Appendix D

The R language compiler

This appendix gives the documentation and complete code for RCOMP, the compiler for the R programming language.

D.1 R Compiler User's Guide

Currently the R compiler does not have a very convenient user interface. But this section describes the interface, such as it is.

The R compiler resides in the files `~mpf/rcomp/*.lisp` at the MIT AI Lab. The files are also listed in §D.4 below, and can be downloaded from the web via the URL `http://www.ai.mit.edu/~mpf/rc/rcomp`. To use RCOMP, start a Common Lisp interpreter, change to a directory containing the source files, and type `(load "loader")`. This will load up the system.

Write your R program in a file such as `program.r`. Then type at Lisp, `(rcompile-file "program.r")`. First, the source will be printed with comments stripped out, and then, after a delay, the entire Pendulum assembly code for the compiled program will be printed. The example program `sch.r` takes about 15 seconds to compile on a Sparc 20.

Alternatively, one can type `(rcompile-file "program.r" :debug t)`, and the full program will be printed out after every tiny little step in the compilation process. The output from this is voluminous, but by doing an incremental search (`^S`/`^R`) in Emacs for the characters "`==>`" in this output, one can easily scroll through it to see what is going on at each compilation step. However, it is not recommended to try this option with very large programs.

When there are compilation problems, it is helpful to try compiling individual subroutines and statements in isolation, using the calls `(rc` *`source`*`)` and `(rcd` *`source`*`)`, where *source* is a Lisp expression that evaluates to the list representing the fragment of source code to be compiled. `Rc` prints out the compiled assembly code, and `rcd` prints out the complete appearance of the program after each compilation step. For example, `(rc '(a += b))` outputs the single Pendulum assembly language instruction "`ADD $2 $3`."

D.2 Compilation technique

The RCOMP compiler was written in Common Lisp, which provides a portable and well-defined language base that runs on many hardware platforms, and a comprehensive standard set of built-in list-manipulation functions, that allow very easy manipulation of program code fragments represented in a Lisp-like (s-expression) format.

The basic principle of RCOMP operation is essentially macro expansion. A high-level language construct is interpreted like a macro; it is expanded in-place to a sequence of compiler-internal constructs, which are in turn expanded to even lower-level constructs until the whole process bottoms out with expressions representing assembly language instructions in the target architecture.

In fact, the earlier versions of RCOMP, which were very limited in their capabilities, actually used Common Lisp's built-in macro expansion facility to do all the work. The entire compiler was implemented as just a set of macro definitions.

However, this raw approach was not capable of performing important compiler functions such as register assignment, since each statement was compiled separately without knowledge of its context. So we replaced the reliance on built-in macro expansion with our own custom macro-like facility which carried an *environment* structure along through the code expansion process. The environment keeps track of the current variable assignments, mapping the lexical variables to registers and stack locations.

Unfortunately, the macro-expansion type of approach is also not capable of performing certain types of code optimizations, in particular optimizations that might operate across statement boundaries, such as peephole optimizations. So the code produced by RCOMP is not particularly well-minimized. The situation could be improved somewhat by adding a post-processing phase to the compilation in which the sequence of assembly instructions is scanned for known patterns that can be simplified. But a more aggressive and thorough approach to optimization would probably benefit from some higher-level knowledge, and this would probably require a different overall compiler architecture than our macro-expansion-based approach.

However, RCOMP was not intended to be a study in advanced compiler techniques. Rather, for us the imperative was to make the compiler particularly simple to modify and extend, in order to easily handle new constructs, implement existing ones differently, or port the entire compiler to a different target reversible instruction set. All these things are easy in our current design.

The reason for having such flexibility is simply that reversible programming languages are still very much experimental and in flux. It is desirable at this point to be able to very rapidly experiment with different high-level constructs and different machine instructions.

Another advantage of the macro-based design is that it was very simple to program, and the compiler code is fairly easy to read. The definition of most language constructs is just a literal expression (using the backtick operator) showing exactly what that construct expands into.

D.3 Internal compiler constructs

We now document a selection of most of the important internal language constructs currently used internally in the R compiler. These constructs are not meant to be used by the end-user in RCOMP programs. However, when testing and debugging the compiler, or for understanding how it works, it may be useful to try compiling code containing these lower-level constructs. Also, for writing standard libraries, direct use of lower-level constructs may enable better-optimized code than the compiler is currently capable of generating given only the highest-level constructs.

Generally, there is nothing in the current RCOMP architecture that actually prevents user programs from being written using any desired mixture of user-level constructs, intermediate- to low-level internal constructs, and target assembly code. However, users should be aware that if they do not stick to the user-level constructs, then their programs may not be portable across changes in the compiler internals and/or the target architecture.

D.3.1 Intermediate-level internal constructs

Of the internal compiler constructs, the following are still relatively high-level.

D.3.1.1 Mid-level variable manipulation

`<-` Bind.

Syntax: `(`*`var`*` <- `*`val`*`)`

Elements:

> ***var*** — Variable to bind.
>
> ***val*** — Expression whose value to bind it to.

Description:

Assuming the variable *var* is initially clear, bind the variable to the value obtained by evaluating the expression *val*.

`->`

Unbind.

Syntax: `(`*`var`*` -> `*`val`*`)`

Elements:

var — Variable to unbind.

val — Expression whose value to unbind it to.

Description:

Assuming the variable *var* initially contains the value that the expression *val* evaluates to, restore *var* to 0.

`with-regvars`

Declare register variables.

Syntax:

```
(with-regvars var-or-vars
        statement_1
        statement_2 ... )
```

Elements:

var-or-vars — A variable or list of variables to declare.

statement*$_1$*, statement*$_2$*, ... — Statements to compile within the context of this declaration.

Description:

This mid-level construct declares a list of variables and forces them into registers, for greater efficiency of the body statements that refer to those variables. The variables are ensured to be declared only within the lexical body of this statement. They are not guaranteed to remain in registers throughout the entire body, only to be in registers intially.

`register`

Advise register allocation.

Syntax: `(register `*`var-or-vars`*`)`

Elements:

> ***var-or-vars*** — A variable or list of variables.

Description:

Advise the compiler to move the listed variables into registers now, rather than later. This is considered to be advice to the compiler, intended to aid optimization. The compiler need not obey the advice, although the current implementation always does.

`scope`

Declare a variable around a body.

Syntax:

```
(scope var-or-vars
       statement₁
       statement₂ ... )
```

Elements:

> ***var-or-vars*** — A variable or list of variables to declare.
>
> ***statement***$_1$***, statement***$_2$***,*** ... — Statements to compile within the context of this declaration.

Description:

Declares the given list of variables around a body. The variables are created at the start of the body and destroyed at the end. Values of all variables are intially 0. The body is assumed to restore the values of all variables to 0 before it completes. If it does not, further program behavior will not obey the intended semantics! It is currently an error if any of the named variables already exist in the environment (currently there is no compiler support for multiple lexical nestings of scopes for variables with the same name).

D.3.1.2 Mid-level environment manipulation

`ensure-green`

Ensure that a body leaves the environment unchanged.

Syntax:

```
(ensure-green
       statement₁
       statement₂ ... )
```

Elements:

statement$_1$***, statement***$_2$***,*** ... — Statements to compile within the context of this declaration.

Description:

Leaves the environment after executing the given body exactly the same as it was before the body. (At least, with regards to its location map.) This may generate code to move variables back to their original locations if they happened to be moved during the course of the body.

`with-location-map` Enforce a location map around a body.

Syntax:
```
(with-location-map locmapdesc
         statement1
         statement2 ... )
```

Elements:

locmapdesc — Description of a location map.

statement$_1$***, statement***$_2$***,*** ... — Statements to compile within the context of this declaration.

Description:

A declaration that ensures that the environment at the start and end of the body maps variables to locations exactly as described by the given location map. This may trigger relocating variables around to be in the appropriate locations. If the set of variables in the current environment is not the same as that in the location map, currently a compiler error is generated.

`with-environment` Enforce an environment around a body.

Syntax:
```
(with-environment envdesc
         statement1
         statement2 ... )
```

Elements:

envdesc — Description of an environment.

statement$_1$***, statement***$_2$***,*** ... — Statements to compile within the context of this declaration.

Description:

Just like `with-location-map`, except it takes an entire environment description (currently this is the same as an environment object) as its input, rather than just a location map description. Only affects the location map of the current environment, and not other properties of the environment.

`environment`

Enforce an environment.

Syntax: `(environment` *`envdesc`*`)`

Elements:

envdesc — Description of an environment.

Description:

Ensure that the environment immediately following this statement is equivalent to the given environment—that is, has the same location map. This may involve moving variables around to be in the appropriate locations. If the set of variables in the current environment is not the same as the set in the given environment, currently a compiler error is signaled.

`locmap`

Enforce a location map.

Syntax: `(locmap` *`locamapdesc`*`)`

Elements:

locmapdesc — Description of a location map.

Description:

Just like `environment`, but takes a location map description instead of a full environment.

D.3.1.3 Mid-level control-flow constructs

`infloop` — Infinite loop.

Syntax:
```
(infloop
    statement_1
    statement_2 ... )
```

Elements:

***statement*$_1$, *statement*$_2$, ...** — Statements to compile inside the loop.

Description:

Generates an "infinite loop" segment of code—if processing inside the loop runs sequentially off the end, it comes back to the beginning. If the infinite loop is encountered from the outside, we skip over it.

Actually, the present compilation of this construct does not actually enforce that the loop is infinite; entry/exit points could be inserted into the middle of the loop body.

`skip` — Unexecutable code segment.

Syntax:
```
(skip
    statement_1
    statement_2 ... )
```

Elements:

***statement*$_1$, *statement*$_2$, ...** — Statements to compile in the skipped section.

Description:

Generates a segment of code that should be skipped over if encountered; it should not be executed. This is primarily only useful for preventing static data objects from being executed. Really this is implemented exactly like `infloop`. But the connotation is different. In `infloop` we are trying to keep the PC on the *inside*, in `skip` we are trying to keep it on the *outside*. But in both cases, the PC actually just stays on whichever side it was originally. (Unless we cheat and construct brach-pairs connecting inside and outside.)

D.3.1.4 Mid-level branching constructs

`bcs-branch-pair`

Pair of binary conditional switching branches.

Syntax:

```
(bcs-branch-pair
            toplabel (vara1 opa vara2)
            botlabel (varb1 opb varb2)
    statement₁
    statement₂ ... )
```

Elements:

toplabel — Label of the upper branch in the pair.

vara1, vara2 — Variables to compare at the top of the branch.

opa — Operation to compare with at the top of the branch.

botlabel — Label of the lower branch in the pair.

varb1, varb2 — Variables to compare at the bottom of the branch.

opb — Operation to compare with at the bottom of the branch.

statement$_1$***, statement***$_2$***,*** ... — Statements to compile in between the two branches.

Description:

This construct implements the common low-level reversible control-flow situation in which one has a pair of switching (that is, paired) conditional branches pointing to each other, at the top and bottom of a block of code, and the condition being tested is a binary (two-operand) function of two given variables. The operation may be any of the relational operators `=`, `<`, `>`, `<=`, `>=`, `!=` which have the expected C-like meanings of signed integer comparison.

Depending on how `bcs-branch-pair` is used, it can be used to implement either reversible `if` statements or loops. The code is potentially a loop if the second condition can succeed even if the first condition does not.

An important feature of `bcs-branch-pair` is that it ensures that the environment is conserved by the body, despite whatever juggling of registers is done inside the body. This is important because otherwise, we could not at compile time predict what the environment would be after the branch, because we would not know whether the branch would be taken or not.

`twin-us-branch`

Pair of unconditional switching branches.

Syntax: `(twin-us-branch` *toplabel botlabel*
statement$_1$
statement$_2$ ...)

Elements:

toplabel — Label of the upper branch in the pair.

botlabel — Label of the lower branch in the pair.

statement$_1$***, statement***$_2$***,*** ... — Statements to compile in between the two branches.

Description:

This construct implements the common low-level reversible control-flow situation in which one has a pair of switching (paired) unconditional branches pointing to each other, at the top and bottom of a block of code. Such a structure may surround a infinite loop, or the body of a subroutine whose entry/exit point is somewhere in its middle. If the structure is encountered from outside, we just jump over it. This construct knows that the environment after the branch pair is the same as the one before it.

D.3.1.5 Miscellaneous mid-level constructs

`with`

Execute body under some temporary condition.

Syntax: `(with` *tmp-statement*
statement$_1$
statement$_2$ ...)

Elements:

tmp-statement — Temporary statement to do before the body and undo after the body.

statement$_1$***, statement***$_2$***,*** ... — Statements to do after doing and before undoing the given statement.

Description:

Given any statement *tmp-statement* that the compiler knows how to invert, performs that statement, then performs the body statements *statement*$_i$, then performs the inverse operation of the statement to undo its effects. Does not yet handle all possible varieties of *tmp-statement*.

`undo` — Undo a given statement.

Syntax: `(undo statement)`

Elements:

statement — Statement to undo.

Description:

Execute a given statement in reverse, thereby undoing its effects.

D.3.2 Low-level constructs

D.3.2.1 Low-level manipulation of variables

`relocate` — Relocate a variable in the environment.

Syntax: `(relocate var loc)`

Elements:

var — The variable to relocate.

loc — Specifier for the register or stack location to relocate to.

Description:

This construct relocates the given variable to a specific register or stack location. The *loc* is currently represented as a list, with either the form `(reg regno)`, where *regno* is the register number, 0–31, or `(stack offset)`, where *offset* is the stack offset, which should be a negative integer specifying the location relative to the current value of the stack pointer (which currently is register 1).

The relocation of a variable involves both changing its location assignment in the environment, and generating code that actually moves the variable's value from the old location to the new location.

If a different variable is already assigned to the given location, `relocate` moves it out of the way. (By exchanging it with the old location of *var* if there was one, or by moving it to a fresh location.)

`new-var-at` — Create a new variable at a given location.

Syntax: `(new-var-at` *`varname location`*`)`

Elements:

> ***varname*** — The name of the variable to create.
>
> ***location*** — Location specifier to create the new variable at.

Description:

Create a new variable named *varname* in the current environment, and assign it to location *location*. If another variable was assigned to *location*, move that variable to some other location.

`vacate`

Ensure that a given location is unassigned.

Syntax: `(vacate` *`loc`*`)`

Elements:

> ***loc*** — Specifier of location to vacate.

Description:

If a variable is assigned to the given location, move it to somewhere else. If there are any free registers, the variable is moved to one of them, otherwise it is moved to the next available location on the stack.

`get-in-register`

Force a variable into a register.

Syntax: `(get-in-register` *`var`*`)`

Elements:

> ***var*** — The variable to registerify.

Description:

Force the given variable to relocate to a register if it isn't in one already. Picks an arbitrary free register for it to live in, or if no registers are free, boots the least-recently-moved register variable back out to the stack, and moves our variable there.

`add-to-env`

Add a variable to the environment.

Syntax: `(add-to-env` *`varname`*`)`

Elements:

varname — Name of variable to create.

Description:

Explicitly creates a variable of the given name in the current environment, initially unassigned to any particular location. It is currently an error if a variable with the given name already exists.

`tell-loc`

Set the raw location of a variable.

Syntax: `(tell-loc` *`var loc`*`)`

Elements:

var — The variable whose location to set.

loc — The location to set it to.

Description:

Explicitly sets the location of the given variable in the current environment to the one given. Does not move the variable's contents.

`remove-var`

Remove a variable from the environment.

Syntax: `(remove-var` *`var`*`)`

Elements:

var — The variable to remove.

Description:

Explicitly removes a variable from the current environment. Dangerous to subsequent program correctness if the contents of *var* are non-zero.

D.3.2.2 Raw manipulation of environments

`declare-green`

Explicitly restore the environment after a body.

Syntax:
```
(declare-green
      statement1
      statement2 ... )
```

Elements:

***statement*$_1$, *statement*$_2$, ...** — Statements to compile within the context of this declaration.

Description:

Explicitly set the environment after the body completes back to what it was before the body. Does not move variables appropriately. This is a very low-level operation.

`declare-locmap`

Explicitly set the current location map.

Syntax: `(declare-locmap locmapdesc)`

Elements:

locmapdesc — Location map description.

Description:

Explicitly set the entire variable-location map of the current environment to the one given. Does not move contents of variables' locations appropriately. This is a very low-level operation.

`declare-environment`

Explicitly set the current environment.

Syntax: `(declare-environment envdesc)`

Elements:

envdesc — Environment description.

Description:

Explicitly set the entire current environment to the one given. Does not move contents of variables' locations appropriately. This is a very low-level operation.

D.3.2.3 Raw register/stack manipuation

This section describes constructs for direct manipulation (moving and exchanging contents) of register and stack locations.

`swaploc`

Swaps the contents of two register/stack locations.

Syntax: `(swaploc` *loc*$_1$ *loc*$_2$`)`

Elements:

loc$_1$ ***loc***$_2$ — Specifiers for the two locations to swap.

Description:

Swaps the contents of the two specified locations. See `relocate` for the format of a location specifier. Note this function does not change variable location assignments, only the values of the specified locations.

`moveloc`

Move the contents of one location to another.

Syntax: `(moveloc` *loc*$_1$ *loc*$_2$`)`

Elements:

loc$_1$ — Source location.

loc$_2$ — Destination location.

Description:

Assuming loc_2 is empty, moves the contents of loc_1 to it, leaving loc_1 empty. If loc_2 is not empty, the effect will be different.

`exregstack`

Exchange register with stack location.

Syntax: `(exregstack` *reg stackloc*`)`

Elements:

reg — A register location specifier.

stackloc — A stack location specifier.

Description:

Exchanges the contents of a given register location with those of a stack location. Does not change the environment. The current implementation works fine but leads to some suboptimization. (See the code comment.)

`swapregs`

Exchange the contents of two registers.

Syntax: (`swapregs` *r1 r2*)

Elements:

r1*, *r2 — Specifiers of register locations to swap.

Description:

Swaps the contents of two registers. Despite the absence of direct hardware support for this operation in the current PISA spec, the current implementation cleverly swaps the two registers in-place anyway by performing a sequence of three XOR's of one register into another. I learned this particular trick from Charles Isbell's handwritten assembly-language programs for early versions of Pendulum.

`movereg`

Moves one register to another.

Syntax: (`movereg` *r1 r2*)

Elements:

r1 — Source register.

r2 — Destination register.

Description:

Assuming *r2* is initially empty, moves *r1* to *r2*, leaving *r1* empty. Implemented like `swapregs`, but takes advantage of the assumption that *r2* is empty to eliminate one of the 3 XORs. If *r2* is not empty, the outcome will not fit the intended semantics.

`with-stack-top`

Temporarily go to top of stack.

Syntax: (`with-stack-top`
 $statement_1$
 $statement_2$ ...)

Elements:

***statement*$_1$, *statement*$_2$, ...** — Statements to execute in the context of the adjusted stack pointer.

Description:

Execute the given body within a context where the stack pointer is reset to the top of the stack, that is, beyond all currently-allocated variables. This is used in the implementation of subroutine calls. The environment is not adjusted, so any entries in the environment that might have been referring to stack-allocated variables will be present but invalid in the body. Higher-level constructs that use `with-stack-top` need to cope with this.

`with-SP-adjustment`

Temporarily adjust stack pointer.

Syntax:
```
(with-SP-adjustment amt
    statement1
    statement2 ... )
```

Elements:

amt — Literal amount by which to adjust stack pointer.

***statement*$_1$, *statement*$_2$, ...** — Statements to execute in the context of the adjusted stack pointer.

Description:

Execute the body within a context where the stack pointer is temporarily adjusted by the explicit amount *amt*. The stack grows downwards in memory, so negative amounts correspond to moving towards the "top" (most short-lived) part of the stack.

D.3.2.4 Low-level control flow constructs

`_if`

Specialized if statement.

Syntax:
```
(_if (reg1 op reg2) then
    statement1
    statement2 ... )
```

Elements:

reg1*, *reg2 — Registers to compare.

op — Binary relation to use for comparison.

***statement*$_1$, *statement*$_2$, ...** — Statements to execute if the condition is true.

Description:

Like `if`, but the condition must be a simple binary relation (with no nested subexpressions) between two explicit registers *reg1* and *reg2*.

`_ifelse`

Specialized if/else statement.

Syntax:

```
(_ifelse (reg1 op reg2) then
         (if-statement1
          if-statement2 ... )
         (else-statement1
          else-statement2 ... ))
```

Elements:

reg1*, *reg2 — Registers to compare.

op — Binary relation to use for comparison.

***if-statement*$_1$, *if-statement*$_2$, ...** — Statements to execute if the condition is true.

***else-statement*$_1$, *else-statement*$_2$, ...** — Statements to execute if the condition is false.

Description:

Like `_if`, but there is a second "else" body that will be executed if the condition fails.

`gosub`

Low-level subroutine call.

Syntax: `(gosub subname)`

Elements:

subname — Name of subroutine to call.

Description:

Calls the subroutine of the given name. This is a low-level operation that doesn't know anything about stacks, arguments, or calling conventions. It just transfers control.

`rgosub`

Low-level reverse subroutine call.

Syntax: `(rgosub` *`subname`*`)`

Elements:

subname — Name of subroutine to call.

Description:

Like `gosub`, but runs the subroutine in reverse.

`portal`

Subroutine entry/exit point.

Syntax: `(portal` *`label`*`)`

Elements:

label — Name to give this portal.

Description:

Declares a subroutine entry/exit point to exist at this point in the program, with the name *label*. A caller can call *label* and have control transferred to this point. Then, if control loops back around to this point again, the portal will switch control back to the caller.

The portal currently works by using register 2 to store the information needed to return to the caller. So the value in this register needs to be preserved during the body of the subroutine.

D.3.2.5 Low-level branching constructs

`sbra-pair`

Low-level pair of switching unconditional branches.

Syntax: `(sbra-pair` *`toplabel botlabel`*
`statement`$_1$
`statement`$_2$ ... `)`

Elements:

toplabel — Label of the upper branch in the pair.

botlabel — Label of the lower branch in the pair.

statement $_1$**,** ***statement*** $_2$**,** ... — Statements to compile in between the two branches.

Description:

Lower-level analogue of `twin-us-branch` that does not worry about maintaining the environment across all the branching.

`sbra`

Switching branch.

Syntax: `(sbra` *`thislabel otherlabel`*`)`

Elements:

thislabel — Label for this branch.

otherlabel — Label for the other branch that we switch to.

Description:

Unconditional switching branch labeled *thislabel* that exchanges control between here and another such branch labeled *otherlabel*.

The semantics is: If we arrive at this statement from *otherlabel*, then continue in normal sequential execution. If we arrive at this statement sequentially, then branch to *otherlabel*. If we arrive at this statement some other way, the behavior is some other, nonsensical thing. So you can see that these "switching" branches have to come in pairs.

`sbez`

Switching branch on equal to zero.

Syntax: `(sbez` *`thislabel var otherlabel`*`)`

Elements:

thislabel — Label for this branch.

var — Variable to test.

otherlabel — Label for the other branch that we switch to.

Description:

This is an example of a *conditional* switching branch. It tests *var* and if it is zero, it behaves like `sbra`, otherwise it is a no-op. If we branch into this statement but its condition does not succeed, the resulting behavior will be nonsensical.

`sbne` — Switching branch on not equal to zero.

Syntax: `(sbne` ***`thislabel var otherlabel`***`)`

Elements:

> ***thislabel*** — Label for this branch.
>
> ***var*** — Variable to test.
>
> ***otherlabel*** — Label for the other branch that we switch to.

Description:

This is an example of a *conditional* switching branch. It tests *var* and if it is nonzero, it behaves like `sbra`, otherwise it is a no-op. If we branch into this statement but its condition does not succeed, the resulting behavior will be nonsensical.

`bcs-branch` — Binary conditional switching branch.

Syntax: `(bcs-branch (`***`var1 op var2`***`)` ***`thislabel otherlabel`***`)`

Elements:

> ***var1, var2*** — Variables to compare.
>
> ***op*** — Operation to compare with.
>
> ***thislabel*** — Label for this branch.
>
> ***otherlabel*** — Label for the other branch that we switch to.

Description:

This is a general two-operand conditional switching branch. It compares the two variables using the operation *op*, which may be any of the relational operators `=`, `<`, `>`, `<=`, `>=`, `!=`, which have the expected C-like meanings of signed integer comparison. If the comparison succeeds, it behaves like `sbra`, otherwise it is a no-op. If we branch into this statement but its condition does not succeed, the resulting behavior will be nonsensical.

`bc-branch` — Non-switching binary conditional branch.

Syntax: `(bc-branch (`*`var1 op var2`*`)` *`destlabel`*`)`

Elements:

> ***var1, var2*** — Variables to compare.
>
> ***op*** — Operation to compare with.
>
> ***destlabel*** — Label of destination.

Description:

This is like `bcs-branch` except that it is not a switching branch; that is, it does not branch to a location that explicitly refers back to it. Instead, the desination will probably be a subroutine entry/exit point which will save away the branch register to allow returning to this location later.

`rbc-branch`

Register binary conditional branch.

Syntax: `(rbc-branch (`*`r1 op r2`*`)` *`destlabel`*`)`

Elements:

> ***r1, r2*** — Registers to compare.
>
> ***op*** — Operation to compare with.
>
> ***destlabel*** — Label of destination.

Description:

Like `bc-branch` except that its arguments must be explicit registers, not variables.

D.3.2.6 Miscellaneous low-level constructs

`_with`

Specialized version of `with`.

Syntax:

```
(_with (var <- val)
          statement_1
          statement_2 ... )
```

Elements:

> ***var*** — Variable name to bind.
>
> ***val*** — Expression whose value to bind it to.

statement$_1$***,*** ***statement***$_2$***,*** ... — Statements to do after doing and before undoing the given statement.

Description:

A version of `with` in which the "statement" to be done before the body (and undone after it) must be a variable-binding. The `_with` construct is more efficient than the general version of `with` for this case, because the *val* expression is not evaluated as many times. However, `_with` may use an amount of temporary space linear in the length of the *val* expression during the course of the body, whereas `with` does not.

`withargs`

Prepare arguments as for a subroutine call.

Syntax:

```
(withargs arg-list
        statement1
        statement2 ... )
```

Elements:

arg-list — List of expressions for the arguments.

statement$_1$***,*** ***statement***$_2$***,*** ... — Statements to do in the context of having prepared the arguments.

Description:

Given a list of "argument" expressions, temporarily places the values of those expressions in the canonical locations where the arguments to a subroutine should go, and compiles the body in that context. This is part of the caller side of the current subroutine calling convention.

D.4 Compiler LISP source code

In this section, we give a verbatim listing of the most recent version of the RCOMP source. This is composed of the following files:

`loader.lisp` — §D.4.1, p. 335. This loads up all the parts of the RCOMP program in an appropriate order.

`util.lisp` — §D.4.2, p. 335. Defines general-purpose utilty functions and macros that we use.

`infrastructure.lisp` — §D.4.3, p. 336. Defines our macro-expansion like facility for defining how to compile language constructs.

`location.lisp` — §D.4.4, p. 341. Defines some functions for working with objects that describe a variable's location in the register file or stack.

`environment.lisp` — §D.4.5, p. 342. Defines the environment objects which map variables to their locations.

Files defining construct-expansion "macros":

- `regstack.lisp` — §D.4.6, p. 346. Defines low-level constructs for direct manipulation of registers and the stack.
- `variables.lisp` — §D.4.7, p. 347. Defines high- to low-level constructs for manipulation of variables in variable assignments (environments).
- `branches.lisp` — §D.4.8, p. 350. Constructs providing intermediate- and low-level support for various kinds of branch structures for control-flow.
- `expression.lisp` — §D.4.9, p. 354. Constructs and low-level functions for expanding nested expressions.
- `clike.lisp` — §D.4.10, p. 359. Defines constructs for various user-level user-level C-like operators.
- `print.lisp` — §D.4.11, p. 362. Defines a few very simple constructs for producing output.
- `controlflow.lisp` — §D.4.12, p. 362. Defines user-level to intermediate-level control flow constructs such as conditionals and looping.
- `subroutines.lisp` — §D.4.13, p. 364. Provides high and low level support for subroutines.
- `staticdata.lisp` — §D.4.14, p. 367. Defines constructs for defining static data objects. Currently this is the only way to provide input to a program.
- `program.lisp` — §D.4.15, p. 367. Defines very high-level constructs for wrapping around the entire program.
- `library.lisp` — §D.4.16, p. 368. Defines constructs that expand into code for standard subroutine libraries. Currently the library is very minimal.

`files.lisp` — §D.4.17, p. 369. Provides support for reading the source code to compile from a file.

`test.lisp` — §D.4.18, p. 370. Miscellaneous functions and R code fragments for exercising the compiler. Some of these may be obsolete.

Let us now present the code.

D.4.1 `loader.lisp`

This very simple file just loads up all the Common Lisp files that make up Rcomp. It is completely devoid of any sophisticated options. Perhaps someday we should use one of the popular CL system-definition facilities instead.

```
;;; -*- Package: user -*-
(in-package "USER")

;;;----------------------------------------------------------------------
;;; This is the loader for the reversible compiler system.  Currently all
;;; files are just in the USER package.

;; Load up the system.
(load "util")            ; General-purpose utilities.
(load "infrastructure")  ; Mechanism for defining and compiling constructs.
(load "location")        ; Describing locations where variables are stored.
(load "environment")     ; Mapping variables to their locations.
(load "regstack")        ; Direct manipulation of registers and the stack.
(load "variables")       ; Creating, destroying, moving variables.
(load "branches")        ; Branches, branch pairs, labels.
(load "expression")      ; Binding variables to multiply-nested expressions.
(load "clike")           ; C-like assignment-operator statements.
(load "print")           ; Data output.
(load "controlflow")     ; High-level conditionals & loops.
(load "subroutines")     ; Support for subroutine calls.
(load "staticdata")      ; Static data definitions.
(load "program")         ; Highest-level constructs.
(load "library")         ; Standard library of R routines.
(load "files")           ; File compiler.

(load "test")            ; Example programs.

;; Command to reload the system.
(defun l () (load "loader"))
```

D.4.2 `util.lisp`

Defines some completely non-application-specific Lisp functions and macros that we use in Rcomp.

```
;;; -*- Package: user -*-
(in-package "USER")
;;;----------------------------------------------------------------------
;;; General utilities.

;;;----------------------------------------------------------------------
;;; Some abbreviations for Common Lisp entities.
```

```
;; Don't you agree that MULTIPLE-VALUE-BIND's name is too long?
(defmacro mvbind (&rest args)
  `(multiple-value-bind . ,args))

;; Same here.
(defmacro dbind (&rest args)
  `(destructuring-bind . ,args))

;;;---------------------------------------------------------------------
;;; Boolean shtuff.  Silly, but hey.

(defconstant true t)
(defconstant false nil)

(defmacro true! (&rest places)
  `(setf . ,(mapcan #'(lambda (place) (list place true)) places)))
(defmacro false! (&rest places)
  `(setf . ,(mapcan #'(lambda (place) (list place false)) places)))

;; Convert an arbitrary object to a true-false value.
(defun true? (obj) (if obj true false))
(defun false? (obj) (eq obj false))

;;;---------------------------------------------------------------------
;;; List manipulation.

;; REPL - Replace first occurrence.  FUNC is called on each item of LIST in
;; succession until it returns non-NIL, at which point a new list is
;; returned in which the guilty item is replaced by the value which was
;; returned by FUNC.  The new list shares its tail with the old.  If FUNC
;; never returns non-nil then a copy of LIST is returned.
(defun repl (list func)
  (if list
      (let ((v (funcall func (car list))))
        (if v (cons v (cdr list))
          (cons (car list) (repl (cdr list) func))))))

;; In this version of REPL, FUNC is passed not only each item of LIST,
;; but also the item's index (as per NTH or ELT).
(defun repl2 (list func &optional (firstindex 0))
  (labels
      ((helper (list index func)
          (if list
              (let ((v (funcall func (car list) index)))
                (if v (cons v (cdr list))
                  (cons (car list) (helper (cdr list) (1+ index) func)))))))
    (helper list firstindex func)))
```

D.4.3 infrastructure.lisp

This file is the core of our macro-style compilation architecture. The expansion of any given language construct is defined using the `defconstruct` macro, defined below.

The core function that does the work of compilation is `rcomp`. It recursively expands the macro definitions, while keeping track of the environment, until the process bottoms out in statements that cannot be further expanded.

```
;;; -*- Package: user -*-
```

```
(in-package "USER")
;;;----------------------------------------------------------------------
;;; Compilation infrastructure.

;; Given an object, return non-nil IFF it could possibly be an
;; infix-operator statement.
(defun infix-form? (form)
  (and (listp form)
       (>= (length form) 2)
       (symbolp (second form))
       (get (second form) 'is-infix)))

;; Given a form, get it into the canonical form where the operator is
;; first.
(defun canonicalize (form)
  (if (infix-form? form)
      `(,(second form) ,(first form) . ,(cddr form))
    form))

;; Given an object, if it's an operator (construct) symbol, return
;; its definition.
(defun definition (operator)
  (and (symbolp operator)
       (get operator 'construct-definition)))

;; Given an operator (construct symbol), return the opposite operator.
;; (Which will undo the effect of the given operator.)
(defun opposite (operator)
  (get operator 'opposite))

;; Given an object, return non-NIL iff it may potentially be a
;; single form statement (not a label atom) with a definition.
(defun statement? (form)
  (and (listp form)
       (not (null form))
       (or (definition (car form))
           (infix-form? form))))

;; Guess whether an object may be a list of statements/primitives.
(defun list-of-statements? (obj)
  (and (listp obj)
       (not (statement? obj))
       (not (null obj))
       (statement? (car obj))))

;; DEFCONSTRUCT - Define how a particular construct is to be compiled.
;; Given a construct name symbol CNAME, lambda list LAMBDA-LIST, and body
;; statements BODY, define CNAME to be a reversible language construct with
;; structure given by LAMBDA-LIST and compilation generated by the BODY.
;; During compilation the BODY gets executed with the variables mentioned
;; in the LAMBDA-LIST bound to corresponding parts of the item to be
;; compiled, and with the variable ENV bound to the variable-location
;; environment in effect at the start of the statement.  The body should
;; return 2 values: the first is a list of statements to which this
;; statement is equivalent.  The second value indicates the environment in
;; effect after the given statement(s).  It may be NIL meaning that the
;; source as a high-level statement does not affect the environment after
;; the statement, although the compiled lower-level statements might.

(defmacro defconstruct (cname lambda-list &body body)
  (let ((opposite
```

```
          (if (eq (car body) :opposite)
              (prog1
                (cadr body)
                (setf body (cddr body)))
            cname)))
     `(setf (get ',cname 'opposite) ',opposite
            (get ',cname 'construct-definition)
            #'(lambda (args env)
                (let ((form (cons ',cname args)))
                  (destructuring-bind ,lambda-list args
                    . ,body))))))

(defmacro definfix ((leftarg opname &rest rightargs) &body body)
  `(progn
     (defconstruct ,opname (,leftarg . ,rightargs)
       . ,body)
     (true! (get ',opname 'is-infix))))

;; Given a statement or a list of statements to compile and an optional
;; initial environment (which defaults to the empty environment), return an
;; equivalent list of compiled statements and the environment in effect
;; after them.
(defun rcomp (source &optional startenv)
  (when (null startenv) (setf startenv (empty-env)))
  (setf source (canonicalize source))
  (cond
   ((null source)
    (values source startenv))
   ((statement? source)
    ;; Source is a single non-label statement with a definition.
    (let ((def (definition (first source))))
      (mvbind (compiled endenv)
              ;; Compile it once.
              (funcall def (cdr source) startenv)
         (mvbind (recomp reenv)
                 ;; Try compiling it further.
                 (rcomp compiled startenv)
           (values recomp
                   (or endenv reenv))))))
   ((form-list? source)
    ;; Source is a list of statements.
    (mvbind (firstcomp firstendenv)
            ;; Compile first statement.
            (rcomp (first source) startenv)
       (mvbind (restcomp restendenv)
               ;; Compile remaining statements in environment from
               ;; first statement.
               (rcomp (rest source) firstendenv)
         (values (if (listp firstcomp)
                     (append firstcomp restcomp)
                   (cons firstcomp restcomp))
                 restendenv))))
   (t
    ;; In all other cases just compile the source to itself
    ;; and leave the environment unchanged.
    (values (list source) startenv))))

;;
;; This version of RCOMP, for debugging purposes, prints out the entire
;; state of the partially-compiled program after each individual code
;; transformation.
```

```
;;
;; WHOLE represents the entire current state of the compilation,
;; represented as a cons cell whose CDR is the current partially-compiled
;; source, which MUST be a LIST of statements, not a single statement.
;; POINTER is a pointer to the cons cell whose CDR is the part of the
;; source that remains to be compiled.  In general, the CAR of this CDR
;; will be an ENV statement giving the current environment.
;;
(defun rcomp-repl (whole &optional (pointer whole))
  (myprint (cdr whole)
           (if (eq (caadr pointer) 'env)
               (cddr pointer)
             (cdr pointer)))            ;Print thuh whole shebang.
  (format t "~&Ready: ")
  ;; (clear-input) (finish-output) ;These don't seem to work right.
  ;; (read-line)
  (let ((source (cdr pointer))          ;Remaining source to compile.
        startenv)
    (if (and (listp source)             ;List of statements.
             (listp (car source))       ;Non-label statement.
             (eq (caar source) 'env))   ;Special (ENV <env>) statement.
        (setf startenv (cadar source))  ;Get the <env>.
      (progn
        ;; Invent an ENV statement and insert it.
        (format t "~&Default environment.~%")
        (setf startenv (empty-env))
        (setf (cdr pointer)             ;Alter our object as follows.
              `((env ,startenv)
                . ,source))
        (myprint (cdr whole) (cddr pointer))))
    ;; Now STARTENV is the current env, and current source obj is just
    ;; after the initial env statement.
    (setf source (cddr pointer))
    ;; If no statements left to compile, we're done.
    (when (null source)
      (return-from rcomp-repl whole))
    (let ((form (car source)))
      ;; From here on we approximately mirror structure of RCOMP.
      ;; If first form is an infix form, canonicalize it.
      (when (infix-form? form)
        ;(format t "~&Canonicalize.~%")
        (setf form (canonicalize form)
              (car source) form)
        ;(myprint (cdr whole))
        )
      (cond
       ;; If first item is label: do nothing with it.
       ((atom form)
        (format t "~&Label.~%")
        (setf (cdr pointer) `(,form
                              (env ,startenv)
                              . ,(cdr source)))
        (rcomp-repl whole (cdr pointer)))
       (t ;; Else first item is a non-label STATEMENT.
        (let ((first (first form)))
          (if (symbolp first)
              ;; Assume source code is a single statement, FIRST is the symbol
              ;; naming the statement type, for dispatching.
              (if (eq first 'env)
                  (progn
                    (format t "~&Environment override.~%")
```

```
                 (setf (cdr pointer) source)
                 (rcomp-repl whole pointer))
               (let ((def (definition first)))
                 (if (null def)
                     ;; No definition for this.  Assume it's a final
                     ;; assembly instruction and doesn't change the
                     ;; environment.
                     (progn
                       (format t "~&Final.~%")
                       (setf (cdr pointer) `(,form
                                             (env ,startenv)
                                             . ,(cdr source)))
                       (rcomp-repl whole (cdr pointer)))
                   ;; OK, we do have a definition for it.
                   (mvbind (compiled endenv)
                           ;; Call the transformer function.
                           (funcall def (cdr form) startenv)
                           ;; Insert result.
                           (format t "~&Expand ~s.~%" first)
                           (if (and endenv (null compiled))
                               (setf (cdr pointer)
                                     `((env ,endenv)
                                       . ,(cdr source)))
                             (setf (cddr pointer)
                                   (if endenv
                                       `(,@(if compiled
                                               (if (not (form-list? compiled))
                                                   (list compiled)
                                                 compiled))
                                            (env ,endenv)
                                            . ,(cdr source))
                                     `(,@(if compiled
                                             (if (not (form-list? compiled))
                                                 (list compiled)
                                               compiled))
                                          . ,(cdr source)))))
                           ;; Try compiling same thing again.
                           (rcomp-repl whole pointer)))))
           ;; The first item isn't a symbol so assume it's a statement
           ;; and treat the form as a list of statements.
           (progn
             (setf source (append form (cdr source))
                   (cddr pointer) source)
             (format t "~&Insert statement list.~%")
             (rcomp-repl whole pointer))))))))

;; The user-level compiler-debugging routine.
(defun rcomp-debug (source)
  (rcomp-repl (cons nil (list source))))

(defun rcd (source)
  (rcomp-debug source))

;; non-nil iff obj could be a list of forms (not incl. label syms)
(defun form-list? (obj)
  (and (listp obj)
       (not (and (car obj) (symbolp (car obj))))
       (not (and (cadr obj) (symbolp (cadr obj))))))

;; Expand the source code to its compilation once, but not
;; recursively.  This is for debugging.
```

```
(defun expand1 (source &optional env)
  (let ((def (get (first source) 'construct-definition)))
    (if (null def)
        (values (list source) env)
      (funcall def (cdr source) env))))

;; For testing.
(defun myprint (code &optional pointer)
  #|(format t "~&~@
             --------~@
             Program:~@
             --------~%")|#
  (format t "~&~%")
  (if (list (car code))
      (dolist (s code)
        (if (eq s (car pointer)) (format t "==>"))
        (cond
         ((atom s) ;Interpret atoms as labels.
          (format t "~s:~15T" s))
         ((and (symbolp (car s))
               (not (get (car s) 'construct-definition))
               (not (eq (car s) 'env))
               (not (and (symbolp (cadr s))
                         (get (cadr s) 'construct-definition))))
          (format t "~16T")
          (dolist (w s)
            (if (register? w)
                (format t "$~s " (cadr w))
              (format t "~:w " w)))
          (format t "~%"))
         (t
          (format t "~16T~:w~%" s))))
    (pprint code))
  (format t "~&~%")
  (values))

(defun rc (source &optional startenv)
  (mvbind (prog env)
          (rcomp source startenv)
          (myprint prog)
          (format t "~&~%Final environment:")
          (print env))
  (values))
```

D.4.4 location.lisp

Functions for working with "location" objects which give the locations of variables. Currently these are just implemented as simple list structures.

```
;;; -*- Package: user -*-
(in-package "USER")
;;; ----------------------------------------------------------------------
;;; A LOCATION object indicates where a variable is stored.
;;;
;;; The current implementation uses list structures.  If a <LOCATION> is
;;; NIL, then the variable exists in the environment but has no storage
;;; location (and is therefore also unbound).  If a <LOCATION> is (REG
;;; <regno>) then the variable is located in register number <REGNO>.  If
```

```
;;; <LOCATION> is (STACK <offset>) then the variable is located on the
;;; stack at the address SP+<OFFSET>, where SP is the current value of the
;;; stack pointer register.
;;; -------------------------------------------------------------------

;; Return non-nil iff the object OBJ is a register location.
(defun register? (obj)
  (and (listp obj) (cdr obj) (null (cddr obj))
       (eq (car obj) 'reg)))

(defun stackloc? (obj)
  (and (listp obj) (cdr obj) (null (cddr obj))
       (eq (car obj) 'stack)))

;; Return non-nil iff the object OBJ is a null location (meaning the
;; location of a variable that is not located anywhere).
(defun null-loc? (obj)
  (null obj))

;; Return the given stack location's offset from the current stack
;; pointer.
(defun offset (stackloc)
  (cadr stackloc))

(defun location? (obj)
  (or (register? obj) (stackloc? obj)))
```

D.4.5 environment.lisp

Defines environment objects, which keep track of variables' location assignments. This is currently implemented as a full-fleged CLOS object.

```
;;; -*- Package: user -*-
(in-package "USER")
;;; ===================================================================
;;; This file defines the interface to and implementation of ENVIRONMENT
;;; objects.  An environment object determines what variables are present
;;; in the R environment at a given point in the program, and where they
;;; are stored.  The environment also maintains identifiers for static
;;; objects.  This file provides the programmer's interface to environment
;;; objects, and should be used in lieu of manipulating the underlying
;;; structures directly.  This is intended to reduce errors and allow
;;; environments to be reimplemented at a later time.
;;; ===================================================================

(defun empty-locmap () '())

(defclass environment ()
  ((variable-locations
    :type list ;More specifically an ALIST from identifiers to locations.
    :initform (empty-locmap)
    :initarg :locmap
    :accessor locmap
    :documentation "An ALIST of the form ((<var1> . <location1>) ...).
       Each VAR is a symbol, and only appears once in the alist.  The
       variables that have most recently been created or moved appear at
       the front of the alist.")
   (static-value-identifiers
    :type list
```

```
    :initform nil
    :initarg :staticvals
    :accessor staticvals
    :documentation "A list of identifier symbols that denote static
       data values permanently located in memory.")
   (static-array-identifiers
    :type list
    :initform nil
    :initarg :staticarrays
    :accessor staticarrays
    :documentation "A list of identifier symbols denoting static arrays.")
   )
  (:documentation "An environment specifies the meanings of identifiers at
     a given point during the compilation of a program."))

(defmacro make-environment (&rest args)
  `(make-instance 'environment . ,args))

(defun empty-env ()
  (make-environment))

(defun copy-environment (env)
  (make-environment :locmap (copy-alist (locmap env))
                    :staticvals (copy-list (staticvals env))
                    :staticarrays (copy-list (staticarrays env))))

(defmethod env-to-list ((env environment))
  `(:locmap ,(locmap env)
    :staticvals ,(staticvals env)
    :staticarrays ,(staticarrays env)))

(defmethod print-object ((env environment) stream)
  (write (env-to-list env) :stream stream))

;; Return an environment like ENV, but with VAR bound to location LOC
;; in the location map.
(defmethod set-loc (var loc (env environment))
  (setf env (copy-environment env)
        (locmap env) `((,var .,loc).,(remove (assoc var (locmap env))
                                             (locmap env))))
  env)

;; Return an environment that is just like the given environment ENV but
;; with the variable VAR removed from the location map.
(defmethod remove-var (var (env environment))
  (setf env (copy-environment env)
        (locmap env) (remove (assoc var (locmap env)) (locmap env)))
  env)

;; Return non-nil iff the variable VAR exists in the environment ENV.
(defmethod defined-in-env? (var (env environment))
  (assoc var (locmap env)))

;; Return the location of variable VAR in environment ENV, or nil if VAR
;; does not exist in the environment.  This is not guaranteed to be
;; distinct from the null location.  (The DEFINED-IN-ENV function can be
;; used to distinguish the two cases.)
(defmethod location (var (env environment))
  (cdr (assoc var (locmap env))))

;; Return the variable stored at the given location in the given
```

```
;; environment, or nil if none.
(defmethod var-at-loc (loc (env environment))
  (car (rassoc loc (locmap env) :test #'equal)))

;; Return non-nil iff the two environments E1 and E2 contain the exact same
;; set of variables.
(defmethod equal-vars? ((e1 environment) (e2 environment))
  (let ((answer t))
    (dolist (v (append (mapcar #'car (locmap e1)) (mapcar #'car (locmap e2))))
        (if (not (and (assoc v (locmap e1)) (assoc v (locmap e2))))
            (setf answer nil)))
    answer))

;; Return non-nil iff the two environments E1 and E2 are equivalent, in the
;; sense that they have the same variables and assign them to the same
;; locations.
(defmethod equal-env? ((e1 environment) (e2 environment))
  (let ((answer t))
    (dolist (v (append (mapcar #'car (locmap e1)) (mapcar #'car (locmap e2))))
      (if (not (equal (assoc v (locmap e1)) (assoc v (locmap e2))))
          (setf answer nil)))
    answer))

;; Return the first variable in environment E1 that is not located in the
;; same place in environment E2.
(defmethod first-misloc ((e1 environment) (e2 environment))
  (dolist (v (mapcar #'car (locmap e1)))
    (if (not (equal (cdr (assoc v (locmap e1))) (cdr (assoc v (locmap e2)))))
        (return v))))

;; Return the lowest-numbered available register location in the given
;; environment.  Available means not containing any variable.  Registers 0
;; and 1 are never available because 0 is reserved to contain 0 and 1 is
;; reserved to be the stack pointer.  Returns nil if no registers are
;; available.
(defmethod next-avail-reg ((env environment))
  (let ((i 2))
    (loop
     (if (not (rassoc `(reg ,i) (locmap env) :test #'equal))
         (return `(reg ,i)))
     (incf i)
     (if (= i 32) (return)))))

;; Return the first stack location ABOVE THE CURRENT STACK
;; POINTER that is available (doesn't contain a variable) in the given
;; environment.  The stack grows down, so ABOVE means MORE NEGATIVE THAN.
(defmethod next-avail-stack ((env environment))
  (let ((i -1))
    (loop
     (if (not (rassoc `(stack ,i) (locmap env) :test #'equal))
         (return `(stack ,i)))
     (decf i))))

;; Return a variable in the given environment that was least recently
;; created or moved.
(defmethod least-recently-moved ((env environment))
  ;; Currently this information is maintained by putting newly-created or
  ;; moved variables on the front of the alist, so we just return the last
  ;; variable on the list.
  (car (nth (1- (length (locmap env))) (locmap env))))
```

```
;; Return non-nil if VAR is located in a register in environment ENV.
(defmethod in-register? (var (env environment))
  (register? (location var env)))

;; Return the index of the topmost stack location at SP or below that
;; has a variable in it.  In other words, where is
;; the top of the stack in relation to the current stack pointer.
(defmethod top-of-stack ((env environment))
  (let ((h 0))
    (dolist (loc (locmap env))
      (if (eq (cadr loc) 'stack)
          (if (< (caddr loc) h)
              (setf h (caddr loc)))))
    h))

(defmethod add-staticval (name (env environment))
  (setf env (copy-environment env)
        (staticvals env) (cons name (staticvals env)))
  env)

(defmethod add-staticarray (name (env environment))
  (setf env (copy-environment env)
        (staticarrays env) (cons name (staticarrays env)))
  env)

(defmethod static-id? (obj (env environment))
  (and (symbolp obj)
       (or (member obj (staticvals env))
           (member obj (staticarrays env)))))

(defmethod dynamic-var? (obj (env environment))
  (and (symbolp obj)
       (not (static-id? obj env))))

(defmethod static-array? (obj (env environment))
  (member obj (staticarrays env)))

(defun negated-sym? (obj)
  (and (symbolp obj)
       (eq #\- (elt (symbol-name obj) 0))))

(defun positive-of (negsym)
  (intern (subseq (symbol-name negsym) 1)))

(defmethod literal? (obj (env environment))
  (or (numberp obj)
      ;; Static array identifiers are literals because they stand for their
      ;; addresses, and are left in the form of symbols which are processed
      ;; directly by the assembler.
      (static-array? obj env)
      ;; If FOO is a static array, -FOO is a literal also.
      (and (negated-sym? obj)
           (static-array? (positive-of obj) env))
      ;; Expression is the address of a static-val.
      (static-val-addr? obj env)))

(defmethod static-val? (obj (env environment))
  (member obj (staticvals env)))

(defmethod static-val-addr? (obj (env environment))
  (and (listp obj)
```

```
       (eq (car obj) '&)
       (null (cddr obj))
       (static-val? (second obj) env)))
```

D.4.6 regstack.lisp

This file defines low-level constructs for directly manipulating registers and the stack.

```
;;; -*- Package: user -*-
(in-package "USER")

;;;-------------------------------------------------------------------
;;; Register/stack manipulation.

(defconstruct relocate (var loc)
  (let ((oldloc (location var env)))
    (if (not (equal oldloc loc))        ;If not already there.
        (if (null oldloc)
            `((vacate ,loc)
              (tell-loc ,var ,loc))
          (if (null loc)
              `(tell-loc ,var ,loc)
            (let ((oldv (var-at-loc loc env)))
              (if oldv
                  ;; If new location occupied, swap.
                  `((swaploc ,oldloc ,loc)
                    (tell-loc ,var ,loc)
                    (tell-loc ,oldv ,oldloc))
                ;; Not occupied, just move.
                (if (null-loc? oldloc)
                    `(tell-loc ,var ,loc)
                  `((moveloc ,oldloc ,loc)
                    (tell-loc ,var ,loc))))))))))

;;; loc1 and loc2 should be register or stack locations.
(defconstruct swaploc (loc1 loc2)
  (if (and (register? loc1)
           (register? loc2))
      `(swapregs ,loc1 ,loc2)
    (if (register? loc1)
        `(exregstack ,loc1 ,loc2)
      (if (register? loc2)
          `(exregstack ,loc2 ,loc1)
        ;; We need a temporary register to facilitate the stack exchange;
        ;; we choose reg. 31 for no particular reason.  The net change to
        ;; it is nil.  This all works but could probably be made
        ;; considerably more efficient.
        `((exregstack (reg 31) ,loc1)
          (exregstack (reg 31) ,loc2)
          (exregstack (reg 31) ,loc1))))))

;; Assuming loc2 is clear, move loc1 to it.
(defconstruct moveloc (loc1 loc2)
  (if (and (register? loc1)
           (register? loc2))
      `(movereg ,loc1 ,loc2)
    (if (register? loc1)
        `(exregstack ,loc1 ,loc2)
      (if (register? loc2)
```

```
          `(exregstack ,loc2 ,loc1)
        ;; We need a temporary register to facilitate the stack exchange;
        ;; we choose reg. 31 for no particular reason.  The net change to
        ;; it is nil.  This all works but could probably be made
        ;; considerably more efficient.
        `((exregstack (reg 31) ,loc1)
          (exregstack (reg 31) ,loc2)
          (exregstack (reg 31) ,loc1))))))

;; Would save a lot of ADDI instructions if I changed this to modify the
;; stack pointer before but not after; and instead change the environment
;; to reflect correct new variable locations and amount of stack adjustment
;; from original.  But perhaps it would be better to leave the stack
;; pointer alone and get rid of adjacent ADDIs via a later peephole
;; optimization or something.
(defconstruct exregstack (reg stackloc)
  `(with-SP-adjustment ,(offset stackloc)
        (exch ,reg (reg 1))))

;; Given a register, push its contents onto the stack.  Not currently used.
(defconstruct push (reg)
  `((exch ,reg (reg 1)) ;; <-- Convention: r1 is stack pointer.
    (++ (reg 1))))

;;;--------------------------------------------------------------------
;;; Pure register manipulation.

;; swapregs R1 R2 - Given two registers, swap their contents.
(defconstruct swapregs (r1 r2)
  ;; Implementation for architectures that support XOR'ing regs, but not
  ;; swapping regs directly.  Another way uses +=, -=, and NEG but takes 4
  ;; instructions.
  `((,r1 ^= ,r2)
    (,r2 ^= ,r1)
    (,r1 ^= ,r2)))

;; Fast way to move r1 to r2 when r2 is known to be empty!  Otherwise
;; behavior is "undefined" (actually in this implementation r1 gets r2, but
;; r2 ends up with r2^r1).
(defconstruct movereg (r1 r2)
  `((,r2 ^= ,r1)
    (,r1 ^= ,r2)))
```

D.4.7 `variables.lisp`

Defines high- to low-level constructs for manipulation of variables in variable assignments (environments).

```
;;; -*- Package: user -*-
(in-package "USER")
;;; ---------------------------------------------------------------------
;;; Constructs for variable-environment manipulation (creating/destroying
;;; variables, changing their locations, etc.)
;;; ---------------------------------------------------------------------

;;;--------------------------------------------------------------------
;;; User-level constructs.

;; Create a new variable VAR, bind it to VAL, execute the BODY, and then
```

```
;; unbind it from VAL and get rid of it.  VAL may be an expression, but it
;; must evaluate to whatever value VAR actually has at the end of the BODY,
;; or else all bets are off!
;;
;; 6/3/97 - Now LET is slightly more general---VAR can be put into a
;; relationship with VAL in any of a number of ways... <-, <->, ^=, +=...
;; <-, ^=, += are all equivalent given that SCOPE forces VAR to initially
;; be zero, but <-> is different... It sets VAR by swapping it with VAL,
;; which obviously must be a location of some sort.  Afterwards VAR is
;; restored to zero by swapping it back.  Note that in this case, if VAR is
;; not left at zero by the BODY, this is fine and results in VAL being
;; side-effected.  In other words, this kind of LET is effectively
;; temporarily giving VAL a new name which pulls it into a register if say
;; it was originally an array entry.  Another kind of operation (not yet
;; defined) would have VAL be a variable and assign VAR to be truly a
;; synonym for that exact same variable.

(defconstruct let ((var ~ val) &body body)
  (if (eq ~ '<-)
      `(scope ,var
          (_with (,var <- ,val)
                 . ,body))
    `(scope ,var
        (with (,var ,~ ,val)
              . ,body))))

;; Declare some vars that should be allocated register locations as soon
;; as they are created.
(defconstruct with-regvars (var-or-vars &body body)
  `(scope ,var-or-vars
     (register ,var-or-vars)
     . ,body))

;; User-level hint to compiler: put the following variables in registers
;; now rather than later.
(defconstruct register (var-or-vars)
  (let ((varlist (if (listp var-or-vars) var-or-vars (list var-or-vars))))
    (mapcar #'(lambda (var) `(get-in-register ,var)) varlist)))

;;;-----------------------------------------------------------------------
;;; Intermediate-level constructs.  Not recommended for casual users.

(defconstruct with-location-map (locmapdesc &body body)
  `((locmap ,locmapdesc)
    ,@body
    (locmap ,locmapdesc)))

;; WITH-ENVIRONMENT envdesc body - Ensure that the environment, as far as
;; location maps go, is equivalent to the one specified by environment
;; description ENVDESC, both at the beginning and at the end of the body.
(defconstruct with-environment (envdesc &body body)
  `((environment ,envdesc)
    ,@body
    (environment ,envdesc)))

;; ENVIRONMENT envdesc - Ensure that the environment is equivalent to the
;; one specified by environment description ENVDESC.  ENVDESC must describe
;; an environment object.  Currently the only supported kind of description
;; is an environment object itself.  Environments are equivalent if they
;; have the same variables in the same locations.  ENVIRONMENT will move
;; variables around as necessary to make the environments match, but it
```

```
;; will not create or destroy any variables.  If the environments cannot be
;; made to match, currently a compiler error is generated.
(defconstruct environment (envdesc)
  (if (equal-env? env envdesc)
      (values '() envdesc)  ;Tell RCOMP the requested form of the description.
    (if (equal-vars? env envdesc)       ;Do the envs have the same variables?
        ;; Relocate the first mis-located variable, and try again.
        (let ((v (first-misloc env envdesc)))
          (if (null (location v envdesc))
              `(environment ,(set-loc v (location v env) envdesc))
            `((relocate ,v ,(location v envdesc))
              (environment ,envdesc))))
      (error "Environments ~s and ~s don't match." env envdesc))))

(defconstruct locmap (locmapdesc)
  (setf env2 (copy-environment env)
        (locmap env2) locmapdesc)
  `(environment ,env2))

(defconstruct declare-locmap (locmapdesc)
  (setf env (copy-environment env)
        (locmap env) locmapdesc)
  (values nil env))

(defconstruct declare-environment (envdesc)
  (setf env (copy-environment envdesc))
  (values nil env))

;; The given variable should exist throughout the body of the scope, no
;; more, no less.  The body must leave the variable clear, or else!
(defconstruct scope (var &body body)
  (let ((vlist (if (listp var) var (list var))))
    `(with ,(mapcar #'(lambda (var) `(add-to-env ,var)) vlist)
           ,@body))
  ;; Note danger if var is not actually clear at end of BODY!
  )

;; ENSURE-GREEN enforces environmental correctness -- it leaves the
;; environment just the way it found it. (With regards to its location map.)
(defconstruct ensure-green (&body body)
  `(,@body
    (locmap ,(locmap env))))

;; DECLARE-GREEN declares that the environment in effect when the body
;; is entered will necessarily be in effect when it ends.  Don't use this
;; when it isn't true!
(defconstruct declare-green (&body body)
  (values body env))

;; Make the LOCATION be clear, and associate it with a new variable VARNAME.
(defconstruct new-var-at (varname location)
  (if (defined-in-env? varname env)
      (error "Variable ~s is already in the environment!" varname)
    `((vacate ,location)
      (tell-loc ,varname ,location))))

;; Make a particular location LOC become available (empty, and no variable
;; assigned to it).
(defconstruct vacate (loc)
  (let ((v (var-at-loc loc env)))
    (if v
```

```
        ;; Location is occupied by variable V.
        (let ((reg (next-avail-reg env)))
          ;; If any registers are available, move V there.
          (if reg `(relocate ,v ,reg)
            ;; Else move V to the next available stack location.
            (let ((s (next-avail-stack env)))
              (if s `(relocate ,v ,s))))))))

;; Arrange for the given variable, which should already be present in the
;; environment, to be located in a register (instead of on the stack).
(defconstruct get-in-register (var)
  (if (symbolp var)
      (let ((l (location var env)))
        (if (not (register? l))         ;If it's not already in a register,
            (let ((reg (next-avail-reg env)))
              (if reg       ;If there is a register avaiable, put it there.
                  (if (null-loc? l)
                      `(tell-loc ,var ,reg)
                    `(relocate ,var ,reg))
                ;; Else boot out the least-recently moved variable.
                (let* ((victim (least-recently-moved env))
                       (loc (location victim env)))
                  (if (null-loc? l)
                      `((vacate ,loc)
                        (tell-loc ,var ,loc))
                    `(relocate ,var ,loc))))))))

;;;----------------------------------------------------------------------
;;; Primitive environment-manipulation constructs.

;; Create the given variable in the environment, but don't give it a
;; location quite yet.
(defconstruct add-to-env (var) :opposite remove-var
  (if (defined-in-env? var env)
      (error "Variable ~s already exists!" var)
    (values '() (set-loc var nil env))))

;; Change the current environment to have the location of variable VAR as
;; being LOC.  Generates no code.  This construct is dangerous if the old
;; location of VAR has a non-zero runtime value, and is not associated with
;; any other variable.
(defconstruct tell-loc (var loc)
  (values '() (set-loc var loc env)))

;; Assuming that a variable is empty, remove it from the environment!
;; (Danger, Will Robinson!)  This causes grave problems if the runtime
;; value of the variable is not zero.  But currently we generate no runtime
;; code to notice that condition, so watch out!
(defconstruct remove-var (varname) :opposite add-to-env
  (values '() (remove-var varname env)))
```

D.4.8 branches.lisp

Constructs providing intermediate- and low-level support for various kinds of branch structures for control-flow.

```
;;; -*- Package: user -*-
(in-package "USER")
```

```
;;;-----------------------------------------------------------------
;;; Support for various kinds of branches.
;;; These constructs are not intended to appear in source code,
;;; but are rather used to implement higher-level control-flow
;;; constructs.
;;;-----------------------------------------------------------------

;;;-----------------------------------------------------------------
;;; Relatively high-level branch constructs.

;; paired binary conditional switching branches.  The body in between the
;; two branches must conserve the environment or else we won't have a
;; definite compile-time idea of the environment after the branch pair
;; because we don't know whether the branch succeeds or fails at run-time
;; or not.  Note the vars must really be variables and not literals or
;; registers.
(defconstruct bcs-branch-pair (toplab (vara1 ~a vara2)
                               botlab (varb1 ~b varb2)
                               &body body)
  `(;; All the variables involved need to be in registers before we start.
    (get-in-register ,vara1)
    (get-in-register ,vara2)
    (get-in-register ,varb1)
    (get-in-register ,varb2)
    (bcs-branch (,vara1 ,~a ,vara2) ,toplab ,botlab)
    ;; Since the vars were already in registers, BCS-BRANCH will not have
    ;; modified the environment after the branch point.
    (ensure-green ;Complain if the body doesn't clean up after itself.
     ,@body)
    ;; Due to the above GET-IN-REGISTERs and the ENSURE-GREEN, the b
    ;; variables will already be in registers here, so this BCS-BRANCH will
    ;; not need to modify the environment at all from the current one,
    ;; which is identical to the one just before the branch point.
    (bcs-branch (,varb1 ,~b ,varb2) ,botlab ,toplab)))

;; (The following construct is currently not used.)  Twin (meaning with
;; identical tests and variables) binary-operator conditional switching
;; branches.  The body in between the two branches must conserve the
;; environment or else we won't have a definite compile-time idea of the
;; environment after the branch pair because we don't know whether the
;; branch succeeds or fails at run-time or not.
(defconstruct twin-bcs-branch ((var1 ~ var2) toplab botlab &body body)
  `((get-in-register ,var1)
    (get-in-register ,var2)
    (bcs-branch (,var1 ,~ ,var2) ,toplab ,botlab)
    ;; bcs-branch had better not itself change the environment
    ;; after it branches!
    (ensure-green
     ,@body)
    (bcs-branch (,var1 ,~ ,var2) ,botlab ,toplab)))

;; Twin unconditional switching branches.
;; Now that the environment contains information other than
;; the locations of run-time variables, does it make
;; sense for it to be completely green?  Declaring static variables
;; inside the body might be expected to be able to affect the
;; outside world.  Or, maybe it shouldn't.  Haven't decided yet.
(defconstruct twin-us-branch (toplab botlab &body body)
  ;; TWIN-US-BRANCH is GREEN on the outside because encountering it from
  ;; the outside you just jump over it, and the environment doesn't change.
  `(declare-green
```

```
    (sbra-pair ,toplab ,botlab . ,body))
  ;; We have no idea what the environment is inside the first bra, though
  ;; (cuz we might slip into the middle as a subroutine call), so we can't
  ;; put an ensure-green or anything in there.  The environment in effect
  ;; at this point, which just leeches in from above, is usually wrong.
  ;; Still, the body needs to be careful that whatever environment is in
  ;; effect at its end is the same as the one it assumes at its top.  But
  ;; this is a job for a higher level to worry about.
  )

;;;----------------------------------------------------------------------
;;; lower-level branch constructs.

;;; Note to self: I think that the way branches and environments
;;; interact may be incorrect.  If the variables mentioned in the
;;; branch are not in registers, then the environment needs to change
;;; to allow the branch to be done---so the environment declared
;;; at the place we're branching to may be wrong.  Need to go thru
;;; and fix carefully.  Really, need a more sophisticated approach
;;; to how environments are kept track of during compilation of
;;; complex control-flow structures.

;; This low-level thing doesn't worry about environments at all.
;; That's a job for higher-level dudes that use it.
(defconstruct sbra-pair (toplab botlab &body body)
  `((sbra ,toplab ,botlab)
    ,@body
    (sbra ,botlab ,toplab)))

;;
;; SBRA: Switching branch (unconditional).  Branch is
;; from label thislab to otherlab.
;;
;; Semantics is: if we branch to this statement from
;; otherlab, then continue forwards normally.  If we
;; arrive at this statement normally, then branch to
;; otherlab.  If we arrive at this statement some other
;; way, results are undefined.
;;
(defconstruct sbra (thislab otherlab)
  `((label ,thislab)  ;Convention: label precedes labeled statement.
    (bra ,otherlab))) ;Assume that the hardware gives the semantics we want.

(defconstruct sbez (thislab var lab)
  ;; There's no built-in BEZ, so we reserve reg $0 to be zero
  ;; so we can just use BEQ instead.
  `(bcs-branch (,var = (reg 0)) ,thislab ,lab))

(defconstruct sbnz (thislab var lab)
  ;; There's no built-in BNZ, so we reserve reg $0 to be zero
  ;; so we can just use BNE instead.
  `(bcs-branch (,var != (reg 0)) ,thislab ,lab))

(defconstruct bcs-branch ((var1 ~ var2) thislab otherlab)
  (cond
   ;; If the variables aren't in registers, get them there.  The only thing
   ;; is, I'm not sure it makes sense to do any environment manipulation at
   ;; this level because it makes it difficult for the higher level stuff
   ;; to ensure thtat the environment is consistent at both ends of the
   ;; actual branch point.
   ((and (member ~ '(= !=)) (numberp var2) (zerop var2))
```

```
     ;; To compare for equality with zero, compare with $0 (reserved for 0).
     `(bcs-branch (,var1 ,~ (reg 0)) ,thislab ,otherlab))
    ((and (or (register? var1)
              (in-register? var1 env)
              (eql var1 0))
          (or (register? var2)
              (in-register? var2 env)
              (eql var2 0)))
     `((label ,thislab)
       ;; This assumes that BC-BRANCH has the right switching sort of
       ;; semantics, and that it doesn't insert any instructions before the
       ;; actual branch.
       (bc-branch (,var1 ,~ ,var2) ,otherlab)))
    ;; Does this make sense?
    ((and (symbolp var1) (defined-in-env? var1 env)
          (not (in-register? var1 env)))
     `((get-in-register ,var1)
       (bcs-branch (,var1 ,~ ,var2) ,thislab ,otherlab)))
    ((and (symbolp var2) (defined-in-env? var2 env)
          (not (in-register? var2 env)))
     `((get-in-register ,var2)
       (bcs-branch (,var1 ,~ ,var2) ,thislab ,otherlab)))
    ((and (symbolp var1) (not (defined-in-env? var1 env)))
     `((add-to-env ,var1)
       (bcs-branch (,var1 ,~ ,var2) ,thislab ,otherlab)))
    ((and (symbolp var2) (not (defined-in-env? var2 env)))
     `((add-to-env ,var2)
       (bcs-branch (,var1 ,~ ,var2) ,thislab ,otherlab)))
    ;; I could go on and handle expressions and literals as well, but again
    ;; I'm concerned that this isn't the right level.
    ))

;; binary conditional branch
(defconstruct bc-branch ((var1 ~ var2) destlab)
  (cond
    ((and (or (register? var1) (eql var1 0))
          (or (register? var2) (eql var2 0)))
     `(rbc-branch (,var1 ,~ ,var2) ,destlab))
    ;; If the variables are in registers, just look at the registers.
    ((and (symbolp var1) (in-register? var1 env))
     `(bc-branch (,(location var1 env) ,~ ,var2) ,destlab))
    ((and (symbolp var2) (in-register? var2 env))
     `(bc-branch (,var1 ,~ ,(location var2 env)) ,destlab))
    ;; We can't really do any environment manipulation at this level
    ;; because the instructions manipulated will come between us and the
    ;; label inserted by BCS-BRANCH.
    (t
     (error "BC-BRANCH can only cope with variables residing in registers!")
     )))

;;;-----------------------------------------------------------------------
;;; Branching primitives.

;; A label is a tag that gives an address in code that is a target
;; for branching.
(defconstruct label (labname)
  labname)

(defconstruct bez (reg lab)
  ;; There's no built-in BEZ, so we reserve reg $0 to be zero
  ;; so we can just use BEQ instead.
```

```
  `(rbc-branch (,reg = (reg 0)) ,lab))

(defconstruct bnz (reg lab)
  ;; Similar, use BNE.
  `(rbc-branch (,reg != (reg 0)) ,lab))

(defparameter *bc-branch-instructions*
  '((!= . bne) (= . beq)))

(defparameter *zerocmp-branch-instructions*
  '((>= . bgez) (<= . blez) (> . bgtz) (< . bltz)))

;; "register binary conditional branch"
;; tests whether two registers satisfy some relation ~ and if so
;; branch to DESTLAB.
(defconstruct rbc-branch ((reg1 ~ reg2) destlab)
  (cond
   ((assoc ~ *bc-branch-instructions*)
    `(,(cdr (assoc ~ *bc-branch-instructions*))
      ,reg1 ,reg2 ,destlab))
   ((and (eql reg2 0)
         (assoc ~ *zerocmp-branch-instructions*))
    `(,(cdr (assoc ~ *zerocmp-branch-instructions*))
      ,reg1 ,destlab))))
```

D.4.9 expression.lisp

Constructs and low-level functions for expanding nested expressions.

```
;;; -*- Package: user -*-
(in-package "USER")

;; This version of WITH allows any temporary effect, not just
;; variable binding, to be done and undone around the body.
(defconstruct with (statement &body body)
  `(,statement
    ,@body
    (undo ,statement)))

;; Given any statements, do their reverse (undoing their effects).  Don't
;; depend too much on this always working yet.
(defconstruct undo (&rest statements)
  (unless (null statements)
    (dbind (first . rest) statements
      (if (list-of-statements? first)
          `(undo ,@first . ,rest)
        (let ((statement (canonicalize first)))
          `((undo . ,rest)
            (,(opposite (first statement)) . ,(rest statement))))))))

;; Maps "expanding" binary operators to their do/undo statements.
(defparameter *expanding*
  '((& ^=&) (<< ^=<<) (>> ^=>>) (* +=* -=*) (*/ +=*/ -=*/) (_ <-_ ->_)))
(defun forw (binop)
  (cadr (assoc binop *expanding*)))
(defun revers (binop)
  (or (caddr (assoc binop *expanding*))
      (cadr (assoc binop *expanding*))))
```

```
(defun expression? (obj)
  (and (listp obj)
       (not (location? obj))
       (not (statement? obj))))

;; This version of with doesn't evaluate expressions as many times.
;; but uses up linear space during body.  It only handles <- (bind) type
;; statements though.
(defconstruct _with ((var <- val) &body body)
  (cond
   ((or (not (expression? val))
        (literal? val env))
    `(with (,var <- ,val)
           . ,body))
   ((null (cddr val)) ; No more than 2 words in value expression.
    (cond
     ((eq (first val) '*)
      ;; These expansions are a bit questionable because what if the body
      ;; tries to look at the dereferenced value also?  It will see 0 (or
      ;; whatever was in VAR) instead.  But the alternative, of introducing
      ;; yet another temporary and swapping the contents back before doing
      ;; the body, seems too inefficient.
      (if (or (register? (second val))
              (dynamic-var? (second val) env))
          `((,var <->* ,(second val))
            ,@body
            (,var <->* ,(second val)))
        (let ((tv (gentemp)))
          `(_with (,tv <- ,(second val))
              `((,var <->* ,tv)
                ,@body
                (,var <->* ,tv))))))))
   (t
    (dbind (a1 ~ a2) val  ;But what about other expressions?
      (let ((rb (revers ~))
            (fb (forw ~)))
        (cond
         ((and (numberp a1) (numberp a2))
          `((,var <- ,(funcall ~ a1 a2)) ;Warning: this is too simplistic.
            ,@body
            (,var -> ,(funcall ~ a1 a2))))
         ((eq ~ '+)
          `(with ((,var += ,a1)
                  (,var += ,a2))
            ,@body))
         ((and (eq ~ '+) (not (expression? a1)))
          `(_with (,var <- ,a2)
              (with (,var += a1)
                 ,@body)))
         ((and (eq ~ '+) (not (expression? a2)))
          `(_with (,var <- ,a1)
              (with (,var += ,a2)
                 ,@body)))
         ((and (eq ~ '-) (not (expression? a2)))
          `(_with (,var <- ,a1)
              (with (,var -= ,a2)
                 ,@body)))
         ((and (expression? a1) (expression? a2))
          (let ((tv1 (gentemp))
                (tv2 (gentemp)))
            `(let (,tv1 <- ,a1)
```

```
              (let (,tv2 <- ,a2)
                (,var ,fb ,tv1 ,tv2)
                ,@body
                (,var ,rb ,tv1 ,tv2)))))
        ((expression? a1)
         (let ((tv (gentemp)))
           `(let (,tv <- ,a1)
              (,var ,fb ,tv ,a2)
              ,@body
              (,var ,rb ,tv ,a2))))
        ((expression? a2)
         (let ((tv (gentemp)))
           `(let (,tv <- ,a2)
              (,var ,fb ,a1 ,tv)
              ,@body
              (,var ,rb ,a1 ,tv))))
        (t
         `((,var ,fb ,a1 ,a2)
           ,@body
           (,var ,rb ,a1 ,a2)))))))))

;;;----------------------------------------------------------------------
;;; Variable binding and unbinding.  For most purposes this takes the
;;; place of assignment.

;; Semantics of BIND: assuming that VAR is already clear,
;; set it to VAL.
(definfix (var <- val) :opposite ->
  ;; Implemented using +=, but ^= would also work.
  (cond
   ((or (symbolp val) (numberp val) (register? val) (literal? val env))
    `(,var += ,val))
   ((expression? val)
    ;; Binary expression.
    (destructuring-bind (a1 ~ a2) val   ;What about other syntaxes?
      (cond
       ((and (numberp a1) (numberp a2))
        `(,var += ,(funcall ~ a1 a2)))  ; Really too simplistic.
       ((eq ~ '+)
        `((,var += ,a1)
          (,var += ,a2)))
       ((eq ~ '-)
        `((,var += ,a1)
          (,var -= ,a2)))
       ((eq ~ '^)
        `((,var ^= ,a1)
          (,var ^= ,a2)))
       ((eq ~ '*)
        `(,var +=* ,a1 ,a2))
       ((assoc ~ *expanding*)
        `(,var ,(forw ~) ,a1 ,a2))
       ((extract form env :lvalues 1))
       (t
        (error "Don't know how to compile ~s." form)))))))

;; Assuming that VAR=VAL, restore it to zero.
(definfix (var -> val) :opposite <-
  (cond
   ((or (symbolp val) (numberp val) (register? val) (literal? val env))
    `(,var -= ,val))
   ((expression? val)
```

```
    ;; Binary expression.
    (destructuring-bind (a1 ~ a2) val
      (cond
       ((and (numberp a1) (numberp a2))
        `(,var -= ,(funcall ~ a1 a2)))  ; Really too simplistic.
       ((eq ~ '+)
        `((,var -= ,a1)
          (,var -= ,a2)))
       ((eq ~ '-)
        `((,var -= ,a1)
          (,var += ,a2)))
       ((eq ~ '^)
        `((,var ^= ,a1)
          (,var ^= ,a2)))
       ((eq ~ '*)
        `(,var -=* ,a1 ,a2))
       ((assoc ~ *expanding*)
        ;; Use the appropriate reverse op if different from forward one.
        `(,(revers ~) ,var ,a1 ,a2))
       ((extract form env :lvalues 1))
       (t
        (error "Don't know how to compile ~s." form)))))))


;;; ----------------------------------------------------------------------
;;; New thingy.  constructs all use this same function EXTRACT to
;;; automatically replace located variables with their locations, move
;;; stack variables into registers before operating on them, create
;;; temporary variables for subexpressions and compute their values.

;; EXTRACT - you give it a form, and it returns code that
;; creates appropriate temporary stuff around it and gets variables
;; in registers together with a reduced version of the original
;; form.
;;    RELEVANT-TERMS is a list of the indices (as per NTH or ELT) of those
;; terms that are candidates for expanding.  If not provided, all terms are
;; considered fair game.
;;    LVALUES is similarly the index of or a list of the indices of
;; a term or terms that are considered to be "lvalues", that is,
;; "destinations" where the value of the term is changed by the
;; statement.  Anything in LVALUES is automatically also a candidate
;; for expanding.
;;    The indices in both RELEVANT-TERMS and LVALUES refer to the
;; indices the terms have *after* any canonicalization.
;;    NIL is returned if EXTRACT can't do anything.

(defun extract (form env &key lvalues (relevant-terms
                                       (labels ((countlist (n)
                                                  (if (>= n 0)
                                                      (cons n
                                                        (countlist
                                                         (1- n))))))
                                         (countlist (length form)))))
  (setf form (canonicalize form));So we can forget about infix.
  (labels ((lvalue? (index)
             "Return non-NIL if the given index is the index of
              a term that is an LVALUE (modifiable term)."
             (or (eql index lvalues)
                 (and (listp lvalues)
                      (member index lvalues))))
           (candidate? (index)
```

```
        "Return non-NIL if the given index is the index of a
         term that is a candidate for expansion."
        (or (member index relevant-terms)
            (lvalue? index))))
;; First we locate the first term that is an expression or
;; a literal, and convert it into a temporary variable.
(let (before)
  (setf (cdr form)
        (repl2 (cdr form)
              #'(lambda (term index)
                  (if (and (candidate? index)
                           (or (expression? term)
                               (literal? term env)))
                      (let ((tv (gentemp)))
                        (if (lvalue? index)
                            (setf before `(,tv <-> ,term))
                          (setf before `(,tv <- ,term)))
                        tv)))
              1))
  (when before
    (return-from extract `(let ,before ,form))))
;; If any term is a static value identifier, wrap the statement in
;; a binding of a temporary to the value's address and replace the
;; term with a dereferencing expression.
(let (before)
  (setf (cdr form)
        (repl2 (cdr form)
              #'(lambda (term index)
                  (if (and (candidate? index)
                           (static-val? term env))
                      (let ((tv (gentemp)))
                        (setf before `(,tv <- (& ,term)))
                        `(* ,tv))))
              1))
  (when before
    (return-from extract `(let ,before ,form))))
;; Now look for variables and make sure they are in the environment.
;; If not, add them, but don't remove them afterwards.  This lets
;; user refrain from explicitly adding variables although he will
;; still have to get rid of them manually.
(let ((index 1))
  (dolist (term (cdr form))
    (when (and (dynamic-var? term env)
               (candidate? index)
               (not (defined-in-env? term env)))
      (return-from extract
        `((add-to-env ,term)
          ,form)))
    (incf index)))
;; Next, get any mentioned variables into registers.
(let ((index 1))
  (dolist (term (cdr form))
    (when (and (dynamic-var? term env)
               (candidate? index)
               (not (in-register? term env)))
      (return-from extract
        `((get-in-register ,term)
          ,form)))
    (incf index)))
;; Finally, replace variables with the registers they're in.
(let (found?)
```

```
    (setf (cdr form)
          (repl2 (cdr form)
                 #'(lambda (term index)
                     (when (and (dynamic-var? term env)
                                (candidate? index)
                                (in-register? term env))
                       (true! found?)
                       (location term env)))
                 1))
    (if found? form))))              ;End function EXTRACT.
```

D.4.10 clike.lisp

Defines constructs for various user-level user-level C-like operators.

```
;;; -*- Package: user -*-
(in-package "USER")

;;;----------------------------------------------------------------------
;;; C-like assignment statements.

(definfix (var ++)
  `(,var += +1))

;; "-" statement: negate the given lvalue in place.
(defconstruct - (var)
  (cond
   ((register? var)
    `(neg ,var))
   (t
    (extract form env :lvalues 1))))

(definfix (var <=< val)
  (cond
   ((and (numberp val) (zerop val))
    '())
   ((and (register? var) (static-val-addr? val env))
    `(rl ,var ,(second val)))
   ((and (register? var) (literal? val env))
    `(rl ,var ,val))
   ((and (register? var) (register? val))
    `(rlv ,var ,val))
   (t
    (extract form env
             :relevant-terms (if (not (literal? val env)) '(2))
             :lvalues 1))))

(definfix (var >=> val)
  (cond
   ((and (numberp val) (zerop val))
    '())
   ((and (register? var) (static-val-addr? val env))
    `(rr ,var ,(second val)))
   ((and (register? var) (literal? val env))
    `(rr ,var ,val))
   ((and (register? var) (register? val))
    `(rrv ,var ,val))
   (t
```

```
    (extract form env
             :relevant-terms (if (not (literal? val env)) '(2))
             :lvalues 1))))

(definfix (var += val) :opposite -=
  (cond
   ((and (numberp val) (zerop val))
    '()) ;Optimization: don't add 0
   ((and (register? var) (static-val-addr? val env))
    `(addi ,var ,(second val)))
   ((and (register? var) (literal? val env))
    `(addi ,var ,val))
   ((and (register? var) (register? val))
    `(add ,var ,val))
   (t
    (extract form env
             :relevant-terms (if (not (literal? val env)) '(2))
             :lvalues 1))))

(definfix (var -= val) :opposite +=
  (cond
   ((and (register? var) (numberp val))
    `(,var += ,(- val)))                ;No SUBI instruction.
   ((and (register? var) (static-array? val env))
    `(,var += ,(intern (concatenate 'string "-" (symbol-name val)))))
   ((and (register? var) (static-val-addr? val env))
    `(addi ,var ,(intern (concatenate 'string "-"
                                      (symbol-name (second val))))))
   ((and (register? var)
         (literal? val env)
         (negated-sym? val))
    `(,var += ,(positive-of val)))
   ((and (register? var) (register? val))
    `(sub ,var ,val))
   (t
    (extract form env
             :relevant-terms (if (not (literal? val env)) '(2))
             :lvalues 1))))

;; Very much like +=.  More abstraction?
(definfix (var ^= val)
  (cond
   ((and (numberp val) (zerop val))
    '()) ;Optimization: don't add 0
   ((and (register? var) (numberp val))
    `(xori ,var ,val))
   ((and (register? var) (register? val))
    `(xor ,var ,val))
   (t
    (extract form env
             :relevant-terms (if (not (numberp val)) '(2))
             :lvalues 1))))

(definfix (dest ^=& src1 src2)
  (cond
   ((and (register? dest) (register? src1) (register? src2))
    `(andx ,dest ,src1 ,src2))
   (t
    (extract form env :lvalues 1))))

(definfix (dest ^=<< src1 src2)
```

```
  (cond
   ((and (register? dest) (register? src1) (register? src2))
    `(sllvx ,dest ,src1 ,src2))
   (t
    (extract form env :lvalues 1))))

;; This is a logical shift right.
(definfix (dest ^=>> src1 src2)
  (cond
   ((and (register? dest) (register? src1) (register? src2))
    `(srlvx ,dest ,src1 ,src2))
   (t
    (extract form env :lvalues 1))))

;; (base _ offset) gets transformed into this, where dest is
;; some temporary register.
(definfix (dest <->_ base offset)
  ;; THIS IS WRONG!  Extracting before expanding runs the danger
  ;; that the variable assignments used during the extraction
  ;; could be invalidated during the body of the WITH, I think 6/3/97
  (or (extract form env :lvalues 1)
      `(with (,base += ,offset)
          (,dest <->* ,base))))

(definfix (dest <-_ base offset) :opposite ->_
  `(,dest <-* (,base + ,offset)))

(definfix (dest ->_ base offset) :opposite <-_
  `(,dest ->* (,base + ,offset)))

(definfix (dest <-* ptr)
  ;; THIS IS WRONG!  Extracting before expanding runs the danger
  ;; that the variable assignments used during the extraction
  ;; could be invalidated during the header of the LET.
  (or
   (extract form env :lvalues 1)
   (let ((tv (gentemp)))
     `(let (,tv <->* ,ptr)
        (,dest <- ,tv)))))

(definfix (dest ->* ptr)
  (or
   (extract form env :lvalues 1)
   (let ((tv (gentemp)))
     `(let (,tv <->* ,ptr)
        (,dest -> ,tv)))))

(definfix (left <->* rightptr)
  (cond
   ((extract form env :lvalues 1))
   ((and (register? left) (register? rightptr))
    `(exch ,left ,rightptr))))

(definfix (var +=* val1 val2) :opposite -=*
  `(call mult ,var ,val1 ,val2))

(definfix (var -=* val1 val2) :opposite +=*
  `(rcall mult ,var ,val1 ,val2))

(definfix (var +=*/ val1 val2) :opposite -=*/
  `(call _smf ,val1 ,val2 ,var)) ;Note dest is last.
```

```
(definfix (var -=*/ val1 val2) :opposite +=*/
  `(rcall _smf ,val1 ,val2 ,var))

(definfix (left <-> right)
  (cond
   ((and (location? left) (location? right))
    `(swaploc ,left ,right))
   ((and (listp right)
         (>= (length right) 3)
         (dbind (base ~ offset) right
           (if (eq ~ '_)
               `(,left <->_ ,base ,offset)))))
   ((and (listp left)
         (>= (length left) 3)
         (dbind (base ~ offset) left
           (if (eq ~ '_)
               `(,right <->_ ,base ,offset)))))
   ((extract form env :lvalues '(1 2)))))

(defun << (a b)
  (ash a b))
```

D.4.11 print.lisp

Defines a few very simple constructs for producing output.

```
;;; -*- Package: user -*-
(in-package "USER")

(defconstruct printword (val)
  `((rawprint 0)
    (rawprint ,val)))

(defconstruct println ()
  `((rawprint 1)))

(defconstruct rawprint (val)
  (cond
   ((register? val)
    `(output ,val))
   (t
    (extract form env))))
```

D.4.12 controlflow.lisp

Defines user-level to intermediate-level control flow constructs such as conditionals and looping.

```
;;; -*- Package: user -*-
(in-package "USER")

;;; ----------------------------------------------------------------------
;;; High-level control flow constructs suitable for user use.

;; if CONDEXPR then
```

```
;;    BODY...
;; [else BODY2...]
;;
;; CONDEXPR is evaluated; if result is nonzero body is executed.  In either
;; case, CONDEXPR is then evaluated in reverse.  The value should be the
;; same whether or not BODY was executed, or else behavior undefined.

;; NOTE 6/26/97: IF and all its subsidiary branching constructs need to be
;; completely cleaned up and reorganized.  One big thing is that code for
;; computing EXPR in a condition expression of a form like (EXPR > 0) needs
;; to be wrapped around the entire IF.  And things like (EXPR1 > EXPR2)
;; need to be transformed into ((EXPR1 - EXPR2) > 0).

(defconstruct if (condexpr then &body body)
  (cond
   ((member 'else body)
    (let ((ifpart (subseq body 0 (position 'else body)))
          (elsepart (subseq body (1+ (position 'else body)))))
      `(ifelse ,condexpr ,ifpart ,elsepart)))
   ((and (listp condexpr)
         (null (cdddr condexpr))
         (member (second condexpr) '(= != > <= < >=)))
    `(_if ,condexpr then . ,body))
   (t
    ;; The current version is appropriate only for testing an arbitrary
    ;; value to see if it is non-zero.  For other kinds of conditions,
    ;; other implementations would be more appropriate.
    (if (symbolp condexpr)
        `(_if (,condexpr != 0) then . ,body)
      (let ((tv (gentemp)))
        `(let (,tv <- ,condexpr)
           (_if (,tv != 0) then . ,body)))))))

;; for VAR = START to END
;;    BODY...
;;
;; Semantics: START and END are expressions.  They are each evaluated once
;; forwards at the beginning of the loop, and once in reverse at the end of
;; the loop.  They should return the same value both times.
;;
;; VAR is a fresh variable whose scope is the BODY.  It is set to START,
;; and then the BODY is executed.  If VAR is ever END after executing the
;; body, then the construct immediately terminates.  Otherwise, VAR is
;; incremented by 1 and the BODY is executed again.  START may be equal to
;; END, in which case the BODY is executed exactly once.  If the values of
;; START and END ever change during evaluation, or if BODY ever sets VAR to
;; START-1, the behavior of the entire program becomes undefined.

(defconstruct for (var = start wordto end &body body)
  (let ((top (gentemp "_FORTOP"))       ;Loop entry point.
        (bot (gentemp "_FORBOT"))       ;Bottom of loop.
        (stvar (gentemp "_FORSTART"))
        (endvar (gentemp "_FOREND")))   ;Loop boundary values.
    `(let (,stvar <- ,start)
          (let (,endvar <- (,end + 1))
               (scope ,var
                      (,var <- ,stvar)
                      ;; The loop itself.
                      (bcs-branch-pair ,top (,var != ,stvar)
                                       ,bot (,var != ,endvar)
                        ,@body
```

```
                      (,var ++))
                    (,var -> ,endvar))))))

;;;---------------------------------------------------------------------
;;; Medium-level control flow constructs not intended
;;; for direct user use.

;; InfLoop: Unconditional branch from bottom of body back to top.
;; Also, if we hit it from outside we jump over it.
(defconstruct infloop (&body body)
  `(twin-us-branch ,(gentemp "_LOOPTOP") ,(gentemp "_LOOPBOT")
                   . ,body))

(setf (get '=  'opposite) '!=
      (get '!= 'opposite) '=
      (get '>  'opposite) '<=
      (get '<= 'opposite) '>
      (get '<  'opposite) '>=
      (get '>= 'opposite) '<
      )

;;; _if (VAR ~ VAL) then BODY
(defconstruct _if ((reg1 ~ reg2) then &body body)
  (let ((l1 (gentemp "_IFTOP"))
        (l2 (gentemp "_IFBOT")))
    `(twin-bcs-branch (,reg1 ,(opposite ~) ,reg2) ,l1 ,l2
        ;; The body sure better not change whether var=0.
        ,@body)))

(defconstruct _ifelse ((reg1 ~ reg2) ifstuff elsestuff)
  (let ((iftop (gentemp "_IFTOP"))
        (ifbot (gentemp "_IFBOT"))
        (elsetop (gentemp "_ELSETOP"))
        (elsetop (gentemp "_ELSEBOT")))
    `((bcs-branch (,reg1 ,(opposite ~) ,reg2) ,iftop ,elsetop)
      (ensure-green . ,ifstuff)
      (sbra ,ifbot ,elsebot)
      (sbra ,elsetop ,iftop)
      (ensure-green . ,elsestuff)
      (bcs-branch (,reg1 ,~ ,reg2) ,elsebot ,ifbot))
    ))
```

D.4.13 subroutines.lisp

Provides high and low level support for subroutines.

```
;;; -*- Package: user -*-
(in-package "USER")

;;;---------------------------------------------------------------------
;;; Subroutine calling support.
;;;---------------------------------------------------------------------

;;;---------------------------------------------------------------------
;;; User-level constructs.

;; Defsub: Implements subroutine entry/return conventions.
(defconstruct defsub (subname arglist &body body)
```

```
  (let ((bodyenv (entryenv arglist env)))
  ;; We wrap it in a branch pair so that if we encounter it from the
  ;; outside we jump over it, and if it runs off its end it comes back to
  ;; the beginning.  This latter behavior facilitates calling a subroutine
  ;; with a single switching-branch to its entry/exit point.
  `((twin-us-branch ,(gentemp "_SUBTOP") ,(gentemp "_SUBBOT")
      ;; At start and end of body, environment is as according
      ;; to subroutine calling convention.
      (declare-environment ,bodyenv)
      (portal ,subname) ;Entry/exit point.
      ,@body
      (environment ,bodyenv)))))

;;;--------------------------------------------------------------------
;;; Subroutine calling convention support.

;; NOTE: Currently this does not work right for more than 29 arguments
;; (i.e. when some args need to go on the top of the stack instead of
;; in registers!).
(defconstruct call (subname &rest actualargs)
  `(withargs ,actualargs
     (with-stack-top
        (gosub ,subname))))

(defconstruct rcall (subname &rest actualargs)
  `(withargs ,actualargs
     (with-stack-top
        (rgosub ,subname))))

(defconstruct with-stack-top (&body body)
  (let ((offset (top-of-stack env)))
    `(with-sp-adjustment ,offset
        . ,body)))

(defconstruct with-SP-adjustment (amt &body body)
  ;; AMT must be a literal number.
  `(((reg 1) += ,amt)
    ,@body
    ((reg 1) -= ,amt)))

;; Call a subroutine at a low level with no mention of arguments.
(defconstruct gosub (subname) :opposite rgosub
  ;; A switching branch to the SWAPBRN in the portal should do the trick.
  `(bra ,subname))

(defconstruct rgosub (subname) :opposite gosub
  `(rbra ,subname))

;; Portal: Entry/exit point of a subroutine.
(defconstruct portal (label)
  `((label ,label)
    ;; We always use register $2 for storing our subroutine offsets, by
    ;; convention.
    (swapbr (reg 2))  ;retvar<->BR
    (neg (reg 2))     ;retvar = -retvar
    ))

;;; WITHARGS below needs some work.  It currently can only prepare the 29
;;; arguments that we can fit into registers.  I was intending that if
;;; there are more arguments they should be passed on the stack.  This is
;;; not too hard, but I'm not sure it's worth it.
```

```
;; Prepare arguments in conventional locations as for a subroutine call.
(defconstruct withargs (actualargs &body body)
  (let ((result `((vacate (reg 2))
                  . ,body))
        (r 1))
    (dolist (a actualargs)
      (if (and (symbolp a) (not (static-id? a env)))
          (if (defined-in-env? a env)
              (push `(relocate ,a ,(argno-to-location r)) result)
            (setf result
                  `((new-var-at ,a ,(argno-to-location r))
                    ,@result
                    (remove-var ,a))))
        (setf result
              (let ((tv (gentemp)))
                `((new-var-at ,tv ,(argno-to-location r))
                  ;; This prevents evaluating A from causing
                  ;; earlier-placed registers to change their locations.
                  ;; 6/3/97- But ENSURE-GREEN really enforces more than
                  ;; just this, unfortunately.
                  (ensure-green
                   (,tv <- ,a))
                  ,@result
                  (,tv -> ,a)
                  ;;^-DANGER! Assumes subroutine didn't change value of A.
                  (remove-var ,tv)))))
      (incf r))
    result))

;; On subroutine entry/exit, the environment contains:
;; A return-address variable located in register 2.
;; All the arguments in registers 3,4,... until we run out,
;; and then stack locations -1,-2,... (below top of stack).
;; If not all the registers were used for arguments, then
;; there is a variable for each unused one (above 2), used
;; to ensure that all these other registers are restored to
;; their original state upon exit.
(defun entryenv (arglist origenv)
  (let (locmap (r 0))
    (dolist (a (cons '_RET arglist))
      (push
       `(,a . ,(argno-to-location r))
       locmap)
      (incf r))
    (setf r (+ r 2))
    (if (<= r 31)
        (loop
         (push `(,(intern (concatenate 'string "_R" (princ-to-string r)))
                 reg ,r) locmap)
         (incf r)
         (if (> r 31) (return))))
    (setf locmap (reverse locmap))
    (let ((env (copy-environment origenv)))
      (labels
          ((is-arg? (name) (member name arglist)))
        (setf (locmap env) locmap
              (staticvals env) (remove-if #'is-arg? (staticvals env))
              (staticarrays env) (remove-if #'is-arg? (staticarrays env))))
      env)))
```

```
;; Convert an argument number (0 and up) to a location (reg <regno>)
;; or (stack <offset>).  Argument 0 is the return address.
(defun argno-to-location (argno)
  (if (<= argno 29)
      `(reg ,(+ argno 2))
    `(stack ,(- 29 argno))))
```

D.4.14 staticdata.lisp

Defines constructs for defining static data objects. Currently this is the only way to provide input to a program.

```
;;; -*- Package: user -*-
(in-package "USER")

;;;----------------------------------------------------------------------
;;; Constructs for declaring static data.

;; Define NAME to refer to a static word of data in memory
;; of value VALUE.
(defconstruct defword (name value)
  `(skip
    (staticval ,name)
    (label ,name)
    (dataword ,value)))

(defconstruct defarray (name &rest elements)
  `(skip
    (staticarray ,name)
    (label ,name)
    . ,(mapcar #'(lambda (elem) `(dataword ,elem)) elements)))

;;;----------------------------------------------------------------------

;; VALUE is a word of data that should be included at
;; this point in the program in literal form.
(defconstruct dataword (value)
  `(data ,value))

(defconstruct staticval (name)
  (values nil (add-staticval name env)))

(defconstruct staticarray (name)
  (values nil (add-staticarray name env)))

;; If the flow of control gets to code surrounded by SKIP it will skip over
;; the contents without executing them.
(defconstruct skip (&body body)
  ;; Implemented by an unconditional branch pair around the body.
  `(sbra-pair ,(gentemp "_PRESKIP") ,(gentemp "_POSTSKIP")
            . ,body))
```

D.4.15 program.lisp

Defines very high-level constructs for wrapping around the entire program.

```
;;; -*- Package: user -*-
(in-package "USER")

;;-----------------------------------------------------------------------
;;; Highest-level constructs.

(defconstruct defmain (progname &body body)
  `(;; Always include the standard library of subroutines.
    (standard-library)
    ;; We surround the whole program with a branch pair because I don't
    ;; think that our current idea of START/FINISH boundary instructions
    ;; can be non-noops on the real machine without dissipation. This
    ;; also skips over main if control somehow comes down from above.
    (twin-us-branch _MAINTOP _MAINBOT
      ;; Execution starts and ends with exactly 0 dynamic variables.
      (with-location-map ,(empty-locmap)
        (declare-startpoint ,progname)
        ;; To begin execution, the PC should initally be set to this label.
        (label ,progname)
        (start)
        ,@body
        (finish)))))

;;-----------------------------------------------------------------------

;; Defprog: a whole program with subroutines and a main routine.
;; Now deprecated in favor of defsub + defmain (6/26/97).
(defconstruct defprog (progname subs &body main)
  `(;; Always include the standard library of subroutines.
    (standard-library)
    ;; Include user subroutines.
    ,@subs
    ;; We surround the whole program with a branch pair because I don't
    ;; think that our current idea of START/FINISH boundary instructions
    ;; can be implemented without dissipation.
    (twin-us-branch _MAINTOP _MAINBOT
     ;; Execution starts and ends with exactly nothing in the environment.
     (with-location-map ,(empty-locmap)
       (declare-startpoint ,progname)
       ;; To begin execution, the PC should initally be set to this label.
       (label ,progname)
       (start)
       ,@main
       (finish)))))

;;-----------------------------------------------------------------------

(defconstruct declare-startpoint (labname)
  `(.start ,labname))
```

D.4.16 library.lisp

Defines constructs that expand into code for standard subroutine libraries. Currently the library is very minimal.

```
;;; -*- Package: user -*-
(in-package "USER")

(defconstruct standard-library ()
```

```
  ;; List of standardly available subroutines.
  '((def-smf)))

;; Define the simplest normal integer multiplication function.
;; This adds the low word of the product of unsigned integers
;; M1 and M2 into PROD.

(defconstruct def-mult ()
  '(defsub mult (m1 m2 prod)
    ;; Use grade-school algorithm.
    (for pos = 0 to 31                  ; For each of the 32 bit-positions,
       (if (m1 & (1 << pos)) then       ;   if that bit of m1 is 1, then
         (prod += (m2 << pos))))))      ;     add m2, shifted over to that
                                        ;       position, into p.

;; Define the signed multiplication-by-fraction function, which takes
;; two signed integers M1 and M2, and adds the high word of their true
;; integer product into PROD.  This is like multiplying M1 by M2 when
;; M2 is considered to represent a fraction with numerator explicit and
;; denominator 2^31.
;;
;; This version of the function was tested in C and seemed to work
;; satisfactorily, although more testing is needed.  Need to compare
;; with upper bits of true (64-bit-wide) product. 6/26/97

(defconstruct def-smf ()
  '(defsub _smf (m1 m2 prod)
     (with-regvars (m1p m2p mask shifted bit p)
       (with ((mask <- 1)
              (m1p <- m1) (if (m1 < 0) then (- m1p))
              (m2p <- m2) (if (m2 < 0) then (- m2p)))
          (mask <=< 31)
          (for position = 1 to 31
             (mask >=> 1)
             (with (bit <- (m1p & mask))
                (if bit then
                  (with (shifted <- (m2p >> position))
                     (p += shifted)))))
          (if (m1 < 0) then (- p))
          (if (m2 < 0) then (- p))
          (prod += p)))))
```

D.4.17 `files.lisp`

Provides support for reading the source code to compile from a file.

```
;;; -*- Package: user -*-
(in-package "USER")

;;;----------------------------------------------------------------------
;;; File compilation code.

(defun rcompile-file (filename &key debug)
  (let (source)
    (with-open-file (stream filename)
       (loop
        (let ((next-form
               (read stream nil :eof)))
```

```
          (cond
           ((eq next-form :eof)
            (setf source (reverse source))
            (return))
           (t
            (push next-form source))))))
    (format t "~&Source:~%")
    (myprint source)
    (if debug
        (rcomp-debug source)
      (rc source))))
```

D.4.18 `test.lisp`

Miscellaneous functions and program fragments for exercising the compiler. Some of these may be obsolete.

```
;;; -*- Package: user -*-
(in-package "USER")
;;;----------------------------------------------------------------------
;;; Testing code.

(defun test-sch ()
  (rcompile-file "sch.r"))

(defun test-sch-debug ()
  (rcompile-file "sch.r" :debug t))

(defparameter *test*
  '(defprog example1
     ()
     (let p = (3 * 5))))

(defparameter *test2*
  '(defprog example-program-2 ()
     (call mult 3 5 p)))

(defun test ()
  (rc *test*))

;; My original very simple MULT routine.
(defparameter *mult-orig*
  '(defsub mult (m1 m2 prod)
      ;; Use grade-school algorithm.
      (for pos = 0 to 31               ; For each of the 32 bit-positions,
         (if (m1 & (1 << pos)) then    ;   if that bit of m1 is 1, then
            (prod += (m2 << pos))))))  ;     add m2, shifted over to that
                                       ;       position, into p.

;; I'd like this code to compile to produce exactly my hand-compiled-
;; and -optimized version of the MULT routine.  Currently, it won't though.
(defparameter *mult-opt*
  '(defsub mult-opt (m1 m2 product)
     (with-register-vars ((limit = 32) (mask = 1) shifted bit position)
        (for position = 0 until limit
           (with (bit <- (m1 & mask))
              (if bit then
                 (with (shifted <- (m2 << position))
                    (product += shifted))))
```

```
          (mask <=< 1))))
  )

;; Hand-compiled, optimized multiply routine.  We could actually
;; include this in programs if we want.
(defparameter *mult-hand*
  '(alloctop
    (bra   allocbot)
    alloc4
    (swbrn $2)     ; This sub-subroutine frees
    (addi  $1  +1) ; 4 registers for use in the
    (exch  $31 $1) ; MULT subroutine.  It leaves
    (addi  $1  +1) ; the stack pointer pushed
    (exch  $30 $1) ; above, but we don't mind.
    (addi  $1  +1)
    (exch  $29 $1)
    (addi  $1  +1)
    (exch  $28 $1)
    allocbot
    (bra   alloctop)
    ;; This subroutine's arguments are in registers $3, $4, and $5.
    subtop
    (bra   subbot)              ; MULT top.
    mult
    (swbrn $2)                  ; Subroutine entry/exit point.
    (exch  $2 $1)               ; Push return address.
    (bra   alloc4)              ; Allocate 4 registers ($28-$31).
    (addi  $31 32)              ; limit <- 32
    (addi  $2 1)                ; mask <- 1
    fortop
    (bne   $30 $0 forbot)       ; unless (position != 0) do
    (andx  $28 $3 $2)           ;     bit <- m1&mask
    iftop
    (beq   $28 $0 ifbot)        ;     if (bit != 0) then
    (sllvx $29 $4 $30)          ;        shifted <- m2<<position
    (add   $5 $29)              ;        product += shifted
    (sllvx $29 $4 $30)          ;        shifted -> m2<<position
    ifbot
    (beq   $28 $0 iftop)        ;     end if
    (andx  $28 $3 $2)           ;     bit -> m1&mask
    (rl    $2 1)                ;     mask <=< 1 (rotate left by 1)
    (addi  $30 +1)              ;     i++
    forbot
    (bne   $30 $31 fortop)      ; and repeat while (position != limit).
    (sub   $30 $31)             ; position -> limit
    (addi  $2 -1)               ; mask -> 1
    (addi  $31 -32)             ; limit -> 32
    (rbra  alloc4)              ; Deallocate 4 registers ($28-$31).
    (exch  $2 $1)               ; Pop return address.
    subbot
    (bra   subtop)              ; MULT bottom.
    ))

(defparameter *mult-frac*
  ;; Like mult, but interprets the multiplier (1st arg) to be a
  ;; number between 1 and -1, on a scale where 2^31 = 1, -2^31 = -1.
  '(defsub mult-frac (m1 m2 prod)
      (for pos = 0 to 31
         (if (m1 & (1 << pos)) then
            (prod += (m2 >> (31 - pos))))))
```

```
;;; Interesting question: does MULT-FRAC yield the same result
;;; independently of the order of m1 and m2?  I know the original
;;; MULT did.  I think it does.

;; This optimized version the same number of instructions as mult-opt.
;; Put this in the standard library?
(defparameter *mult-frac-opt*
  '(defsub mult-frac-opt (m1 m2 product)
     (with-registers ((limit = 32) (mask = 1) shifted bit position)
        (for position = 0 until limit
           (mask >=> 1)
           (with (bit <- (m1 & mask))
              (if bit then
                 (with (shifted <- (m2 >> position))
                    (product += shifted))))
           )))
  )

;;; Now, what to do if integers are signed?
;;; What to do: for actual multiplication, only look at bits
;;; 30-0 of multiplier.  Branch on bit 31.  If 1, then
;;; subtract shifted multiplicand from product instead of
;;; adding it.

(defparameter *signed-mult-frac-opt*
  '(defsub signed-mult-frac-opt (m1 m2 product)
     (with-registers ((limit = 32) mask shifted bit position)
        (mask <- (1 << 31))
        (for position = 1 until limit
           (mask >=> 1)
           (with (bit = (m1 & mask))
              (if bit then
                 (with (shifted = (m2 >> position))
                    (if (m2 > 0)
                        (product += shifted)
                      else
                        (product -= shifted))))))
        (mask -> 1))))


;; This will be the first version of SIGNED-MULT-FRAC to actually
;; be compilable. (NOT. -6/26)
(defparameter *smf-first*
  '(defsub smf-first (m1 m2 prod)
     (with-regvars (mask shifted bit)
       (with (mask <- 1)
          (mask <=< 31)
          (for position = 1 to 31
             (mask >=> 1)               ;Rotate right by 1 bit
             (with (bit <- (m1 & mask))
                (if bit then
                  (with (shifted <- (m2 >> position))
                     (if (m2 > 0)
                         (prod += shifted)
                       else
                         (prod -= shifted))))))))))

;; OK, now THIS version is the new target. 6/26
(defparameter *smf-new*
  '(defsub smf-new (m1 m2 prod)
     (with-regvars (m1p m2p mask shifted bit p)
```

```
      (with ((mask <- 1)
             (m1p <- m1) (if (m1 < 0) then (- m1p))
             (m2p <- m2) (if (m2 < 0) then (- m2p)))
         (mask <=< 31)
         (for position = 1 to 31
            (mask >=> 1)               ;Rotate right by 1 bit
            (with (bit <- (m1 & mask))
               (if bit then
                 (with (shifted <- (m2 >> position))
                    (p += shifted)))))
         (if (m1 < 0) then (- p))
         (if (m2 < 0) then (- p))
         (prod += p)))))

;;; Stuff for Schroedinger program.
;; Point function.
(defparameter *pfunc*
  '(defsub pfunc (dest src i alphas epsilon)
      ((dest _ i) += (((alphas _ i) << 1) */ (src _ i)))
      ((dest _ i) -= (epsilon */ (src _ ((i + 1) & 127))))
      ((dest _ i) -= (epsilon */ (src _ ((i - 1) & 127)))))
  )

;; Wavefunction step.
(defparameter *schstep*
  '(defsub schstep (psiR psiI alphas epsilon)
      (for i = 0 until 128             ;For each point i,
         (call pfunc psiR psiI i))      ; psiR[i] += func(psiI,i)
      (for i = 0 until 128             ;For each point i,
         (rcall pfunc psiI psiR i)))    ; psiI[i] -= func(psiR,i)
  )

;; Data for the schroedinger simulation:
;; Epsilon, alphas, and psis.
(defparameter *schdata*
  '((array epsilon 203667001)
    (array alphas
           458243442 456664951 455111319 453582544 452078627 450599569
           449145369 447716027 446311542 444931917 443577149 442247239
           440942188 439661994 438406659 437176182 435970563 434789802
           433633899 432502854 431396668 430315339 429258869 428227257
           427220503 426238607 425281569 424349389 423442068 422559605
           421701999 420869252 420061363 419278332 418520159 417786845
           417078388 416394790 415736049 415102167 414493143 413908977
           413349669 412815220 412305628 411820895 411361019 410926002
           410515843 410130542 409770099 409434515 409123788 408837920
           408576909 408340757 408129463 407943027 407781450 407644730
           407532868 407445865 407383720 407346432 407334003 407346432
           407383720 407445865 407532868 407644730 407781450 407943027
           408129463 408340757 408576909 408837920 409123788 409434515
           409770099 410130542 410515843 410926002 411361019 411820895
           412305628 412815220 413349669 413908977 414493143 415102167
           415736049 416394790 417078388 417786845 418520159 419278332
           420061363 420869252 421701999 422559605 423442068 424349389
           425281569 426238607 427220503 428227257 429258869 430315339
           431396668 432502854 433633899 434789802 435970563 437176182
           438406659 439661994 440942188 442247239 443577149 444931917
           446311542 447716027 449145369 450599569 452078627 453582544
           455111319 456664951)
    ;; This is the shape of the initial wavefunction.
    (array psis
```

```
          2072809 3044772 4418237 6333469 8968770 12546502 17338479
          23669980 31921503 42527251 55969298 72766411 93456735 118573819
          148615999 184009768 225068513 271948808 324607187 382760978
          445857149 513053161 583213481 654924586 726530060 796185813
          861933650 921789572 973841548 1016350163 1047844835 1067208183
          1073741824 1067208183 1047844835 1016350163 973841548 921789572
          861933650 796185813 726530060 654924586 583213481 513053161
          445857149 382760978 324607187 271948808 225068513 184009768
          148615999 118573819 93456735 72766411 55969298 42527251 31921503
          23669980 17338479 12546502 8968770 6333469 4418237 3044772
          2072809 1393998 926112 607804 394060 252382 159681 99804 61622
          37586 22647 13480 7926 4604 2642 1497 838 463 253 136 73 38 20
          10 5 2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
          0 0 0 0 0 0 0 0 0 0))
  )


(defparameter *sch*
  '((DEFWORD EPSILON 203667001)
    (DEFARRAY ALPHAS 458243442 456664951 455111319 453582544
      452078627 450599569 449145369 447716027 446311542 444931917
      443577149 442247239 440942188 439661994 438406659 437176182
      435970563 434789802 433633899 432502854 431396668 430315339
      429258869 428227257 427220503 426238607 425281569 424349389
      423442068 422559605 421701999 420869252 420061363 419278332
      418520159 417786845 417078388 416394790 415736049 415102167
      414493143 413908977 413349669 412815220 412305628 411820895
      411361019 410926002 410515843 410130542 409770099 409434515
      409123788 408837920 408576909 408340757 408129463 407943027
      407781450 407644730 407532868 407445865 407383720 407346432
      407334003 407346432 407383720 407445865 407532868 407644730
      407781450 407943027 408129463 408340757 408576909 408837920
      409123788 409434515 409770099 410130542 410515843 410926002
      411361019 411820895 412305628 412815220 413349669 413908977
      414493143 415102167 415736049 416394790 417078388 417786845
      418520159 419278332 420061363 420869252 421701999 422559605
      423442068 424349389 425281569 426238607 427220503 428227257
      429258869 430315339 431396668 432502854 433633899 434789802
      435970563 437176182 438406659 439661994 440942188 442247239
      443577149 444931917 446311542 447716027 449145369 450599569
      452078627 453582544 455111319 456664951)
    (DEFARRAY PSIR 2072809 3044772 4418237 6333469 8968770 12546502
      17338479 23669980 31921503 42527251 55969298 72766411 93456735
      118573819 148615999 184009768 225068513 271948808 324607187
      382760978 445857149 513053161 583213481 654924586 726530060
      796185813 861933650 921789572 973841548 1016350163 1047844835
      1067208183 1073741824 1067208183 1047844835 1016350163
      973841548 921789572 861933650 796185813 726530060 654924586
      583213481 513053161 445857149 382760978 324607187 271948808
      225068513 184009768 148615999 118573819 93456735 72766411
      55969298 42527251 31921503 23669980 17338479 12546502 8968770
      6333469 4418237 3044772 2072809 1393998 926112 607804 394060
      252382 159681 99804 61622 37586 22647 13480 7926 4604 2642
      1497 838 463 253 136 73 38 20 10 5 2 1 0 0 0 0 0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
    (DEFARRAY PSII 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 0 0 0 0)
    (DEFSUB PFUNC (DEST SRC I ALPHAS EPSILON)
```

```
      ((DEST _ I) += ((ALPHAS _ I) */ (SRC _ I)))
      ((DEST _ I) -= (EPSILON */ (SRC _ ((I + 1) & 127))))
      ((DEST _ I) -= (EPSILON */ (SRC _ ((I - 1) & 127)))))
    (DEFSUB SCHSTEP (PSIR PSII ALPHAS EPSILON)
      (FOR I = 0 TO 127 (CALL PFUNC PSIR PSII I))
      (FOR I = 0 TO 127 (RCALL PFUNC PSII PSIR I)))
    (DEFSUB PRINTWAVE (WAVE)
      (FOR I = 0 TO 127 (PRINTWORD (WAVE _ I))) (PRINTLN))
    (DEFMAIN SCHROED
      (FOR I = 1 TO 50
           (FOR J = 1 TO 20 (CALL SCHSTEP PSIR PSII ALPHAS EPSILON))
           (CALL PRINTWAVE PSIR) (CALL PRINTWAVE PSII)))))

(defparameter *sch-frag1*
  '((DEFWORD EPSILON 203667001)
    (DEFARRAY ALPHAS 458243442 456664951)
    (DEFARRAY PSIR 2072809 3044772)
    (DEFARRAY PSII 0 0)
    (DEFSUB PFUNC (DEST SRC I ALPHAS EPSILON)
      ((DEST _ I) += ((ALPHAS _ I) */ (SRC _ I)))
      ((DEST _ I) -= (EPSILON */ (SRC _ ((I + 1) & 127))))
      ((DEST _ I) -= (EPSILON */ (SRC _ ((I - 1) & 127)))))
    (DEFSUB SCHSTEP (PSIR PSII ALPHAS EPSILON)
      (FOR I = 0 TO 127 (CALL PFUNC PSIR PSII I))
      (FOR I = 0 TO 127 (RCALL PFUNC PSII PSIR I)))
    (DEFSUB PRINTWAVE (WAVE)
      (FOR I = 0 TO 127 (PRINTWORD (WAVE _ I))) (PRINTLN))
    (DEFMAIN SCHROED
      (FOR I = 1 TO 50
           (FOR J = 1 TO 20 (CALL SCHSTEP PSIR PSII ALPHAS EPSILON))
           (CALL PRINTWAVE PSIR) (CALL PRINTWAVE PSII)))))

(defparameter *sch-frag2*
  '((DEFWORD EPSILON 203667001)
    (DEFARRAY ALPHAS 458243442)
    (DEFARRAY PSIR 2072809)
    (DEFARRAY PSII 0)
    (CALL SCHSTEP PSIR PSII ALPHAS EPSILON)))

(defparameter *sch-frag3*
  '((DEFWORD EPSILON 203667001)
    (DEFARRAY ALPHAS 458243442)
    (CALL SCHSTEP ALPHAS EPSILON)))
```