td>
</tr>
<tr>
<td>srl1</td>
<td>d,s,i16</td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>d,s1,s2</td>
<td>(traps on overflow)</td>
</tr>
<tr>
<td>subi</td>
<td>d,s,i16</td>
<td>(traps on overflow) (sign-extended i16 DATA)</td>
</tr>
<tr>
<td>subu</td>
<td>d,s1,s2</td>
<td></td>
</tr>
<tr>
<td>subui</td>
<td>d,s,i16</td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td>i16(s),d</td>
<td>(sign-extended i16 OFFSET)</td>
</tr>
<tr>
<td>trap</td>
<td>i27</td>
<td></td>
</tr>
<tr>
<td>xor</td>
<td>d,s1,s2</td>
<td></td>
</tr>
<tr>
<td>xori</td>
<td>d,s,i16</td>
<td></td>
</tr>
</tbody>
</table>
### A.3 Pseudo-Integer Instructions

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operands</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>div</td>
<td>fd,fs1,fs2</td>
<td>traps on div-by-zero</td>
</tr>
<tr>
<td>divu</td>
<td>fd,fs1,fs2</td>
<td>traps on div-by-zero</td>
</tr>
<tr>
<td>movfp2i</td>
<td>d,f</td>
<td></td>
</tr>
<tr>
<td>movi2fp</td>
<td>f,r</td>
<td></td>
</tr>
<tr>
<td>mult</td>
<td>fd,fs1,fs2</td>
<td>traps on overflow</td>
</tr>
<tr>
<td>multu</td>
<td>fd,fs1,fs2</td>
<td>traps on overflow</td>
</tr>
</tbody>
</table>

### A.4 Floating Point Instructions

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operands</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>Fd,Fs1,Fs2</td>
<td></td>
</tr>
<tr>
<td>addf</td>
<td>fd,fs1,fs2</td>
<td></td>
</tr>
<tr>
<td>bfpf</td>
<td>label</td>
<td></td>
</tr>
<tr>
<td>bftp</td>
<td>label</td>
<td></td>
</tr>
<tr>
<td>cvtd2f</td>
<td>f,F</td>
<td></td>
</tr>
<tr>
<td>cvtd2i</td>
<td>f,F</td>
<td></td>
</tr>
<tr>
<td>cvtf2d</td>
<td>F,f</td>
<td></td>
</tr>
<tr>
<td>cvtf2i</td>
<td>f1,f2</td>
<td></td>
</tr>
<tr>
<td>cvti2d</td>
<td>F,f</td>
<td></td>
</tr>
<tr>
<td>cvti2f</td>
<td>f1,f2</td>
<td></td>
</tr>
<tr>
<td>divd</td>
<td>Fd,Fs1,Fs2</td>
<td></td>
</tr>
<tr>
<td>divf</td>
<td>fd,fs1,fs2</td>
<td></td>
</tr>
<tr>
<td>eqd</td>
<td>F1,F2</td>
<td></td>
</tr>
<tr>
<td>eqf</td>
<td>f1,f2</td>
<td></td>
</tr>
<tr>
<td>ged</td>
<td>F1,F2</td>
<td></td>
</tr>
<tr>
<td>gef</td>
<td>f1,f2</td>
<td></td>
</tr>
<tr>
<td>gtd</td>
<td>F1,F2</td>
<td></td>
</tr>
<tr>
<td>gtf</td>
<td>f1,f2</td>
<td></td>
</tr>
<tr>
<td>ld</td>
<td>F,i16(s)</td>
<td>(sign-extended i16 OFFSET)</td>
</tr>
<tr>
<td>led</td>
<td>F1,F2</td>
<td></td>
</tr>
<tr>
<td>lef</td>
<td>f1,f2</td>
<td></td>
</tr>
<tr>
<td>lf</td>
<td>f,i16(s)</td>
<td>(sign-extended i16 OFFSET)</td>
</tr>
<tr>
<td>ltd</td>
<td>F1,F2</td>
<td></td>
</tr>
<tr>
<td>ltf</td>
<td>f1,f2</td>
<td></td>
</tr>
<tr>
<td>movd</td>
<td>F1,F2</td>
<td></td>
</tr>
<tr>
<td>movf</td>
<td>f1,f2</td>
<td></td>
</tr>
<tr>
<td>multd</td>
<td>Fd,Fs1,Fs2</td>
<td></td>
</tr>
<tr>
<td>multf</td>
<td>fd,fs1,fs2</td>
<td></td>
</tr>
<tr>
<td>ned</td>
<td>F1,F2</td>
<td></td>
</tr>
<tr>
<td>nef</td>
<td>f1,f2</td>
<td></td>
</tr>
<tr>
<td>sd</td>
<td>i16(s),F</td>
<td>(sign-extended i16 OFFSET)</td>
</tr>
<tr>
<td>sf</td>
<td>i16(s),f</td>
<td>(sign-extended i16 OFFSET)</td>
</tr>
<tr>
<td>subd</td>
<td>Fd,Fs1,Fs2</td>
<td></td>
</tr>
<tr>
<td>subf</td>
<td>fd,fs1,fs2</td>
<td></td>
</tr>
</tbody>
</table>
This appendix contains a listing of the standard library routines supported by the runtime system of \( \muFX/DLX \). This is a subset of the FX-91 standard library.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>symbol</td>
<td>(-(-&gt; (name) symbol))</td>
<td>builtin constructor</td>
</tr>
<tr>
<td>and</td>
<td>(-(-&gt; (bool bool) bool))</td>
<td>builtin operator</td>
</tr>
<tr>
<td>or</td>
<td>(-(-&gt; (bool bool) bool))</td>
<td>builtin operator</td>
</tr>
<tr>
<td>backspace</td>
<td>char</td>
<td>constant</td>
</tr>
<tr>
<td>newline</td>
<td>char</td>
<td>constant</td>
</tr>
<tr>
<td>page</td>
<td>char</td>
<td>constant</td>
</tr>
<tr>
<td>space</td>
<td>char</td>
<td>constant</td>
</tr>
<tr>
<td>tab</td>
<td>char</td>
<td>constant</td>
</tr>
<tr>
<td>equiv?</td>
<td>(-(-&gt; (bool bool) bool))</td>
<td></td>
</tr>
<tr>
<td>and?</td>
<td>(-(-&gt; (bool bool) bool))</td>
<td></td>
</tr>
<tr>
<td>or?</td>
<td>(-(-&gt; (bool bool) bool))</td>
<td></td>
</tr>
<tr>
<td>not?</td>
<td>(-(-&gt; (bool) bool))</td>
<td></td>
</tr>
<tr>
<td>not</td>
<td>(-(-&gt; (bool) bool))</td>
<td></td>
</tr>
<tr>
<td>char=?</td>
<td>(-(-&gt; (char char) bool))</td>
<td></td>
</tr>
<tr>
<td>char&lt;?</td>
<td>(-(-&gt; (char char) bool))</td>
<td></td>
</tr>
<tr>
<td>char&gt;?</td>
<td>(-(-&gt; (char char) bool))</td>
<td></td>
</tr>
<tr>
<td>char&lt;=?</td>
<td>(-(-&gt; (char char) bool))</td>
<td></td>
</tr>
<tr>
<td>char=-?</td>
<td>(-(-&gt; (char char) bool))</td>
<td></td>
</tr>
<tr>
<td>char-ci=?</td>
<td>(-(-&gt; (char char) bool))</td>
<td></td>
</tr>
<tr>
<td>char-ci&lt;?</td>
<td>(-(-&gt; (char char) bool))</td>
<td></td>
</tr>
<tr>
<td>char-ci&gt;?</td>
<td>(-(-&gt; (char char) bool))</td>
<td></td>
</tr>
<tr>
<td>char-ci&lt;=?</td>
<td>(-(-&gt; (char char) bool))</td>
<td></td>
</tr>
<tr>
<td>char-ci&gt;=?</td>
<td>(-(-&gt; (char char) bool))</td>
<td></td>
</tr>
<tr>
<td>char-alphabetic?</td>
<td>(-(-&gt; (char) bool))</td>
<td></td>
</tr>
<tr>
<td>char-numeric?</td>
<td>(-(-&gt; (char) bool))</td>
<td></td>
</tr>
<tr>
<td>char-whitespace?</td>
<td>(-(-&gt; (char) bool))</td>
<td></td>
</tr>
<tr>
<td>char-lower-case?</td>
<td>(-(-&gt; (char) bool))</td>
<td></td>
</tr>
<tr>
<td>char-upper-case?</td>
<td>(-(-&gt; (char) bool))</td>
<td></td>
</tr>
<tr>
<td>char-upcase</td>
<td>(-(-&gt; (char) char))</td>
<td></td>
</tr>
<tr>
<td>char-downcase</td>
<td>(-(-&gt; (char) char))</td>
<td></td>
</tr>
<tr>
<td>char-&gt;int</td>
<td>(-(-&gt; (char) int))</td>
<td></td>
</tr>
<tr>
<td>int-&gt;char</td>
<td>(-(-&gt; (int) char))</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Comment</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>=</td>
<td>(-&gt; (int int) bool)</td>
<td>builtin predicate</td>
</tr>
<tr>
<td>&lt;</td>
<td>(-&gt; (int int) bool)</td>
<td>builtin predicate</td>
</tr>
<tr>
<td>&gt;</td>
<td>(-&gt; (int int) bool)</td>
<td>builtin predicate</td>
</tr>
<tr>
<td>&lt;=</td>
<td>(-&gt; (int int) bool)</td>
<td>builtin predicate</td>
</tr>
<tr>
<td>&gt;=</td>
<td>(-&gt; (int int) bool)</td>
<td>builtin predicate</td>
</tr>
<tr>
<td>+</td>
<td>(-&gt; (int int) int)</td>
<td>builtin operator</td>
</tr>
<tr>
<td>-</td>
<td>(-&gt; (int int) int)</td>
<td>builtin operator</td>
</tr>
<tr>
<td>/</td>
<td>(-&gt; (int int) int)</td>
<td>builtin operator</td>
</tr>
<tr>
<td>remainder</td>
<td>(-&gt; (int int) int)</td>
<td>builtin operator</td>
</tr>
<tr>
<td>modulo</td>
<td>(-&gt; (int int) int)</td>
<td>builtin operator</td>
</tr>
<tr>
<td>neg</td>
<td>(-&gt; (int) int)</td>
<td>builtin operator</td>
</tr>
<tr>
<td>abs</td>
<td>(-&gt; (int) int)</td>
<td>builtin operator</td>
</tr>
<tr>
<td>null?</td>
<td>(generic (t) (-&gt; ((listof t)) bool))</td>
<td></td>
</tr>
<tr>
<td>null</td>
<td>(generic (t) (-&gt; () (listof t)))</td>
<td></td>
</tr>
<tr>
<td>cons</td>
<td>(generic (t) (-&gt; (t (listof t)) (listof t)))</td>
<td></td>
</tr>
<tr>
<td>car</td>
<td>(generic (t) (-&gt; ((listof t)) t))</td>
<td></td>
</tr>
<tr>
<td>cdr</td>
<td>(generic (t) (-&gt; ((listof t)) (listof t)))</td>
<td></td>
</tr>
<tr>
<td>set-car!</td>
<td>(generic (t) (-&gt; ((listof t) t) unit))</td>
<td></td>
</tr>
<tr>
<td>set-cdr!</td>
<td>(generic (t) (-&gt; ((listof t) (listof t)) unit))</td>
<td></td>
</tr>
<tr>
<td>length</td>
<td>(generic (t) (-&gt; ((listof t)) int))</td>
<td></td>
</tr>
<tr>
<td>append</td>
<td>(generic (t) (-&gt; ((listof t) (listof t) (listof t)))</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Comment</td>
</tr>
<tr>
<td>------------</td>
<td>---------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>reverse</td>
<td>(generic (t))</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(-&gt; ((listof t)) (listof t))</td>
<td></td>
</tr>
<tr>
<td>list-tail</td>
<td>(generic (t))</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(-&gt; ((listof t) int) (listof t))</td>
<td></td>
</tr>
<tr>
<td>list-ref</td>
<td>(generic (t))</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(-&gt; ((listof t) int) t))</td>
<td></td>
</tr>
<tr>
<td>map</td>
<td>(generic (t1 t2))</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(-&gt; ((-&gt; (t1 t2) (listof t1)) (listof t2)))</td>
<td></td>
</tr>
<tr>
<td>for-each</td>
<td>(generic (t1 t2))</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(-&gt; ((-&gt; (t1 t2) (listof t1)) unit))</td>
<td></td>
</tr>
<tr>
<td>reduce</td>
<td>(generic (t1 t2))</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(-&gt; ((-&gt; (t1 t2) t2) (listof t1) t2) t2)</td>
<td></td>
</tr>
<tr>
<td>list-&gt;string</td>
<td>(-&gt; ((listof char)) string)</td>
<td></td>
</tr>
<tr>
<td>string-&gt;list</td>
<td>(-&gt; (string) (listof char))</td>
<td></td>
</tr>
<tr>
<td>pair</td>
<td>(generic (t1 t2))</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(-&gt; (t1 t2) (pairof t1 t2)))</td>
<td></td>
</tr>
<tr>
<td>left</td>
<td>(generic (t1 t2))</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(-&gt; ((pairof t1 t2)) t1))</td>
<td></td>
</tr>
<tr>
<td>right</td>
<td>(generic (t1 t2))</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(-&gt; ((pairof t1 t2)) t2))</td>
<td></td>
</tr>
<tr>
<td>ref</td>
<td>(generic (t))</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(-&gt; (t) (refof t)))</td>
<td></td>
</tr>
<tr>
<td>:=</td>
<td>(generic (t))</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(-&gt; ((refof t) t) unit))</td>
<td></td>
</tr>
<tr>
<td>make-string</td>
<td>(-&gt; (int char) string)</td>
<td></td>
</tr>
<tr>
<td>string-length</td>
<td>(-&gt; (string) int)</td>
<td></td>
</tr>
<tr>
<td>string-ref</td>
<td>(-&gt; (string int) char)</td>
<td></td>
</tr>
<tr>
<td>string-set!</td>
<td>(-&gt; (string int char) unit)</td>
<td></td>
</tr>
<tr>
<td>string-fill!</td>
<td>(-&gt; (string char) unit)</td>
<td></td>
</tr>
<tr>
<td>string=?</td>
<td>(-&gt; (string string) bool)</td>
<td></td>
</tr>
<tr>
<td>string&lt;?</td>
<td>(-&gt; (string string) bool)</td>
<td></td>
</tr>
<tr>
<td>string&gt;?</td>
<td>(-&gt; (string string) bool)</td>
<td></td>
</tr>
<tr>
<td>string&lt;=?</td>
<td>(-&gt; (string string) bool)</td>
<td></td>
</tr>
<tr>
<td>string&gt;=?</td>
<td>(-&gt; (string string) bool)</td>
<td></td>
</tr>
<tr>
<td>string-ci=?</td>
<td>(-&gt; (string string) bool)</td>
<td></td>
</tr>
<tr>
<td>string-ci&lt;?</td>
<td>(-&gt; (string string) bool)</td>
<td></td>
</tr>
<tr>
<td>string-ci&gt;?</td>
<td>(-&gt; (string string) bool)</td>
<td></td>
</tr>
<tr>
<td>string-ci&lt;=&gt;?</td>
<td>(-&gt; (string string) bool)</td>
<td></td>
</tr>
<tr>
<td>string-ci&gt;=?</td>
<td>(-&gt; (string string) bool)</td>
<td></td>
</tr>
<tr>
<td>substring</td>
<td>(-&gt; (string int int) string)</td>
<td></td>
</tr>
<tr>
<td>substring</td>
<td>(-&gt; (string string) string)</td>
<td></td>
</tr>
<tr>
<td>string-append</td>
<td>(-&gt; (string string) string)</td>
<td></td>
</tr>
<tr>
<td>string-copy</td>
<td>(-&gt; (string) string)</td>
<td></td>
</tr>
<tr>
<td>string-&gt;vector</td>
<td>(-&gt; (string) (vectorof char))</td>
<td></td>
</tr>
<tr>
<td>vector-&gt;string</td>
<td>(-&gt; ((vectorof char)) string)</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Comment</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>sym-&gt;string</td>
<td>(-&gt; (sym) string)</td>
<td></td>
</tr>
<tr>
<td>string-&gt;sym</td>
<td>(-&gt; (string) sym)</td>
<td></td>
</tr>
<tr>
<td>sym=?</td>
<td>(-&gt; (sym sym) bool)</td>
<td>built</td>
</tr>
<tr>
<td>hash</td>
<td>(-&gt; (sym) int)</td>
<td></td>
</tr>
<tr>
<td>make-vector</td>
<td>(generic (t) (-&gt; (int t) (vectorof t)))</td>
<td></td>
</tr>
<tr>
<td>vector-length</td>
<td>(generic (t) (-&gt; ((vectorof t)) int))</td>
<td></td>
</tr>
<tr>
<td>vector-ref</td>
<td>(generic (t) (-&gt; ((vectorof t) int) t))</td>
<td></td>
</tr>
<tr>
<td>vector-set!</td>
<td>(generic (t) (-&gt; ((vectorof t) int t) unit))</td>
<td></td>
</tr>
<tr>
<td>vector-fill!</td>
<td>(generic (t) (-&gt; ((vectorof t) t) unit))</td>
<td></td>
</tr>
<tr>
<td>vector-&gt;list</td>
<td>(generic (t) (-&gt; ((vectorof t)) (listof t)))</td>
<td></td>
</tr>
<tr>
<td>list-&gt;vector</td>
<td>(generic (t) (-&gt; ((listof t)) (vectorof t)))</td>
<td></td>
</tr>
<tr>
<td>vector-map</td>
<td>(generic (t1 t2) (-&gt; ((-&gt; (t1) t2) (vectorof t1)) (vectorof t2)))</td>
<td></td>
</tr>
<tr>
<td>vector-map2</td>
<td>(generic (t1 t2 t3) (-&gt; ((-&gt; (t1 t2 t3) (vectorof t1) (vectorof t2)) (vectorof t3)))</td>
<td></td>
</tr>
<tr>
<td>vector-reduce</td>
<td>(gen : t1 t2) (-&gt; ((-&gt; (t1 t2) t2) (vectorof t1) t2) t2)</td>
<td></td>
</tr>
<tr>
<td>scan</td>
<td>(generic (t) (-&gt; ((-&gt; (t t) t) (vectorof t)) (vectorof t)))</td>
<td></td>
</tr>
<tr>
<td>segments-scan</td>
<td>(generic (t) (-&gt; ((-&gt; (t t) t) (vectorof bool) (vectorof t)) (vectorof t)))</td>
<td></td>
</tr>
<tr>
<td>compress</td>
<td>(generic (t1) (-&gt; ((vectorof bool) (vectorof t)) (vectorof t)))</td>
<td></td>
</tr>
<tr>
<td>expand</td>
<td>(generic (t1) (-&gt; ((vectorof bool) (vectorof t) (vectorof t)) (vectorof t)))</td>
<td></td>
</tr>
<tr>
<td>eoshift</td>
<td>(generic (t1) (-&gt; (int (vectorof t) (vectorof t)) (vectorof t)))</td>
<td></td>
</tr>
</tbody>
</table>
C \(\mu\text{FX}/\text{DLX} \) Compiler Implementation

This appendix contains a snapshot as of February 12, 1992 of the 17 source files which implement the compiler and ocode simulator in Mini-FX. All of these files are available via FTP.

The files included in this appendix are as follows:

<table>
<thead>
<tr>
<th>Filename</th>
<th>Module</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>compiler/asm.fx</td>
<td>Support</td>
<td>Assemble and unparse ocode representation</td>
</tr>
<tr>
<td>compiler/bits.fx</td>
<td>Support</td>
<td>Low-level data manipulation utilities</td>
</tr>
<tr>
<td>compiler/dlxsim.fx</td>
<td>Emulate</td>
<td>Emulation support</td>
</tr>
<tr>
<td>compiler/exp.fx</td>
<td>Parse</td>
<td>(\mu\text{FX} ) parse tree definition</td>
</tr>
<tr>
<td>compiler/exp2ic.fx</td>
<td>Translate</td>
<td>Translate expressions to intermediate code</td>
</tr>
<tr>
<td>compiler/ic2oc.fx</td>
<td>Generate</td>
<td>Generate ocode by recursive descent of icode</td>
</tr>
<tr>
<td>compiler/icode.fx</td>
<td>Generate</td>
<td>Icode representation definition</td>
</tr>
<tr>
<td>compiler/lib.fx</td>
<td>Runtime</td>
<td>Definitions of (\mu\text{FX} ) primitives</td>
</tr>
<tr>
<td>compiler/misc.fx</td>
<td>Support</td>
<td>Miscellaneous utilities</td>
</tr>
<tr>
<td>compiler/oc2txt.fx</td>
<td>Output</td>
<td>Produce DLX assembly code text</td>
</tr>
<tr>
<td>compiler/ocode.fx</td>
<td>Generate</td>
<td>Ocode representation definition</td>
</tr>
<tr>
<td>compiler/optimize.fx</td>
<td>Optimize</td>
<td>Optimization of intermediate code</td>
</tr>
<tr>
<td>compiler/parse.fx</td>
<td>Parse</td>
<td>Parse (\mu\text{FX} ) Syntax</td>
</tr>
<tr>
<td>compiler/system.fx</td>
<td>Emulate</td>
<td>Garbage collector and allocator for ocode emulation</td>
</tr>
<tr>
<td>compiler/table.fx</td>
<td>Support</td>
<td>Symbol table utility</td>
</tr>
<tr>
<td>compiler/toplevel.fx</td>
<td>Support</td>
<td>Top-level user interface to (\mu\text{FX}/\text{DLX} )</td>
</tr>
<tr>
<td>compiler/ty_recon.fx</td>
<td>Reconstruct</td>
<td>Type reconstruction algorithm</td>
</tr>
</tbody>
</table>

The index at the end of this document contains entries for procedures, shared variables, and runtime entry points.
C.1 compiler/asm.fx

The contents of the file compiler/asm.fx:

```
;; -e- Mode: Scheme; Package: SCHEME -e-

;; asm.fx -- assemble and unparse ocode from code generator. "Assemble"
;; means prepare for emulation; unparse means prepare for output to file
;; for dlxsim assembler.

;; We factor instructions according to the type of operands they use.
;; The following shorthands are used to name these types: i = integer
;; register; c = integer constant; f = single-precision FP; g = double
;; FP; d = integers in FP registers (e.g., for div instruction).

;; The two functions exported by this file are asm-ocode and
;; unparse-ocode. asm-ocode takes an operation name (symbol) and a
;; set of operands and returns a thunk that, when applied, mutates the
;; state of the DLI machine (see dlxsim.fx) according to the
;; instruction. unparse-ocode takes an operation name and set of
;; 'rands and returns a string representing the instruction in
;; official DLI assembler syntax.

;; asm-ocode and unparse-ocode use asm-??? (assemble) and unp-???
;; (unparse) functionals for different classes of instructions. The
;; asm-??? functionals return a thunk which, when applied, mutates
;; DLI machine state according to the instruction. The asm-???
;; functionals try to do expensive operations (e.g., matches on
;; operands) in an environment enclosing the returned thunk. The
;; unp-??? functionals return a string which is the unparsing of the
;; operation.

(define unp-oname
  (lambda (op)
    (let (((oname (down-sym op)))
          (string-append oname (pad oname 8)))))
```

27
(define unparse-reg
  (lambda (rn)
    (if ("*rr-pretty")
        (cond ((= rn ZERO) "ZERO")
              ((= rn VAL) "VAL")
              ((= rn ENV) "ENV")
              ((= rn FP) "FP")
              ((= rn SP) "SP")
              ((= rn ARG0) "ARG0")
              ((= rn ARG1) "ARG1")
              ((= rn ARG2) "ARG2")
              ((= rn ARG3) "ARG3")
              ((= rn ARG4) "ARG4")
              ((= rn ARG5) "ARG5")
              ((= rn ARG6) "ARG6")
              ((= rn ARG7) "ARG7")
              ((= rn ARG8) "ARG8")
              ((= rn ATENF) "ATENF")
              ((= rn RETADR) "RETADR")
              (else (string-append "r" (int->string rn)))))
    (string-append "r" (int->string rn)))))

(define unparse-freg
  (lambda (n) (string-append "f" (int->string n))))

(define unparse-dreg
  (lambda (n) (string-append "d" (int->string n))))

(define asm-iii
  ;; Three integer registers (e.g., add, subu)
  (lambda (fn rands)
    (match rands
      ((rrr->rands d s1 s2)
       (lambda () (set-reg! d (fn (get-reg s1) (get-reg s2)))))))

(define unse-iii
  (lambda (op rands)
    (let ((oname (unp-oname op))
          (match rands
            ((rrr->rands d s1 s2)
             (string-append oname (unparse-reg d) "," (unparse-reg s1) "," (unparse-reg s2)))))))

(define asm-iic
  ;; Two int reg's and a constant (e.g., xor)
  (lambda (fn rands)
    (match rands
      ((rrr->rands d s1 s2)
       (let ((val (eval-immed s2))
            (lambda () (set-reg! d (fn (get-reg s1) val))))))))
(define unp-iic
  (lambda (op rands)
    (let ((oname (unp-oname op)))
      (match rands
        ((rrri->rands" d s1 s2)
         (string-append oname
                      (unparse-reg d) "", " (unparse-reg s1) ", " s2))))))

(define asm-bi
  ;; Branches that take one operand (e.g., jal, bfpf)
  (lambda (fn rands)
    (match rands
      ((i->rands" 1)
       (let ((val (eval-immed 1)))
         (lambda () (:= *npc* (fn val)))))
      ((r->rands" r)
       (lambda () (:= *npc* (fn r))))))

(define unp-bi
  (lambda (op rands)
    (let ((oname (unp-oname op)))
      (match rands
        ((i->rands" 1)
         (string-append oname 1))
        ((r->rands" r) (string-append oname (unparse-reg r))))))

(define asm-b2
  ;; Branches that take two operands (e.g., bneq)
  (lambda (fn rands)
    (match rands
      ((ri->rands" r 1)
       (let ((val (eval-immed 1)))
         (lambda () (:= *npc* (fn r val)))))))

(define unp-b2
  (lambda (op rands)
    (let ((oname (unp-oname op)))
      (match rands
        ((ri->rands" r 1)
         (string-append oname (unparse-reg r) ", " 1))))))

(define asm-load
  ;; All load instructions except lhi and FP loads
  (lambda (fn rands)
    (match rands
      ((load->rands" d offset base)
       (lambda ()
         (set-reg! d (fn (get-mem (+ offset (get-reg base))))))))))
(define unp-load
  (lambda (op rands)
    (let ((oname (unp-oname op)))
      (match rands
        ((load->rands* d ofst base)
         (string-append oname
                       (unparse-reg d) " , "
                       (int->string ofst) "(" (unparse-reg base) ")"))))))

(define asm-lhi
  (lambda (rands)
    (match rands
      ((ri->rands* d s)
       (let ((val (* (eval-immed s) two-16)))
        (lambda () (set-reg! d val)))))))))

(define unp-lhi
  (lambda (op rands)
    (match rands
      ((ri->rands* d s) (string-append "lhi " (unparse-reg d) " , " s))))))

(define asm-store
  ;; All store instructions except FP stores
  (lambda (fn rands)
    (match rands
      ((store->rands* offset base v)
       (lambda ()
        (let* ((addr (+ offset (get-reg base)))
               (old-val (get-mem addr))
               (val (get-reg v)))
         (set-mem! addr (fn old-val val)))))))))

(define unp-store
  (lambda (op rands)
    (let ((oname (unp-oname op)))
      (match rands
        ((store->rands* offset base v)
         (string-append oname
                       (int->string ofst) "(" (unparse-reg base) ", "
                       (unparse-reg v)))))))))

(define asm-fff
  ;; Instructions taking three single FP registers (e.g., addf
  (lambda (fn rands)
    (match rands
      ((rrr->rands* d s1 s2)
       (lambda () (set-freg! d (fn (get-freg s1) (get-freg s2)))))})
(define unp-fff
  (lambda (op rands)
    (let ((oname (unp-oname op)))
      (match rands
        ((rrr->rands d s1 s2)
          (string-append oname (unparse-freg d) "", " (unparse-freg s1) ", "
                           (unparse-freg s2)))))
    )))

(define asm-ggg
  ;; Instructions taking three double FP reg's (e.g., divd)
  (lambda (fn rands)
    (match rands
      ((rrr->rands d s1 s2)
        (lambda () (set-dreg! d (fn (get-dreg s1) (get-dreg s2))))))))

(define unp-ggg
  (lambda (op rands)
    (let ((oname (unp-oname op)))
      (match
        rands
        ((rrr->rands d s1 s2)
          (string-append oname (unparse-dreg d) ",", " (unparse-dreg s1) "., "
                           (unparse-dreg s2))))))

(define asm-fp-rel
  ;; Floating point relational operations modify the fp boolean flag
  (lambda (fn c rands)
    (let ((get (cond ((sym=? c 'f) get-freg) ((sym=? c 'g) get-dreg))))
      (match rands
        ((rr->rands s1 s2)
          (lambda () (:= *fp-cond* (fn (get s1) (get s2)))))))))

(define unp-fp-rel
  (lambda (op c rands)
    (let ((unp (cond ((sym=? c 'f) unparse-freg) ((sym=? c 'g) unparse-dreg)))
      (oname (unp-oname op)))
      (match rands
        ((rr->rands s1 s2)
          (lambda () (string-append oname (unp s1) (unp s2))))))))

(define asm-cnv
  (let ((get (lambda (c) (cond ((or (sym=? c 'd) (sym=? c 'f)) get-freg)
                             ((sym=? c 'g) get-dreg)
                             ((sym=? c 'i) get-reg))))
    (st! (lambda (c) (cond ((or (sym=? c 'd) (sym=? c 'f)) set-freg!)
                             ((sym=? c 'g) set-dreg!)
                             ((sym=? c 'i) set-reg!)))
    (lambda (fn d s rands)
      (let ((g (get s)) (s! (st! d)))
        (match rands
          ((rr->rands dst src) (lambda () (s! dst (fn (g src))))))))))
(define unp-cnv
  (let ((unp (lambda (c) (cond ((or (sym=? c 'd) (sym=? c 'f)) unparse-freg)
                              ((sym=? c 'g) unparse-dreg)
                              ((sym=? c 'i) unparse-reg))))
       (lambda (op d e rands)
         (let ((op-name (unp-opname op)))
           (match rands
             ((rr->rands- dst src) (string-append op-name ((unp d) dst) ", " ((unp e) src)))))))))

;; Functions that are applied to operands (represented as bignums) to
;; execute low-level ALU ops and return the result (another bignum).
;; These functions are used only when the minifx equivalent doesn't do
;; the right thing, such as when dealing with the inidividual
;; bits of a signed integer.
;;
;; (define mk-bwise
;;   ;; Takes an operation (e.g., and?) and applies it bit-wise to the bits
;;   ;; in the operands. Since we store the operands in bignum rather
;;   ;; than machine int format, we need to convert the operands to
;;   ;; machine ints to make sure negative numbers are handled correctly.
;;   (lambda (op)
;;     (lambda (val1 val2)
;;       (letrec ((loop (loop (x v1 v2 i) (if (= i 32) (mint2bignum x) (loop (+ x (if (op (odd? v1) (odd? v2)) (expt 2 i) 0)) (quotient v1 2) (quotient v2 2) (+ i 1))))))
;;         (loop 0 (bignum2mint val1) (bignum2mint val2) 0))))))
;;
;; For relational operators, must return 1 or 0 rather than #t or #f
;; (define mk-alu-rel
;;   (lambda (fn) (lambda (x y) (if (fn x y) 1 0))))
;;
;; (define mk-alu-urel ;; For unsigned operands
;;   (lambda (fn) (lambda (x y) (if (fn (bignum2mint x) (bignum2mint y)) 1 0))))
;;
;; (define mk-alu-sgn ;; For signed integer operators
;;   (lambda (fn) (lambda (x y) (let ((result (fn x y)))
;;                                  (if (int32? result) result
;;                                      (error "ALU overflow"))))))
;;
;; (define mk-alu-unsgn ;; For unsigned integer operators
;;   (lambda (fn)
;;     (lambda (x y)
;;       (mint2bignum (remainder (fn (bignum2mint x) (bignum2mint y)) two^-32))))
;;
;; (define alu-neq (lambda (x y) (if (= x y) 0 1))))

32
(define alu-xor (mk-bwise (lambda (x y) (or (and x (not y)) (and (not x) y))))

(define alu-or
  ;; The code generator uses an or instruction with the RO register as
  ;; moves, so we want that case to go fast.
  (lambda (x y)
    (cond ((= x 0) y)
          ((= y 0) x)
          (else ((mk-bwise or?) x y))))

(define alu-s-left-l (lambda (x y) (* x (expt 2 y))))

(define alu-s-right-l
  (lambda (x y) (mint2bignum (quotient (bignum2mint x) (expt 2 y))))

(define alu-s-right-a
  (lambda (x y)
    (mint2bignum (+ (quotient (bignum2mint x) (expt 2 y))
                   (if (< x 0) (two-31 0))))))
(define asm
  (lambda (insn)
    (let ((Cop Cop-code inan))
      (rands Cop-rand insn))
    (cond
      ((gym=? op 'add) (asm-iii (mk-alu-sgn +) rands))
      ((gym=? op 'addi) (asm-iic (mk-alu-sgn +) rands))
      ((gym=? op 'addu) (asm-iic (mk-alu-unsgn +) rands))
      ((gym=? op 'addui) (asm-iic (mk-alu-unsgn +) rands))
      ((gym=? op 'and) (asm-iii (mk-bwise and?) rands))
      ((gym=? op 'andi) (asm-iic (mk-bwise and?) rands))
      ((gym=? op 'or) (asm-iii (mk-or rands)))
      ((gym=? op 'ori) (asm-iic (mk-or rands)))
      ((gym=? op 'seq) (asm-iii (mk-alu-rel =) rands))
      ((gym=? op 'seqi) (asm-iic (mk-alu-rel =) rands))
      ((gym=? op 'sequ) (asm-iic (mk-alu-urel =) rands))
      ((gym=? op 'sequi) (asm-iic (mk-alu-urel =) rands))
      ((gym=? op 'seq) (asm-iii (uk-alu-rel =) rands))
      ((gym=? op 'seqi) (asm-iic (uk-alu-rel =) rands))
      ((gym=? op 'sequ) (asm-iic (uk-alu-urel =) rands))
      ((gym=? op 'sequi) (asm-iic (uk-alu-urel =) rands))
      ((gym=? op 'slt) (asm-iii (mk-alu-rel <) rands))
      ((gym=? op 'slli) (asm-iic (mk-alu-rel <) rands))
      ((gym=? op 'slt) (asm-iic (uk-alu-rel <) rands))
      ((gym=? op 'slli) (asm-iic (uk-alu-rel <) rands))
      ((gym=? op 'sgt) (asm-iii (mk-alu-rel >) rands))
      ((gym=? op 'sgti) (asm-iic (mk-alu-rel >) rands))
      ((gym=? op 'sgt) (asm-iic (uk-alu-rel >) rands))
      ((gym=? op 'sgti) (asm-iic (uk-alu-rel >) rands))
      ((gym=? op 'sge) (asm-iii (mk-alu-rel >=) rands))
      ((gym=? op 'sgei) (asm-iic (mk-alu-rel >=) rands))
      ((gym=? op 'sgeu) (asm-iic (mk-alu-urel >=) rands))
      ((gym=? op 'sgeui) (asm-iic (mk-alu-urel >=) rands))
      ((gym=? op 'sgt) (asm-iii (uk-alu-rel >) rands))
      ((gym=? op 'sgti) (asm-iic (uk-alu-rel >) rands))
      ((gym=? op 'sge) (asm-iii (uk-alu-rel >=) rands))
      ((gym=? op 'sgei) (asm-iic (uk-alu-rel >=) rands))
      ((gym=? op 'sgeu) (asm-iic (uk-alu-urel >=) rands))
      ((gym=? op 'sgeui) (asm-iic (uk-alu-urel >=) rands))
      ((gym=? op 'slt) (asm-iii (uk-alu-rel <) rands))
      ((gym=? op 'slli) (asm-iic (uk-alu-rel <) rands))
      ((gym=? op 'sgt) (asm-iii (uk-alu-rel >) rands))
      ((gym=? op 'sgti) (asm-iic (uk-alu-rel >) rands))
      ((gym=? op 'sge) (asm-iii (uk-alu-rel >=) rands))
      ((gym=? op 'sgei) (asm-iic (uk-alu-rel >=) rands))
      ((gym=? op 'sgeu) (asm-iic (uk-alu-urel >=) rands))
      ((gym=? op 'sgeui) (asm-iic (uk-alu-urel >=) rands))
      ((gym=? op 'atel) (asm-iic (mk-alu-sgn <=) rands))
      ((gym=? op 'ateli) (asm-iic (mk-alu-sgn <=) rands))
      ((gym=? op 'sne) (asm-iii (mk-alu-sgn <=) rands))
      ((gym=? op 'snei) (asm-iic (mk-alu-sgn <=) rands))
      ((gym=? op 'sneu) (asm-iic (mk-alu-sgn <=) rands))
      ((gym=? op 'sneui) (asm-iic (mk-alu-sgn <=) rands))
      ((gym=? op 'sra) (asm-iii (mk-alu-sgn <=) rands))
      ((gym=? op 'sr) (asm-iic (mk-alu-sgn <=) rands))
      ((gym=? op 'sraei) (asm-iic (mk-alu-sgn <=) rands))
      ((gym=? op 'sr) (asm-iic (mk-alu-sgn <=) rands))
      ((gym=? op 'sraei) (asm-iic (mk-alu-sgn <=) rands))
      ((gym=? op 'sr) (asm-iic (mk-alu-sgn <=) rands))
      ((gym=? op 'sub) (asm-iii (mk-alu-unsgn >=) rands))
      ((gym=? op 'subi) (asm-iic (mk-alu-unsgn >=) rands))
      ((gym=? op 'sub) (asm-iic (mk-alu-unsgn >=) rands))
      ((gym=? op 'subi) (asm-iic (mk-alu-unsgn >=) rands))
      ((gym=? op 'xor) (asm-iii (mk-alu-xor rands))
      ((gym=? op 'xori) (asm-iic (mk-alu-xor rands))
      ()))
      )
((sym=? op 'beqz) (asm-b2 (lambda (r d) (if (= (get-reg r) 0) d (- *npc*))) rands))
((sym=? op 'bfpt) (asm-b1 (lambda (d) (if (~ *fp-cond*) (~ *npc*) d)) rands))
((sym=? op 'bfpf) (asm-b1 (lambda (d) (if (~ *fp-cond*) d (~ *npc*))) rands))
((sym=? op 'bnez) (asm-b2 (lambda (r d) (if (= (get-reg r) 0) (~ *npc*) d)) rands))
((sym=? op 'j) (asm-b1 id rands))
((sym=? op 'jr) (asm-b1 get-reg rands))
((sym=? op 'jal) (asm-b1 (lambda (d) (begin (set-reg! RETADR (+ (~ *pc*) 4)) d)) rands))
((sym=? op 'jalr) (asm-b1 (lambda (r) (begin (set-reg! RETADR (+ (~ *pc*) 4)) (get-reg r))) rands))
((sym=? op 'lb) (asm-load (lambda (v) (sext8->32 (remainder v two^8))) rands))
((sym=? op 'lbu) (asm-load (lambda (v) (remainder v two^8)) rands))
((sym=? op 'lh) (asm-load (lambda (v) (sext16->32 (remainder v two^16))) rands))
((sym=? op 'lhi) (asm-lhi rands))
((sym=? op 'lhu) (asm-load (lambda (v) (remainder v two^16)) rands))
((sym=? op 'lw) (asm-load id rands))
((sym=? op 'sb) (asm-store (let ((fn (set-bit-field 0 8))) (lambda (ov v) (fn ov (remainder v two^8))) rands))
((sym=? op 'sh) (asm-store (let ((fn (set-bit-field 0 16))) (lambda (ov v) (fn ov (remainder v two^16))) rands))
((sym=? op 'sw) (asm-store (lambda (ov v) v) rands))
((sym=? op 'div) (asm-fff quotient rands))
((sym=? op 'divu) no-emulate)
((sym=? op 'mult) (asm-fff (mk-alu-sgn *)) rands))
((sym=? op 'multu) no-emulate)
(((sym=? op 'addd) (asm-ddd fl+ rands))
((sym=? op 'addf) (asm-fff fl+ rands))
((sym=? op 'divd) (asm-ddd fl/ rands))
((sym=? op 'divf) (asm-fff fl/ rands))
((sym=? op 'eqd) (asm-fp-rel 'g fl= rands))
((sym=? op 'eqf) (asm-fp-rel 'f fl= rands))
((sym=? op 'ged) (asm-fp-rel 'g fl>= rands))
((sym=? op 'gef) (asm-fp-rel 'f fl>= rands))
((sym=? op 'gtd) (asm-fp-rel 'g fl> rands))
((sym=? op 'gtf) (asm-fp-rel 'f fl> rands))
((sym=? op 'ltd) (asm-fp-rel 'g fl< rands))
((sym=? op 'ltf) (asm-fp-rel 'f fl< rands))
((sym=? op 'led) (asm-fp-rel 'g fl<= rands))
((sym=? op 'lef) (asm-fp-rel 'f fl<= rands))
((sym=? op 'multd) (asm-ddd fl* rands))
((sym=? op 'multf) (asm-fff fl* rands))
((sym=? op 'ned) (asm-fp-rel 'g (lambda (x y) (not (fl= x y))) rands))
((sym=? op 'nef) (asm-fp-rel 'f (lambda (x y) (not (fl= x y))) rands))
((sym=? op 'subd) (asm-ddd fl- rands))
((sym=? op 'subf) (asm-fff fl- rands))
((sym=? op 'ld) no-emulate)
((sym=? op 'lf) no-emulate)
((sym=? op 'sd) no-emulate)
((sym=? op 'sf) no-emulate)
((sym=? op 'trap) no-emulate)
((sym=? op 'nop) (lambda 0) the-unit))
(define unparse-ocode
  (lambda (insn)
    (let ((op (op-code insn))
          (rands (op-rands insn)))
      (cond
       ((sym=? op 'labeldef)  
        (string-append (match rands ((label->rands 1 l) l)) ":
                     "))
       ((sym=? op 'stringdef) 
        (string-append ".asciz \\" (match rands ((string->rands s s)) "\\"
                     (char->string #\newline) ".align 2\n                     "))
       ((sym=? op 'worddef)    
        (string-append ".word 
                     (int->string (match rands ((word->rands w w)))))
       (memq op '(add addu and or seq sequ sge sgtu sle sleu  
                  sll slt slti smlu sne sneu sra srl sub subu xor))
        (unp-iii op rands))
       (memq op '(addi addui audi oni aeqi seqi sgei sgeui sgti sgtui slii  
                  salui salui siti situi snei sncui srai srii subi subui xori))
        (unp-iic op rands))
       (memq op '(beqz beuez))  
        (unp-b2 op rands))
       (memq op '(bfpt bfpf beuez j jr jal jalr)) (unp-b1 op rands))
       (memq op '(lb lbu lh lhu lw)) (unp-load op rands))
       ((sym=? op 'lhi) (unp-lhi op rands))
       (memq op '(sh shw sw)) (unp-store op rands))
       (memq op '(div divu mult multu)) (unp-fff op rands))
       (memq op '(addf divf multf subf)) (unp-fff op rands))
       (memq op '(addd divd multd subd)) (unp-fff op rands))
       (memq op '(eqf gef gtf lef lef nef)) (unp-fp-rel 'f op rands))
       (memq op '(eqd ged gtd ltd led led ned)) (unp-fp-rel 'g op rands))
       ((sym=? op 'nop) "nop")
       ((sym=? op 'cvtb2f) (unp-cnv op 'f 'g rands))
       ((sym=? op 'cvtb2i) (unp-cnv op 'd 'g rands))
       ((sym=? op 'cvtf2d) (unp-cnv op 'g 'f rands))
       ((sym=? op 'cvtf2i) (unp-cnv op 'd 'f rands))
       ((sym=? op 'cvtf2i) (unp-cnv op 'f 'd rands))
       ((sym=? op 'movd) (unp-cnv op 'g 'rands))
       ((sym=? op 'movf) (unp-cnv op 'f 'rands))
       ((sym=? op 'movfp2i) (unp-cnv op 'i 'd rands))
       ((sym=? op 'movi2fp) (unp-cnv op 'd 'i rands)))))

;; (memq op '(1d lf sd sf)) ??)
;; (sym=? op 'trap) ??)

C.2 compiler/bits.fx

The contents of the file compiler/bits.fx:

;; -- Mode: Scheme; Package: SCHEME --
;;; Bit banging stuff
;;;
(define two-8 (expt 2 8))
(define two-15 (expt 2 15))
(define two-16 (expt 2 16))
(define two-31 (expt 2 31))
(define two-32 (expt 2 32))

(define int16? ; (-> (int) bool)
  (lambda (n) (and (<= n (- two-15 1)) (>= n (- 0 two-15)))))

(define int32?
  (lambda (n) (and (<= n (- two-31 1)) (>= n (- 0 two-31)))))

(define mk-bignum2mint
  (lambda (size-in-bits)
    (let ((two-n (expt 2 size-in-bits)))
      (lambda (bn)
        (if (>= bn 0) bn (+ two-n bn))))))

(define bignum2mint (mk-bignum2mint 32))

(define mk-bignum2bits
  (lambda (size-in-bits)
    (letrec ((loop
      (lambda (s v i)
        (if (= i size-in-bits)
          s
          (loop (string-append (if (odd? v) "1" "0") s) (quotient v 2) (+ i 1))))))
      (lambda (bn) (loop "" (bignum2signed bn) 0))))))

(define bignum2bits (mk-bignum2bits 32))

(define mk-mint2bignum
  (lambda (size-in-bits)
    (let ((two-n (expt 2 size-in-bits))
          (two-n-1 (expt 2 (- size-in-bits 1))))
      (lambda (mi)
        (if (<= mi (- two-n-1 1)) mi (- mi two-n))))))

(define mint2bignum (mk-mint2bignum 32))

(define get-bit-field
  ;; bit-field : int * int -> (int -> int)
  ;; Access a bit field of an integer.
  (lambda (position size)
    (let ((shift (expt 2 position))
          (mask (expt 2 size))
          (lambda (n)
            (remainder (quotient n shift) mask))))))
(define set-bit-field
  ;;; set-bit-field : int * int -> (int * int -> int)
  ;;; Initialize a bit field of an integer. Assumes the field currently
  ;;; contains zero, because we don't need to clear it first in that case.
  (lambda (position size)
    (let ((shift (expt 2 position))
           (mask (expt 2 size)))
      (lambda (n value)
        (if (or (< value 0)
               (>= value mask))
            (error "bit field out of range" position size n value)
            (+ n (* value shift)))))))

;; Parse and unpars hex numbers to make examination of bit-operations
;; easier.
;;
;;
(define h2i
  ;; Convert string of hex numbers into integer. Requires string to
  ;; be in form "0xh..." where h is in [0-9a-z]
  (letrec ((loop (lambda (s i v)
                  (if (= i (string-length s))
                      v
                      (let* ((c (string-ref s i))
                              (h (if (char-numeric? c)
                                   (- (char->int c) (char->int \0))
                                   (+ (- (char->int c) (char->int \a)) 10))))
                              (loop s (+ i 1) (+ (* v 16) h)))))
        (lambda (s) (loop s 2 0))))

(define i2h
  (letrec ((loop (lambda (v l)
                   (if (= v 0)
                       l
                       (let* ((h (remainder v 16))
                               (c (if (<= h 9)
                                    (int->char (+ h (char->int \0)))
                                    (int->char (+ (- h 10) (char->int \a)))))
                               (loop (quotient v 16) (cons c l))))))
       (lambda (v)
         (if (= v 0)
             "0x0"
             (list->string (cons \0 (cons \x (loop v (null)))))))))

C.3 compiler/dlxsim.fx

The contents of the file compiler/dlxsim.fx:

;; -- Mode: Scheme; Package: SCHEME --
(define *semispace-size* (ref 500)) ; Slots in each space
(define *stack-size* (ref 500)) ; Slots in stack
(define *noisy-gc* (ref #t)) ; Pretty-print registers
(define *program-start* (ref 0)) ; Addr of 1st word of code
(define *break-points* (ref 'list)))

(define *entire-memory-size* (ref 0)) ; Total words in simulator mem
(define *mem* (ref (make-vector 0))
(define *end-program* (ref 0)) ; Addr of last word of code
(define *this-semispace* (ref 0)) ; (Tagged) address of first
(define *other-semispace* (ref 0)) ; word in space
(define *this-semispace-ends* (ref 0)) ; beyond last word in space
(define *other-semispace-ends* (ref 0))

(define *reg* (generate-vector 32 (lambda (index) 0)))
(define *freg* (generate-vector 32 (lambda (index) 0.0))
(define *fp-cond* (ref 0))
(define *pc* (ref 0))
(define *npc* (ref 0))
(define *halt-emulate?* (ref #f))
(define *label-table* (ref (null)))

(define *instruction-count* (ref 0))
(define *num-gcs* (ref 0))
(define *gc-words-copied* (ref 0))
(define *total-allocation* (ref 0))
(define *total-allocs* (ref 0))
(define *max-stack-size* (ref 0))
;; Create a vector to represent the DLX machine's memory. If required
;; memory size hasn't grown much since last time, then don't cons up
;; the new vector so we avoid generating garbage.
(let ((old-len (vector-length (*mem*)))
  (if (or (< old-len (*entire-memory-size*))
      (> (- old-len 600) (*entire-memory-size*)))
    (:= *mem*
      (generate-vector (*entire-memory-size*) (lambda (i) 0)))
    (:= *entire-memory-size* old-len)))

;; Now that we have a memory vector, assemble code into that vector
(asm-pass2 (*ocode-list*)); Assmbl emulator thunks in2 mem vec

(define restart-emulator
  (lambda ()
    (begin
      ;; Zero out registers
      (letrec ((oop (lambda Mi (if (>= i 32)
              the-unit
            (begin (set-reg! i 0)
              (set-frog! i 0.0)
              (loop (+ i 1)))))
        (loop 0))

      ;; Initialize important system registers
      (set-reg! SP (- (* (*entire-memory-size*) 4) 4))
      (set-reg! HP (+ (* *end-program*) 1))
      (goto (*program-start*)); Set pc registers
      (:= *halt-emulate?* #f)

      ;; Set up heap
      (:= *this-semispace* (get-reg HP))
      (:= *this-semispace-end* (+ (get-reg HP) (* *semispace-size* 4)))
      (:= *other-semispace* (*this-semispace-end*))
      (:= *other-semispace-end* (+ (*other-semispace*) (* *semispace-size* 4)))

      ;; Clear statistics variables
      (:= *num-gcs* 0)
      (:= *gc-words-copied* 0)
      (:= *total-allocation* 0)
      (:= *total-allocs* 0)
      (:= *instruction-count* 0)
      (:= *max-stack-size* 0)

      ;; Set up an initial frame to return through
      (set-reg! ATEMP FrameSize)
      (allocate-block-of-memory)
      (set-reg! FP (get-reg ATEMP))
      (set-slot! (get-reg FP) 0 0)
      (set-slot! (get-reg FP) 1 SP)
      (set-slot! (get-reg FP) 2 -16); Return to __EXIT system routine
      (set-slot! (get-reg FP) 3 0)
      (save-regs-into-frame)
**C.3 compiler/dlxsim.fx**

---

```scheme
;; Set-up an initial closure for calling START_1
(set-reg! ATEMP 4) ; 2 words (tagged)
(allocate-block-of-memory)
(set-reg! ARGO (get-reg ATEMP)); Create dummy closure
(set-slot! (get-reg ARGO) 0 (*program-start*))
(set-slot! (get-reg ARGO) 1 0)))

;; Routines to do emulations after machine has been set up
;;
(define emulate
  (lambda ()
    (begin (emulate-one-instruction)
      (if (*halt-emulate?*) the-unit (emulate)))))

(define rerun
  ;; rerun the compiled program from the top...
  (lambda () (begin (restart-emulator) (emulate))))

(define step
  ;; Single-step program, printing out instruction just emulated.
  (lambda ()
    (let ((temp (*verbose-flag*)))
      (begin (:= *verbose-flag* #f)
        (display-one-instruction (*pc*)
         (nstep 1)
         (display-one-instruction (*pc*)
         (:= *verbose-flag* temp))))))

(define nstep
  (lambda (n)
    (letrec ((loop (lambda (i)
      (cond ((>= i n) the-unit)
        ((*halt-emulate?*)
          (begin (display "Execution terminated.")
            the-unit))
        (else (begin (emulate-one-instruction)
          (loop (+ i 1))))))
      (begin (loop 0) (dump))))))

;;
;; Two passes of assembler
;;
```

---

42
(define asm-pass1
  ;; Calculate values of labels and enter them into symbol table. Set
  ;; *end-program* to (untagged) address of 1st word after program code.
  (lambda (ocode)
    (letrec ((loop (lambda (ocode pc)
          (if (null? ocode)
              pc
              (match (car ocode)
                ((ocode° 'labeldef (label->rands" label))
                  (begin (enter-label label pc)
                          (loop (cdr ocode) pc)))
                ((ocode" 'stringdef -) (loop (cdr ocode) (+ pc 4)))
                (_ (loop (cdr ocode) (+ pc 4)))))))
      (: *end-program* (loop ocode (* program-start*))))))

(define asm-pass2
  ;; Place emulation thunks into memory
  (lambda (ocode)
    (letrec ((loop (lambda (ocode pc)
          (if (null? ocode)
              the-unit
              (match (car ocode)
                ((ocode° 'labeldef _) (loop (cdr ocode) pc))
                ((ocode" 'stringdef (string->rands" string))
                  (begin (set-mem! PC string)
                          (loop (cdr ocode) (+ pc 4))))
                ((ocode" 'worddef (word->rands" word))
                  (begin (set-mem! PC word)
                          (loop (cdr ocode) (+ pc 4))))
                (_
                  (begin (set-mem! PC (asm (car ocode)))
                          (loop (cdr ocode) (+ pc 4))))))))
      (loop ocode 0))))

;;
;; Symbol table for labels -- set up for two-way mapping
;;
(define enter-label
  (lambda (l v) (= *label-table* (cons (tuple 1 v) (* label-table*))))))

(define label2num
  (lambda (l)
    (letrec ((loop (lambda (lst)
            (cond ((null? lst) (error "Unknown label: 1"))
                  ((string=? l (tuple-ref (car lst) 0))
                    (tuple-ref (car lst) 1))
                  (else (loop (cdr lst)))))))
      (loop (* label-table*))))))

43
(define num2label
  (lambda (n)
    (letrec ((loop (lambda (1st)
                     (cond ((null? lst) "")
                           ((= n (tuple-ref (car lst) 1))
                            (tuple-ref (car lst) 0))
                           (else (loop (cdr lst))))))
     (loop (- *label-table*)))))

;; Routines to change state of emulator according to instructions
;;
(define goto
  (lambda (new-pc)
    (begin
      (if (memq new-pc "break-points")
        (error "Break-point: type (proceed) to continue.")
        (:= *pc* new-pc)
        (:= *npc* (+ new-pc 4)))))

(define emulate-one-instruction
  (lambda ()
    (begin
      (:= *instruction-count* (+ (- *instruction-count*) 1))
      (if (~verbose-flag*)
        (display-one-instruction (~ *pc*)
        the-unit)
      (if (< (~ *pc*) 0)
        (begin
          ;; fake system subroutines have negative addresses...
          ;; they're also always called with jal's (retaddr in r31)
          (system-routine (~ *pc*)
            ; call the service
            ( goto (get-reg RETADR))) ; fake the return.
          (let ((old-pc (~ *pc*))
            (begin (goto (~ *npc*)
                      ((get-mem old-pc))))))))

;; Procedures to access and change emulator state
;;
(define get-mem
  (lambda (address)
    (if (= (remainder address 4) 0)
      (vector-ref (~ *mem*) (quotient address 4))
      (error "Unaligned read.")
      )))

(define set-mem!
  (lambda (address val)
    (if (= (remainder address 4) 0)
      (vector-set! (~ *mem*) (quotient address 4) val)
      (error "Unaligned write.")
      )))
(define get-slot (lambda (ptr slot) (get-mem (+ ptr (otag slot))))
(define set-slot! (lambda (ptr slot v) (set-mem! (+ ptr (otag slot)) v)))

(define get-reg  (lambda (regnum) (vector-ref *reg* regnum))
(define set-reg! (lambda (regnum val) (vector-set! *reg* regnum val)))
(define get-freg (lambda (fregnum) (vector-ref *freg* fregnum)))
(define set-freg! (lambda (fregnum val) (vector-set! *freg* fregnum val)))

(define get-dreg
  (lambda (fregnum)
    (if (odd? fregnum)
      (error "Bad double register" fregnum)
      (vector-ref *frog* fregnum))))

(define set-dreg!
  (lambda (fregnum val)
    (if (odd? fregnum)
      (error "Bad double register" fregnum)
      (begin (vector-set! *frog* fregnum val)
              (vector-set! *freg* (+ fregnum 1) -3.21))))

;; Routines to inspect the state of the emulator
(define dump
  ;; Display register values
  (lambda ()
    (letrec ((cols 4) (width 10)
      (loop (lambda (i)
        (if (> i 32)
            the-unit
            (let ((rval (i2h (get-reg i))))
              (begin (display (pad rval width)) (display rval)
                     (if (= (remainder i cols) (- cols 1))
                        (newline)
                        (the-unit))
                     (loop (+ i 1))))))))

(define find-insn
  (lambda (ocode pc)
    (cond ((sym=? (op-code (car ocode)) 'labeldef)
      (find-insn (cdr ocode) pc))
      ((<= pc 0) (car ocode))
      ((sym=? (op-code (car ocode)) 'stringdef)
       (find-insn (cdr ocode) (- pc 4)))
      (else (find-insn (cdr ocode) (- pc 4))))))
(define display-one-instruction
  (lambda (pc-i)
    (if (< pc-i 0)
        (begin (display "**** ") (display (num2label pc-i)) (newline))
        (let ((l (num2label pc-i)))
            (begin
                (newline)
                (if (> (string-length l) 0)
                    (begin (display " ") (display l) (display ":") (newline))
                the-unit)
                (if (< pc-i 1000) (display "0") the-unit)
                (if (< pc-i 100) (display "0") the-unit)
                (if (< pc-i 10) (display "0") the-unit)
                (display pc-i) (display "...
                (display (unparse-opcode (find-insn (*opcode-list*) pc-i))))))))

(define disasm ;; print program in memory (to see if it's uncorrupted)
  (lambda ()
    (letrec ((loop (lambda (i)
        (if (>= i (*end-program*))
            the-unit
            (begin (display-one-instruction i)
                  (loop (+ i 4)))))))
      (loop (* program-start*)))))

(define pb ;; print heap block
  (lambda (p)
    (letrec ((blksize (do-itag (get-slot p -1)))
        (loop (lambda (i)
            (begin (loop (+ i 1))))))
      (begin
        (display "Pointer=") (display p)
        (display " Blocksize=") (display blksize) (newline)
        (loop 0))))))

46
(define eval-immed
  ;; this is *very* crude, but sufficient for now...
  (define av-parse-expr
    (lambda (toks)
      (let ((c (string-ref (car (list-tail (list-tail toks) 2)) 0)))
        (cond ((char=? c #\( ) ;; (expr) ...
            (begin (:= toks (cdr (list-tail (list-tail toks) 2))))
              (mint2bignum (avpe-recurse toks)))
          ((char-numeric? c) ;; number
            (let ((n (av-parse-number (list-tail (list-tail (list-tail toks) 2)))))
              (begin (:= toks (cdr (list-tail (list-tail toks) 2)))
                    n)))))
        ;; LABEL | LABEL + 1
        (let ((n (label2num (list-tail (list-tail (list-tail toks) 2))))
          (if (and (> (length (list-tail toks)) 3)
            (char=? (string-ref (list-tail (list-tail (list-tail toks) 2)) 0) #\+)) ;; LABEL + 1
            (begin (:= toks (list-tail (list-tail (list-tail toks) 2)))
                  (+ 1 n))
                (begin (:= toks (cdr (list-tail toks)))
                      n))))))))
  (define avpe-recurse
    (lambda (toks)
      (let ((n (bignum2mint (av-parse-expr toks))))
        (if (not (char=? (string-ref (list-tail (list-tail (list-tail toks) 2)) 0) #\+))
          (error "invalid av-expr syntax")
          (cond ((char=? (string-ref (list-tail (list-tail (list-tail toks) 2)) 0) #\&)
            (begin (:= toks (list-tail (list-tail (list-tail toks) 2)))
                   (modulo n 65536)))
            ((char=? (string-ref (list-tail (list-tail (list-tail toks) 2)) 0) #\>)
              (begin (:= toks (list-tail (list-tail (list-tail toks) 2)))
                     (quotient n 65536)))
            (else (error "invalid av-expr syntax"))))))))
  (define avpe-recurse
(define av-parse-number
  ;; Assume we got a decimal number
  (lambda (s)
    (letrec ((loop
               (lambda (n i)
                 (if (= i (string-length s))
                     n
                     (loop (+ (* 10 n)
                                   (- (char->int (string-ref s i)) (char->int #\0)))
                                   (+ i 1)))))))
    (loop 0 0))))

;; A crude lexer for a crude parser...

(define av-tokenize
  ;; Real simple lexer for arithmetic expressions in assembly code
  (lambda (s)
    (letrec ((loop
               (lambda (s i)
                 (let ((Q
                        (av-kill-whitespace s i)))
                   (if (> j (string-length s)) (null)
                        (let ((k (av-next-tok s j)))
                          (cons (substring s j k)
                                (loop s k))))))
    (loop s 0))))

(define av-kill-whitespace
  (lambda (s i)
    (if (and (< i (string-length s)) (char-whitespace? (string-ref s i)))
      (av-kill-whitespace s (+ i 1))
      i))

(define av-next-tok
  (lambda (s i)
    (let ((c (string-ref s i))
      (len (string-length s)))
      (cond
       ((char=? c #\A) (+ i 1))
       ((char=? c #\#) (+ i 1))
       ((av-sym? c) (letrec ((loop (lambda (i)
                                  (if (or (= i len) (not (av-sym? (string-ref s i))))
                                      i
                                      (loop (+ i 1)))))
                           (loop (+ i 1)))))
       ((av-punct? c) (letrec ((loop (lambda (i)
                                      (if (or (= i len) (not (av-punct? (string-ref s i))))
                                          i
                                          (loop (+ i 1)))))
                            (loop (+ i 1))))))

..
(define av-sym?
  (lambda (c) (or (char-alphabetic? c) (char-numeric? c) (char=? c #\_)))))

(define av-punct?
  (lambda (c) (and (not (av-sym? c)) (not (char=? c #\.))
                   (not (char=? c #\()) (not (char-whitespace? c))))))

C.4 compiler/exp.fx

The contents of the file compiler/exp.fx:

; -- Mode: Scheme; Package: SCHEME; --

; Data types for use by the front-end of the compiler
; (parser, type reconstructor, exp2ic)

; Expressions

(define-datatype exp
  (variable->exp (refof type) sym) ; E :=
  (bool->exp (refof type) bool) ; I
  (char->exp (refof type) char) ; | Bool
  (int->exp (refof type) int) ; | Char
  (string->exp (refof type) string) ; | String
  (sym->exp (refof type) sym) ; | (symbol Sym)
  (conditional->exp (refof type) exp exp exp) ; | (if E1 E2 E3)
  (begin->exp (refof type) (listof exp)) ; | (begin E1 ...)
  (abstraction->exp (refof type) (listof sym) exp) ; | (lambda (I*) E)
  (combination->exp (refof type) exp (listof exp)) ; | (EO E*)
  (binder->exp (refof type) (listof definition) exp) ; | (let ((I E)* E)
  (recursion->exp (refof type) (listof definition) exp) ; | (letrec ((I E)* E)
  )

(define expression-type
  (lambda (exp)
    (match exp
      ((variable->exp type-ptr _) ("type-ptr"))
      ((bool->exp _ _) boolean-type)
      ((char->exp _ _) character-type)
      ((int->exp _ _) integer-type)
      ((string->exp _ _) string-type)
      ((sym->exp _ _) symbol-type)
      ((conditional->exp type-ptr _ _) ("type-ptr"))
      ((begin->exp type-ptr _ _) ("type-ptr"))
      ((abstraction->exp type-ptr _ _) ("type-ptr"))
      ((combination->exp type-ptr _ _) ("type-ptr"))
      ((binder->exp type-ptr _ _) ("type-ptr"))
      ((recursion->exp type-ptr _ _) ("type-ptr"))
      )))

(define-datatype definition ; (I E)
  (make-definition sym exp))
(define (definition-name d)
  (match d
    (((make-definition name value) name)))

(define (definition-value d)
  (match d
    (((make-definition name value) value)))

; Types

(define-datatype type
  (base->type sym) ; base type (bool, int, string, symbol, unit)
  (tvariable->type tvariable) ; type variable
  (compound->type sym ; ->, listof, etc.
    (listof type))
  (unknown->type) ; marking yet unconstrained type variables)

(define boolean-type (base->type (symbol bool)))
(define character-type (base->type (symbol char)))
(define integer-type (base->type (symbol int)))
(define string-type (base->type (symbol string)))
(define symbol-type (base->type (symbol sym)))
(define unit-type (base->type (symbol unit)))
(define unknown-type (unknown->type))

(define same-constructor? eq?)
(define arrow-constructor (symbol ->))

(define make-arrow-type
  (lambda (arg-types body-type)
    (compound->type arrow-constructor (cons body-type arg-types))))

(define arrow-takes
  (lambda (ty)
    (match ty
      (((compound->type '-> (cons' bt at)) at)))))

(define arrow-returns
  (lambda (ty)
    (match ty
      (((compound->type '-> (cons' bt at)) bt)))))

(define is-arrow-type?
  (lambda (ty)
    (match ty
      (((compound->type '-> _) #t)
        (_ #f)))))

; Type names are symbols
(define same-name? sym=?)

; Type schemas
(define-datatype schema
    (make-schema (listof tvariable) type))

(define-datatype tvar-or-schema
    (tvar->tvar-or-schema tvariable)
    (schema->tvar-or-schema schema))

(define (schema-generics s)
    (match s
        ((make-schema "generics typ" generics)))

(define (schema-type s)
    (match s
        ((make-schema "generics typ" typ)))

; Unit value
(define the-unit (:= (ref 0) 0))

C.5 compiler/exp2ic.fx

The contents of the file compiler/exp2ic.fx:

;;;; Mode: Scheme; Package: SCHEME ;;-
;;;; exp2ic.fx -- convert expr's to icode.
;;;; Depth-first, syntax-driven translation of expressions into
;;;; intermediate code. The depth-first translation is stack hungry,
;;;; but we don't expect extremely deep expressions.
;;;; The resulting intermediate code is stored on the list "icode-list."
;;;; The order of the icode on this list is preserved in the ultimate
;;;; machine code, i.e., the order of the list represents flow of
;;;; control through the icode.
;;;; One tricky part is how to generate code for lambda's and letrec's
;;;; because the code for the body does not belong in-line with the call
;;;; to the body. To handle this, we have a list called
;;;; "icode-to-be-emitted" which is a list of blocks of icode to be
;;;; emitted later. When a lambda body is compiled, the icode for the
;;;; body is put on the icode-to-be-emitted list, while the icode for
;;;; the closure is put into the current block of icode.

(define icode-list (ref (null))) ; intermediate code collected on this list
(define generate-icode
  (lambda (exp)
    (let* ((ignore (reset-label-counter))
           (startlabel (new-label "START")))
      (begin
        (lib-init)
        (: icode-to-be-emitted (null))
        (: library-icode (null))
        (: library-count 0)
        (: sym-label-env (mk-empty-env add-sym))

        ;; Initialize icode list with translation of a return of the
        ;; top-level expression along with a label for the start of
        ;; the program.
        (: icode-list
          (list (labeldef->icode startlabel)
                (return->icode (translate exp standard-c-t-env))))

        ;; Translating the top-level expression caused some code,
        ;; e.g., the bodies of lambda's and letrec's, to be put on
        ;; an auxiliary list to be emitted later. Now's the time to
        ;; emit it.
        (emit-delayed-ic))))

(define icode-to-be-emitted
  ;; Icode to be emitted later is put on this list in blocks. A block
  ;; is a list of icode. Each block is emitted in order, i.e., the
  ;; order of the list specifies control flow. No order between blocks
  ;; is guaranteed. Usually each block starts with a labeldef->icode so
  ;; that other blocks can refer to it.
  (ref (null)))

(define emit-ic-later
  (lambda (codelist)
    (:= icode-to-be-emitted (cons codelist (- icode-to-be-emitted))))

(define emit-delayed-ic
  (lambda ()
    (letrec ((flatten ; turns a list of lists into a list
               (lambda (iclist-list iclist)
                (if (null? iclist-list)
                    iclist
                    (flatten (cdr iclist-list)
                             (append (car iclist-list)
                                     (append (cdr iclist-list) (null))))))))
      (begin
        (:= icode-list (append (" icode-list"
                              (flatten " icode-to-be-emitted") (null)))
        (:= icode-list (append (" icode-list"
                              (flatten " library-icode") (null))))))

52
(define library-icode
  ;; The library functions also generate delayed icode. To make the output
  ;; clearer (i.e., to hide the ugly library stuff at the end of the outputed
  ;; code), we put the library icode on a separate list. Before translating
  ;; library code, call swap-delayed-icode-lists to swap the
  ;; icode-to-be-emitted list with another list for library code. When the
  ;; library code is done, the lists should be swapped back.
  (ref (null)))

(define library-count
  ;; When compiling library code, we may have to compile other library
  ;; procedures. To make sure that the delayed icode list gets set
  ;; back to the user's list when we stop compiling library code, we
  ;; keep count of how deep we are into compiling library calls within
  ;; library calls.
  (ref 0))

(define swap-delayed-icode-lists
  (lambda ()
    (let ((tmp (- icode-to-be-emitted)))
      (begin (:= icode-to-be-emitted (- library-icode))
             (:= library-icode tmp))))

(define enter-library
  (lambda ()
    (begin (if (= 0 (- library-count)) (swap-delayed-icode-lists) the-unit)
            (:= library-count (+ (- library-count) 1))))

(define leave-library
  (lambda ()
    (begin (:= library-count (- (- library-count) 1))
           (if (= 0 (- library-count)) (swap-delayed-icode-lists) the-unit))))
(define translate
  (lambda (exp c-t-env)
    (match exp
      ((int->exp _ n) (trans-integer n))
      ((char->exp _ c) (trans-integer (char->int c)))
      ((bool->exp _ b) (trans-integer (if b 1 0)))
      ((string->exp _ s) (trans-string s))
      ((sym->exp _ s) (trans-symbol s))
      ((variable->exp tr var) (trans-variable c-t-env (~ tr) var))
      ((conditional->exp tr t c a) (trans-conditional c-t-env (~ tr) t c a))
      ((begin->exp tr exprs) (trans-begin c-t-env (~ tr) exprs))
      ((combination->exp tr op args) (trans-call c-t-env (~ tr) op args))
      ((abstraction->exp tr args b) (trans-lambda c-t-env (~ tr) args b))
      ((binder->exp tr defs b) (trans-let c-t-env (~ tr) defs b))
      ((recursion->exp tr dots b) (trans-letrec c-t-env (~ tr) dels b))))

;; Integers
;;
;; (define trans-integer (lambda (n) (int->icode integer-type n)))

;; String
;;
;; (define trans-string
;;  ;; Since strings are vectors of characters, we need a way to translate
;;  ;; a string literal into a vector. We do this by making a symbol out
;;  ;; of the string literal, and using the sym->string to build the char
;;  ;; vector at run-time. (sym->string is in lib.fx)
;;
;;  (lambda (s)
;;    (let ((s2s-type (parse-type '(-> (sym) string))))
;;      (translate (combination->exp ;; combination
;;                  (ref string-type) ;; this is its type
;;                  (variable->exp (ref s2s-type) 'sym->string) ;; this is the function being called
;;                  (list (sym->exp (ref symbol-type) (string->sym s))))))
;;      ;; these are the arguments.
;;      standard-c-t-env)))))))

;; (symbol Name)
;;
;; (define trans-symbol
;;  (lambda (s)
;;    (labelref->icode symbol-type ((` sym-label-env) s))))

54
(define add-sym
  ;; If symbol not already in the environment, then make a label for
  ;; it and put the new label in the environ. Look in table.fx to see
  ;; how mk-empty-env and mk-binder work.
(lambda (key)
  (let (((label (new-label "SYM")))
        begin
    (emit-ic-later (list (labeldef->icode label) (sym->icode key)))
    (:= sym-label-env ((mk-binder sym=? key label (~ sym-label-env)))
     label)))

(define sym-label-env (ref (mk-empty-env add-sym)))

;; Identifiers

;; (define trans-variable
(lambda (c-t-env ty var)
  (match (c-t-lookup c-t-env var)
  ;; compile a normal var ref as its access path...
  ([(back+over->binding" back over) (var->icode ty back over)])
  ;; but compile primes as if they were (lambda (x y) (+ x y)).
  ([(primitive->binding' prim)
     (if (prim-constant? prim)
         (prim-constant-form prim) ;; For space, newline, etc. constants
         (prim-closure-form prim))])));; For primitive routines (e.g., +)

;; (if E1 E2 E3)

;; (define trans-conditional
(lambda (c-t-env ty tst con alt)
  (op->icode ty 'if (list (translate tst c-t-env)
                           (translate con c-t-env)
                           (translate alt c-t-env))))))

;; (begin E*)

;; (define trans-begin
(lambda (c-t-env ty exprs)
  (op->icode ty 'begin (map (lambda (x) (translate x c-t-env), exprs)))

;; (define trans-let
(lambda (c-t-env ty var val)
  (match (c-t-lookup c-t-env var)
  ;; compile a var ref as its access path...
  ([(back+over->binding" back over) (var->icode ty back over)])
  ;; but compile primes as if they were (lambda (x y) (+ x y)).
  ([(primitive->binding' prim)
     (if (prim-constant? prim)
         (prim-constant-form prim) ;; For space, newline, etc. constants
         (prim-closure-form prim))])));; For primitive routines (e.g., +)
;; (lambda (I* E) -- create a procedure
;;
(define trans-lambda
  (lambda (c-t-env ty formals body)
    (let ((label (new-label "LAMBDA")))
      (begin
        ;; 1. Translate body but emit it later.
        (emit-ic-later (list (labeldef->icode label)
            (body->icode
            ty
            (length formals)
            (return->icode
            (translate body (c-t-bind formals c-t-env))))))
        ;; 2. Translate the creation of the closure now
        (trans-closure ty label))))

(define trans-closure
  ;; The ENV reg is defined (as a reg number) in backend/gen.fx
  (lambda (ty proc-label)
    (alloc->icode ty 2 (list (labelref->icode unknown-type proc-label)
        (reg->icode unknown-type ENV)))))

;; (let ((I E)*) EO)
;; For now, compile this as ((lambda I* EO) E*);
;; Can be fancier in the future
;;
(define trans-let
  (lambda (c-t-env ty defa body)
    (let ((names (map definition-name defs))
            (vale (map definition-value
                defa)))
      (translate (combination->exp
            (ref (expression-type body))
            (abstraction->exp
            (ref (make-arrow-type (map expression-type vals)
            (expression-type body)))
            names
            body)
            vals)
            c-t-env))))
The ENV reg is defined (as a reg number) in backend/gen.fx

(define trans-letrec
  (lambda (c-t-env ty defs body)
    (let* ((label (new-label "LETREC"))
           (new-env (c-t-bind (map definition-name defs) c-t-env))
           (trans-vals (lambda (d) (translate d new-env))))
      (begin
        ;; 1. Translate the body of the letrec, but emit it later
        (emit-ic-later
         (list (labeldef->icode label)
               (letrec->icode
                (length defs)
                (map trans-vals (map definition-value defs))
                (return->icode (translate body new-env))))
        ;; 2. Translate a call to the letrec body
        (op->icode ty
          'call
          (list (trans-closure (make-arrow-type '() ty) label))))))
    ;; (EO E*)

(define trans-call
  (lambda (c-t-env ty op operands)
    (match op
      ((variable->exp"_ var)
        (match (c-t-lookup c-t-env var)
          ((primitive->binding" prim)
            ;;
            ;; Yes, all this was to pull out this one case.
            ;; Primitives get compiled in-line instead of as calls.
            ;;
            (trans-primitive-combination ty prim operands c-t-env))
          (_ (trans-unknown-combination ty op operands c-t-env))))
      (_ (trans-unknown-combination ty op operands c-t-env))))

(define trans-primitive-combination
  ;; Trans a call to a primitive procedure
  (lambda (ty prim operands c-t-env)
    ((prim-inline-form prim)
      ty
      (map (lambda (a) (translate a c-t-env)) operands))))

(define trans-unknown-combination
  ;; Trans a call to an "unknown" (i.e. computed) procedure
  (lambda (ty operator operands c-t-env)
    (op->icode ty
      'call
      (cons (translate operator c-t-env)
        (map (lambda (a) (translate a c-t-env)) operands))))
;; new-label : symbol -> sexpr
;;
(define label-counter (ref 0))

(define reset-label-counter
  (lambda () (:= label-counter 0)))

(define new-label
  (lambda (prefix)
    (begin (:= label-counter (+ (' label-counter) 1))
      (string-append (string-append prefix "_")
      (int->string (- label-counter))))))

;;
;; The stuff that follows is all for displaying intermediate code
;; (lists of labeled trees) prettily.
;;
(define unparse-icode
  (lambda (code)
    (letrec ((up-ic-list (lambda (cl) (map unparse-ic cl))))
      (match code
        ((labeldef->icode- lab) lab)
        ((noop->icode-) 'nop)
        ((int->icode" -n) n)
        ((string->icode" s) (list 'string-lit s))
        ((sym->icode" s) (list 'symbol-lit s))
        ((word->icode" w) (list 'word-lit w))
        ((labelref->icode" -lab) lab)
        ((reg->icode" -reg) (list 'r reg))
        ((op->icode" _ s clist) (cons s (map unparse-icode clist))))
        ((alloc->icode" _ i clist) (list 'alloc i (map unparse-icode clist)))
        ((var->icode" _ back over) (list 'var back over))
        ((body->icode" _ n b) '(body (,n) ,(unparse-icode b)))
        ((letrec->icode" n 1 e) ' (letrec ,(map unparse-icode 1) ,(unparse-icode e)))
        ((return->icode" code) (list 'return (unparse-icode code))))))

(define display-icode
  (lambda (code)
    (match code
      ((labeldef->icode" lab)
        (begin (newline) (display lab) (display ":")))
      (_ (begin (display " ") (pp (unparse-icode code))))))
(define old-display-icode ;; for list-of-trees-style stuff
(lambda (code)
(letrec
((sp (lambda () (display " ")))))
(print-ic-list (lambda (cl)
  (begin
    (display "( "
    (for-each (lambda (e) (print-icods e) (sp)) cl)
    (display ")")))
  (print-icode
   (lambda (code)
     (match code
      ((labeldef->icode lab)
        (begin
          (display lab) (display ":") )))
      ((noop->icode)
        (begin
t          ; no-op; ignore it.
          the-unit))
      ((int->icode n)
        (begin
          (display n)))
      ((string->icode s)
        (begin
          (display ".string ")(display s)))
      ((sym->icode s)
        (begin
          (display ".symbol ")(display s)))
      ((word->icode w)
        (begin
          (display ".word ")(display s)))
      ((labelref->icode lab)
        (begin
          (display lab)))
      ((reg->icode reg)
        (begin
          (display "r ")(display reg)))
      ((op->icode s clist)
        (begin
          (display s) (sp) (print-ic-list clist)))
      ((alloc->icode i clist)
        (begin
          (display "alloc/")(display i) (print-ic-list clist)))
      ((var->icode back over)
        (begin
          (display "vat(")(display back) (display ",")
          (display over) (display ")")))
      ((body->icode n b)
        (begin
          (display ".body/")(display n)
          (display " ") (print-icode b)))
      ((letrec->icode n 1 e)
        (begin
          (display "letrec/"
            (display n)
            (display ""
              (for-each (lambda (x)
                (begin
                  (print-icode x) (display " \"
                    ;; the // terminates icode trees to make
                    ;; them more easily readable.
                    1)
                (display " ")
              (print-icode e))))
            (print-icode e)))
      ((return->icode code)
        (begin
          (print-icode code))))
     (match code
      ((labeldef->icode _)
        (print-icode code)) ; special hack for indenting
      (_ (begin
          (display " ")
          (print-icode code)))))))))))

59
The contents of the file compiler/ic2oc.fx:

```scheme
;; -== Node: Scheme; Package: SCHEME -=-

;; gen.fx -- translate list of trees to DLX ocode

;; The code generation is by recursive-decent tree rewriting. The
;; intermediate code tree is traversed depth-first by various
;; routines. When one of these routines recognizes the tree it is
;; looking at, it emits code for the tree and re-writes the tree into
;; another tree that indicates where the result was put. For example,
;; a (op->icode t1 + (reg->icode _ 1) (reg->icode _ 2)) will be recognized
;; by the binary operation routine, which will emit the code
;; "add r3, r2, r1" and will rewrite the tree into (reg->icode t1 3).

;; The top level translation procedure is gencode. Its function is to
;; translate root nodes into no-ops, and to translate intermediates
;; and leaves into either reg->icode nodes or int->icode nodes for
;; small integers. Passing up the small integers allows us to take
;; advantage of operations w/ special forms for small immediates. The
;; various routines called by gencode first translate the subtrees of
;; a node by calling gencode recursively, then translating the
;; simplified node through case analysis. Thus we use
;; recursive-decent parsing to "divide and conquer."

(define ocode-list (ref (null))) ; generated code will go here.

(define emit (lambda (op rands) (= ocode-list (cons (insn op rands) (- ocode-list)))))

(define emit-lw (lambda (dst slot block) ; reg#:int slot#:int reg#:int
  (emit 'lw (load->rands dst (otag slot) block))))

(define emit-sw (lambda (slot block src) ; reg#:int slot#:int reg#:int
  (emit 'sw (store->rands (otag slot) block src))))

(define emit-error ; for better error reporting (and locating!)
  (lambda (s)
    (begin
      (display s)
      (newline)
      (emit 'error (err->rands s)))))
``
;;; input comes from icode-list; output goes to ocode-list.

(define generate-ocode
  (lambda ()
    (begin
      (:= ocode-list (null))
      (init-reg-allocator)
      (for-each (lambda (x) (gencode x)) (~ icode-list))
      (:= ocode-list (reverse-list (~ ocode-list))))))

;;; Top-level code generators -- translate any kind of icode node.

(define gencode ;->(icode) icode)
  ;; Translate roots into noop->icode; all others into either
  ;; reg->icode or int->icode (for small ints). In the register case,
  ;; gencode will allocate a register, and gencode's caller must
  ;; deallocate it.
  (lambda (code)
    (match code
      ((znoop->icod*-) code) code)
      ((labeldef->icode- 1) (gen-labdef 1))
      ((body->icode- ty num-args body) (gen-body ty num-args body))
      ((letrec->icode- nvars vars body) (gen-letrec nvars vars body))
      ((return->icode- e) (gen-return e))
      ((string->icode- s) (gen-string s))
      ((sym->icode- s) (gen-symbol s))
      ((word->icode- w) (gen-word w))
      ((op->icode- ty o 1) (gen-op ty o 1))
      ((alloc->icode- ty i codelist) (gen-alloc ty i codelist))
      ((labelref->icode- ty 1) (gen-labref ty 1))
      ((int->icode- ty n) (gen-intref ty n))
      ((reg->icode- _ r) code)
      ((var->icode- ty back over) (gen-varref ty back over))))

(define full-gencode ; (-> (icode) icode)
  ;; Same as gencode except no int->icode are returned
  (lambda (code)
    (let ((code2 (gencode code)))
      (match code2
        ((int->icode- ty n)
          (let ((target (allocate-reg)))
            (begin
              (emit 'addi (rri->rands target ZERO (int->string (itag n))))
              (reg->icode ty target)))
            (~ code2))))))
(define target-gencode
  ;; Same as gencode except value is stored into specified register
  (lambda (code target)
    (let ((ty (match (gencode code)
                    ((reg->icode ty r)
                     (if (= r target)
                         ty
                         (begin (emit 'or (rrr->rands target r ZERO))
                                (deallocate-reg r)
                                ty))
                    ((int->icode ty i)
                     (begin
                      (emit 'addi (rri->rands target ZERO (int->string (itag i))))
                      ty))
                    (_ unit-type)))
      (reg->icode ty target))))

;; Translation for roots of intermediate code
;;
(define gen-labdef
  (lambda (1) (begin (emit 'labeldef (label->rands 1)) (noop->icode))))

(define gen-string
  (lambda (s) (begin (emit 'stringdef (string->rands s)) (noop->icode))))

(define gen-symbol
  (lambda (s) (begin (emit 'stringdef (string->rands (sym->string s)))
                  (noop->icode))))

(define gen-word
  (lambda (w) (begin (emit 'worddef (word->rands w)) (noop->icode))))

(define gen-return
  ;; Emit code to compute return result into VAL register, then emit
  ;; code to jump to caller (return address taken from activation
  ;; frame).
  (lambda (e)
    (begin
      (target-gencode e VAL)
      (emit-lw ATEMP 2 FP)
      (emit 'jr (r->rands ATEMP))
      (emit 'nop (nop->rands))
      (noop->icode))))
(define gen-body
  (lambda (ty num-args body)
    (let ((save-args (map (lambda (x) (reg->icode unknown-type x))
                           (integers-between ARG1 (+ ARG0 num-args))))
         (stack-env? (stack-allocate-environment-frame? (ictype body) body)))
      (begin
        ;; Allocate new env frame, link into static chain (available in
        ;; closure passed in ARG0), and save parameter values into it.
        (gen.alloc-block stack-env? (+ num-args 1) ENV) ; Alloc env frame
        (emit-lw ATEMP 1 ARG0) ; Lnk in2 static chain
        (emit-sw 0 ENV ATEMP) ; Link new frame into static chain
        (gen-fill-block ENV 1 save-args) ; Sv args in new env
        (gencode body)))))

(define gen-letrec
  ;; Same as lambda except we fill the new environment with evaluations
  ;; of the letrec variables instead of with argument registers.
  (lambda (num-vars vars body)
    (let ((stack-enh? (stack-allocate-environment-frame? (ictype body) body)))
      (begin
        (gen.alloc-block stack-env? (+ num-vars 1) ENV) ; Alloc env frame
        (emit-lw ATEMP 1 ARG0) ; Lnk in2 static chain
        (emit-sw 0 ENV ATEMP) ; Link new frame into static chain
        (gen-fill-block ENV 1 vars) ; Fill frame slots w/ letrec variables
        (gencode body)))))

(define stack-allocate-environment-frame?
  (lambda (ty body) #f))
(let ((result (allocate-reg))) ; Alloc a reg to put result
  (begin (emit 'or (rrr->rands result VAL ZERO))
    (reg->icode ty result))))

(define stack-allocate-activation-frame?
  (lambda (ty arglist) #f))

(define gen-jump
  ;; Generates a jump rather than a call to a closure. The first
  ;; argument in arg list is the closure to jump to.
  (lambda (ty arglist)
    (begin
      (simulate-stackframe-push)
      (gen-arguments arglist)
      (emit-lw ATEMP 0 ARGO)
      (emit 'jr (r->rands ATEMP))
      (emit 'nop (nop->rands))
      (simulate-stackframe-pop)
      (int->icode ty 0))))

(define gen-arguments
  ;; compile args into arg regs. return last-used arg reg.
  (lambda (arglist)
    (letrec ((loop (lambda
                    (l r)
                    (if (null?
                        1)
                        the-unit
                        (begin (allocate-specific-reg r)
                                (target-gencode
                                 (car
                                  1)
                                r)
                                (loop (cdr
                                       1)
                                       (+ r 1)))))))
      (loop arglist ARGO))))

(define gen-intref ; (-> (int) icode)
  ;; Small integer values are are passed up. For larger integers, a
  ;; register is allocated for the int and code is emitted to move the
  ;; int into the register.
  (lambda (ty n)
    (cond
      ; (= n 0) (reg->icode ty ZERO)) NO WRK <= way dst rgs R pckd
      (() (inc16? (itag n)) (int->icode ty n))
      (else (let ((target (allocate-rag))
                  (tn (int->string (itag u))))
               (begin (emit 'addui (rri->rands target ZERO tn))
                      (begin (emit 'lhi (ri->rands target (msw tn)))
                             (emit 'ori (rri->rands target target (lsw tn))))
                      (reg->icode ty target)))))))
(define gen-labref ; (-> (string) icode)
  ;; Allocate reg and emit code to move value of label into it.
  (lambda (ty lab)
    (let ((target (allocate-reg)))
      (begin
        (emit 'lhi (ri->rands target (msw lab)))
        (emit 'ori (rri->rands target target (lsw lab)))
        (reg->icode ty target))))

(define gen-varref
  (lambda (ty back over)
    (let ((target (allocate-reg)))
      (begin (emit-lw target over (localref back))
        (reg->icode ty target))))))

(define localref ; (-> (int) int)
  ;; Emits code to walk up static chain. Returns register pointing to
  ;; frame; local variable can be accessed by loading relative to this
  ;; pointer.
  (lambda (back)
    (letrec ((walk-up (lambda (a)
                          (if (= a 0)
                            the-unit
                            (begin
                              (emit-lw ATEMP 0 ATEMP)
                              (walk-up (- a 1))))))
      (if (= back 0)
        ENV
        (begin (emit-lw ATEMP 0 ENV)
          (walk-up (- back 1))
          ATEMP))))))

;;
;; Translation for operators
;;
(define gen-if
  (lambda (ty args)
    (let ((treg (reg-num (full-gencode (car args))))
          (con (car (cdr args)))
          (alt (car (cdr (cdr args))))
          (elselabel (new-label "ELSE"))
          (joinlabel (new-label "JOIN"))
          (begin
            (emit 'beqz (ri->rands treg elselabel))
            (emit 'nop (nop->rands))
            (deallocate-reg treg)
            (target-gencode con VAL)
            (emit 'j (i->rands joinlabel))
            (emit 'nop (nop->rands))
            (emit 'labeldef (label->rands elselabel))
            (target-gencode alt VAL)
            ;; (emit 'j (i->rands joinlabel)) ; let this case fall through...
            ;; (emit 'nop (nop->rands))
            (emit 'labeldef (label->rands joinlabel))
            (let ((result (allocate-reg))) ; Alloc a reg for result
              (begin (emit 'or (rrr->rands result VAL ZERO))
                      (reg->icode ty result)))))))

(define gen-begin
  ;; Gen code for a list of expressions, returning value of last in
  ;; list. Incoming list must have at least one element.
  (lambda (ty codelist)
    (let ((result (gencode (car codelist))))
      (if (null? (cdr codelist)) result
        (begin (match result ((reg->icode - r) (deallocate-reg r)) (_ the-unit))
               (gen-begin ty (cdr codelist)))))))

(define gen-alloc
  ;; Generate code to allocate and fill a block of memory whose size
  ;; is fixed at compile time. Used for allocating cons cells, pairs,
  ;; and closures.
  (lambda (ty size codelist)
    (let ((blkptr (allocate-reg)))
      (begin (gen-alloc-block #f size blkptr) ; Gen call to blk alloc'er
              (gen-fill-block blkptr 0 codelist) ; Fill blk w/ values
              (reg->icode ty blkptr)))))
(define gen-alloc-block
  ;; Emit code to call allocation routine, putting address of
  ;; allocated block in register whose number is passed in blkptr.
  (lambda (stack-alloc? size blkptr)
    (begin
      (emit 'ori (rri->rands ATEMP ZERO (int->string (itag size))))
      (if stack-alloc?
        (begin
          (emit 'jal (i->rands "_SALLOC"))
          (emit 'jal (i->rands "_ALLOC")))
        (emit 'nop (nop->rands))
        (emit 'or (rrr->rands blkptr ZERO ATEMP))))))

(define gen-fill-block
  ;; Emit code to fill a block with values. The values to fill the
  ;; block with are the results of evaluating codelist. The slots to
  ;; put these results start at offset(blkptr) and go up from there.
  ;; "offset" is slot number.
  (lambda (blkptr offset codelist)
    (if (null? codelist)
      the-unit
      (let ((src (reg-num (full-gencode (car codelist))))
        (begin
          (emit-sw offset blkptr src)
          (deallocate-reg src)
          (gen-fill-block blkptr (+ offset 1) (cdr codelist)))))))

(define gen-begin-activation
  ;; Emit code to allocate and fill an activation frame. Updates FP
  ;; register (and SP if stack-allocate? is true). Trashes ATEMP and
  ;; RETADR.
  (lambda (stack-allocate? Fsize return-label)
    (gen-alloc-block stack-allocate? Fsize ATEMP)
    (emit-sw 0 ATEMP FP) ; Save old frame pointer
    (emit 'or (rrr->rands FP ZERO ATEMP)) ; Load new frame into FP
    (if stack-allocate?
      (begin
        (emit 'addi (rri->rands ATEMP SP (int->string (* 4 (+ Fsize 1)))))
        (emit-sw 1 FP ATEMP)) ; Save old value of stack pointer
        (emit-sw 1 FP SP))
        (emit 'lhi (ri->rands ATEMP (msw return-label)))
        (emit 'ori (rrr->rands ATEMP ATEMP (lsw return-label)))
        (emit-sw 2 FP ATEMP) ; Save return address
        (emit-sw 3 FP ENV) ; Save environment pointer
        (emit 'jal (i->rands "_SAVE"))
        (emit 'nop (nop->rands))))))
(define gen-end-activation
  ;; Emit code to restore registers saved in activation frame.
  ;; Trashes ATEMP and RETADR.
  (lambda ()
    (emit 'jal (i->rands "_RESTORE"))
    (emit 'nop (nop->rands))
    (emit-lw ENV 3 FP)
    (emit-lw SP 1 FP)
    (emit-lw FP 0 FP)))

(define gen-not
  (lambda (ty args)
    (let ((reg (reg-num (full-gencode (car args))))
         (begin
          (emit 'xori (rrr->rands reg reg (int->string (itag 1))))
          (reg->icode ty reg)))))

(define gen-assign ; (-> ((listof icode)) icode)
  (lambda (ty args)
    (let ((sreg (reg-num (full-gencode (cadr args))))
           (dreg (reg-num (full-gencode (car args))))
           (begin
            (emit-sw 0 dreg sreg)
            (deallocate-reg dreg)
            (deallocate-reg sreg)
            (int->icode ty (itag 0)))))))
(define gen-load ; (-> ((listof icode)) icode)
  ;; The first operand is a base pointer, i.e., a tagged pointer to a
  ;; block of memory created by alloc. The second is an integer index
  ;; pointing to a slot inside that block of memory. Those slots are
  ;; numbered consecutively 0, 1, 2, ... The index must be turned
  ;; into a byte offset, which is 4x the index number, then added into
  ;; the base, either with an explicit add or with the
  ;; load-with-displacement ocode.
  (lambda (ty args)
    (let ((base (reg-num (full-gencode (car args))))
          (disp (gencode (cadr args))))
      (match disp
        ((int->icode _ i) ;
          (begin
            (if (int16? (otag i))
                (emit-lw base i base)
                (let ((dispr (reg-num (full-gencode disp))))
                  (emit 'slli (rri->rands dispr dispr "i"))
                  (emit 'add (rrr->rands base base dispr))
                  (emit-lw base 0 base)
                  (deallocate-reg dispr))
                (reg->icode ty base))))
        (let ((dispr (reg-num (full-gencode disp))))
          (emit 'slli (rri->rands dispr dispr "i"))
          (emit 'add (rrr->rands base base dispr))
          (emit-lw base 0 base)
          (deallocate-reg dispr)
          (reg->icode ty base))))))
(define gen-vec-alloc
  ;; Generate code to allocate and fill a block of memory where both the
  ;; size and the contents of the block are computed at compile time. All
  ;; slots of the block are filled with the same value. Used for allocating
  ;; vectors.
  (lambda (ty codelist)
    (let* ((blkptr (allocate-reg))
      (size (reg-num (full-gencode (car codelist))))
      (filler (reg-num (full-gencode (cadr codelist))))
      (loop (new-label "LOOP"))
      (test (new-label "TEST")))
      (begin
       (emit 'or (rrr->rands ATEMP ZERO size))
       (emit 'jal (i->rands "...LLOC"))
       (emit 'nop (nop->rands))
       (emit 'or (rrr->rands blkptr ZERO ATEMP))
       (emit 'j (i->rands test))
       (emit 'nop (nop->rands))
       (emit 'labeldef (label->rands loop))
       (emit-sw 0 ATEMP filler)
       (emit 'addi (rri->rands ATEMP ATEMP (int->string 4)))
       (emit 'labeldef (label->rands test))
       (emit 'subi (rri->rands size size (int->string 2)))
       (emit 'bnez (ri->rands size loop))
       (emit 'nop (nop->rands))
       (deallocate-reg size)
       (deallocate-reg filler)
       (reg->icode ty blkptr)))
    ))

(define gen-vec-length
  (lambda (ty codelist)
    (let ((vec (reg-num (full-gencode (car codelist))))
      (begin
       (emit-lw vec -1 vec)
       (reg->icode ty vec))))))
(define gen-voc-set!
  (lambda (ty codelist)
    (let* ((vec (rag-gui
                  (full-gencode (car codelist))))
            (idxz (gencode (ccadr codelist)))
            (val (reg-num (full-gencode (ccadr (ccdr codelist))))))
      (begin
        (match idx
          ((int->icode' _ i)
           (if (int16? (otag i))
               (emit-sw i vec val)
               (let ((dispr (reg-num (full-gencode idx))))
                 (begin
                   (emit 'slli (rrr->rands dispr dispr "1")
                   (emit 'add (rrr->rands vec vec dispr))
                   (emit-sw 0 vec val)
                   (deallocate-reg dispr))))
          ((reg->icode' _ dispr)
           (begin
             (emit 'slli (rrr->rands dispr dispr "1")
             (emit 'add (rrr->rands vec vec dispr))
             (emit-sw 0 vec val)
             (deallocate-reg dispr)))
          (deallocate-reg vp_)
          (deallocate-reg vac)
          (int->icode ty 0)))))

(define gen-sym2string
  (lambda (ty codelist)
    (let ((sym (reg-num (iuli-gencode (car codelist))))
          (begin
            (emit 'or (rrr->rands ATEMP sym ZERO))
            (emit 'jal (i->rands "._SYM2STRING")
            (emit 'nop (nop->rands))
            (emit 'or (rrr->rands sym ATEMP ZERO))
            (reg->icode ty sym))))))

(define gen-put-char
  (lambda (ty codelist)
    (begin
      (target-gencode (ccar codelist) ATEMP)
      (emit 'jal (i->rands "._TPUTCCHAR") ; arg is tagged, so use tagged vers.
      (emit 'nop (nop->rands))
      (int->icode ty 0)))))
(define gen-op ; (-> (op (listof icode)) icode)
  ;; The code generators for each type of operator are kept in a table
  ;; indexed by the operation symbol. As mentioned before, code
  ;; generation is by recursive decent: we translate sub-trees first by
  ;; calling gencode, then translate the operator.
  (lambda (ty op args) ((op-table op) ty args)))

(define make-binop-gen
  ;; Code generators for alu binary op's all look the same, so abstract.
  (lambda (op opi)
    (lambda (ty args)
      (let ((1 (reg-num (full-gencode (car args))))
            (r (gencode (cadr args))))
        (begin
          (match r
            ((int->icode" n) ;
              (emit opi (rri->rands 1 1 (int->string (itag n)))))
            ((reg->icode" rreg) ;; Do the general thing
              (begin (emit op (rrr->ranids 1 1 rreg))
                (deallocate-reg rreg)))
            (reg->icode ty 1))))))

(define make-binop-gen2
  ;; Binary operators w/ no built-in ALU operation
  (lambda (emitter)
    (lambda (ty args)
      (let ((M (reg-num (full-gencode (car args))))
            (r (reg-num (full-gencode (cadr args))))
            (emitter 1 r)
            (deallocate-reg r)
            (reg->icode ty 1))))))

(define emit-mul
  (lambda (l r)
    (begin (emit 'movi2fp (rr->rands FP0 1))
            (emit 'srai (rr->rands ATTEMP r "1")
            (emit 'movi2fp (rr->rands FP1 ATTEMP))
            (emit 'mult (rrr->rands FP2 FP0 FP1))
            (emit 'movfp2i (rr->rands 1 FP2)))))

(define emit-div
  (lambda (l r)
    (begin (emit 'movi2fp (rr->rands FP0 1))
            (emit 'movi2fp (rr->rands FP1 r))
            (emit 'div (rrr->rands FP2 FP0 FP1))
            (emit 'movfp2i (rr->rands 1 FP2))
            (emit 'slli (rr->rands 1 1 "1"))))
(define emit-remainder
  (lambda (l r)
    (begin (emit 'movi2fp (rr->rands FP0 1))
      (emit 'movi2fp (rr->rands FP1 r))
      (emit 'div (rrr->rands FP2 FP0 FP1))
      (emit 'mult (rrr->rands FP3 FP2 FP1))
      (emit 'movfp2i (rr->rands ATEMP FP3))
      (emit 'sub (rrr->rands 1 1 ATEMP))))

(define op-table
  (let ((empty (lambda (bad-op)
                  (emit-error (string-append "bad op:
                                 " (sym->string key)))))
         (op-list
          '((÷ ,(make-binop-gen 'add 'addi))
            (- ,(make-binop-gen 'sub 'subi))
            (seq ,(make-binop-gen 'seq 'seqi))
            (slt ,(make-binop-gen 'slt 'slti))
            (sle ,(make-binop-gen 'sle 'slei))
            (sgt ,(make-binop-gen 'sgt 'sgti))
            (sge ,(make-binop-gen 'sge 'sgei))
            (* ,(*op-binop-gen 'mul 'muli))
            (/ ,(make-binop-gen2 emit-div))
            (remainder ,(*op-binop-gen2 emit-remainder))
            (:= ,gen-assign)
            (+ ,gen-load)
            (vec-alloc ,gen-vec-alloc)
            (vec-set ,gen-vec-set!)
            (vec-length ,gen-vec-length)
            (sym2string ,gen-sym2string)
            (put-char ,gen-put-char)
            (not ,gen-not)
            (call ,gen-call)
            (jump ,gen-jump)
            (if ,gen-if)
            (begin ,gen-begin))))
    (list2env empty sym=? op-list)))

;; Tags:
;; The least significant bit of each machine word is used as a tag so that
;; the garbage collector can distinguish pointers from integers. All
;; integer values are shifted up one bit position (i.e., multiplied by
;; two) to have a zero tag. Pointers (which all point to things on word
;; boundaries, and so are all multiples of four) are given a tag of 1. To
;; dereference a pointer, one must subtract one from it before indirecting
;; through it.
;;
;; (define itag (lambda (i) (* 2 i)))           ; tag an integer
;;
;; (define de-itag (lambda (i) (quotient i 2)))

73
(define otag
    ;; For referencing into object slots. Takes a slot number and
    ;; returns an offset which, when added to a pointer, addresses the
    ;; desired data value. (Remember slot -1, the size of the object,
    ;; is stored at address "tagged_pointer - 1," slot 0 at
    ;; "tagged_pointer + 3," etc.)
    (lambda (slot) (+ (* slot 4) 3)))

(define de-otag (lambda (i) (/ (- i 3) 4)))

(define int16? : (-> (int) bool)
    (lambda (n) (and (<= n 32767) (>= n -32768))))

(define msw
    (lambda (l)
        (string-append "(" (string-append l ")")))

(define lsw
    (lambda (l) (string-append "(" (string-append l ")"))

(define FrameSize 28)

;; Symbolic names for some registers (local to this file with one
;; exception noted below).

(define ZERO 0)
(define VAL 1)
(define ENV 2)      ;; this def is used in exp2ic.fx
(define FP 3)
(define SP 4)
(define HP 5)
(define ARG0 6)
(define ARG1 7)
(define ARG2 8)
(define ARG3 9)
(define ARG4 10)
(define ARG5 11)
(define ARG6 12)
(define ARG7 13)
(define ARG8 14)
(define ARG9 15)
(define TOPARG 29)
(define ATEMP 30)
(define RETADR 31)

(define FP0 0)
(define FP1 1)
(define FP2 2)
(define FP3 3)
(define FP4 4)
(define FP5 5)
;; Register allocation
;;

;; The following def's are local to the allocator

(define *free-reg-se* (ref (generate-vector 0 (lambda (i) #t)))))
(define *free-reg-stack* (ref '()))
(define *num-allocatable* (+ 1 (- TOPARG ARGO))

(define allocatable-reg? (lambda (r) (and (<= ARG0 r) (<= r TOPARG))))
(define reg-free? (lambda (r) (and (allocatable-reg? r)
                                (vector-ref (- *free-reg-se* r) ARGO))))

(define take-reg! (lambda (r) (vector-set! *free-reg-se* (- r ARG0) #f)))
(define put-reg! (lambda (r) (vector-set! *free-reg-se* (- r ARG0) #t)))

;; The following def's are exported by the allocator, but are local to this file.

(define init-reg-allocator (lambda ()
  (begin (: *free-reg-stack* '())
          (: *free-reg-se* (generate-vector *num-allocatable* (lambda (i) #t))))))

(define allocate-reg (lambda ()
  (letrec ((loop (lambda (r) (cond ((> r TOPARG) (error "Out of registers.")
                                 ((reg-free? r) (begin (take-reg! r) r))
                                 (else (loop (+ r 1)))))))
    (loop ARGO))))

(define allocate-specific-reg (lambda (r)
  (if (reg-free? r)
    (begin (take-reg! r) r)
    (error "Register needed twice.")))))

(define deallocate-reg (lambda (reg)
  (if (allocatable-reg? reg)
    (put-reg! reg)
    the-unit)))

(define highest-used-reg (lambda ()
  (letrec ((loop (lambda (r) (cond ((< r ARGO) ARGO)
                                 ((not (reg-free? r)) r)
                                 (else (loop (- r 1)))))))
    (loop TOPARG))))

75
C.7 compiler/icode.fx

The contents of the file compiler/icode.fx:

;;; -*- Mode: Scheme; Package: SCHEME -*- 

(define datatype icode
  ;; tree roots:
  ;; ;
  (labeldef->icode string) ; label definitions
  (body->icode type int icode) ; stack-act stack-env
  (letrec->icode int (listoe icode) icode) ; num-vars vars body
  (noop->icode) ; no-op
  (string->icode string) ; immediate strings
  (sym->icode sym) ; immediate symbols

;; intermediate nodes
;; ;
  (return->icode icode) ; return an e'
  (op->icode type sym (listoe icode)) ; primitive operations
  (alloc->icode type int (listoe icode)) ; mem allocation

;; leaf nodes
;; ;
  (labelref->icode type string) ; refs to labels
  (int->icode type int) ; immediate integers
  (reg->icode type int) ; register ref (for codegen...)
  (var->icode type int int) ; var refs (back, over)
  (word->icode string) ; For labels used as ints. This
                        ; node can also be a int'

(define noop (noop->icode)) ; only need one of these...

(define reg-num ; quick access to register numbers
  (lambda (icode) (match icode ((reg->icode _ n) n))))

(define ictype

;; Find the high-level type associated with a piece of icode. Note that
;; this operation is not appropriate for all icodes. For example, there
;; is no type associated with labeldef->icode. The only root with a
;; type slot is the body slot: this type does not give the type of the
;; expression, but rather is just a way to communicate down the types
;; of the arguments. return->icode does not have a type slot because
;; the type of that expression is just the type of the value being returned.
;; word->icode does not have a type slot because it is used for internal
;; compiler stuff.
(lambda (ic)
  (match ic
    ((body->icode ty _) ty)
    ((letrec->icode _ ic) (ictype ic))
    ((return->icode ic) (ictype ic))
    ((op->icode ty _) ty)
    ((alloc->icode ty _) ty)
    ((labelref->icode ty _) ty)
    ((int->icode ty _) ty)
    ((reg->icode ty _) ty)
    ((var->icode ty _) ty))))

C.8 compiler/lib.fx

The contents of the file compiler/lib.fx:

;;; Mode: Scheme; Package: SCHEME

;; lib.fx -- compile-time environments and built-ins

;; This file contains the compile-time environments and built-in
;; primitives for mico-FX built-ins. There are two compile time
;; environments, one for the type reconstructor (standard-type-env)
;; and another for the exp2ic translator (standard-c-t-env). This
;; file exports the following variables:

;;; lib-init:(-> () unit)
;;; Initializes stuff internal to library; needs to be called once
;;; per compilation.

;;; c-t-lookup:(-> (c-t-env sym) binding)
;;; Returns the binding for sym; aborts with error if not found.

;;; c-t-bind:(-> ((listof sym) c-t-env) c-t-env)
;;; Returns new environment with a new lexical level containing
;;; the list of symbols pushed onto the old environment.

;;; standard-c-t-env:c-t-env
;;; Standard environment defined by microFX (containing bindings
;;; for +, -, etc.).

;;; The c-t-env abstract type is defined by operations in this file. The
;;; environment itself is an abstract type defined by two operations:
The "binding" data type is returned by c-t-lookup. back+over is for variables defined by the code being compiled: back indicates how far down the static chain the variable resides (0 => variable is in current environment), over indicates how far over in the environment frame the variable resides.

"prim" is an abstract data type used for variables defined by microFX. It has the operations:

- prim-constant?: (-> (prim) bool) Returns true iff primitive is a constant (eg, the-unit) rather than a function (eg, +).
- prim-inline-form: (-> (prim) ((listof icode)) icode)) For functional primitives, returns a function that takes a list of icode which are the arguments to the function and returns icodes that computes the function on those arguments.
- prim-closure-form: (-> (prim) icode) For functional primitives, returns icode which is a closure for the indicated primitive.

Note that prim-inline-form and prim-closure-form can modify the library-icode list, a list of auxiliary icodes needed by library functions.

(define lib-init (lambda () (:= prim-closure-env (mk-empty-env add-prim-closure))))

(define-datatype prim

; The following data structure is used inside this module to hold important data about the built-ins in the library. For built-ins coded in assembly language, the structure holds the label of the routine. For library routines coded in micro-FX, the structure contains the unparsed micro-FX expression for the routine. For library routines that can be inlined (e.g., +, - *), the structure holds both an unparsed expression for the routine plus a procedure that takes a list of arguments (in icode form) and returns icode for the inlined procedure. The first is used in situations like '+, while the second is used in situations like (+ x y).

(asm->prim ty label) ; prim coded in assy lang (in "microFX/runtime")
(cnst->prim icode) ; prim is a constant
(lib->prim sexp) ; prim coded in microFX
(inline->prim sexp (-> ((listof icode)) icode))) ; inline-able prim
(define-datatype binding
    ;; Bindings held in the compile-time environment. Primitives are
    ;; wrapped up as uniqueof's so we can have a table of them (using eq?).
    ;; The type (uniqueof t) denotes the set of values for which each
    ;; element of type is distinguishable. In other words, it's a promise
    ;; to the compiler that there's no sharing of elements, so eq? will
    ;; works. The current minifx interpreter doesn't actually do anything
    ;; with this information, but there's always tomorrow...)
    ;;
    ;; (primitive->binding (uniqueof prim))
    ;; (back+over->binding int int))

    ;; Code implementing c-t-env used by exp2ic. We don't use our
    ;; usual environments here so we can encode back+over in the
    ;; lambda's that make up the environment.
    (define empty-c-t-env (lambda (var) (error "unbound variable" var)))

    (define c-t-lookup (lambda (c-t-env var) (c-t-env var)))

    (define c-t-add-prims
        (lambda (ops prima env)
            (letrec '(loop (lambda (var ops prima)
                (if (null? ops)
                    (env var)
                    (if (same-variable? (car ops) var)
                        (primitive->binding (car prims))
                        (loop var (cdr ops) (cdr prims)))))
                (lambda (var) (loop var ops prima)))))

    (define c-t-bind
        (lambda (formals c-t-env)
            (lambda (var)
                (letrec ((loop (lambda (i formals)
                    (if (null? formals)
                        (match (c-t-lookup c-t-env var)
                            (((back+over->binding back over) 
                                (back+over->binding (+ back 1) over))
                                (binding binding))
                        (if (same-variable? var (car formals))
                            (back+over->binding 0 i)
                            (loop (+ i 1) (cdr formals)))
                        (loop i formals))))))
                (loop 1 formals))))

    (define prim-constant?
        (lambda (prim)
            (match (value prim)
                (((cnst->prim ic) #t)
                    (_ #f)))))

    (define prim-constant-form ;; (-> (prim) icode)
        (lambda (prim)
            (match (value prim)
                (((cnst->prim ic) ic)))))
(define prim-inline-form ;; (-) (prim) (-) (ty (listof iode)) iode))
(lambda (prim)
  (match (value prim)
    ((inline->prim "emitter) emitter)
     (_ (lambda (ty args)
         (op->icode ty 'call (cons (prim-closure-form prim) args)))))))))

(define prim-closure-form ;; (-) (prim) iode)
(lambda (prim) (("prim-closure-tv prim)))

;; Closures for library routines are created only once and are
;; memoized in prim-closure-env, defined below.

(define add-prim-closure
  ;; If primitive not already in the environment, then make a closure
  ;; iode for it and insert that iode into the table
  (lambda (prim)
    (begin (enter-library) ;; Trans lib fn's w/ separate delay list
      (let ((closure (make-prim-closure prim)))
        (begin
          (:=
            prim-closure-env
            ((mk-binder eq?) prim closure
             (- prim-closure-ynv)))
          (leave-library) ;; Back to normal
            closure)))))))

(define prim-closure-env (ref (mk-empty-env add-prim-closure)))

(define make-prim-closure
  ;; There's no closure for this prim yet (first-time reference). Make one.
  (lambda (prim)
    (match (value prim)
      ((asm->prim "ty label) ; asm prim: grab closure.
        (trans-closure ty label))
       ((inline->prim sexp ...) ; inline prim: create and compile one.
         (let* ((exp (parse sexp))
          (type (reconstruct-top exp)))
            (translate exp standard-c-t-env))
       ((lib->prim sexp) ; same for lib prims.
         (let* ((exp (parse sexp))
           (type (reconstruct-top exp)))
             (translate exp standard-c-t-env)))))

;; Functions to create iode for inlined primitive operations

(define mk-op-prim ;; + * quotient ...
  (lambda (op) (lambda (ty arglist) (op->icode ty op arglist))))

(define mk-cons-prim ;; Data constructor: cons, pi, ref
  (lambda (size)
    (lambda (ty arglist) (alloc->icode ty size arglist))))
(define mk-sel-prim
  ;; Projection functions: car, cdr, left, right, etc.
  ;; Tagging implements a check for (car (null)) or (cdr (null))!
  (lambda (offset)
    (lambda (ty arglist)
      (op->icode ty '" (list (car arglist)
                          (int->icode integer-type offset)))))))

(define assign-prim ; ; For :=
      (lambda (ty arglist) (op->icode ty ':= arglist))))

(define neg-prim
  (lambda (ty arglist) (op->icode ty 'sub (cons (int->icode ty 0) arglist))))

(define null-prim
  ;; The empty list is represented as immediate 0. It's untagged so
  ;; the garbage collector doesn't try to follow it.
  (lambda (ty arglist) (int->icode ty 0)))

(define null?-prim
  (lambda (ty arglist)
    (op->icode ty 'seq (list (car arglist) (int->icode integer-type 0)))))

(define sym->string-prim
  (lambda (ty arglist) (op->icode ty 'sym2string arglist)))

(define cvt-prim ;; Pass argument straight thru -- for type coercion
      (lambda (ty arglist) (car arglist)))

(define put-char-prim
  (lambda (ty arglist) (op->icode ty 'put-char arglist)))

(define void-prim
  ;; Cons up a fake argument. In real life, this routine should generate
  ;; code to crash the system.
  (lambda (ty arglist) (int->icode ty 0)))

(define void-name
); ; Make it hard for the user to get at the void function
     (string->symbol "the void function")

(define no-prim
  (lambda (ty arglist) (error "Unimplemented primitive used."))))

;; Most built in functions are written in micro-FX code. This code is
;; listed below. These expressions are evaluated in the standard
;; compile time environment, built-ins can call one another.

;; Char prims
(define mk-char-ci-pred
  (lambda (op) '(lambda (x y) (,op (char-downcase x) (char-downcase y))))))

(define 1-char-alphabetic?
  '(lambda (x) (or (char-lower-case? x) (char-upper-case? x))))
(define 1-char-numeric?
  '(lambda (x)
    (and (> (char->int x) (char->int #\0))
         (<= (char->int x) (char->int #\0)))))

(define 1-char-whitespace?
  '(lambda (x)
    (or (= (char->int x) (char->int space))
        (= (char->int x) (char->int tab))
        (= (char->int x) (char->int page))
        (= (char->int x) (char->int newline)))))

(define 1-char-lower-case?
  '(lambda (x)
    (and (>= (char->int x) (char->int #\a))
         (<= (char->int x) (char->int #\z)))))

(define 1-char-upper-case
  '(lambda (x)
    (and (>= (char->int x) (char->int #\A))
         (<= (char->int x) (char->int #\Z)))))

(define 1-char-upcase
  '(lambda (x)
    (if (char-lower-case? x)
        (int->char (+ (char->int #\A) (- (char->int x) (char->int #\a))))
        x)))

(define 1-char-downcase
  '(lambda (x)
    (if (char-upper-case? x)
        (int->char (+ (char->int #\a) (- (char->int x) (char->int #\A))))
        x)))

;; List prims

;; (define 1-set-car!
;; (define 1-set-cdr!

(define 1-length
  '(lambda (1)
    (letrec ((loop (lambda (1st len)
                      (if (null? 1st) len (loop (cdr 1st) (+ len 1))))))
      (loop 1 0))))

;; (define 1-append

(define 1-reverse ; minifx doesn't have this built-in.
  '(lambda (1)
    (letrec ((loop (lambda (1st rev)
                      (if (null? 1st)
                          rev
                          (loop (cdr 1st) (cons (car 1st) rev))))))
      (loop 1 (null)))))

82
;;; (define 1-list-tail
;;; (define 1-list-ref
;;; (define 1-map
;;; (define 1-for-each
;;; (define 1-reduce
(define 1-list->string '(lambda (l) (vector->string (list->vector l))))
(define 1-string->list '(lambda (s) (vector->list (string->vector s))))

;;; Strings
;;; (define 1-string-fill!
;;; (define 1-string=?
;;; (define 1-string<?
;;; (define 1-string>?
;;; (define 1-string<=?
;;; (define 1-string>=?
;;; (define 1-string-ci=?
;;; (define 1-string-ci<?
;;; (define 1-string-ci>?
;;; (define 1-string-ci<=?
;;; (define 1-string-ci>=?
(define 1-string-append
  '(lambda (sl s2)
    (letrec ((r (make-string (+ (string-length sl) (string-length s2)) #\I))
      (loop (lambda (s i j)
        (if (>= i (string-length s))
          the-unit
          (begin
            (string-set! r j (string-ref s i))
            (loop s (+ i 1) (+ j 1))))))
    (begin (loop sl 0 0) (loop s2 0 (string-length s2)) r))))

;;; (define 1-string-copy

;;; Syms
;;; (define 1-sym->string
;;; (define 1-string->sym
;;; (define 1-sym=?
;;; (define 1-hash

;;; Vectors
(define 1-make-vector (mk-op-prim 'vec-alloc))
(define 1-vector-length (mk-op-prim 'vec-length))
(define 1-vector-ref (mk-op-prim '-))
(define 1-vector-set! (mk-op-prim 'vec-set))

;;; (define 1-vector-fill!

(define 1-vector->list
  '(letrec ((v21 (lambda (v i l)
        (if (= i -1)
          l
          (v21 v (- i 1) (cons (vector-ref v i) l))))))
    (lambda (v) (v21 v (- (vector-length v) 1) (null))))))

83
C.8 compiler/lib.fx

C \(\mu\)FX/DLX COMPILER IMPLEMENTATION

```
(define l-list->vector
  '(letrec ((12v (lambda (l i v))
    (if (= i (vector-length v))
      v
      (begin (vector-set! v i (car l))
        (12v (cdr l) (+ i 1) v))))))

(lambda (l)
  (if (= (length l) 0)
    (make-vector (length l) ,void-name))
    (12v l 0 (make-vector (length l) (car l))))))

;;; (define l-vector-map
;;; (define l-vector-map2
;;; (define l-vector-reduce
;;; (define l-scan
;;; (define l-segmented-scan
;;; (define l-compress
;;; (define l-expand
;;; (define l-eoshift

;;; Unparsers
(define l-unparse-bool '(lambda (x) (if x "#t" "#f")))
(define l-unparse-char '(lambda (x) (string-append "#\" (make-string 1 x))))
(define l-unparse-unit '(lambda (x) "#u"))

(define l-unparse-int
  '(lambda (x)
    (letrec ((loop (lambda (l i))
      (if (= i 0)
        (vector->string (list->vector l))
        (loop (cons (int->char (+ (remainder i 10) (char->int #\0)))
          l))
        (/ i 10))))))

       (if (= x 0)
         "0"
       (if (< x 0)
         (string-append "-" (loop (null) x))
         (loop (null) x))))))

(define l-unparse-string
  '(lambda (x) (string-append "\" (string-append x "\\"))))

(define l-unparse-symbol
  '(lambda (x) (string-append "(symbol " (string-append (sym->string x) ")"))))
```
(define 1-unparse-list
  '(lambda (pr x)
     (letrec ((loop
                  (lambda (s x)
                    (if (null? x)
                        (string-append s "")
                        (loop (string-append s (string-append " " (pr (car x)))
                              (cdr x))))
                    (loop "(list" x))))))

(define 1-unparse-vector
  '(lambda (pr x)
     (string-append "(list->vector "
                    (string-append (unparse-list pr (vector->list x)) "")"))))

(define 1-unparse-pair
  '(lambda (prl prr x)
     (string-append "(pair "
                    (string-append (prl (left x)) (string-append (prr (right x)) "")"))))

(define 1-put-string
  '(lambda (s)
     (letrec ((loop
                    (lambda (i)
                      (if (> i (string-length s))
                          the-unit
                          (begin (put-char (string-ref s i)) (loop (+ i 1)))))))
      (loop 0))))

;; The list "standard-prim-bindings" contains typing and other
;; information about micro-fx built-ins. This list is used to
;; initialize two other lists, the standard type environment used by
;; the type checker and the standard compile-time environment used by
;; the expression to icode translator.

;; Each entry of the list has the form "(op type prim)" where op is
;; the name of the built-in (a symbol), type is the type of the
;; primitive (an sexp), and prim is the translation information for
;; the built-in (a "prim" sum-of-products).

;; The following routines make it easy to define entries in
;; standard-prim-bindings:

(define in0
  (lambda (op type icode-emitter)
    (list op type
       (unique (inline->prim ' (lambda () (.op)) icode-emitter))))))

(define in1
  (lambda (op type icode-emitter)
    (list op type
       (unique (inline->prim ' (lambda (x) (.op x)) icode-emitter))))))

85
(define in2
  (lambda (op type icode-emitter)
    (list op type
          (unique (inline->prim '(lambda (x y) ,(op x y)) icode-emitter))))))

(define in3
  (lambda (op type icode-emitter)
    (list op type
          (unique (inline->prim '(lambda (x y z) ,(op x y z)) icode-emitter))))))

(define cnst
  (lambda (op type icode)
    (list op type (unique (cnst->prim icode))))))

(define lib
  (lambda (op type sexp)
    (list op type (unique (lib->prim sexp))))))

(define asm
  (lambda (op type label)
    (list op type
          (unique (asm->prim (instantiate-schema (parse-schema type)) label))))))

(define standard-bindings
  '(
    ,(cnst 'the-unit 'unit (int->icode unit-type 0))
    ,(asm 'printf '(generic (t) (-> (sym t) unit)) "PRINTF")
    ,(inl 'put-char '-> (char) unit) put-char-prim
    ,(lib 'put-string '-> (string) unit) 1-put-string
    ,(in0 void-name '-> () t) void-prim)

;; Booleans
    ,(in2 'equiv? '-> (bool bool) bool) (mk-op-prim 'seq))
    ,(in2 'and? '-> (bool bool) bool) (mk-op-prim 'and))
    ,(in2 'or? '-> (bool bool) bool) (mk-op-prim 'or))
    ,(in1 'not? '-> (bool) bool) (mk-op-prim 'not))
    ,(in1 'not? '-> (bool) bool) (mk-op-prim 'not))

86
;;; Characters
(cnst 'backspace 'char (int->icode character-type 8))
(cnst 'newline 'char (int->icode character-type 10))
(cnst 'page 'char (int->icode character-type 12))
(cnst 'space 'char (int->icode character-type 32))
(cnst 'tab 'char (int->icode character-type 9))
(in2 'char=? '(-> (char char) bool) (mk-op-prim 'seq))
(in2 'char<? '(-> (char char) bool) (mk-op-prim 'slt))
(in2 'char>? '(-> (char char) bool) (mk-op-prim 'sgt))
(in2 'char<=? '(-> (char char) bool) (mk-op-prim 'sle))
(in2 'char=>? '(-> (char char) bool) (mk-op-prim 'sge))
(lib 'char-ci=? '(-> (char char) bool) (mk-char-ci-pred char=?))
(lib 'char-ci?> '(-> (char char) bool) (mk-char-ci-pred char>)
(lib 'char-ci<? '(-> (char char) bool) (mk-char-ci-pred char<))
(lib 'char-ci==? '(-> (char char) bool) (mk-char-ci-pred char==))
(lib 'char-alphabetic? '(-> (char) booi) l-char-alphabetic?)
(lib 'char-numeric? '(-> (char) booi) l-char-numeric?)
(lib 'char-whitespace? '(-> (char) booi) l-char-whitespace?)
(lib 'char-lower-case? '(-> (char) booi) l-char-lower-case?)
(lib 'char-upper-case? '(-> (char) booi) l-char-upper-case?)
(lib 'char-upcase '(-> (char) char) l-char-upcase)
(lib 'char-downcase '(-> (char) char) l-char-downcase)
(in2 'char->int '(-> (char) int) cvt-prim)
(in2 'int->char '(-> (int) char) cvt-prim)

;;; Integers
(in2 '=' '(-> (int int) bool) (mk-op-prim 'eq))
(in2 '< '(-> (int int) bool) (mk-op-prim 'slt))
(in2 '> '(-> (int int) bool) (mk-op-prim 'sgt))
(in2 '<= '(-> (int int) bool) (mk-op-prim 'sle))
(in2 '>= '(-> (int int) bool) (mk-op-prim 'sgge))
(in2 '+' '(-> (int int) int) (mk-op-prim '+))
(in2 '-' '(-> (int int) int) (mk-op-prim '-'))
(in2 '*' '(-> (int int) int) (mk-op-prim '*'))
(in2 '/' '(-> (int int) int) (mk-op-prim '/'))
(in2 'quotient '(-> (int int) int) (mk-op-prim '/')) ; helps in testing...
(in2 'neg '(-> (int int) int) cvt-prim)
(in2 'remainder '(-> (int int) int) (mk-op-prim 'remainder))
(in2 'modulo '(-> (int int) int) (mk-op-prim 'modulo))
(lib 'abs '(-> (int int) 'lambda (x) (if (< x 0) (- 0 x) x)))
;;; Lists
,(inl 'null? '(generic t) (-> ((listof t)) bool)) null?-prim
,(in0 'null '((generic t) (-> () (listof t))) null-prim)
,(in2 'cons '((generic t) (-> (t (listof t)) (listof t))) (mk-cons-prim 2))
,(in1 'car '((generic t) (-> ((listof t)) t)) (mk-sel-prim 0))
,(in1 'cdr '((generic t) (-> ((listof t)) (listof t))) (mk-sel-prim 1))
,(in2 'set-car '((generic t) (-> ((listof t) t) unit)) no-prim)
,(in2 'set-cdr '((generic t) (-> ((listof t) t) unit)) no-prim)
,(lib 'length '((generic t) (-> ((listof t) int)) int)) l-length
,(in2 'append '((generic t) (-> ((listof t) (listof t)) (listof t))) no-prim)
,(in1 'reverse '((generic t) (-> ((listof t)) (listof t))) no-prim)
,(in2 'list-tail '((generic t) (-> ((listof t) int) (listof t))) no-prim)
,(in2 'list-ref '((generic t) (-> ((listof t) int) t)) no-prim)
,(in3 'map
 '((generic t1 t2) (-> ((t1 t2) (listof t1)) (listof t2)))
 no-prim)
,(in3 'for-each
 '((generic t1 t2) (-> ((t1 t2) (listof t1)) unit))
 no-prim)
,(in3 'reduce
 '((generic t1 t2) (-> ((t1 t2) (listof t1) t2) t2))
 no-prim)
,(lib 'list->string '((listof char) string) l-list->string)
,(lib 'string->list '((string) (listof char)) l-string->list)

;;; Ordered pairs
,(in2 'pair '((generic t1 t2) (-> (t1 t2) (pairof t1 t2))) (mk-cons-prim 2))
,(in1 'left '((generic t1 t2) (-> ((pairof t1 t2)) t1)) (mk-sel-prim 0))
,(in1 'right '((generic t1 t2) (-> ((pairof t1 t2)) t2)) (mk-sel-prim 1))

;;;Refs
,(in1 'ref '((generic t) (-> (t) (refof t))) (mk-cons-prim 2))
,(in1 '=' '((generic t) (-> ((refof t)) t)) (mk-sel-prim 0))
,(in2 ':= '((generic t) (-> ((refof t) t) unit)) assign-prim)
;; Strings
,(in2 'make-string '(-> (int char) string) l-make-vector)
,(in2 'string-length '(-> (string) int) l-vector-length)
,(in2 'string-ref '(-> (string int) char) l-vector-ref)
,(in2 'string-set! '(-> (string int char) unit) l-vector-set!)
,(in2 'string-fill! '(-> (string char) unit) no-prim)
,(in2 'string=? '(-> (string string) bool) no-prim)
,(in2 'string<? '(-> (string string) bool) no-prim)
,(in2 'string>=? '(-> (string string) bool) no-prim)
,(in2 'string-ci=? '(-> (string string) bool) no-prim)
,(in2 'string-ci<? '(-> (string string) bool) no-prim)
,(in2 'string-ci>=? '(-> (string string) bool) no-prim)
,(in2 'string-ci=>? '(-> (string string) bool) no-prim)
,(in3 'substring '(-> (string int int) string) no-prim)
,(lib 'string-append '(-> (string string) string) l-string-append)
,(in1 'string-copy '(-> (string) string) no-prim)
,(in1 'string->vector '(-> (string) (vectorof char)) cvt-prim)
,(in1 'vector->string '(-> ((vectorof char)) string) cvt-prim)

;; Syms
,(in1 'sym->string '(-> (sym) string) sym->string-prim)
,(in1 'string->sym '(-> (string) sym) no-prim)
,(in2 'sym=? '(-> (sym sym) bool) (mk-op-prim 'seq))
,(in1 'hash '(-> (sym int) no-prim)
;; Vectors
,(in2 'make-vector '(generic (t) -> (int t) (vectorof t))) 1-make-vector
,(in1 'vector-length
   '(generic (t) -> ((vectorof t)) int))
   1-vector-length)
,(in2 'vector-ref
   '(generic (t) -> ((vectorof t) int) t))
   1-vector-ref)
,(in3 'vector-set!
   '(generic (t) -> ((vectorof t) int t) unit))
   1-vector-set!)
,(in2 'vector-fill! 'generic (t) (-) ((vectorof t) t) unit)) no-prim)
,(lib 'vector->list
   '(generic (t) -> ((vectorof t)) (listof t)))
   1-vector->list)
,(lib 'list->vector
   '(generic (t) -> ((listof t)) (vectorof t)))
   1-list->vector)
,(in2 'vector-map
   '(generic (t1 t2) -> ((-> (t1) t2) (vectorof t1)) (vectorof t2)))
   no-prim)
,(in3 'vector-map2
   '(generic (t1 t2 t3)
       -> ((-> (t1) t2) (vectorof t1) (vectorof t2) (vectorof t3)))
       no-prim)
,(in3 'vector-reduce
   '(generic (t1 t2) -> ((-> (t1) t2) (vectorof t1) t2) t2))
   no-prim)
,(in2 'scan
   '(generic (t) -> ((-> t t) (vectorof t)) (vectorof t)))
   no-prim)
,(in3 'segmented-scan
   'generic (t)
   -> ((-> t t) (vectorof t) (vectorof t) (vectorof t)))
   no-prim)
,(in2 'compress
   'generic (t) ((vectorof bool) (vectorof t)) (vectorof t)))
   no-prim)
,(in3 'expand
   'generic (t)
   ((vectorof bool) (vectorof t) (vectorof t)) (vectorof t)))
   no-prim)
,(in3 'eoshift
   'generic (t) ((int (vectorof t) (vectorof t))) (vectorof t)))
   no-prim)
;; Unparsers
, (lib 'unparse-bool '(-> (bool) string) 1-unparse-bool)
, (lib 'unparse-char '(-> (char) string) 1-unparse-char)
, (lib 'unparse-int '(-> (int) string) 1-unparse-int)
, (lib 'unparse-string '(-> (string) string) 1-unparse-string)
, (lib 'unparse-symbol '(-> (sym) string) 1-unparse-symbol)
, (lib 'unparse-unit '(-> (unit) string) 1-unparse-unit)
, (lib 'unparse-list
   '((generic (t) (-> ((-> (t) string) (listof t))) string))
   1-unparse-list)
, (lib 'unparse-vector
   '((generic (t) (-> ((-> (t) string) (vectorof t))) string))
   1-unparse-vector)
, (lib 'unparse-pair
   '((generic (r l)
     (-> ((-> (l) string) (-> (r) string) (pairof r l))) string))
   1-unparse-pair)
)

;; Define standard-type-environment and standard-c-t-env.

(define sb-name (lambda (binding) (car binding)))
(define sb-type (lambda (binding) (parse-schema (cadr binding))))
(define sb-prim (lambda (binding) (cadr (cdr binding))))

(define standard-type-environment
  (extend-by-schemas empty-type-environment
    (map sb-name standard-bindings)
    (map sb-type standard-bindings)))

(define standard-c-t-env
  (c-t-add-prims (map sb-name standard-bindings)
    (map sb-prim standard-bindings)
    empty-c-t-env))

C.9 compiler/misc.fx

The contents of the file compiler/misc.fx:

;;; Mode: Scheme; Package: SCHEME

(define id (lambda (x) x))

(define max (lambda (n1 n2) (if (> n1 n2) n1 n2)))

;;; List routines
(define integers-between
  (lambda (lo hi)
    (if (> lo hi) (null) (cons lo (integers-between (+ 1 lo) hi)))))
(define reverse-list
   ; minifx doesn't have this built-in.
   (lambda (lst)
      (letrec ((rlist (lambda (l r)
               (if (null? l)
                 r
                 (rlist (cdr l) (cons (car l) r))))))
       (rlist lst (null)))))

(define reduce-left
   (lambda (fn lst seed)
      (letrec ((loop (lambda (l v)
               (if (null? l)
                 v
                 (loop (cdr l) (fn v (car l))))))
       (loop lst seed)))))

(define for-each
   (lambda (proc l)
      (if (null? l)
        the-unit
        (begin (proc (car l))
               (for-each proc (cdr l)))))

(define for-each-2
   (lambda (proc lst1 lst2)
      (if (null? lst1)
        the-unit
        (begin
           (proc (car lst1) (car lst2))
           (for-each-2 proc (cdr lst1) (cdr lst2))))))

(define 1st car)
(define 2nd cadr)
(define 3rd caddr)
(define 4th caddr)

;; Vector stuff

;; String stuff

(define map-string
   (lambda (proc str)
      (letrec ((len (string-length str))
               (loop (lambda (i)
                        (if (< i len)
                            (begin (string-set! str i (proc (string-ref str i)))
                                   (loop (+ i 1)))
                                   str)))
       (loop 0)))))
(define char->string
  (lambda (c)
    (let ((s "x"))
      (begin (string-set! s 0 c)
        (string-copy s)))))

(define down-string
  (lambda (s) (map-string char-downcase s)))

(define up-string
  (lambda (s) (map-string char-upcase s)))

(define pad
  (lambda (s sz) (substring " " 0 (- sz (string-length s)))))

(define down-sym
  (lambda (s) (down-string (symbol->string s))))

(define up-sym
  (lambda (s) (up-string (symbol->string s))))

;; Stream stuff

(define copy-input-stream-to-output-stream
  (let ((terminators (char-set #\newline)))
    (lambda (fin fout)
      (let ((line (read-string terminators fin)))
        (if (eso-object? line)
          the-unit
          (begin
            (*scheme-read-char* fin) ; clear the newline
            (display line fout)
            (newline fout)
            (copy-input-stream-to-output-stream fin fout)))))))

;; Wrappers for uniqueof functions (may not be the right semantics for
;; all data types...)

(define unique (lambda (W) x))

(define value (lambda (W x) x))

;; Vector stuff

(define generate-vector
  (lambda (size proc)
    (letrec ((ans (make-vector size))
      (loop
        (lambda (i)
          (if (= i size)
            ans
            (begin (vector-set! ans i (proc i))
            (loop (+ i 1)))))))
    (loop 0))))
C.10  compiler/oc2txt.fx

The contents of the file compiler/oc2txt.fx:

```
;; Faked-up implementation of tuples:
(define tuple list)
(define tuple-ref list-ref)
```

```
;; oc2txt.fx -- output ocode to a file in official DLX assembly format

(define print-one-instruction
  (lambda (x strm)
    (begin
      (match x
        ((ocode "labeldef (label->rands" lab))
          (display (string-append lab "") strm))
        (_ (display (string-append " " (unparse-ocode x)) strm)))
    (newline strm))))

;; Routines for printing stuff, including unparsing results from micro-FX
;; program run in DLX simulator

(define extract-value
  ;; Unparse result from DLX simulator
  (lambda (word type)
    (match (prune type)
      ((base->type "bool"
          (bool->sexp (not (= word 0))))
        ((base->type "char"
            (char->sexp (int->char (quotient word 2))))
          ((base->type "int"
              (int->sexp (quotient word 2))))
            ((base->type "string"
                (string->sexp (extract-string word)))
              ((base->type "sym"
                  (sym->sexp (extract-symbol word)))
                (base->type "unit"
                  (sym->sexp 'the-unit)))
```
((compound->type'->)
(unparse-type type))

((compound->type'pairof et)
(pair->sexp (cons (extract-value (get-slot word 0) (car et))
(extract-value (get-slot word 1) (cadr et)))))

((compound->type'listof et)
(list->sexp (extract-list word (car et)))))

((compound->type'vectorof et)
(vector->sexp (extract-vec word (car et)))))

(_, '(unrecognized type,(unparse-type type),word)))

(define extract-string
(lambda (word)
(let* ((vc (extract-vec word (parse-type 'char))))
(list->string (vector->list vc))))

(define extract-symbol
(lambda (word) (string->sym (get-mem word))))

(define extract-list
(lambda (word type)
(if (= word 0)
(null)
(cons (extract-value (get-slot word 0) type)
(extract-list (get-slot word 1) type))))

(define extract-vec
(lambda (word type)
(letrec ((len (de-itag (get-slot word 1)))
(loop
(lambda (v i)
(if (= i len)

(v
(begin
(vector-set! v i (extract-value (get-slot word i) type))
(loop v (+ i 1))))))))

(loop (make-vector len 0)))))

(define type-to-printf-format
;; A munged version of unparse-type (../frontend/parse.fx) that prints
;; out a printf-like format string for printing the result of the
;; computation. The printf-like code is written in DLX assembly code
;; in the runtime directory.
(lambda (type)
(if (recognize-type? type)
 (string-append "%" (type2printf type))
 (string-append "unrecognized type"))))
C.11 compiler/ocode.fx  C  \(\mu\)FX/DLX COMPILER IMPLEMENTATION

(define recognize-type?  
(lambda (type)  
  (match (prune type)  
    ((base->type" t) (if (memq t '(int bool char string sym unit)) #t #f))  
    ((compound->type" 'listof _) #t)  
    ((compound->type" t sub-types)  
      (and (memq t '(listof pairof vectorof refof ->))  
        (reduce-left and? (map recognize-type? sub-types) #t)))  
    ((unknown->type") #f))))

(define type2printf  
(lambda (type)  
  (match (prune type)  
    ((base->type" 'int) "d")  
    ((base->type" 'bool) "b")  
    ((base->type" 'char) "c")  
    ((base->type" 'string) "vc") ; strings are vec's of chars  
    ((base->type" 'sym) "a")  
    ((base->type" 'unit) "u")  
    ((compound->type" 'listof q)  
      (match (prune (car q))  
        ((tvariable->type" _) "ld")  
        (q1 (string-append "l" (type2printf q1))))))  
    ((compound->type" 'pairof operands)  
      (string-append  
        "p"  
        (string-append (type2printf (car operands))  
          (type2printf (cadr operands)))))  
    ((compound->type" 'vectorof operands)  
      (string-append "v" (type2printf (car operands)))))  
    ((compound->type" 'refof operands)  
      (string-append "r" (type2printf (car operands)))))  
    ((compound->type" '-> operands) "F"))))

C.11  compiler/ocode.fx

The contents of the file compiler/ocode.fx:

;; -*- Mode: Scheme; Package: SCHEME -*-

;; object code format

(define insn cons)
(define op-code car)
(define op-rands cdr)
(define ocode" cons")
(define-datatype rands
  (rrr->rands int int int)
  (rrri->rands int int string)
  (rr->rands int int)
  (rri->rands int int string)
  (r->rands int)
  (i->rands string)
  (load->rands int int int)
  (store->rands int int int)
  (nop->rands)
  (symbol->rands sym)
  (string->rands string)
  (label->rands string)
  (word->rands string)
  (err->rands string))

C.12 compiler/optimize.fx

The contents of the file compiler/optimize.fx:

;;;; -*- Mode: Scheme; Package: SCHEME -*-
;;;; icode optimizer pass

(define optimize-icode
  (lambda ()
    (:= icode-list (map prop-returns-down (~ icode-list)))))

;;;; Tail call handling
;;;;
;;;; A tail call is any call that is immediately followed by a return (in
;;;; execution order). We find them by pushing returns downward in the
;;;; tree, looking for (return (call foo)) and changing that to (jump foo).
;;;;
;;;; To push returns downward in a tree, we use the transformations:
;;;;
;;;;  (return (if a b c))     -->  (if a (return b) (return c))
;;;;  (return (begin e1 e2 ... en)) --> (begin e1 e2 ... (return en))
;;;;
(define prop-returns-down
  (lambda (ic)
    (match ic
      ((body->icode ty n b) (body->icode ty n (prop-returns-down b)))
      ((letrec->icode n args b)
        (letrec->icode n args (prop-returns-down b)))))

97
C.13 compiler/parse.fx

The contents of the file compiler/parse.fx:

```scheme
;;; -e- Mode: Scheme; Package: SCHEME -e-

;; Expression and type parsers

(define parse
  (lambda (sexp) (parse-exp sexpr)))

;; Parse a single expression
(define parse-exp
  (lambda (sexp)
    (match sexp
      ((sym->sexp" sym) (variable->exp (ref unknown-type) sym))
      ((char->sexp" c) (char->exp (ref character-type) c))
      ((bool->sexp" b) (bool->exp (ref boolean-type) b))
      ((int->sexp" n) (int->exp (ref integer-type) n))
      ((string->sexp" s) (string->exp (ref string-type) s))
      ;; First thing in list is a SYMBOL
      ('(,(sym->sexp" head) ,@_) ((get-parser-for-keyword head) sexpr))
      ;; Procedure call is the default
      ('(,operator ,@operands) (parse-combination operator operands))
      (_ (error "unrecognized expression" sexpr)))))
```

C.13 compiler/parse.fx

C.13  compiler/parse.fx

```

((return->icode" (op->icode" ty 'call args))
  (op->icode ty 'jump args))

((return->icode" (op->icode" ty 'if tac))
  (op->icode ty
    'if
    (list (car tac)
      (prop-returns-down (return->icode (cadr tac)))
      (prop-returns-down (return->icode (caddr tac))))))

((return->icode" (op->icode" ty 'begin exprs))
  (letrec ((p-r-down-last
    (lambda (l)
      (if (null? (cdr l))
        (list (prop-returns-down (return->icode (car l))))
        (cons (car l) (p-r-down-last (cdr l))))))))

(_ ic))))
```
;;; Parse a definition
(define parse-definition
  (lambda (sexpr)
    (match sexpr
      ('(define ,name ,value)
        (make-definition (parse-formal name) (parse-exp value)))
      (_ (error "invalid definition" sexpr))))

;;; check-out a formal parameter; make sure it's not a reserved word.
(define parse-formal
  (lambda (sexpr)
    (match sexpr
      ((sym->sexp name)
        (if (memq name (- all-keywords))
          (error "attempt to use reserved word as variable name" sexpr)
          name))
      (_ (error "invalid variable name" sexpr))))

;;; Most of the rest of this file concerns itself with special forms
;;; (expressions of the form (reserved-word ...)). Define-keyword is a
;;; function that defines a reserved word, associating it with a function
;;; that can parse the named construct.

;;; List of parsing functions. Each checks to see if the new keyword
;;; is its own, and either parses the whole thing or passes the buck.
(define keyword-table
  (ref (lambda (head)
    (lambda (sexpr) ;; Procedure call is the default
      (match sexpr
        ('(,operator ,Goperands) (parse-combination operator operands)) (_ (error "this shouldn't happen")))))))

(define all-keywords (ref (null))) ; list of keywords.

(define get-parser-for-keyword
  (lambda (name) ((- keyword-table) name)))

(define define-keyword
  (lambda (keyword parser)
    (let ((current-table (' keyword-table)))
      (begin (: keyword-table
        (lambda (head)
          (if (eq? head keyword) parser (current-table head))))
        (: all-keywords (cons keyword (- all-keywords))
          keyword)))

    ;; And here are the parsing functions...
(define-keyword 'symbol
  (lambda (sexpr)
    (match sexpr
      ('(symbol ,sym->sexp name)) (sym->exp (ref symbol-type) name))
      (_ (parse-error sexpr))))

(define-keyword 'call
  (lambda (sexpr)
    (match sexpr
      ('(call ,operator ,operands) (parse-combination operator operands))
      (_ (parse-error sexpr)))))

(define parse-combination
  (lambda (operator operands)
    (combination->exp (ref unknown-type)
      (parse-exp operator)
      (map parse-exp operands))))

(define-keyword 'if
  (lambda (sexpr)
    (match sexpr
      ('(if ,test ,con ,alt)
       (conditional->exp (ref unknown-type)
         (parse-exp test)
         (parse-exp con)
         (parse-exp alt))
       (_ (parse-error sexpr)))))))

(define-keyword 'begin
  (lambda (sexpr)
    (match sexpr
      ('(begin) (parse-exp '(null)))
      ('(begin ,exp) (parse-exp exp))
      ('(begin ,expss) (begin->exp (ref unknown-type) (map parse-exp expss))
       (_ (parse-error sexpr))))))
(define-keyword 'lambda
(lambda (sexpr)
  (match sexpr
   ('(lambda (@formals) ,body)
     (abstraction->exp (ref unknown-type)
      (map parse-formal formals)
      (parse-exp body)))
   (_ (parse-error sexpr)))))

;; LET is not simply sugar because handled specially during typechecking
;; (let ((I E)* E0)
(define-keyword 'let
(lambda (sexpr)
  (letrec ((parse-binding-spec
            (lambda (bspec)
              (match bspec
               ('(,name ,value)
                 (make-definition (parse-formal name) (parse-exp value)))
               (_ (error "invalid binding specifier" bspec))))))
   (match sexpr
    ('(let (,Obspecs) ,body)
     (binder->exp (ref unknown-type)
      (map parse-binding-spec bspecs)
      (parse-exp body)))
    (_ (parse-error sexpr)))))

(define-keyword 'letrec
(lambda (sexpr)
  (letrec ((parse-binding-spec
            (lambda (bspec)
              (match bspec
               ('(,name ,value)
                 (make-definition (parse-formal name) (parse-exp value)))
               (_ (error "invalid binding specifier" bspec))))))
   (match sexpr
    ('(letrec (,sym->sexp"_",J) ,body)
     (parse-error sexpr)) ; LETREC w/ no body
    ('(letrec ,bspecs) ,body)
     (recursion->exp (ref unknown-type)
      (map parse-binding-spec bspecs)
      (parse-exp body)))
    (_ (parse-error sexpr)))))

;; Sugars

101
;; (and) ==> #t
;; (and E) ==> E
;; (and E0 E+) ==> (if E0 (and E+) #f)
;;
(define-keyword 'and
(lambda (sexpr)
  (match sexpr
   ('(and ,@exp-list)
    (parse-exp (letrec ((recur (lambda (exps)
                                 (match exps
                                   (null)      '#t)
                                   (',exp)     exp)
                                   ((cons first rest)
                                    '(if ,first ,(recur rest) #f))))))
    (recur exp-list)))))
  (_ (parse-error sexpr))))

;; (or) ==> #f
;; (or E) ==> E
;; (or E0 E+) ==> (if E0 #t (or E+))
;;
(define-keyword 'or
(lambda (sexpr)
  (match sexpr
   ('(or ,@exp-list)
    (parse-exp (letrec ((recur (lambda (exps)
                                 (match expa
                                   (null)   '(i)
                                   (',exp)  exp)
                                   ((cons first rest)
                                    '(if ,first it ,)(recur rest))))
                        (recur exp-list)))))
    (recur exp-list)))))
  (_ (parse-error sexpr))))

;; (list E*)
;;
(define-keyword 'list
(lambda (sexpr)
  (match sexpr
   ('(list ,@exp-list)
    (parse-exp (letrec ((recur (lambda (exps)
                                 (match exps
                                   (null) '(null))
                                   (',exp) exp)
                                   ((cons first rest)
                                    '(if ,first ,)(recur rest))))))
    (recur exp-list)))))
  (_ (parse-error sexpr))))

(define parse-error
(lambda (sexpr)
  (error "invalid expression syntax" sexpr)))
;; Unparser

(define unparse
  (lambda (exp)
    (unparse-exp exp)))

(define unparse-exp
  (letrec ((unparse-binding-specs
                (lambda (dots)
                  (map (lambda (dot)
                         '((definition-name dot)
                           (unparse-exp (definition-value dot)))
                         dots))))
    (lambda (exp)
      (match exp
        ((variable->exp _ var) (sym->sexp var))
        ((bool->exp _ b) (bool->sexp b))
        ((int->exp _ n) (int->sexp n))
        ((string->exp _ s) (string->sexp s))
        ((char->exp _ c) (char->sexp c))
        ((sym->exp _ name) 'symbol (sym->sexp name))
        ((conditional->exp _ test consequent alternate)
         'if (unparse-exp test)
          (unparse-exp consequent)
          (unparse-exp alternate))
        ((begin->exp _ exprs)
         'begin (map unparse-exp exprs))
        ((abstraction->exp _ formals body)
         '(lambda ,0(formals) ,unparse-exp body))
        ((combination->exp _ operator operands)
         ',unparse-exp operator
         ,0(map unparse-exp operands))
        ((binder->exp _ defs body)
         'let ,0(unparse-binding-specs defs)
         ,unparse-exp body))
        ((recursion->exp _ defs body)
         'letrec ,0(unparse-binding-specs defs)
         ,unparse-exp body))))))
;; Generic pretty printer.
;; ppriflt-type is a function on types to include the reconstructed
;; type information in the output.
;;
;; (define ppriflt-type

(define pprint-ezp-gen
  (letrec ((pp-binding-specs
    (lambda (defs ppriflt-type)
      (map (lambda (def)
            "(',(definition-name def)
             ,(pprint-exp-gen (definition-value def) ppriflt-type)))
        defs)))
   (lambda (exp ppriflt-type)
     (match exp
       ((variable->exp ty var)
        (list 'variable->exp (pprint-type ty) (sym->sexp var)))
       ((bool->exp ty b)
        (list 'bool->exp (pprint-type ty) (bool->sexp b)))
       ((int->exp ty n)
        (list 'int->exp (pprint-type ty) (int->sexp n)))
       ((char->exp ty c)
        (list 'char->exp (pprint-type ty) (char->sexp c)))
       ((string->exp ty s)
        (list 'string->exp (pprint-type ty) (string->sexp s)))
       ((sym->exp ty name)
        (list 'sym->exp (pprint-type ty) (sym->sexp name)))
       ((conditional->exp ty test consequent alternate)
        (list 'conditional->exp (pprint-type ty)
             (pprint-exp-gen test ppriflt-type)
             (pprint-exp-gen consequent ppriflt-type)
             (pprint-exp-gen alternate ppriflt-type)))
       ((begin->exp ty exprs)
        (cons 'begin->exp (cons (pprint-type ty)
                                (map (lambda (e) (pprint-exp-gen e ppriflt-type))
                                      exprs)))))
       ((abstraction->exp ty formals body)
        (list 'abstraction->exp (pprint-type ty)
             "(',@formals) (pprint-exp-gen body ppriflt-type))")
       ((combination->exp ty operator operands)
        (list 'combination->exp (pprint-type ty)
             (pprint-exp-gen operator ppriflt-type)
             (map (lambda (op) (pprint-exp-gen op ppriflt-type)) operands)))
       ((binder->exp ty defts body)
        (list 'binder->exp (pprint-type ty)
             (pp-binding-specs defts ppriflt-type)
             (pprint-exp-gen body ppriflt-type)))
       ((recursion->exp ty defts body)
        (list 'recursion->exp (pprint-type ty)
             (pp-binding-specs defts ppriflt-type)
             (pprint-exp-gen body ppriflt-type)))))))

;; pprinter.
;;
;; (define (pprint-exp exp) (pprint-exp-gen exp (lambda (ty) 'ty)))
;; pprint with types.
;;
(define (pprint-exp-types exp)
  (pprint-exp-gen exp (lambda (tref) (unparse-type (- tref)))))

;; Type expression parser
;;
(define parse-type
  (lambda (sexpr)
    (match sexpr
      ((sym->sexp sym) (base->type sym))
      ('-> (,.arg-types) .result-type)
      (compound->type arrow-constructor
        (cons (parse-type result-type)
              (map parse-type arg-types)))
      ('(,(sym->sexp name) .types)
        (compound->type name (map parse-type types)))
      (_ (error "invalid type expression syntax" sexpr)))))

;; Type expression unparser
;;
(define unparse-type
  (lambda (type)
    (match (prune type)
      ((base->type sym) (syz->sexp sym))
      ((compound->type constructor operands)
        (if (same-constructor? constructor arrow-constructor)
          '(- (,.unparse-type (cdr operands)))
          ,(unparse-type (car operands)))
      ('(,(sym->sexp constructor) .operands)
        (compound->type constructor (map unparse-type operands)))
      ((tvariable->type tvar)
        (sym->sexp (tvariable->sym tvar)))
      ((unknown->type)
        "(*unknown*)"))))

;; Parse a type schema (generic (I* T))
;;
(define parse-schema
  (lambda (sexpr)
    (match sexpr
      ('(generic (,Cnames) ,type)
        (let ((names (map (lambda (name)
                                (match name
                                  ((sym->sexp name) name)
                                  (_ (error "invalid type schema parameter" name)))))
                  names))
          (let ((tvars (map new-tvariable names)))
            (make-schema tvars
              (substitute-for-names (map tvariable->type tvars) names
                                      (parse-type type)))
              (_ (make-schema (null) (parse-type sexpr))))))))
(define substitute-for-names
  ; substitute-for-names is a kludge, to be used only by initialization
  ; code. Other ways to do this: (1) change the type parser to take an
  ; environment argument; (2) generalize substitute-into-type so that it
  ; can substitute for either names or tvars; (3) change the
  ; representation of schemas so that the generic variables in the type
  ; are not tvars but rather names.
  (lambda (types names type)
    (match type
      (((tvariable->type' _) type) ; shouldn't happen
        (if (null? ts)
            type
            (if (same-name? name (car ns))
                (car to)
                (loop (cdr ts) (cdr ns))))>
      ((base->type' name)
        (letrec ((loop (lambda (ts ns)
            (if (null? ts)
                type
                (if (same-name? name (car ns))
                    (car ts)
                    (loop (cdr ts) (cdr ns))))))))
      ((compound->type c args)
        (compound->type c (map (lambda (arg)
            (substitute-for-names types names arg))
            args)))
      (_, (error "this shouldn't happen" type)))))

(define unparse-schema
  (lambda (s)
    (match s
      (((make-schema' tvars type)
        '((generic ,0(map sym->sexp (map tvariable->sym tvars)))
          ,(unparse-type type))))))

C.14 compiler/system.fx

The contents of the file compiler/system.fx:

;;; -- Mode: Scheme; --

;;; System routines, including garbage collector, used by DLX simulator
;;; System routines (which don't live in simulated memory) have
;;; negative addresses so we can take their addresses, etc, and not get
;;; confused.
(define enter-system-routine-labels
  (lambda ()
    (enter-label "ALLOC" -4)
    (enter-label "SAVE" -8)
    (enter-label "RESTORE" -12)
    (enter-label "EXIT" -16)
    (enter-label "PUTCHAR" -20)
    (enter-label "SYM2STRING" -24)
    (enter-label "SALLOC" -28)
    (enter-label "SFREE" -32)
    (enter-label "BZERO" -36)))

(define system-routine
  ;; Call sys routines by passing calling this function w/ routine's addr
  (lambda (funcnumber)
    (cond ((= funcnumber -4) (allocate-block-of-memory))
          ((= funcnumber -8) (save-regs-into-frame))
          ((= funcnumber -12) (restore-regs-from-frame))
          ((= funcnumber -16) (done-emulating))
          ((= funcnumber -20) (putchar))
          ((= funcnumber -24) (sym2string))
          ((= funcnumber -28) (stack-allocate-block))
          ((= funcnumber -32) (stack-free-block))
          ((= funcnumber -36) (zero-block))
          (else (error "unknown system routine called."))))

;; Here are the system routines. They're basically the same as the
;; versions in microFX/runtime (though some of the names may have
;; changed through negligence).

;; _SYM2STRING
(define sym2string
  ;; Turn a symbol into a vector of characters
  (lambda ()
    (letrec ((str (get-mem (get-reg ATEMP)))
      (fill-block
        (lambda (i)
          (if (> i (string-length str))
              the-unit
              (begin (set-slot! (get-reg ATEMP) i
                (itag (char->int (string-ref str i))))
              (fill-block (+ i 1)))))))
    (begin (set-reg! ITEMP (itag (string-length str)))
            (allocate-block-of-memory)
            (fill-block 0))))

;; _PUTCHAR
(define putchar (lambda () (display (int->char (de-itag (get-reg ATEMP))))))

;; _EXIT
(define done-emulating (lambda () (:= *halt-emulate?* #t)))
(define save-regs-into-frame
  (lambda ()
    (letrec ((f (get-reg FP))
      (loop (lambda (reg slot)
          (if (= reg 30)
            the-unit
            (begin (set-slot! f slot (get-reg reg))
              (loop (+ 1 reg) (+ 1 slot))))))
      (loop 6 4))))

(define restore-regs-from-frame
  (lambda ()
    (letrec ((f (get-reg FP))
      (loop (lambda (reg slot)
          (if (= reg 30)
            the-unit
            (begin (set-reg! reg (get-slot f slot))
              (loop (+ 1 reg) (+ 1 slot))))))
      (loop 6 4))))

;; For the allocator and garbage collector, see the commentary in
;; "microFI/runtime/alloc.s. This version is sufficiently similar
;; that those comments should apply here.

(define zero-block
  (lambda ()
    (letrec ((loop (lambda (blk len)
          (if (= len 0)
            the-unit
            (begin (set-slot! blk (- len 1) 0)
              (loop blk (- len 1))))))
      (loop (get-reg ATEMP) (de-itag (get-slot (get-reg ATEMP) -1))))))

(define stack-allocate-block
  (lambda ()
    (let* ((nslots (de-itag (get-reg ATEMP)))
      (new-sp (- (get-reg SP) (+ (* nslots 4) 4))) ; Extra slot for size
      (stack-size (- (- (* entire-memory-size*) (quotient new-sp 4)) 1))
      (new-block (+ new-sp 5))) ; Tagged ptr to new GC block
      (begin
        (if (> stack-size (* max-stack-size*))
          (= *max-stack-size* stack-size)
          the-unit)
        (if (> stack-size (* stack-size*))
          (error "Stack overflow."
            (begin
              (= *total-allocation* (+ 1 (+ (* *total-allocation*) nslots)))
              (set-slot! new-blk -1 (get-reg ATEMP)) ; Size
              (set-reg! ATEMP new-blk)
              (zero-block)
              (set-reg! SP new-sp))))))
;;; _SFREE
(define st::c\_free-block
(lambda ()
  (let* ((nslots (de-itag (get-slot (get-reg ATEMP) -1)))
         (new-sp (+ (get-reg SP) (+ (* nslots 4) 4))))
    (if (>= new-sp (* (\*entire-memory-size\) 4))
        (error "Stack underflow.")
        (set-reg! SP new-sp))))

;;; _ALLOC
(define allocate-block-of-memory
(lambda ()
  (let* ((nslots (de-itag (get-reg ATEMP)))
         (p (allocate-raw-block-of-memory (+ nslots 1))))
    (set-slot! p -1 (get-reg ATEMP)) ; (tagged)
    (set-reg! ATEMP p)
    (zero-block))))

(define allocate-raw-block-of-memory
(lambda (nslots)
  (let ((block (get-reg HP))
        (begin (set-rag! HP (+ block (* nslots 4))) ; 4 bytes per slot
          (if (<= (get-reg HP) (^this-semispace-end*))
              (begin
                (:*total-allocation* (- *total-allocation* nslots))
                block)
              (begin
                (let ((stack-atemp (get-reg ATEMP)))
                  (interpreter-gc)
                  (set-reg! ATEMP stack-atemp))
                (:*num-gcs* (+ (- *num-gcs* 1))
                 (:*gc-words-copied*
                  (+ (- *gc-words-copied*)
                   (quotient (- (get-reg HP) (^this-semispace-end*)) 4)))
                (if (> (+ (get-reg HP) (* nslots 4)) (^this-semispace-end*))
                  (error "out of memory!")
                  (allocate-raw-block-of-memory nslots))))))))

(define stack-depth
(letrec ((loop (lambda (p count)
                  (if (even? p)
                      count
                      (loop (get-slot p 0) (+ count 1))))))))

(define interpreter-gc
(lambda ()
  (begin
    (if (~*ncis-gc*)
      (begin (newline)
        (display "- - - - - - Beginning garbage-collection"))
      the-unit)))

109
;; Create frame on stack to save old values of registers
(set-reg! ATEMP (itag FrameSize))
(stack-allocate-block)
(set-slot! (get-reg ATEMP) 0 (get-reg FP))
(set-reg! FP (get-reg ATEMP))
(set-slot! (get-reg FP) 3 (get-reg ENV))
(save-regs-into-frame)

;; flip the semispaces
(let ((old-start (- *this-semispace*))
      (old-end (- *this-semispace-end*))
      (begin (:= *this-semispace* (~ other-semispace*))
              (:= ~other-semispace-old-start)
              (:= *other-semispace-end* (~ other-semispace-end*))
              (:= ~other-semispace-end* old-end))
      (set-reg! HP (~ this-semispace*))
      (set-reg! FP (rafte-copy (get-reg FPM))
                  (set-reg's V4J (maybe-copy (get-reg VAL)))
      (restore-regs-from-frame)
      (set-reg! ATEMP (get-reg FP))
      (set-reg! ENV (get-slot (get-reg FP) 3))
      (set-reg! FP (get-slot (get-reg FP) 0))
      (stack-free-block)
      (if (~ noisy-gc)
          (begin
            (newli le)
            (display "-----
            (display quotient (~ this-semispace-end* (get-reg HP)) 4))
            (display "/")
            (display (~ semispace-size*))
            (display " words free.
            the-unit))))))

(define in-thisspace?
  (lambda (p) (and (#= p (~ this-semispace*))
                  (< p (~ this-semispace-end*))))))

(define in-otherspace?
  (lambda (p) (and (#= p (~ other-semispace*))
                  (< p (~ other-semispace-end*))))))

(define in-stack?
  (lambda (p) (> p (get-reg SP)))))
(define maybe-copy
  ;; Copy heap blocks into new heap space, returning new result
  (lambda (p)
    (cond ((even? p) p) ; don't copy atoms
      ((in-thisspace? p) p) ; optimization: obj's in thisS already scanned
      ((not (in-otherspace? p))
        (begin (scan-transitively p)) ; only scan blocks not in old space
        ((odd? (get-slot p -1)) (get-slot p -1)) ; return forward addr
      (else ;; ok, we've got a live one.
        (let ((neup (get-reg HP))
              (nslots (+ (de-itag (get-slot p -1)) 1))) ; untagged wordcount
          (begin (set-reg! HP (+ neup (* nslots 4)))
            (set-slot! newp -1 (get-slot p -1)) ; size field
            (copy-block newp p (- nslots 1)) ; data fields
            (set-slot! p -1 newp) ; set forward ptr.
            (scan-transitively newp)
            newp))))))

(define copy-block ; copies from slot 0 to slot "slot - 1", inclusive.
  (lambda (new old slot)
    (if (<= slot 0)
      the-unit
      (begin (set-slot! new (- slot 1) (get-slot old (- slot 1)))
        (copy-block new old (- slot 1)))))

(define scan-transitively
  (lambda (p)
    (letrec ((loop (lambda (slot)
        (if (< slot 0)
          the-unit
          (begin
            (set-slot! p slot (maybe-copy (get-slot p slot)))
            (loop (- slot 1)))))))
    (loop (- (de-itag (get-slot p -1)) 1)))))

C.15  compiler/table.fx

The contents of the file compiler/table.fx:

;;
;; polymorphic symbol-tables (compile-time environments)
;;
;; There are three components to this package: a function to create an
;; empty environment "(mk-empty-env empty-function)", a function to create a
;; "binder" (i.e. a function to enter key/value pairs into the table)
;; "(mk-binder equality-comparator)", and a polymorphic lookup function
;; "(lookup key env)".
;; types used in the comments below:
;;
;;;; empty-flag : empty-type
;;;; empty-type == value-type
;;;; key : key-type
;;;; env : key-type -> value-type

;;;; mk-empty-env : (poly (key-type value-type)
;;;;           (-> ((-> (key-type) value-type)) env))
;;;;
;;;; (define mk-empty-env
;;;;     (lambda (empty-fn) (lambda (key) (empty-fn key)))))

;;;; mk-binder : (poly (key-type value-type)
;;;;        (-> ((-> (key-type key-type) bool))
;;;;            (-> (key-type value-type env) env))
;;;; here, "env" is a macro, borrowing the definitions of key-type and
;;;; value-type from the poly params.
;;;;
;;;; (define mk-binder
;;;;     (lambda (key=?)
;;;;         (lambda (key value env)
;;;;             (lambda (new-key)
;;;;                 (if (key=? key new-key)
;;;;                     value
;;;;                     (lookup new-key env)))))

;;;; lookup : (poly (key-type value-type)
;;;;        (-> (key-type (-> (key-type) value-type))
;;;;            value-type))
;;;;
;;;; (define lookup (lambda (key env) (env key)))
;;; The contents of the file compiler/toplevel.fx:
;; -e- Mode: Scheme; Package: SCHEME -*-
;;
;; (define *verbose-flag* (ref #f)) ;; makes the interpreter noisy...
;; (define *silent-flag* (ref #f)) ;; makes the interpreter SILENT.

C.16 compiler/toplevel.fx
;;; parse a microFI s-expression
;;; (define test-parse
;;;   (lambda (sexp)
;;;     (pprint-exp (parse sexp))))

;;; parse a microFI s-expression
;;; (should reproduce input)
;;; (define test-parse-simple
;;;   (lambda (sexp)
;;;     (unparse (parse sexp))))

;;; type-check a microFI s-expression
;;; (define check
;;;   (lambda (e)
;;;     (unparse-type (reconstruct-top (parse e)))))

;;; type-check a microFI s-expression and display the expression tree
;;; annotated with the reconstructed type information.
;;; (define show-type-check
;;;   (lambda (e)
;;;     (let ((parse-tree (parse e)))
;;;       (begin (reconstruct-top parse-tree)
;;;              (pprint-exp-types parse-tree)))))

;;; compile expression to icode.
;;; (define itest-compile
;;;   (lambda (sexpr)
;;;     (let* ((exp (parse sexpr))
;;;            (type (reconstruct-top exp)))
;;;       (begin (newline) (display "Type: ") (write (unparse-type type)) (newline)
;;;              (generate-icode exp)
;;;              (display-icode-list)))))

;;; compile expression to optimized icode.
;;; (define otest-compile
;;;   (lambda (sexpr)
;;;     (let* ((exp (parse sexpr))
;;;            (type (reconstruct-top exp)))
;;;       (begin (newline) (display "Type: ") (write (unparse-type type)) (newline)
;;;              (generate-icode exp)
;;;              (optimize-icode)
;;;              (display-icode-list))))
;; compile expression to ocode (prints assembly code).
;;
(define test-compile
  (lambda (sexpr)
    (let* ((exp (parse sexpr))
           (type (reconstruct-top exp)))
      (begin
        (newline) (display "Type: ") (write (unparse-type type)) (newline)
        (generate-icode exp)
        (optimize-icode)
        (generate-ocode)
        (newline) (display "Object code: ") (newline)
        (display-ocode-list (current-output-port))
        (newline))))

;; compile to ocode, then interpret ocode.
;;
(define run
  (lambda (sexpr)
    (let* ((exp (parse sexpr))
           (type (reconstruct-top exp)))
      (begin
        (if (~ *silent-flag*)
          the-unit
          (begin (newline)
            (display "Type: ") (write (unparse-type type)) (newline)))
          (generate-icode exp)
        (optimize-icode)
        (generate-ocode)
        (if (~ *silent-flag*)
          the-unit
          (begin (newline) (display "Running: ") (newline)))
          (init-emulator)
        (rerun)
        (if (~ *silent-flag*)
          the-unit
          (begin (newline) (show-stats)))
        (extract-value (get-reg VAL) type)))))
;; compile to ocode, then interpret ocode. free-up spare memory first!
;; (worse for debugging, but better for automatic testing.)
;;
(define run-u
  (lambda (sexpr)
    (let* ((exp (parse sexpr))
           (type (reconstruct-top exp)))
      (begin
        (if (~ *silent-flag*)
          the-unit
          (begin (newline)
                (display "Type: ") (write (unparse-type type)) (newline)))
        (generate-icode exp)
        (optimize-icode)
        (generate-ocode)
        (if (~ *silent-flag*)
          the-unit
          (begin (newline) (display "Running: ") (newline)))
        (begin (newline) (display "Running: ") (newline)))
      (begin (newline) (show-stats)))
    (extract-value (get-reg VAL type))))

;; like run, but verbose (prints instruction stream as it executes).
;;
(define runv
  (lambda (sexpr)
    (let ((old-vflaq (~ *verbose-flag*))
      (begin
        (:= *verbose-flag* ft)
        (let ((retval (run sexpr))
          (begin (:= *verbose-flag* old-vflaq)
                  retval))))))

;; print out various statistics about the run (currently only gc stats).
;;
(define show-stats
  (lambda ()
    (begin ()
      (newline)
      (display "#gc's=") (display (~ *num-gcs*))
      (display " words copied by gc=") (display (~ *gc-words-copied*))
      (display " words allocated=") (display (~ *total-allocation*))
      (display " total-allocs=") (display (~ *total-allocs*))
      (newline))))
;; compile for dlxsim into the file "fx.s".
;;
(define fx
  (lambda (sexpr) (fx-with-outname sexpr "fx.s")))

;; compile for dlxsim into a named file.
;;
(define fx-with-outname
  (lambda (sexpr outname)
    (let ((prologname (string-append compiler-directory "/prolog.code")))
      (epilogname (string-append compiler-directory "/epilog.code")))
    (old-pp " *rr-pretty*)
    (let* ((exp (parse sexpr))
      (type (reconstruct-top exp))
      (fout (open-output-file outname)))
      (begin
        (= *rr-pretty* #f) ; No printing for DLX-ASM output
        (newline) (display "Type:" ) (write (unparse-type type)) (newline)
        ;; Compile program
        (generate-icodes exp)
        (optimize-icode)
        (generate-ocode)
        ;; Copy prolog to output file
        (newline fout)
        (let ((prolog (open-input-file prologname)))
          (copy-input-stream-to-output-stream prolog fout)
          (close-input-port prolog))
        ;; write compiled code (as text) to output file
        (display-ocode-list fout)
        ;; Output format string to print the result (see printf.s in runtime)
        (display "RESULT_FORMAT:" fout) (newline fout)
        (display " .ascii " fout)
        (write (type-to-printf-format type) fout)
        (newline fout)
        (display " .byte 0x0a,0" fout) (newline fout)
        (display " .align 2" fout) (newline fout)
        ;; Copy epilog to output file
        (newline fout)
        (let ((epilog (open-input-file epilogname)))
          (copy-input-stream-to-output-stream epilog fout)
          (close-input-port epilog))
        (close-output-port fout)
(display "Object code has been written to file ") (display outname) (newline) (:= *rr-pretty* old-pp)))))

(define listing
  (lambda () (do-listing (* ocode-list) "fx.asm")))))

(define do-listing
  (lambda (l outname)
    (letrec ((fout (open-output-file outname))
      (loop (lambda (pc 1)
        (if (null? 1)
            the-unit
            (let ((new-pc (match (car 1)
                           ((ocode 'labeldef _) pc)
                           ((ocode 'stringdef _) (+ pc 4))
                           (_ (+ pc 4))))
              (begin
                (newline fout)
                (if (= pc new-pc)
                    the-unit
                    (begin
                      (if (< pc 1000) (display "0" fout) the-unit)
                      (if (< pc 100) (display "0" fout) the-unit)
                      (if (< pc 10) (display "0" fout) the-unit)
                      (display (unparse-ocode (car 1)) fout)
                      (display (unparse-ocode (car 1)) fout)
                      (loop new-pc (cdr 1))))))))
     (begin (loop 0 1)
      (close-output-port fout)
      (newline)
      (display "Listing has been written to file ") (display outname)
      (newline))))))

;; some shorthand...
;;
(define tc test-compile)
(define ic itest-compile)
(define oc otest-compile)

;; print-out the list of icode.
;;
(define display-icode-list
  (lambda ()
    (begin
      (newline) (display "Icode: ") (newline)
      (for-each display-icode (~ icode-list))
      (newline))))
C vsFX/DLX COMPILER IMPLEMENTATION

C.17 compiler/ty_recon.fx

The contents of the file compiler/ty_recon.fx:

;;; print-out the list of ocode.
;;
(define display-ocode-list
 (lambda (strm)
   (newline) (display "Code: ") (newline)
   (for-each
     (lambda (x) (print-one-instruction x strm))
     (ocode-list))))

;;; Mode: Scheme; ---
;;; modified 11/14/90 by jwo for mini-fx(90) version
; Contains the occurs-check fix

; Correction for future: when printing out type
; clashes, should substitute types.

; Most, but not all, of a type reconstruction program.

; Type reconstruction

(define reconstruct-top
 (lambda (e)
   (begin
     (reset-tvariable-counter!)
     (reconstruct e standard-type-environment))))
(define reconstruct
(lambda (exp tenv)
  (match exp
    ((variable->exp type-ptr var)
     (memoize-type type-ptr (reconstruct-variable var tenv)))
    ((bool->exp type-ptr _)
     (memoize-type type-ptr boolean-type))
    ((int->exp type-ptr _)
     (memoize-type type-ptr integer-type))
    ((char->exp type-ptr _)
     (memoize-type type-ptr character-type))
    ((string->exp type-ptr _)
     (memoize-type type-ptr string-type))
    ((sym->exp type-ptr _)
     (memoize-type type-ptr symbol-type))
    ((conditional->exp type-ptr test con alt)
     (memoize-type type-ptr (reconstruct-conditional test con alt tenv)))
    ((begin->exp type-ptr exprs)
     (memoize-type type-ptr (reconstruct-begin exprs tenv)))
    ((abstraction->exp type-ptr formals body)
     (memoize-type type-ptr (reconstruct-abstraction formals body tenv)))
    ((combination->exp type-ptr op args)
     (memoize-type type-ptr (reconstruct-combination op args tenv)))
    ((binder->exp type-ptr defs body)
     (memoize-type type-ptr (reconstruct-binder defs body tenv)))
    ((recursion->exp type-ptr dotsa body)
     (memoize-type type-ptr (reconstruct-recursion dotsa body tenv))))
)

(define set-type!
(lambda (exp type)
  (match exp
    ((variable->exp type-ptr _)
     (memoize-type type-ptr type))
    ((bool->exp type-ptr _)
     (memoize-type type-ptr type))
    ((int->exp type-ptr _)
     (memoize-type type-ptr type))
    ((char->exp type-ptr _)
     (memoize-type type-ptr type))
    ((string->exp type-ptr _)
     (memoize-type type-ptr type))
    ((sym->exp type-ptr _)
     (memoize-type type-ptr type))
    ((conditional->exp type-ptr _ _)
     (memoize-type type-ptr type))
    ((begin->exp type-ptr _)
     (memoize-type type-ptr type))
    ((abstraction->exp type-ptr _ _)
     (memoize-type type-ptr type))
    ((combination->exp type-ptr _ _)
     (memoize-type type-ptr type))
    ((binder->exp type-ptr _ _)
     (memoize-type type-ptr type))
    ((recursion->exp type-ptr _ _)
     (memoize-type type-ptr type))))

(define memoize-type
(lambda (type-ptr type)
  (begin (:= type-ptr type) type)))
(define reconstruct-variable
  (lambda (var tenv)
    (let ((tvar-or-schema (tlookup tenv var)))
      (match tvar-or-schema
        ((tvar->tvar-or-schema tvar) (tvariable->type tvar))
        ((schema->tvar-or-schema schema) (instantiate-schema schema))))))

(define reconstruct-conditional ;; if
  (lambda (test con alt tenv)
    (begin (unify! (reconstruct test tenv) boolean-type)
      (let ((con-type (reconstruct con tenv))
           (alt-type (reconstruct alt tenv))
           (begin (unify! con-type alt-type)
                   (con-type)))
      (begin (unify! boolean-type con-type)
              (con-type)))))

(define reconstruct-begin
  (lambda (exprs tenv)
    (begin (map (lambda (exp) (reconstruct exp tenv)) exprs)
           (expression-type (car (list-tail exprs (- (length exprs) 1)))))))

(define reconstruct-abstraction ;; lambda
  (lambda (vars body tenv)
    (let ((new-tvars (map no'-t-variables vars)))
      (make-arrow-type
       (map tvariable->type new-tvars)
       (reconstruct body
        (+extend-by-tvariables tenv vars new-tvars))))))

(define reconstruct-combination ;; call
  (lambda (r, args tenv)
    (let V(arg-types)
      (map (lambda (arg) (reconstruct arg tenv)) args)
      (result-type (tvariable->type (new-tvariable (symbol result))))
      (begin (unify! (reconstruct op tenv)
                      (make-arrow-type arg-types result-type)))
      (result-type))))

(define reconstruct-binder ;; let
  (lambda (defs body tenv)
    (reconstruct body
      (extend-by-schemas tenv
      (map definition-name defs)
      (map (lambda (binding) (compute-schema (reconstruct binding tenv) tenv))
      (map definition-value defs)))))

121
(define reconstruct-recursion ; letrec
(lambda (def body tenv)
  (let* ((names (map definition-name defs))
         (tvars (map new-tvariable names))
         (dummy-tenv (extend-by-tvariables tenv names tvars))
         (types (map (lambda (def)
                       (reconstruct (definition-value def) dummy-tenv))
                    defs))
         (new-tenv (extend-by-schemas
tenv names
         (map (lambda (t) (compute-schema t tenv)) types)))))
  (begin (for-each-2 unity! (map tvariable->type tvars) types)
          (reconstruct body new-tenv)))))

; Note: the use of UNIFY!-LIST rather than FOR-EACH-2 fails to
; correctly type (or find a type error in) the following example:
; (check '(letrec ((a (lambda () 3))
;               (b (if (a 1 2)))
;       4))

; Type schemas
(define compute-schema ; Function GEN from handout
  (lambda (type tenv)
    (make-schema (generic-tvariables type tenv)
                 type))))

; NOTE: generic-tvariables looks not only at tvariables in the
; given type, but also at tvariables in the leaves of
; the fully unwound version of the given type. This interacts with
; a similar unwinding at instantiation time to appropriately handle
; generalization. There is potential confusion in that the returned
; list may contain types that are not manifestly in TYPE but are in
; the fully unwound tree associated with it.

(define generic-tvariables ; Compute FTV(type) - FTE(tenv)
  (lambda (type tenv)
    (match (prune type)
      ((variable->type tvar)
       (if (generic-variable? tvar tenv)
        (list tvar)
        (null)))
      ((compound->type _ operands)
       (letrec ((loop (lambda (ops tvars)
                         (if (null? ops)
                           tvars
                           (loop (cdr ops)
                              (union (generic-tvariables (car ops) tenv)
                                  tvars))))))
        (loop operands (null))))
      ((base->type _ ) (null))
      (_ (error "this shouldn't happen" type))))

122
(define (union 11 12)
  (cond ((null? 11) 12)
        ((null? 12) 11)
        (((in-tvariable-list? (car 11)) 12) (union (cdr 11) 12))
        (else (cons (car 11) (union (cdr 11) 12))))

; [The following use of MEMQ is a Mini-FX type error and an abstraction violation, but it works & is fast.]

(define in-tvariable-list? memq)

; Instantiate a type schema on a fresh set of type variables.
; [This corresponds to Cardelli's "FreshType".]

(define instantiate-schema
  (lambda (schema)
    (substitute-into-type
     (map (lambda (g) (tvariable->type (new-tvariable (tvariable-id g))))
     (schema-generics schema))
     (schema-type schema)))))

; [The following corresponds to Cardelli's "Fresh"; note the call to prune.]

; Note that this unwinds TYPE out to the leaves when doing the substitution; this guarantees that we don't miss any substitutions because type itself isn't fully unwound.

(define substitute-into-type
  (lambda (types tvars type)
    (let ((type (prune type)))
      (match type
        ((tvariable->type" tvar)
          (letrec ((loop (lambda (ts tvars)
                          (if (null? ts)
                            type
                            (if (same-tvariable? tvar (car tvars))
                              (car ts)
                              (loop (cdr ts) (cdr tvars)))))))
            (loop types tvars)))))
        ((base->type" _) type)
        ((compound->type" c args)
          (compound->type c (map (lambda (arg)
                                      (substitute-into-type types tvars arg))
                                 args)))
        (_ (error "this shouldn't happen" type))))))))
; Type environments.

Environments can be extended in either of two ways:
extend-by-tvariables should be used by lambda and letrec to bind
variables to type variables
extend-by-schemas should be used by let and letrec to bind variables
to type schemas

Once constructed, there are two operations one can perform on a
type environment:
tlookup : tenv * var -> (tvar + schema)
does the usual thing.
generic-tvariable? : tvar * tenv -> bool
returns true iff tvar is not free in the type of any var bound in tenv.

(define-datatype type-environment
  (make-type-env tlookup-proc generic-tvariable?-proc))

(define (tenv-lookup te)
  (match te
    ((make-type-env lookup generic?) lookup)))

(define (tenv-generic? te)
  (match te
    ((make-type-env lookup generic?) generic?)))

(define extend-by-tvariables
  (lambda (outer-tenv vars tvars)
    (extend-tenv
      outer-tenv
      vars
      (map tvar->tvar-or-schema tvars)
      (lambda (tvar)
        ;; tvar is an unconstrained type variable.
        (letrec ((loop (lambda (tvars)
            (if (null? tvars)
              (generic-tvariable? tvar outer-tenv)
              (if (occurs-in-type? tvar
                (tvariable->type (car tvars)))
                ;; (same-tvariable? tvar (car tvars))
                #f
                (loop (cdr tvars)))))))
      (loop tvars)))))

(define extend-by-schemas
  (lambda (outer-tenv vars schemas)
    (extend-tenv outer-tenv
      vars
      (map schema->tvar-or-schema schemas)
      (lambda (tvar)
        (generic-tvariable? tvar outer-tenv))))))
(define extend-tenv ; Students' code should not call this
  (lambda (outer-tenv vars typas generic-tvariable?-proc)
    (make-type-env
      (lambda (var)
        (letrec ((loop (lambda (vars typas)
                          (if (null? vars)
                            (tlookup outer-tenv var)
                            (if (same-variable? var (car vars))
                              (car typas)
                              (loop (cdr vars) (cdr typas)))))))
          (loop vars typas)))
      generic-tvariable?-proc)))

(define empty-type-environment
  (make-type-env
    (lambda (var) (error "unbound variable" var))
    (lambda (tvar) #t)))

(define tlookup
  (lambda (tenv var)
    ((tenv-lookup tenv) var)))

(define same-variable? sym=?)

(define generic-tvariable?
  (lambda (tvar tenv)
    ((tenv-generic? tenv) tvar)))

; Proving the correctness of this implementation of GENERIC-TVARIABLE?
; is tricky.

; A type variable is implemented as a record that contains a ref. The
; global substitution is realized as the collective contents of the
; refs for all type variables.

(define-datatype tvariable
  (make-tvariable id gennum ref) ; id gennum ref
  )

(define tvariable-id
  (lambda (tvar)
    (match tvar
      ((make-tvariable id _ _) id))))

(define tvariable-ref
  (lambda (tvar)
    (match tvar
      ((make-tvariable _ _ r) r))))

(define tvariable-counter (ref 0))

(define reset-tvariable-counter!
  (lambda () (:= tvariable-counter 0)))
(define new-tvariable
  (lambda (id)
    (begin (:= tvariable-counter (+ (~ tvariable-counter) 1))
      (make-tvariable id (~ tvariable-counter) (ref unknown-type)))))

(define (tvariable-binding tvar)
  (~ (tvariable-ref tvar)))

(define extend-substitution!
  (lambda (tvar binding)
    (begin (:= (tvariable-ref tvar) binding)
      #t)))

(define same-tvariable?
  (lambda (tvar1 tvar2)
    (same-ref? (tvariable-ref tvar1) (tvariable-ref tvar2))))

(define unknown-type (unknown->type))

(define tvariable->sym
  (lambda (tvar)
    (match tvar
      ((make-tvariable- id gennum ..)
        (string->sym (string-append (string-append "?" (sym->string id))
          (string-append "=" (int->string gennum))))))))

; Unification
; Has side effects.
; Generates an error if there is no unification.

(define unify!
  (lambda (type1 type2)
    (if (unify!-internal type1 type2)
      the-unit
      (error "type clash" (unparse-type type1) (unparse-type type2))))
(define unify!-internal
  (lambda (types1 types2)
    (let ((types1 (prune types1))
           (types2 (prune types2)))
      ;; Now if a type is a variable, it will be unbound
      (match types1
        ((tvariable->type" v1)
         (match types2
           ((tvariable->type" v2)
            (if (same-tvariable? v1 v2)
                #t
                (extend-substitution! v1 types2)))
           (_
            (if (occurs-in-type? v1 types2)
                $f
                ;Circularity
                (extend-substitution! v1 types2)))))
        ((base->type" c1)
         (match types2
           ((tvariable->type" v2)
            (extend-substitution! v2 types1))
           ((base->type" c2)
            (same-name? c1 c2))
           (_ $f))))
        ((compound->type" con1 args1)
         (match types2
           ((tvariable->type" v2)
            (if (occurs-in-type? v2 types1)
                #f
                (extend-substitution! v2 types1)))
           ((compound->type" con2 args2)
            (if (same-constructor? con1 con2)
                (unify!-list args1 args2)
                $f))
           (_ $f))))))

(define unify!-list
  (lambda (types1 types2)
    (if (null? types1)
        (null? types2)
        (if (null? types2)
            $f
            (if (unify!-internal (car types1) (car types2))
                (unify!-list (cdr types1) (cdr types2))
                $f))))))

; Chase substitutions of tvariables until either a non-tvariable or an
; unbound tvariable is found.
(define prune
  (lambda (type)
    (match type
      ((tvariable->type tvar)
       (match (tvariable-binding tvar)
         ((unknown->type) type)
         (other-type (prune other-type))))
      (_ type))))

; Prevent circular substitutions.

(define occurs-in-type?
  (lambda (tvar type)
    (match (prune type)
      ((tvariable->type tvar2)
       (prune has guaranteed that tvar2 is unbound)
       (same-tvariable? tvar tvar2))
      ((compound->type c args)
       (letrec ((loop (lambda (args)
                       (if (null? args)
                            #f
                            (or (occurs-in-type? tvar (car args))
                                (loop (cdr args)))))))
         (loop args)))
      (_ #f))))

(define the-unit (:= (ref 0) 0))

D \(\mu\)FX/DLX Run-time Implementation

This appendix contains a snapshot as of February 12, 1992 of the 8 source files which implement the runtime system. All of these files are available via FTP.

The files included in this appendix are as follows:

<table>
<thead>
<tr>
<th>Filename</th>
<th>Module</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>runtime/Makefile</td>
<td>Support</td>
<td>Makefile for processing runtime DLX code</td>
</tr>
<tr>
<td>runtime/alloc.s</td>
<td>Runtime</td>
<td>DLX Memory allocation and garbage collection</td>
</tr>
<tr>
<td>runtime/epilog.s</td>
<td>Output</td>
<td>DLX code to follow compiled code</td>
</tr>
<tr>
<td>runtime/frames.s</td>
<td>Runtime</td>
<td>Runtime code for register save and restore</td>
</tr>
<tr>
<td>runtime/lib.s</td>
<td>Runtime</td>
<td>Miscellaneous runtime primitives</td>
</tr>
<tr>
<td>runtime/macros.h</td>
<td>Support</td>
<td>Macros used in prolog.s and epilog.s</td>
</tr>
<tr>
<td>runtime/printf.s</td>
<td>Runtime</td>
<td>DLX code implementing printf utility</td>
</tr>
<tr>
<td>runtime/prolog.s</td>
<td>Output</td>
<td>DLX code to precent compiled code</td>
</tr>
</tbody>
</table>

The index at the end of this document contains entries for procedures, shared variables, and runtime entry points.
D.1 runtime/Makefile

The contents of the file runtime/Makefile:

```plaintext
CPP=cc -E -traditional
#CPP=gcc -E -traditional
#CPP=/lib/cpp -P

all: ../epilog.code ../prolog.code

../epilog.code: macros.h epilog.s alloc.s frames.s printf.s lib.s
 $(CPP) epilog.s \
   | sed -g 's/;.*$$//g' \n   | sed -g 's/ /g' \n   | egrep -v '^[ ]*$$|-$.'|\n, \n  > ../epilog.code

../prolog.code: macros.h prolog.s
 $(CPP) prolog.s \
   | sed -g 's/;.*$$//g' \n   | sed -g 's/ /g' \n   | egrep -v '^[ ]*$$|-$.'|\n, \n  > ../prolog.code
```

D.2 runtime/alloc.s

The contents of the file runtime/alloc.s:

```plaintext
;; alloc.s

;; Here lies the memory allocator and garbage collector.

;; If you have to change it, then be careful, and TEST OFTEN (that
;; means REALLY often), because it's a real pain to debug if it gets
;; broken.

;; LEAVE ALL THIS DATA TOGETHER (for addressability)!
Memory_description:
   .word 0 ; Dummy value

;; Bounds on memory use
_stack_size:
   .word 2048
stack_size:
   .word 2048
```
D.2 runtime/alloc.s

D. µFX/DLX RUN-TIME IMPLEMENTATION

;; Pointers that describe the two semispaces.
_this_semispace:
    .word 0
_this_semispace_end:
    .word 0
_other_semispace:
    .word 0
_other_semispace_end:
    .word 0

;; Statistics
_num_gcs: ; count of gc's performed
    .word 0
_gc_words_copied: ; number of words scanned by all gc's
    .word 0
_total_allocation: ; total $words allocated (ever)
    .word 0
_total_allocs:
    .word 0
_max_stack_size:
    .word 0

;; Flags and misc...
_just_did_a_gc:
    .word 0

;; Address above locations by offsets from Memory_description
#define semispace_size _semispace_size-Memory_description
#define stack_size _stack_size-Memory_description
#define this_semispace _this_semispace-Memory_description
#define this_semispace_end _this_semispace_end-Memory_description
#define other_semispace _other_semispace-Memory_description
#define other_semispace_end _other_semispace_end-Memory_description
#define num_gcs _num_gcs-Memory_description
#define gc_words_copied _gc_words_copied-Memory_description
#define total_allocation _total_allocation-Memory_description
#define total_allocs _total_allocs-Memory_description
#define max_stack_size _max_stack_size-Memory_description
#define just_did_a_gc _just_did_a_gc-Memory_description

;;_init-runtime:
;; Initialize runtime system. Puts startup values in the runtime
;; system's variables and initializes SP, and HP. Trashes ATEMP, RETADR;
;; zeros ARG0--ARG3.
_init-runtime:
    ;; Initialize memory-management registers
    lhi SP,(TOTALMEMSIZE-4)>>16
    ori SP,SP,(TOTALMEMSIZE-4)&0xffff
    lhi HP,(_endprogram+1)>>16 ; tag this pointer.
    ori HP,HP,(_endprogram+1)&0xffff
;; Initialize the semispace descriptions
hi ARG1,(Memory_description)>>16
ori ARG1,ARG1,(Memory_description)&0xffff
lw ARG2,semispace_size(ARG1)
slli ARG3,ARG3,2 ; *4 = byte count
sw this_semispace(ARG1),HP
addu ARG2,HP,ARG3
sw this_semispace_end(ARG1),ARG2
sw other_semispace(ARG1),ARG2
addu ARG2,ARG2,ARG3
sw other_semispace_end(ARG1),ARG2

;; Clear statistics...
sw num_gcs(ARG1),ZERO
sw gc_words_copied(ARG1),ZERO
sw total_allocation(ARG1),ZERO
sw total_allocs(ARG1),ZERO
sw max_stack_size(ARG1),ZERO

or ARG1,ARG1,ZERO
or ARG2,ARG1,ZERO
or ARG3,ARG1,ZERO
jr RETADR
nop

;; ZBLOCK -- zero gc block
;; On entry:
;; ATEMP contains (tagged) pointer to GC block (with size
;; slot filled in).

ZBLOCK:
sw 0(SP),ARG1 ; free-up a temp
sw -4(SP),ARG2
lw ARG1,-1(ATEMP) ; load (tagged) size into ARG2
or ARG2,ZERO,ATEMP
zblock_loop:
beqz ARG1,zblock_done
nop
sw 3(ARG2),ZERO ; zap a word
addui ARG2,ARG2,4 ; inc pointer
j zblock_loop
subui ARG1,ARG1,2 ; dec count
zblock_done:
lw ARG2,-4(SP)
jr RETADR
lw ARG1,0(SP)
D.2 runtime/alloc.s

D. μFX/DLX RUN-TIME IMPLEMENTATION

:: SALLOC -- stack allocation of GC block

:: On entry:
:: ATEMP contains the (tagged) number of words to allocate.

:: On return:
:: ATEMP contains a (tagged) pointer to a new gc block of
:: the requested size. This block has its size slot filled
:: and all data slots set to zero
:: SP decremented to make room for new block
:: All other regs are preserved

.SALLOC:

    .subu    SP, SP, ATEMP
    .subu    SP, SP, ATEMP
    .subui   SP, SP, 4
    .sw      0(SP), ARG1           ; free-up a temp
    .sw      -4(SP), ARG2          ; free-up another temp
    .sw      -8(SP), ARG3          ; free-up another temp

    .lhi     ARG1, (Memory_description)>>16
    .ori     ARG1, ARG1, (Memory_description)&0xffff
    .lw      ARG2, stack_size(ARG1)
    .slli    ARG2, ARG2, 2         ; *4 = byte count
    .lhi     ARG3, (TOTALMEMSIZE-4)>>16
    .ori     ARG3, ARG3, (TOTALMEMSIZE-4)&0xffff
    .subu    ARG2, ARG3, ARG2

    .slt     ARG2, SP, ARG2         ; Stack overflow?
    .bnez    ARG2, stack_overflow
    .nop

    .addui   ARG3, SP, 5            ; Tagged ptr to new block
    .sw      -1(ARG3), ATEMP        ; Save size into size slot

:: Increment statistics

    .srli    ATEMP, ATEMP, 1        ; get untagged wordcount
    .lw      ARG2, total_allocation(ARG1)
    .addu    ARG2, ARG2, ATEMP      ; inc the statistic.
    .addui   ARG2, ARG2, 1          ; (account for size fields too)
    .sw      total_allocation(ARG1), ARG2

    .or      ATEMP, ZERO, ARG3      ; move pointer to ATEMP

    .lw      ARG3, -8(SP)           ; reload the temp registers
    .lw      ARG2, -4(SP)           ; & go zero block
    .j       _ZBLOCK
    .lw      ARG1, 0(SP)
D.2 runtime/alloc.s

sov_msg:
.ascii "Stack overflow. Dying..."
.byte 0x0a, 0x00
.align 2
stack_overflow:
    lhi     ARGO, (printf_closure+1)>>16
    ori     ARGO, ARGO, (printf_closure+1)&0xffff
    lhi     ARG1, (sov_msg)>>16
    ori     ARG1, ARG1, (sov_msg)&0xffff
    lw      ATEMP, 3(ARGO)          ; Jump to printf
    jalr    ATEMP
    nop
    trap    0                        ; Die
    nop

    ;; SFREE -- stack free

SFML:
    lw      ATEMP, -1(ATEMP)          ; Get (tagged) word count
    addu    SP, SP, ATEMP
    addu    SP, SP, ATEMP
    addui   SP, SP, 4                ; (Plus one for size slot).
    lhi     ATEMP, (TOTALMEMSIZE-4)>>16
    ori     ATEMP, ATEMP, (TOTALMEMSIZE-4)&0xffff
    sgt      ATEMP, SP, ATEMP
    bnez    ATEMP, stack_underflow
    nop
    jr       RETADR
    nop

suv_msg:
.ascii "Stack underflow. Dying..."
.byte 0x0a, 0x00
.align 2
stack_underflow:
    lhi     ARGO, (printf_closure+1)>>16
    ori     ARGO, ARGO, (printf_closure+1)&0xffff
    lhi     ARG1, (suv_msg)>>16
    ori     ARG1, ARG1, (suv_msg)&0xffff
    lw      ATEMP, 3(ARGO)          ; Jump to printf
    jalr    ATEMP
    nop
    trap    0                        ; Die
    nop

    ;; _ALLOC -- allocate gc block on heap
    ;; Same as SALLOC except on heap. May initiate a garbage
    ;; collection, which will change the value of pointers into
    ;; the current space.

ALLOC:
    sw      0(SP), ARG1          ; free-up a temp
    sw      -4(SP), ARG2        ; free-up another temp
    sw      -8(SP), ARG3        ; free-up yet another temp

133
lhi ARG1, (Memory_description)>>16
ori ARG1, ARG1, (Memory_description)&0xffff
lw ARG2, this_semspace_end(ARG1) ; get end ptr ready...
or ARG3, ZERO, HP ; get pointer to the new block
addu HP, HP, ATEMP ; inc HP to next free mem
addu HP, HP, ATEMP ; (ATEMP was tagged wordcount...)
addu HP, HP, 4 ; add a word for a size field
sgt ARG2, HP, ARG2 ; is hp over the end?
beqz ARG2, need_no_gc ; if not, we're ok.
nop

;; Uh-oh, we need to do a gc.
lw ARG3, -8(SP) ; reload the temp registers
lw ARG2, -4(SP) ; & go collect (& retry alloc)
j gc
lw ARG1, 0(SP)

need_no_gc:
sw -(1(ARG3), ATEMP) ; stick the size into the new blk

;; Clear recursion-check flag to indicate it worked this time
sw just_did_a_gc(ARG1), ZERO

;; Increment statistics
srli ATEMP, ATEMP, 1 ; change ATEMP to untagged wordcount
lw ARG2, total_allocation(ARG1)
addu ARG2, ARG2, ATEMP ; inc the statistic.
addui ARG2, ARG2, 1 ; (account for size fields too)
sw total_allocation(ARG1), ARG2
lw ARG2, total_allocs(ARG1)
addui ARG2, ARG2, 1
sw total_allocs(ARG1), ARG2
or ATEMP, ZERO, ARG3 ; move pointer to ATEMP
lw ARG3, -8(SP) ; reload the temp registers
lw ARG2, -4(SP) ; & go zero block
j _ZBLOCK
lw ARG1, 0(SP)
;; gc -- garbage collector

;; On entry:
;; HP is garbage.
;; ATEMP (still) contains the (tagged) number of words to allocate.

;; On return:
;; All reachable data has been moved to the (old) other
;; space, and the spaces have been swapped. The root set
;; is all registers except ZERO, HP, SP, ATEMP, and RETADR
;; HP points to usable space in a (new) semispace
;; ATEMP contains a pointer satisfying the alloc request
;; All other regs are preserved modulo forwarding due to copying
;; We return to address in RETADR

gc:

sw 0(SP), ATEMP
sw -4(SP), RETADR
subui SP, SP, 8

;; We'll push a frame onto the stack, making the root set be
;; just FP and VAL (since VAL isn't saved into the frame).
jal _SALLOC ; Allocate frame on stack
ori ATEMP, ZERO, 2*(FrameSize) ; (Tagged) size of frame
sw 3(ATEMP),FP ; Link frame into
or FP,ZERO,ATEMP ; dynamic chain
jal _SAVE
sw 16(FP),ENV ; (Not saved by _SAVE)

lhi ARG1,(Memory_description)>>16
ori ARG1,ARG1,(Memory_description)&0xffff

;; Check to see if we recursed; if so, out of memory...
lw ARG2,just_did_a_gc(ARG1)
bnez ARG2,gc_loop_detected ; EEEEEEEEEEEEEEE EEEEEEEEEEEEEEEEEEE
nop

;; Flip the semispaces
lw ATEMP,this_semispace_end(ARG1)
lw HP,other_semispace_end(ARG1)
sw other_semispace_end(ARG1),ATEMP
sw this_semispace_end(ARG1),HP
lw ATEMP,this_semispace(ARG1)
lw HP,other_semispace(ARG1)
sw other_semispace(ARG1),ATEMP
sw this_semispace(ARG1),HP

;; That cleverly left HP properly initialized. Now scan root set.
jal maybe_copy
or ARG2,ZERO,FP
or FP,ZERO,ATEMP
jal maybe.copy
or ARG2,ZERO,VAL ; VAL isn't in the frame...
or VAL,ZERO,ATEMP

;; At this point, garbage collection is done. Need to update
;; statistics, restore registers and the stack, and retry
;; allocation.

;; mark-up statistics...
lw ARG2,num_gcs(ARG1) ; inc gc count
addui ARG2,ARG2,1
sw num_gcs(ARG1),ARG2

lw ARG3,this_semispace(ARG1) ; get bottom of this space
subu ARG3,HP,ARG3 ; subtract it from current hp
srai ARG3,ARG3,2 ; change bytecount to wordcount
lw ARG2,gc_words_copied(ARG1) ; add this to total GC charge
addu ARG2,ARG2,ARG3
sw gc_words_copied(ARG1),ARG2

;; Set flag so next alloc (called from end of GC) will fail if no mem
sw just_did_a_gc(ARG1),ARG1

;; Restore regs, pop activation frame from dyn chain, free mem it used
jal _RESTORE
sw 15(FP),ENV ; (Not restored by _RESTORE)
or ATEMP, ZERO, FP
lw FP, 3(ATEMP)
jal _SFREE
nop

;; Restore registers we saved right at start of gc
addui SP, SP, 8
lw RETADR, -4(SP)
lw ATEMP, 0(SP)

;; retry the alloc, setting just_did flag so we can detect out-of-mem
j _ALLOC ; go re-try the allocation...
nop
out_mem_msg:
  .ascii "Insufficient memory to process alloc request. Dying..."
  .byte 0x0a, 0x00
  .align 2

gc_loop_detected:
  lhi ARGO, (printf_closure+1)>>16
  ori ARGO,ARGO,(printf_closure+1)&0xffff
  lhi ARG1,(out_mem_msg)>>16
  ori ARG1,ARG1,(out_mem_msg)&0xffff
  lw ATEMP, 3(ARG0) ; Jump to printf
  jalr ATEMP
  nop
trap 0 ; Die
  nop

maybe_copy:
  sw O(SP), RETADR
  subui SP, SP, 4
  or ATEMP,ZERO, ARG2 ; By default, return original value

If not a pointer, then we’re done
andi ARG3,ARG2,1
beqz ARG3, maybe_copy_done

If pointer into this-space, just return it <= it’s scanned already
lw ARG4,this_semispace(ARG1)
sge ARG3,ARG2,ARG4 ; ARG3:= p >= this_semispace
beqz ARG3,check4_semispace_ptr
lw ARG4,this_semispace.end(ARG1)
slt ARG3,ARG2,ARG4 ; ARG3:= p < this_semispace.end
bnez ARG3, maybe_copy_done

nop
;; pointer to otherspace => copy object AND scan object
;; pointer somewhere else => ONLY scan object

check_4_semspace.ptr:
lw ARG4, other_semspace(ARG1)
sge ARG3, ARG2, ARG4 ; ARG3: = p >= other_semspace
beqz ARG3, scan_transitively
lw ARG4, other_semspace_end(ARG1)
slt ARG3, ARG2, ARG4 ; ARG3: = p < other_semspace_end
beqz ARG3, scan_transitively
nop

;; For semispace ptrs, don't chase if it's a forwarded pointer

is_semspace.ptr:
lw ATEMP,-1(ARG2) ; Get size or forwarding ptr
andi ARG4, ATEMP, 1 ; is this forwarding pointer?
bnez ARG4, maybe.copy.done ; yup => just return it.
nop

live: ;; Ok, we've got a live one.
or ARG3, ZERO, ATEMP ; get (tagged) size in words
or ATEMP, ZERO, HP ; quick 'n dirty alloc
addu HP, HP, ARG3 ;
addu HP, HP, ARG3 ; (two times to get byte count)
addui HP, HP, 4 ; (add 4 bytes for size slot)
sw -1(ATEMP), ARG3 ; stash block size (tagged in words)
sw -1(ARG2), ATEMP ; set forwarding address.
jal copy_block ; copy old block to new block
nop

scan_transitively:
;; Here, ATEMP points to a block whose contents need to be
;; "maybe.copied". loop over it, doing the proper thing.
;; ARG2, ARG3 and ARG4 are free.
subui SP, SP, 12 ; save 3 words live data thru recursion
sw 4(SP), ATEMP ; Save original value
or ARG3, ZERO, ATEMP ; block pointer into ARG3
lw ARG4,-1(ARG3) ; (tagged) wordcount in ARG4

scan_transitively_loop:
beqz ARG4, scan_transitively.done ; quit when count is zero
nop
lw ARG2, 3(ARG3) ; get a word
sw 8(SP), ARG3 ; Save registers for recursive call
sw 12(SP), ARG4 ;
jal maybe_copy ; maybe_copy the item
nop
lw ARG4, 12(SP) ; restore regs after recursive call
lw ARG3, 8(SP) ;
sw 3(ARG3), ATEMP ; replace old value
addui ARG3, ARG3, 4 ; inc pointer
j scan_transitively_loop
subui ARG4, ARG4, 2 ; decrement (tagged) count.

scan_transitively.done:
lw ATEMP, 4(SP) ; Restore original value of pointer
addui SP, SP, 12 ; Pop off spaced used during recursion
maybe_copy_done:
    addui SP,SP,4
lw   RETADR,0(SP)
jr   RETADR
nop

;; copy_block -- copy old block to new block.
;; ARG2 points to old block (with forwarding ptr in size slot)
;; ATEMP points to new block (with valid size in size block)
;; ARG2, ARG3 and ARG4 are trashed; all rest (incl. ATEMP) preserved

copy_block:
    lw   ARG3,-1(ATEMP)       ; Get size in words (tagged)
addu ARG3,ARG3,ARG3        ; * 2 = untagged byte count
addu ARG2,ARG2,ARG3        ; point to end of old block
addu ARG3,ATEMP,ARG3       ; point to end of new block

copy_block_loop:
    subu ARG4,ARG3,ATEMP      ; compare ptr to beginning of block
beqz ARG4,copy_block.done  ; done if no more to copy.
nop
lw   ARG4,-1(ARG2)         ; move a word
sw   -1(ARG3),ARG4         ; /
subui ARG2,ARG2,4          ; adjust oldblock pointer
j    copy_block_loop        ; (loop)
subui ARG3,ARG3,4          ; adjust newblock pointer
copy_block.done:
    jr   RETADR              ; ARG2 and ATEMP are what they were.
nop

;; statistics -- print memory statistics.

stat_format:
    .ascii "#gc's=%d words copied by gc=%d words allocated=%d total allocs=%d"
    .byte OxOaO
    .align 2

STATISTICS:
    sw   0(SP), RETADR
    sw   4(SP), ARG0
    sw   8(SP), ARG1
    sw   12(SP), ARG2
    sw   16(SP), ARG3
    sw   20(SP), ARG4
    sw   24(SP), ARG5
    subui SP,SP,28

    lhi ARG0,(printf_closure+1)>>16
    ori ARG0,ARG0,(printf_closure+1)&0xffff

    lhi ARG1,(stat_format)>>16    ; Get format string
    ori ARG1,ARG1,(stat_format)&0xffff ; /

139
D.3 runtime/epilog.s

The contents of the file runtime/epilog.s:

```assembly
lhi ATEMP, (Memory_description)>>16
ori ATEMP, ATEMP, (Memory_description)&0xffff
lw ARG2, num_gcs(ATEMP) ; get count of GC's performed
slli ARG2, ARG2, 1 ; tag the value for PRINTF
lw ARG3, gc_words_copied(ATEMP) ; get GC work estimate
slli ARG3, ARG3, 1 ; tag the value for PRINTF
lw ARG4, total_allocation(ATEMP) ; get total amt allocated
slli ARG4, ARG4, 1 ; tag the value for PRINTF
lw ARG5, total_allocs(ATEMP) ; get total calls to alloc
slli ARG5, ARG5, 1 ; tag the value for PRINTF

;; Do call to printf
lw ATEMP, 3(ARG0) ; Jump to printf
jalr ATEMP
nop

addui SP, SP, 28
lw ARG5, 24(SP)
lw ARG4, 20(SP)
lw ARG3, 16(SP)
lw ARG2, 12(SP)
lw ARG1, 8(SP)
lw ARG0, 4(SP)
lw RETADR, 0(SP)
jr RETADR
nop
```

D.3 runtime/epilog.s

The contents of the file runtime/epilog.s:

```assembly
;; COMPILLED CODE ENDS HERE ..........................................

;; epilog.s (in comp/backend/runtime)

Epilog: What follows makes-up all run-time routines used by the microFX system. We let the DLX simulator do any necessary linking, and if more code is included than is needed, so be it. See macros.h for register definitions.

prolog.s contains initialization code.

#include "macros.h" /* registernames macros, etc */
```
startl_closure: ; Static closure for printf routine
    .word 4
    .word START.i
    .word 0

printf_closure: ; Static closure for printf routine
    .word 4
    .word PRINTF
    .word 0

stats_closure: ; Static closure for statistics printing routine
    .word 4
    .word STATISTICS
    .word 0

#include "alloc.s"
#include "frames.s"
#include "printf.s"
#include "lib.s"

; _endprogram
; Does nothing, just marks the end of the program (and hence the
; beginning of the heap at initialization time). See alloc.s
; to see how this is used.
;
_endprogram:

D.4 runtime/frames.s

The contents of the file runtime/frames.s:

;; frames.s
.SAVE -- save registers into a frame

try:

FP contains a (tagged) pointer to the frame.

On exit:

ITEM and RETADR trashed
r6--r29 are saved into frame according to convention
All other regs are preserved

.SAVE:

sw 111(FP),r29
sw 107(FP),r28
sw 103(FP),r27
sw 99(FP),r26
sw 95(FP),r25
sw 91(FP),r24
sw 87(FP),r23
sw 83(FP),r22
sw 79(FP),r21
sw 75(FP),r20
sw 71(FP),r19
sw 67(FP),r18
sw 63(FP),r17
sw 59(FP),r16
sw 55(FP),r15
sw 51(FP),r14
sw 47(FP),r13
sw 43(FP),r12
sw 39(FP),r11
sw 35(FP),r10
sw 31(FP),r9
sw 27(FP),r8
sw 23(FP),r7
sw 19(FP),r6
jr RETADR
nop
D.5 runtime/lib.s

The contents of the file runtime/lib.s:

```assembly
; ; _RESTORE -- restore regs from a frame.
; ; On entry:
; ; FP points to the frame
; ;
; ; On exit:
; ; r6--r29 are restored from frame
; ; ATEMP and RETADR may be trashed
;
; _RESTORE:
    lw     r29,111(FP)
    lw     r28,107(FP)
    lw     r27,103(FP)
    lw     r26, 99(FP)
    lw     r25, 95(FP)
    lw     r24, 91(FP)
    lw     r23, 87(FP)
    lw     r22, 83(FP)
    lw     r21, 79(FP)
    lw     r20, 75(FP)
    lw     r19, 71(FP)
    lw     r18, 67(FP)
    lw     r17, 63(FP)
    lw     r16, 59(FP)
    lw     r15, 55(FP)
    lw     r14, 51(FP)
    lw     r13, 47(FP)
    lw     r12, 43(FP)
    lw     r11, 39(FP)
    lw     r10, 35(FP)
    lw     r 9, 31(FP)
    lw     r 8, 27(FP)
    lw     r 7, 23(FP)
    lw     r 6, 19(FP)
    jr     RETADR
    nop
```

D.5 runtime/lib.s

The contents of the file runtime/lib.s:

```assembly
;; Print (tagged) character in ARGO
;; callable from micro-FX [has type (-> (char) unit)]

PUTCHAR:
    lw     RETADR, 11(FP)
    sub    VAL, VAL, VAL ; Return value zero
    jr     .TPUTCHAR
    nop
```
;; _PUTCHAR -- put a character using dlx system call
;; On entry:
;; ATEMP is (untagged) character to print
;; On exit:
;; Stack used but SP preserved
;; ATEMP trashed

putchar_format_string:
   .asciiz "%c"
   .align 2

putchar_format:
   .word   putchar_format_string
   .space  4

_TPUTCHAR: ;; Print (tagged) character in ATEMP.
srai ATEMP, ATEMP, 1
   ;; untag the char

_PUTCHAR: ;; Print (untagged) character in ATEMP.
sw   0(SP), r14
   ;; Save old value of r14
subui SP, SP, 4
lhi r14, (putchar_format>>16)
   ;; Pointer to args in r14
ori r14, r14, (putchar_format&0xffff)
sw   4(r14), ATEMP
trap 5
   ;; Call print built-in
nop
addui SP, SP, 4
lw   r14, 0(SP)
   ;; restore r14
jr   RETADR
   ;; Return to caller

;; _SYM2STRING -- implements low-level part of sym->string
;; On entry:
;; ATEMP points to uFX symbol
;;
;; On exit:
;; Stack used but SP preserved
;; ATEMP points to uFX string

_SYM2STRING:
sw   0(SP), ARGO
   ;; free-up some temps
sw   -4(SP), ARG1
   ;; /
sw   -8(SP), ARG2
   ;; /
sw   -12(SP), RETADR
   ;; /
subui SP, SP, 16
or   ARG2, ATEMP, ZERO
   ;; Save pointer to symbol text
subi ARG1, ZERO, 1
   ;; Initialize length counter

len_loop:
lbu ARG0, 0(ATEMP)
   ;; Get next character in symbol
addi ARG1, ARG1, 1
   ;; Inc length counter
addui ATEMP, ATEMP, 1
   ;; Increment symbol pointer
bnez ARG0, len_loop
   ;; nxt char is not '0', loop again
nop

144
The contents of the file runtime/macros.h:

/**
 macros.h: macros for easing the writing of prolog and epilog.
 do /lib/cpp -P prolog.s > prolog.code
     /lib/cpp -P epilog.s > epilog.code
 to use (done in the Makefile).
*/

#define ZERO r0
#define VAL r1
#define ENV r2
#define FP r3
#define SP r4
#define HP r5
#define ARG0 r6
#define ARG1 r7
#define ARG2 r8
#define ARG3 r9
#define ARG4 r10
#define ARG5 r11
#define ARG6 r12
#define ARG7 r13
#define ARG8 r14
D.7 runtime/printf.s

The contents of the file runtime/printf.s:

```c
#define ATEMP r30
#define RETADR r31
#define FrameSize 28 /* (untagged) # words in a frame. */
#define TOTALMEMSIZE 32768

D.7 runtime/printf.s

The contents of the file runtime/printf.s:

;; printf.s (in comp/backend/runtime)

;; PRINTF -- A primitive printf
;; On entry:
;; ARG1 points to a DLX string (uFX_symbol) with grammar %x
;; where x is:
;;   x ::= %   a '%' sign
;;   d   (decimal) integer
;;   b   boolean
;;   c   char
;;   s   string (null-terminated)
;;   F   function
;;   pxx pairs (where x is recursive format)
;;   lx  lists (thus a list of lists of ints is printed by %lld)
;;   rx  refs
;;   vx  vectors
;; ARG2--ARG8 are rest of printf arguments (up to 6)
;; On exit:
;; Uses regular uFX calling conventions, i.e., almost nothing saved.
;; Returns the-unit.

PRINTF:
    nop
printf.loop:
    lb VAL,O(ARG1)
    beqz VAL,end_printf
    nop
    seqi r29,VAL,0x25 ; '
    beqz r29,no_escape
    nop

;; got a %, see what the esc'd char is...
    addui ARG1,ARG1,1
    lbu VAL,O(ARG1)
    beqz VAL,end_printf
    nop
```
seqi r29,VAL,0x62 ; 'b'
bnez r29,bool_out
nop

seqi r29,VAL,0x63 ; 'c'
bnez r29,char_out
nop

seqi r29,VAL,0x64 ; 'd'
bnez r29,dec_out
nop

seqi r29,VAL,0x70 ; 'p'
bnez r29,pair_out
nop

seqi r29,VAL,0x6c ; 'l'
bnez r29,list_out
nop

seqi r29,VAL,0x76 ; 'v'
bnez r29,vec_out
nop

seqi r29,VAL,0x72 ; 'r'
bnez r29,ref_out
nop

seqi r29,VAL,0x73 ; 's'
bnez r29,sym_out
nop

;; Otherwise, just print the escaped character
no_escape:
jal _PUTCHAR ; Print the character
or ATEMP, VAL, ZERO
j printf_loop
addui ARG1,ARG1,1

end_printf:
lw RETADR, 11(FP)
jr RETADR
or VAL, ZERO, ZERO

shiftem:
or ARG2,ZERO,ARG3
or ARG3,ZERO,ARG4
or ARG4,ZERO,ARG5
or ARG5,ZERO,ARG6
or ARG6,ZERO,ARG7
jr RETADR
or ARG7,ZERO,ARG8
To print a char, untag it and use _PUTCHAR

```assembly
char_out:
jal  shiftem          ; Shift arguments down
srai ATEMP, ARG2, 1    ; Untag char to be printed (in D-slot!)
jal  _PUTCHAR
nop
j    printf_loop
addui ARG1, ARG1, 1
```

Use DLX’s printf with %d format

```assembly
dec_out:
lhi  r14,(dec_format)>>16
ori  r14,r14,(dec_format)&0xffff
srai VAL, ARG2, 1
sw   4(r14), VAL
trap 5
nop
jal  shiftem          ; shift the args around...
j   printf_loop
addui ARG1, ARG1, 1
```

```assembly
dec_format_string:
    .asciiz "%d"
    .align 2
dec_format:
    .word dec_format_string
    .space 4
```

Use _PUTCHAR to print a # then a T or F

```assembly
bool_out:
jal  _PUTCHAR          ; Print '
ori  ATEMP, ZERO, 0x23
sne  VAL, ARG2, ZERO
ori  ATEMP, ZERO, 14
sl   ATEMP, ATEMP, VAL
addi ATEMP, ATEMP, 56    ; ATEMP = (VAL ? 'T' : 'F')
jal  _PUTCHAR
nop
jal  shiftem
nop
j    printf_loop
addui ARG1, ARG1, 1
```
;; Use DLX printf here.
sym_out:
    lhi    r14,(string_format)\>>16
    ori    r14,r14,(string_format)&0xffff
    sw     4(r14),ARG2
    trap   5
    nop
    jal    shiftem   ; shift the args around...
    nop
    j      printf_loop
    addui  ARG1,ARG1,1
string_format_string:
    .asciiz "%s"
    .align 2
string_format:
    .word  string_format_string
    .space 4
;; build_format -- build format for recursive call to printf
;; On entry:
;; ARGI points to first character of the old format
;; (excluding % sign). This char is assumed to be one of
;; the compound types (l, v, p, r).
;; On exit:
;; ATEMP points to new format (allocated on stack).
;; ARGI points to last character of old format
;; r14, r15, r16, r17, r18 trashed

build_format:
    ; first calculate length of new format string (minus the % char)
or r14, ARG1, ZERO
ori ATEMP, ZERO, 0
ori r18, ZERO, 0
bf_deeper:
    addi r18, r18, 1 ; Counts nested p's
bf_len_loop:
    addui r14, r14, 1
    lbu r15, 0(r14)
    addi ATEMP, ATEMP, 1
    sequi r16, r15, 0x70 ; 'p'
    bnez r16, bf_deeper
    sequi r16, r15, 0x6c ; 'l'
    bnez r16, bf_len_loop
    sequi r17, r15, 0x72 ; 'r'
    bnez r17, bf_len_loop
    sequi r16, r15, 0x76 ; 'v'
    bnez r16, bf_len_loop
    nop
    subi r18, r18, 1
    bnez r18, bf_len_loop
    nop
bf_len_done: ; length in ATEMP
    ; Now allocate room on stack for the new format string and the length
    ; of the string
    addi r14, ATEMP, 8 ; Include room for '%' & a length
    andi r14, r14, 0xffffc ; Align length to word boundary
    subu SP, SP, r14
    sw 4(SP), r14 ; Save length to make popping easy
    ; Create new format string by appending a '%' to thing passed in
    addui r14, SP, 8 ; Ptr to base of new string
    ori r15, ZERO, 0x25
    sb 0(r14), r15 ; Store a '%'
bf_cp_loop:
    addui ARG1, ARG1, 1
    lbu r15, 0(ARG1)
    addui r14, r14, 1
    subi ATEMP, ATEMP, 1
    sb 0(r14), r15
    bnez ATEMP, bf_cp_loop
    nop
bf_done:
    addui r14, r14, 1 ; Terminate format string
    sb 0(r14), ZERO
    jr RETADR
    addui ATEMP, SP, 8 ; Return pointer to new format string
.pf_recurse:
    j PRINTF
    sw 11(FP), RETADR

;; Call PRINTF recursively to print the thing pointed to
pair_out:
lhi r14,(pair_hdr)>>16 ; Print "(pair 
ori r14,r14,(pair_hdr)&0xffff
trap 5
nop

;; Set up stack-frame for recursive call
jal _SALLOC
ori ATEMP,ZERO,(FrameSize*2) ; Old static chain
sw 3(ATEMP), FP
or FP,ZERO,ATEMP
addi ATEMP, SP, (FrameSize+1)*4 ; Old SP
sw 7(FP), ATEMP /

;; Print thing on left
jal build_format ; Returns ptr to new string in ATEMP
nop ; & ARG1 pting 2 lst chr in old format
or r29, ZERO, ATEMP ; Stash this away for a second
jal _SAVE ; Save reg's during recursive call
nop
or ARG1,ZERO,r29 ; Save new format string
lw ARG2,3(ARG2) ; Pass item to print in ARG2
jal _pf.recurse ; Recursive call to printf
nop
jal _PUTCHAR ; Put space between items
ori ATEMP,ZERO,0x20
jal _RESTORE ; Get back our old registers
nop

;; Now do right hand thing
jal build_format ; Returns ptr to new string in ATEMP
nop ; & ARG1 pting 2 lst chr in old format
or r29, ZERO, ATEMP ; Stash this away for a second
jal _SAVE ; Save reg's during recursive call
nop
or ARG1,ZERO,r29 ; Save new format string
lw ARG2,7(ARG2) ; Pass item to print in ARG2
jal _pf.recurse ; Call PRINTF recursively
nop
jal _PUTCHAR ; Put closing ")"
ori ATEMP,ZERO,0x20
jal _RESTORE
nop
D.7 runtime/printf.s

```assembly
 ;; Finish up
 lw   SP,7(FP)          ; Get rid of activation frame.
 lw   FP,3(FP)          ; /
 jal  shiftem           ; Shift args down
 nop
 j    printf_loop
 addui ARG1, ARG1, 1

pair_hdr_string:
   .asciiz "pair ",
   .align 2
pair_hdr:
   .word  pair_hdr_string
   .space 4

;; CallPRINTF recursively to print the list items.
list_out:
 jal  _PUTCHAR          ; Put a '('
 ori  ATEMP, ZERO, 0x28

;; Set up stack-frame for recursive call
 jal  _SALLOC
 ori  ATEMP, ZERO, (FrameSize*2)
 sw   3(ATEMP), FP      ; Old static chain
 or   FP,ZERO,ATEMP
 addi ATEMP, SP, (FrameSize+1)*4  ; Old SP
 sw   7(FP), ATEMP     ; /

 jal  build_format      ; Returns ptr to new string in ATEMP
 or   r29,ZERO,ATEMP   ; & ARG1 pting 2 lst chr in old format
 beqz ARG2,list_done   ; If list is empty, skip loop
 nop
 j    list_nospace      ; skip space in front of first item

```

152
 xuânrace: __PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print a space in front of item
list.nospace:
   jal  _SAVE
   nop
   or  ATEMP, ZERO, 0x20
   ; (in D-slot!)
   lw  ATEMP, ZERO, 0x20
   ; Pass format string in ARG1
   lw  ARG2, 3(ARG2)
   ; Pass item to print in ARG2
   jal  _pf.recurse
   nop
   jal  _RESTORE
   nop
   lw  ARG2, 7(ARG2)
   bnez  ARG2, list.loop
   ; do again if more items in list
list.done:
   lw  SP,7(FP)
   ; Get rid of activation frame.
   lw  FP,3(FP)
   ; /jal .pf.printf
   lw  ARG2, 7(ARG2)
   bnez  ARG2, list.loop
   ; do again if more items in list
vec.out:
   jai  _PUTCHAR
   ori  ATEMP, ZERO, 0x23
   ; Put a '
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x28
   ; Put a ')' 
   ori  ATEMP, ZERO, 0x29
   ; Shift args down
   nop
   j  printf_loop
vec_done:
   addi  ARGI, ARGI, 1
   ; Call PRINTF recursively to print the vec items.
vec_done:
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print a space in front of item
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jl  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCHAR
   ori  ATEMP, ZERO, 0x20
   ; Print closing ')'
   jal  _PUTCH
vec_loop:
    jal _PUTCHAR ; Print a space in front of item
    ori ATEMP, ZERO, 0x20 ; (in D-slot!)
vec_nospace:
    jal _SAVE
vec_return:
    jal _RESTORE
vec_done:
    lw SP,7(FP) ; Get rid of activation frame.
    lw FP,3(FP) ; /
    jal _PUTCHAR ; Print closing ')
vec_ione:
    lw SP,7(FP) ; Get rid of activation frame.
    lw FP,3(FP) ; /
    jal _PUTCHAR ; Print closing ')
vec_ione:
    lw SP,7(FP) ; Get rid of activation frame.
    lw FP,3(FP) ; /
    jal _PUTCHAR ; Print closing ')
vec_ione:
    lw SP,7(FP) ; Get rid of activation frame.
    lw FP,3(FP) ; /
    jal _PUTCHAR ; Print closing ')

ref_out:
    lhi r14, (ref.hdr)>>16 ; Print "(ref "); Set up stack-frame for recursive call
    ori r14, r14, (ref.hdr)&0xffff
    trap 5
ref_out:
    lhi r14, (ref.hdr)>>16 ; Print "(ref "); Set up stack-frame for recursive call
    ori r14, r14, (ref.hdr)&0xffff
    trap 5
ref_out:
    lhi r14, (ref.hdr)>>16 ; Print "(ref "); Set up stack-frame for recursive call
    ori r14, r14, (ref.hdr)&0xffff
    trap 5
ref_out:
    lhi r14, (ref.hdr)>>16 ; Print "(ref "); Set up stack-frame for recursive call
    ori r14, r14, (ref.hdr)&0xffff
    trap 5
ref_out:
    lhi r14, (ref.hdr)>>16 ; Print "(ref "); Set up stack-frame for recursive call
    ori r14, r14, (ref.hdr)&0xffff
    trap 5
D.8 runtime/prolog.s

The contents of the file runtime/prolog.s:

```assembly
lw     SP,7(FP)     ; Get rid of activation frame.
lw     FP,3(FP)     ; /
jal    _RESTORE
nop
jal    _PUTCHAR     ; Put closing ')
ori    ATEMP, ZERO, 0x29
jal    shiftem     ; Shift args down
nop
j      printf_loop
addui  ARG1, ARG1, 1
ref_hdr_string:
    .asciiz "(ref "
    .align 2
ref_hdr:
    .word  ref_hdr_string
    .align 2
```

```plaintext
... prolog.s (in comp/backend/runtime)
;
; prolog.s: microFX program prolog. This gets prefixed to every compiled file.
;
#include "macros.h"     /**< registername macros, etc */

;; Execution starts right here!
.global PRINTF
.global STATISTICS
;; - is needed due to dlxsim bug :-(
```
program1:

```
;; zero regs so that gc stays happy (all reg's have tagged values)
or   r1, ZERO, ZERO
or   r2, ZERO, ZERO
or   r3, ZERO, ZERO
or   r4, ZERO, ZERO
or   r5, ZERO, ZERO
or   r6, ZERO, ZERO
or   r7, ZERO, ZERO
or   r8, ZERO, ZERO
or   r9, ZERO, ZERO
or  r10, ZERO, ZERO
or  r11, ZERO, ZERO
or  r12, ZERO, ZERO
or  r13, ZERO, ZERO
or  r14, ZERO, ZERO
or  r15, ZERO, ZERO
or  r16, ZERO, ZERO
or  r17, ZERO, ZERO
or  r18, ZERO, ZERO
or  r19, ZERO, ZERO
or  r20, ZERO, ZERO
or  r21, ZERO, ZERO
or  r22, ZERO, ZERO
or  r23, ZERO, ZERO
or  r24, ZERO, ZERO
or  r25, ZERO, ZERO
or  r26, ZERO, ZERO
or  r27, ZERO, ZERO
or  r28, ZERO, ZERO
or  r29, ZERO, ZERO
or  r30, ZERO, ZERO

jal _init_runtime ; Init runtime sys
nop

;; Set-up an initial frame to return through
ori ATEMP, ZERO, FrameSize
jal _ALLOC
nop
or   FP, ATEMP, ZERO ; FP points to frame
sw  3(FP), ZERO
sw  7(FP), SP
lhi ATEMP,(print_res) >> 16 ; place to return to
ori ATEMP, ATEMP, (print_res) & 0xffff ; /
sw 11(FP), ATEMP ; /
sw 15(FP), ZERO
jal _SAVE
nop
```
:: jump to START_1 through statically-created closure.
li ARGO,(start1_closure+1)>>16 ; Get closure to printf
ori ARGO,ARGO,(start1_closure+1)&0xffff ; /
lw ATEMP, 3(ARGO) ; Jump to START_1
jr ATEMP
nop

print_res:
:: Print result by calling printf. Recall that the compiler
gave us a string "RESULT_FORMAT" which is a printf format
for the type of the result. (Reuses activation from from
above.)
li ATEMP,(print_stat)>>16 ; place for printf to return to
ori ATEMP,ATEMP,(print_stat)&0xffff ; /
sw 11(FP),ATEMP ; /
li ARGO,(printf_closure+1)>>16 ; Get closure to printf
ori ARGO,ARGO,(printf_closure+1)&0xffff ; /
lhi ARG1,(RESULT_FORMAT)>>16 ; put the format string in ARG1
ori ARG1,ARG1,(RESULT_FORMAT)&0xffff ; /
or ARG2,ZERO,VAL ; put the returned val in ARG2
lw ATEMP, 3(ARGO) ; Jump to printf
jr ATEMP ; /

print_stat:
:: Print a couple of new-lines
ori ATEMP, ZERO, 10
jal _PUTCHAR
nop
ori ATEMP, ZERO, 10
jal _PUTCHAR
nop

:: Call STATISTICS to print stat's
li ATEMP,(..EXIT)>>16 ; place for stats to return to
ori ATEMP,ATEMP,(..EXIT)&0xffff ; /
sw 11(FP),ATEMP ; /
lhi ARGO,(stats_closure+1)>>16 ; Get closure to stats printing
ori ARGO,ARGO,(stats_closure+1)&0xffff
lw ATEMP, 3(ARGO) ; Jump to statistics
jr ATEMP ; /
nop

.EXIT:
trap 0 ; "exit" sys call
nop

;; COMPILED CODE STARTS HERE........................................
References


Index

| *break-points*, 40 | allocate-raw-block-of-memory, 109 |
| *end-program*, 40 | allocate-reg, 75 |
| *entire-memory-size*, 40 | allocate-specific-reg, 75 |
| *fp-cond*, 40 | alu-neq, 32 |
| *free-reg-stack*, 75 | alu-or, 33 |
| *free-regs*, 75 | alu-s-left-1, 33 |
| *freg*, 40 | alu-s-right-a, 33 |
| *gc-words-copied*, 40 | alu-s-right-1, 33 |
| *halt-emulate?*, 40 | alu-xor, 33 |
| *instruction-count*, 40 | ARG0, 74, 145 |
| *label-table*, 40 | ARG1, 74, 145 |
| *max-stack-size*, 40 | ARG2, 74, 145 |
| *mem*, 40 | ARG3, 74, 145 |
| *noisy-gc*, 40 | ARG4, 74, 145 |
| *npc*, 40 | ARG5, 74, 145 |
| *num-allocatable*, 75 | ARG6, 74, 145 |
| *num-gcs*, 40 | ARG7, 74, 145 |
| *other-semispace*, 40 | ARG8, 74, 145 |
| *other-semispace-end*, 40 | ARG9, 74 |
| *pc*, 40 | arrow-constructor, 50 |
| *program-start*, 40 | arrow-returns, 50 |
| *reg*, 40 | arrow-takes, 50 |
| *rr-pretty*, 40 | asm, 34, 86 |
| *semispace-size*, 40 | asm-b1, 29 |
| *silent-flag*, 113 | asm-b2, 29 |
| *stack-size*, 40 | asm-cnv, 31 |
| *this-semispace*, 40 | asm-fff, 30 |
| *this-semispace-end*, 40 | asm-fp-rel, 31 |
| *total-allocation*, 40 | asm-ggg, 31 |
| *total-allocs*, 40 | asm-iiic, 28 |
| *verbose-flag*, 113 | asm-iii, 28 |
| 1st, 92 | asm-lhi, 30 |
| 2nd, 92 | asm-load, 29 |
| 3rd, 92 | asm-pass1, 43 |
| 4th, 92 | asm-pass2, 43 |
| add-prim-closure, 80 | asm-store, 30 |
| add-sym, 55 | assign-prim, 81 |
| all-keywords, 99 | ATEMP, 74, 146 |
| allocatable-reg?, 75 | av-kill-whitespace, 48 |
| allocate-block-of-memory, 109 | av-next-tok, 48 |
|                        | av-parse-expr, 47 |
|                        | av-parse-number, 48 |
av-punct?, 49
av-sym?, 49
av-tokenize, 48
avpe-recurse, 47
bf_cp_loop, 150
bf_deeper, 150
bf_done, 150
bf_len_done, 150
bf_len_loop, 150
bignum2bits, 38
bignum2mint, 38
binding, 79
block, 13
boolean-type, 50
bool_out, 148
build_format, 150
c-t-add-prims, 79
c-t-bind, 79
c-t-lookup, 79
calling convention, 15
char->string, 93
caracter-type, 50
char_out, 148
check, 114
check_4_semspace_ptr, 138
closure elimination, 18
cnst, 86
compute-schema, 122
copy-block, 111
copy-input-stream-to-output-stream, 93
copy_block, 139
copy_block_done, 139
copy_block_loop, 139
cvt-prim, 81
de-itag, 73
de-otag, 74
deleallocate-reg, 75
dec_format, 148
dec_format_string, 148
dec_out, 148
define-keyword, 99
definition, 49
definition-name, 50
definition-value, 50
design goals, 5
disasm, 46
display-icode, 58
display-icode-list, 118
display-ocode-list, 119
display-one-instruction, 46
displays, 18
do-listing, 118
done-emulating, 107
down-string, 93
down-sym, 93
dump, 45
electronic distribution, 18
emit, 60
emit-delayed-ic, 52
emit-div, 72
emit-error, 60
emit-ic-later, 52
emit-lw, 60
emit-mul, 72
emit-remainder, 73
emit-sw, 60
empty-c-t-env, 79
empty-type-environment, 125
emulate, 42
emulate-one-instruction, 44
end_printf, 147
enter-label, 43
enter-library, 53
enter-system-routine-labels, 107
ENV, 74, 145
environments, 15
eval-immed, 47
exp, 10, 49
experiments, 18
expression-type, 49
extend-by-schemas, 124
extend-by-tvariables, 124
extend-substitution, 126
extend-tenv, 125
extract-list, 95
extract-string, 95
extract-symbol, 95
extract-value, 94
<table>
<thead>
<tr>
<th>Extract from document</th>
</tr>
</thead>
<tbody>
<tr>
<td>extract-vec, 95</td>
</tr>
<tr>
<td>fill-done, 145</td>
</tr>
<tr>
<td>fill-loop, 145</td>
</tr>
<tr>
<td>find-insn, 45</td>
</tr>
<tr>
<td>for-each, 92</td>
</tr>
<tr>
<td>for-each-2, 92</td>
</tr>
<tr>
<td>FP, 74, 145</td>
</tr>
<tr>
<td>FP0, 74</td>
</tr>
<tr>
<td>FP1, 74</td>
</tr>
<tr>
<td>FP2, 74</td>
</tr>
<tr>
<td>FP3, 74</td>
</tr>
<tr>
<td>FP4, 74</td>
</tr>
<tr>
<td>FP5, 74</td>
</tr>
<tr>
<td>FrameSize, 74, 146</td>
</tr>
<tr>
<td>gen-alloc, 66</td>
</tr>
<tr>
<td>gen-alloc-block, 67</td>
</tr>
<tr>
<td>gen-arguments, 64</td>
</tr>
<tr>
<td>gen-assign, 68</td>
</tr>
<tr>
<td>gen-begin, 66</td>
</tr>
<tr>
<td>gen-begin-activation, 67</td>
</tr>
<tr>
<td>gen-body, 63</td>
</tr>
<tr>
<td>gen-call, 63</td>
</tr>
<tr>
<td>gen-end-activation, 68</td>
</tr>
<tr>
<td>gen-fill-block, 67</td>
</tr>
<tr>
<td>gen-if, 66</td>
</tr>
<tr>
<td>gen-intref, 64</td>
</tr>
<tr>
<td>gen-jump, 64</td>
</tr>
<tr>
<td>gen-labdef, 62</td>
</tr>
<tr>
<td>gen-labref, 65</td>
</tr>
<tr>
<td>gen-letrec, 63</td>
</tr>
<tr>
<td>gen-load, 69</td>
</tr>
<tr>
<td>gen-not, 68</td>
</tr>
<tr>
<td>gen-op, 72</td>
</tr>
<tr>
<td>gen-put-char, 71</td>
</tr>
<tr>
<td>gen-return, 62</td>
</tr>
<tr>
<td>gen-string, 62</td>
</tr>
<tr>
<td>gen-sym2string, 71</td>
</tr>
<tr>
<td>gen-symbol, 62</td>
</tr>
<tr>
<td>gen-varref, 65</td>
</tr>
<tr>
<td>gen-vec-alloc, 70</td>
</tr>
<tr>
<td>gen-vec-length, 70</td>
</tr>
<tr>
<td>gen-vec-set!, 70</td>
</tr>
<tr>
<td>gen-word, 62</td>
</tr>
<tr>
<td>gencode, 61</td>
</tr>
<tr>
<td>generate-icode, 52</td>
</tr>
<tr>
<td>generate-ocode, 61</td>
</tr>
<tr>
<td>generate-vector, 93</td>
</tr>
<tr>
<td>generic-tvariable?, 125</td>
</tr>
<tr>
<td>generic-tvariable-list, 122</td>
</tr>
<tr>
<td>get-bit-field, 38</td>
</tr>
<tr>
<td>get-dreg, 45</td>
</tr>
<tr>
<td>get-freg, 45</td>
</tr>
<tr>
<td>get-mem, 44</td>
</tr>
<tr>
<td>get-parser-for-keyword, 99</td>
</tr>
<tr>
<td>get-reg, 45</td>
</tr>
<tr>
<td>get-slot, 45</td>
</tr>
<tr>
<td>goto, 44</td>
</tr>
<tr>
<td>h2i, 39</td>
</tr>
<tr>
<td>highest-used-reg, 75</td>
</tr>
<tr>
<td>history, 2</td>
</tr>
<tr>
<td>HP, 74, 145</td>
</tr>
<tr>
<td>i2h, 39</td>
</tr>
<tr>
<td>ic, 118</td>
</tr>
<tr>
<td>icode, 11, 76</td>
</tr>
<tr>
<td>icode-list, 51</td>
</tr>
<tr>
<td>icode-to-be-emitted, 52</td>
</tr>
<tr>
<td>ictype, 77</td>
</tr>
<tr>
<td>id, 91</td>
</tr>
<tr>
<td>in-otherspace?, 110</td>
</tr>
<tr>
<td>in-stack?, 110</td>
</tr>
<tr>
<td>in-thisspace?, 110</td>
</tr>
<tr>
<td>in-tvariable-list?, 123</td>
</tr>
<tr>
<td>in0, 85</td>
</tr>
<tr>
<td>in1, 85</td>
</tr>
<tr>
<td>in2, 86</td>
</tr>
<tr>
<td>in3, 86</td>
</tr>
<tr>
<td>init-emulator, 40</td>
</tr>
<tr>
<td>init-reg-allocator, 75</td>
</tr>
<tr>
<td>inline allocation, 18</td>
</tr>
<tr>
<td>insn, 96</td>
</tr>
<tr>
<td>instantiate-schema, 123</td>
</tr>
<tr>
<td>int16?, 38, 74</td>
</tr>
<tr>
<td>int32?, 38</td>
</tr>
<tr>
<td>integer-type, 50</td>
</tr>
</tbody>
</table>
integers-between, 91
intermediate forms, 10
interpreter-gc, 109
is-arrow-type?, 50
is_semispaces_ptr, 138
itag, 73
ltest-compile, 114
keyword-table, 99

l-char-alphabetic?, 81
l-char-downcase, 82
l-char-lowercase?, 82
l-char-numeric?, 82
l-char-uppercase, 82
l-char-upper-case?, 82
l-char-whitespace?, 82
l-length, 82
l-list->string, 83
l-list->vector, 84
l-make-vector, 83
l-put-string, 85
l-reverse, 82
l-string->list, 83
l-string-append, 83
l-unparse-bool, 84
l-unparse-char, 84
l-unparse-int, 84
l-unparse-list, 85
l-unparse-pair, 85
l-unparse-string, 84
l-unparse-symbol, 84
l-unparse-unit, 84
l-unparse-vector, 85
l-vector->list, 83
l-vector-length, 83
l-vector-ref, 83
l-vector-set!, 83
label-counter, 58
label2num, 43
leave-library, 53
len_loop, 144
lib, 86
lib-init, 78
library-count, 53
library-icode, 53
list2env, 113
listing, 118
list_done, 153
list-loop, 153
list_nospace, 153
list_out, 152
live, 138
live registers, 15, 18
localref, 65
lookup, 112
lsw, 74
machine, 6
make-arrow-type, 50
make-binop-gen, 72
make-binop-gen2, 72
make-prim-closure, 80
makefile, 129
map-string, 92
max, 91
maybe-copy, 111
maybe_copy, 137
maybe_copy_done, 139
memoize-type, 120
memory block, 13
Memory_description, 129
mint2bignum, 38
mk-alu-rel, 32
mk-alu-sgn, 32
mk-alu-unsgn, 32
mk-alu-urel, 32
mk-bignum2bits, 38
mk-bignum2mint, 38
mk-binder, 112
mk-bwise, 32
mk-char-ci-pred, 81
mk-cons-prim, 80
mk-empty-env, 112
mk-mint2bignum, 38
mk-op-prim, 80
mk-sel-prim, 81
msw, 74
need_no_gc, 134
neg-prim, 81
new-label, 58
new-tvariable, 126
no-prim, 81
<table>
<thead>
<tr>
<th>Function/Operator</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>noop, 76</td>
<td>147</td>
</tr>
<tr>
<td>no_escape, 147</td>
<td></td>
</tr>
<tr>
<td>nstep, 42</td>
<td></td>
</tr>
<tr>
<td>null?-prim, 81</td>
<td></td>
</tr>
<tr>
<td>null-prim, 81</td>
<td></td>
</tr>
<tr>
<td>num2label, 44</td>
<td></td>
</tr>
<tr>
<td>object code, 12</td>
<td></td>
</tr>
<tr>
<td>oc, 118</td>
<td></td>
</tr>
<tr>
<td>occurs-in-type?, 128</td>
<td></td>
</tr>
<tr>
<td>ocode, 12, 96</td>
<td></td>
</tr>
<tr>
<td>ocode-list, 60</td>
<td></td>
</tr>
<tr>
<td>old-display-icode, 59</td>
<td></td>
</tr>
<tr>
<td>op-code, 96</td>
<td></td>
</tr>
<tr>
<td>op-rands, 96</td>
<td></td>
</tr>
<tr>
<td>op-table, 73</td>
<td></td>
</tr>
<tr>
<td>optimize-icode, 97</td>
<td></td>
</tr>
<tr>
<td>otag, 14, 74</td>
<td></td>
</tr>
<tr>
<td>otest-compile, 114</td>
<td></td>
</tr>
<tr>
<td>out_mem_msg, 137</td>
<td></td>
</tr>
<tr>
<td>pad, 93</td>
<td></td>
</tr>
<tr>
<td>pair_hdr, 152</td>
<td></td>
</tr>
<tr>
<td>pair_hdr_string, 152</td>
<td></td>
</tr>
<tr>
<td>pair_out, 151</td>
<td></td>
</tr>
<tr>
<td>parse, 98</td>
<td></td>
</tr>
<tr>
<td>parse-combination, 100</td>
<td></td>
</tr>
<tr>
<td>parse-definition, 99</td>
<td></td>
</tr>
<tr>
<td>parse-error, 102</td>
<td></td>
</tr>
<tr>
<td>parse-exp, 98</td>
<td></td>
</tr>
<tr>
<td>parse-formal, 99</td>
<td></td>
</tr>
<tr>
<td>parse-schema, 105</td>
<td></td>
</tr>
<tr>
<td>parse-type, 105</td>
<td></td>
</tr>
<tr>
<td>pb, 46</td>
<td></td>
</tr>
<tr>
<td>pprint-exp, 104</td>
<td></td>
</tr>
<tr>
<td>pprint-exp-gen, 104</td>
<td></td>
</tr>
<tr>
<td>pprint-exp-types, 105</td>
<td></td>
</tr>
<tr>
<td>prim, 78</td>
<td></td>
</tr>
<tr>
<td>prim-closure-env, 80</td>
<td></td>
</tr>
<tr>
<td>prim-closure-form, 80</td>
<td></td>
</tr>
<tr>
<td>prim-constant?, 79</td>
<td></td>
</tr>
<tr>
<td>prim-constant-form, 79</td>
<td></td>
</tr>
<tr>
<td>prim-inl-case-form, 80</td>
<td></td>
</tr>
<tr>
<td>prim-inl-case-form, 80</td>
<td></td>
</tr>
<tr>
<td>print-one-instruction, 94</td>
<td></td>
</tr>
<tr>
<td>PRINTF, 146</td>
<td></td>
</tr>
<tr>
<td>printf_closure, 141</td>
<td></td>
</tr>
<tr>
<td>printf_loop, 146</td>
<td></td>
</tr>
<tr>
<td>print_res, 157</td>
<td></td>
</tr>
<tr>
<td>print_stat, 157</td>
<td></td>
</tr>
<tr>
<td>procedure invocation, 15</td>
<td></td>
</tr>
<tr>
<td>program_1, 156</td>
<td></td>
</tr>
<tr>
<td>prop-returns-down, 97</td>
<td></td>
</tr>
<tr>
<td>prune, 128</td>
<td></td>
</tr>
<tr>
<td>put-char-prim, 81</td>
<td></td>
</tr>
<tr>
<td>put-reg!, 75</td>
<td></td>
</tr>
<tr>
<td>putchar, 107, 143</td>
<td></td>
</tr>
<tr>
<td>putchar_format, 144</td>
<td></td>
</tr>
<tr>
<td>putchar_format_string, 144</td>
<td></td>
</tr>
<tr>
<td>rands, 97</td>
<td></td>
</tr>
<tr>
<td>recognize-type?, 96</td>
<td></td>
</tr>
<tr>
<td>reconstruct, 120</td>
<td></td>
</tr>
<tr>
<td>reconstruct-abstraction, 121</td>
<td></td>
</tr>
<tr>
<td>reconstruct-begin, 121</td>
<td></td>
</tr>
<tr>
<td>reconstruct-binder, 121</td>
<td></td>
</tr>
<tr>
<td>reconstruct-combination, 121</td>
<td></td>
</tr>
<tr>
<td>reconstruct-conditional, 121</td>
<td></td>
</tr>
<tr>
<td>reconstruct-recursion, 122</td>
<td></td>
</tr>
<tr>
<td>reconstruct-top, 119</td>
<td></td>
</tr>
<tr>
<td>reconstruct-variable, 121</td>
<td></td>
</tr>
<tr>
<td>reconstruction, 10</td>
<td></td>
</tr>
<tr>
<td>reduce-left, 92</td>
<td></td>
</tr>
<tr>
<td>ref_hdr, 155</td>
<td></td>
</tr>
<tr>
<td>ref_hdr_string, 155</td>
<td></td>
</tr>
<tr>
<td>ref_out, 154</td>
<td></td>
</tr>
<tr>
<td>reg-free?, 75</td>
<td></td>
</tr>
<tr>
<td>reg-num, 76</td>
<td></td>
</tr>
<tr>
<td>rerun, 42</td>
<td></td>
</tr>
<tr>
<td>reset-label-counter, 58</td>
<td></td>
</tr>
<tr>
<td>reset-tvariable-counter!, 126</td>
<td></td>
</tr>
<tr>
<td>restart-emulator, 41</td>
<td></td>
</tr>
<tr>
<td>restore-regs-from-frame, 108</td>
<td></td>
</tr>
<tr>
<td>RETADR, 74, 146</td>
<td></td>
</tr>
<tr>
<td>reverse-list, 92</td>
<td></td>
</tr>
<tr>
<td>root set, 15</td>
<td></td>
</tr>
<tr>
<td>run, 115</td>
<td></td>
</tr>
<tr>
<td>run-m, 116</td>
<td></td>
</tr>
<tr>
<td>runv, 116</td>
<td></td>
</tr>
<tr>
<td>same-constructor?, 50</td>
<td></td>
</tr>
<tr>
<td>same-name?, 51</td>
<td></td>
</tr>
<tr>
<td>same-tvariable?, 126</td>
<td></td>
</tr>
<tr>
<td>same-variable?, 125</td>
<td></td>
</tr>
<tr>
<td>save-regs-into-frame, 108</td>
<td></td>
</tr>
</tbody>
</table>
INDEX

sb-name, 91
sb-prim, 91
sb-type, 91
scan-transitively, 111
scan_transitively, 138
scan_transitively_done, 138
scan_transitively_loop, 138
schema, 51
schema-generics, 51
schema-type, 51
set-bit-field, 39
set-dreg!, 45
set-freg!, 45
set-mem!, 44
set-reg!, 45
set-slot!, 45
set-type!, 120
shiftem, 147
show-stats, 116
show-type-check, 114
simulate-stackframe-pop, 76
simulate-stackframe-push, 76
simulator, 6
sov_msg, 133
SP, 74, 145
stack allocation, 14, 18
stack_allocate-activation-frame?, 64
stack-allocate-block, 108
stack-allocate-environment-frame?, 63
stack-depth, 109
stack-free-block, 109
stack_overflow, 133
stack_underflow, 133
standard-bindings, 86
standard-c-t-env, 91
standard-type-environment, 91
startl_closure, 141
STATISTICS, 139
stats_closure, 141
stat_format, 139
step, 42
stop-and-copy, 15
string-type, 50
string_format, 149
string_format_string, 149
substitute-for-names, 106
substitute-into-type, 123
suv_msg, 133
swap-delayed-icode-lists, 53
sym->string-prim, 81
sym-label-env, 55
sym2string, 107
symbol-type, 50
sym_out, 149
syntax, 5
system-routine, 107
tagging, 14
take-reg!, 75
target-gencode, 62
tc, 118
tenv-generic?, 124
tenv-lookup, 124
test-compile, 115
test-parse, 114
test-parse-simple, 114
testing utilities, 8
the-unit, 51, 128
tlookup, 125
TOPARG, 74
TOTALMEMSIZE, 146
trans-begin, 55
trans-call, 57
trans-closure, 56
trans-conditional, 55
trans-integer, 54
trans-lambda, 56
trans-let, 56
trans-letrec, 57
trans-primitive-combination, 57
trans-string, 54
trans-symbol, 54
trans-unknown-combination, 57
trans-variable, 55
translate, 54
tuple, 94
tuple-ref, 94
tvar-or-schema, 51
tvariable, 125
tvariable->sym, 126