1. Definition of ERISCAL. This program takes input written in `ERISCAL`, the `ELTE RISC` assembly language, and translates it into binary files that can be loaded and executed on `ELTE RISC` simulators or hardwares. `ERISCAL` is much simpler than the "industrial strength" assembly languages that computer manufacturers usually provide, because it is primarily intended for the simple demonstration programs. Yet it tries to have enough features to serve also as the back end of compilers for C and other high-level languages.

Instructions for using the program appear at the end of this document. First we will discuss the input and output languages in detail; then we'll consider the translation process, step by step; then we'll put everything together.

2. A program in `ERISCAL` consists of a series of *lines*, each of which usually contains a single instruction. However, lines with no instructions are possible, and so are lines with two or more instructions.

Each instruction has three parts called its label field, opcode field, and operand field; these fields are separated from each other by one or more spaces. The label field, which is often empty, consists of all characters up to the first blank space. The opcode field, which is never empty, runs from the first nonblank after the label to the next blank space. The operand field, which again might be empty, runs from the next nonblank character (if any) to the first blank or semicolon that isn't part of a string or character constant. If the operand field is followed by a semicolon, possibly with intervening blanks, a new instruction begins immediately after the semicolon; otherwise the rest of the line is ignored. The end of a line is treated as a blank space for the purposes of these rules, with the additional proviso that string or character constants are not allowed to extend from one line to another.

The label field must begin with a letter or a digit; otherwise the entire line is treated as a comment. Popular ways to introduce comments, either at the beginning of a line or after the operand field, are to precede them by the character `%` as in TEX, or by `//` as in C++; `ERISCAL` is not very particular. However, Lisp-style comments introduced by single semicolons will fail if they follow an instruction, because they will be assumed to introduce another instruction.

3. `ERISCAL` has no built-in macro capability, nor does it know how to include header files and such things. But users can run their files through a standard C preprocessor to obtain `ERISCAL` programs in which macros and such things have been expanded. (Caution: The preprocessor also removes C-style comments, unless it is told not to do so.) Literate programming tools could also be used for preprocessing.

If a line begins with the special form '`#` ⟨ integer ⟩ ⟨ string ⟩', this program interprets it as a *line directive* emitted by a preprocessor. For example,

```
# 13 "foo.mms"
```

means that the following line was line 13 in the user's source file `foo.mms`. Line directives allow us to correlate errors with the user's original file; we also pass them to the output, for use by simulators and debuggers.

4. `ERISCAL` deals primarily with *symbols* and *constants*, which it interprets and combines to form machine language instructions and data. Constants are simplest, so we will discuss them first.

A *decimal constant* is a sequence of digits, representing a number in radix 10. A *hexadecimal constant* is a sequence of hexadecimal digits, preceded by `#`, representing a number in radix 16:

$$\begin{aligned}
&\langle\,\text{digit}\,\rangle \longrightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}\\
&\langle\,\text{hex digit}\,\rangle \longrightarrow \langle\,\text{digit}\,\rangle \mid \texttt{A} \mid \texttt{B} \mid \texttt{C} \mid \texttt{D} \mid \texttt{E} \mid \texttt{F} \mid \texttt{a} \mid \texttt{b} \mid \texttt{c} \mid \texttt{d} \mid \texttt{e} \mid \texttt{f}\\
&\langle\,\text{decimal constant}\,\rangle \longrightarrow \langle\,\text{digit}\,\rangle \mid \langle\,\text{decimal constant}\,\rangle\langle\,\text{digit}\,\rangle\\
&\langle\,\text{hex constant}\,\rangle \longrightarrow \texttt{\#}\langle\,\text{hex digit}\,\rangle \mid \langle\,\text{hex constant}\,\rangle\langle\,\text{hex digit}\,\rangle
\end{aligned}$$

Constants whose value is 2^{32} or more are reduced modulo 2^{32}.

5. A *character constant* is a single character enclosed in single quote marks; it denotes the ASCII or Unicode number corresponding to that character. For example, `'a'` represents the constant `#61`, also known as `97`. The quoted character can be anything except the character that the C library calls `\n` or *newline*; that character should be represented as `#a`.

⟨ character constant ⟩ ⟶ `'`⟨ single byte character except newline ⟩`'`
⟨ constant ⟩ ⟶ ⟨ decimal constant ⟩ | ⟨ hex constant ⟩ | ⟨ character constant ⟩

Notice that `'''` represents a single quote, the code `#27`; and `'\'` represents a backslash, the code `#5c`. `ERISCAL` characters are never "quoted" by backslashes as in the C language.

In the present implementation a character constant will always be at most 255, since wyde character input is not supported. The present program does not support Unicode directly because basic software for inputting and outputting 16-bit characters was still in a primitive state at the time of writing. But the data structures below are designed so that a change to Unicode will not be difficult when the time is ripe.

6. A *string constant* like `"Hello"` is an abbreviation for a sequence of one or more character constants separated by commas: `'H','e','l','l','o'`. Any character except newline or the double quote mark `"` can appear between the double quotes of a string constant.

7. A *symbol* in `ERISCAL` is any sequence of letters and digits, beginning with a letter. A colon '`:`' or underscore symbol '`_`' is regarded as a letter, for purposes of this definition. All extended-ASCII characters like '`é`', whose 8-bit code exceeds 126, are also treated as letters.

⟨ letter ⟩ ⟶ `A` | `B` | ··· | `Z` | `a` | `b` | ··· | `z` | `:` | `_` | ⟨ character with code value > 126 ⟩
⟨ symbol ⟩ ⟶ ⟨ letter ⟩ | ⟨ symbol ⟩⟨ letter ⟩ | ⟨ symbol ⟩⟨ digit ⟩

In future implementations, when `ERISCAL` is used with Unicode, all wyde characters whose 16-bit code exceeds 126 will be regarded as letters; thus `ERISCAL` symbols will be able to involve Greek letters or Chinese characters or thousands of other glyphs.

8. A symbol is said to be *fully qualified* if it begins with a colon. Every symbol that is not fully qualified is an abbreviation for the fully qualified symbol obtained by placing the *current prefix* in front of it; the current prefix is always fully qualified. At the beginning of an `ERISCAL` program the current prefix is simply the single character '`:`', but the user can change it with the `PREFIX` command. For example,

```
ADD     x,y          % means ADD :x,:y
PREFIX  Foo:         % current prefix is :Foo:
ADD     x,y          % means ADD :Foo:x,:Foo:y
PREFIX  Bar:         % current prefix is :Foo:Bar:
ADD     :x,y         % means ADD :x,:Foo:Bar:y
PREFIX  :            % current prefix reverts to :
ADD     x,Foo:Bar:y  % means ADD :x,:Foo:Bar:y
```

This mechanism allows large programs to avoid conflicts between symbol names, when parts of the program are independent and/or written by different users. The current prefix conventionally ends with a colon, but this convention need not be obeyed.

9. A *local symbol* is a decimal digit followed by one of the letters `B`, `F`, or `H`, meaning "backward," "forward," or "here":

$$\langle\,\text{local operand}\,\rangle \longrightarrow \langle\,\text{digit}\,\rangle\,\texttt{B} \mid \langle\,\text{digit}\,\rangle\,\texttt{F}$$
$$\langle\,\text{local label}\,\rangle \longrightarrow \langle\,\text{digit}\,\rangle\,\texttt{H}$$

The `B` and `F` forms are permitted only in the operand field of `ERISCAL` instructions; the `H` form is permitted only in the label field. A local operand such as `2B` stands for the last local label `2H` in instructions before the current one, or 0 if `2H` has not yet appeared as a label. A local operand such as `2F` stands for the first `2H` in instructions after the current one. Thus, in a sequence such as

```
2H JMP 2F
2H JMP 2B
```

the first instruction jumps to the second and the second jumps to the first.

Local symbols are useful for references to nearby points of a program, in cases where no meaningful name is appropriate. They can also be useful in special situations where a redefinable symbol is needed; for example, an instruction like

```
9H IS 9B+1
```

will maintain a running counter.

10. Each symbol receives a value called its *equivalent* when it appears in the label field of an instruction; it is said to be *defined* after its equivalent has been established. A few symbols, like `Fopen`, are predefined because they refer to fixed constants associated with the `ELTE RISC` hardware or its rudimentary operating system; otherwise every symbol should be defined exactly once. The two appearances of '`2H`' in the example above do not violate this rule, because the second '`2H`' is not the same symbol as the first.

A predefined symbol can be redefined (given a new equivalent). After it has been redefined it acts like an ordinary symbol and cannot be redefined again. A complete list of the predefined symbols appears in the program below.

Equivalents are either *pure* or *register numbers*. A pure equivalent is an unsigned wyde, but a register number equivalent is a nybble value, between 0 and 15. A dollar sign is used to change a pure number into a register number; for example, '`$15`' means register number 15.

11. Constants and symbols are combined into *expressions* in a simple way:

$\langle$ primary expression $\rangle \longrightarrow \langle$ constant $\rangle$ | $\langle$ symbol $\rangle$ | $\langle$ local operand $\rangle$ | `@` |
`(`$\langle$ expression $\rangle$`)` | $\langle$ unary operator $\rangle\langle$ primary expression $\rangle$
$\langle$ term $\rangle \longrightarrow \langle$ primary expression $\rangle$ | $\langle$ term $\rangle\langle$ strong operator $\rangle\langle$ primary expression $\rangle$
$\langle$ expression $\rangle \longrightarrow \langle$ term $\rangle$ | $\langle$ expression $\rangle\langle$ weak operator $\rangle\langle$ term $\rangle$
$\langle$ unary operator $\rangle \longrightarrow$ `*` | `+` | `-` | `~` | `$` | `&`
$\langle$ strong operator $\rangle \longrightarrow$ `*` | `/` | `//` | `%` | `<<` | `>>` | `&`
$\langle$ weak operator $\rangle \longrightarrow$ `+` | `-` | `|` | `^`

Each expression has a value that is either pure, a register number or an indirect version of these. The character `@` stands for the current location, which is always pure. The unary operators `*`,`+`, `-`, `~`, `$`, and `&` mean, respectively, "indirectize", "relativize," "subtract from zero," "complement the bits," "change from pure value to register number," and "take the serial number." Only the first of these, `*`, can be applied to a register number. The last unary operator, `&`, applies only to symbols, and it is of interest primarily to system programmers; it converts a symbol to the unique positive integer that is used to identify it in the binary file output by `ERISCAL`. The unusual operator `+` make a relative value from a value subtracting the current location from it. A relative value is useful if we want to add it to the current location; it is mainly used in the `SRC` field of instructions. Another unusual operator `*` symple gives the information of the compiler that the value of this field will be used indirectly, i.e, addressing mode is 3. It main use is in instruction as `SETL` \$4,*\$5 setting \$4 from address contained in \$5, but it is also used as for example `SETL` \$4,*xxxx meaning that \$4 set from the \$0 relative data `xxxx` contained in wyde after the instruction.

Binary operators come in two flavors, strong and weak. The strong ones are essentially concerned with multiplication or division: `x*y`, `x/y`, `x//y`, `x%y`, `x<<y`, `x>>y`, and `x&y` stand respectively for $(x \times y) \bmod 2^{64}$ (multiplication), $\lfloor x/y \rfloor$ (division), $\lfloor 2^{64}x/y \rfloor$ (fractional division), $x \bmod y$ (remainder), $(x \times 2^{y}) \bmod 2^{64}$ (left shift), $\lfloor x/2^{y} \rfloor$ (right shift), and $x \,\&\, y$ (bitwise and) on unsigned wydes. Division is legal only if $y > 0$; fractional division is legal only if $x < y$. None of the strong binary operations can be applied to register numbers.

The weak binary operations `x+y`, `x-y`, `x|y`, and `x^y` stand respectively for $(x + y) \bmod 2^{64}$ (addition), $(x - y) \bmod 2^{64}$ (subtraction), $x \mid y$ (bitwise or), and $x \oplus y$ (bitwise exclusive-or) on unsigned wydes. These operations can be applied to register numbers only in four contexts: $\langle$ register $\rangle + \langle$ pure $\rangle$, $\langle$ pure $\rangle + \langle$ register $\rangle$, $\langle$ register $\rangle - \langle$ pure $\rangle$ and $\langle$ register $\rangle - \langle$ register $\rangle$. For example, if `x` denotes `$1` and `y` denotes `$10`, then `x+3` and `3+x` denote `$4`, and `y-x` denotes the pure value `9`.

Register numbers within expressions are allowed to be arbitrary wydes, but a register number assigned as the equivalent of a symbol should not exceed 15.

(Incidentally, one might ask why the designer of `ERISCAL` did not simply adopt the existing rules of C for expressions. The primary reason is that the designers of C chose to give `<<`, `>>`, and `&` a lower precedence than `+`; but in `ERISCAL` we want to be able to write things like `o<<24+x<<16+y<<8+z` or `@+yz<<2` or `@+(#100-@)&#ff`. Since the conventions of C were inappropriate, it was better to make a clean break, not pretending to have a close relationship with that language. The new rules are quite easily memorized, because `ERISCAL` has just two levels of precedence, and the strong binary operations are all essentially multiplicative by nature while the weak binary operations are essentially additive.)

12. A symbol is called a *future reference* until it has been defined. `ERISCAL` restricts the use of future references, so that programs can be assembled quickly in one pass over the input; therefore all expressions can be evaluated when the `ERISCAL` processor first sees them.

The restrictions are easily stated: Future references cannot be used in expressions together with unary or binary operators (except the unary `+`, which does nothing); moreover, future references can appear as operands only in instructions that have relative addresses (namely branches, probable branches, `JMP`, `PUSHJ`, `GETA`) or in wyde constants (the pseudo-operation `OCTA`). Thus, for example, one can say `JMP 1F` or `JMP 1B-4`, but not `JMP 1F-4`.

13. We noted earlier that each `ERISCAL` instruction contains a label field, an opcode field, and an operand field. The label field is either empty or a symbol or local label; when it is nonempty, the symbol or local label receives an equivalent. The operand field is either empty or a sequence of expressions separated by commas; when it is empty, it is equivalent to the simple operand field '`0`'.

⟨ instruction ⟩ ⟶ ⟨ label ⟩⟨ opcode ⟩⟨ operand list ⟩
⟨ label ⟩ ⟶ ⟨ empty ⟩ | ⟨ symbol ⟩ | ⟨ local label ⟩
⟨ operand list ⟩ ⟶ ⟨ empty ⟩ | ⟨ expression list ⟩
⟨ expression list ⟩ ⟶ ⟨ expression ⟩ | ⟨ expression list ⟩`,`⟨ expression ⟩

The opcode field contains either a symbolic `ELTE RISC` operation name (like `ADD`), or an *alias operation*, or a *pseudo-operation*. Alias operations are alternate names for `ELTE RISC` operations whose standard names are inappropriate in certain contexts. Pseudo-operations do not correspond directly to `ELTE RISC` commands, but they govern the assembly process in important ways.

There are ?????? alias operations:

⟨ opcode ⟩ ⟶ ⟨ symbolic `ELTE RISC` operation ⟩ | ⟨ alias operation ⟩
| ⟨ pseudo-operation ⟩
⟨ symbolic `ELTE RISC` operation ⟩ ⟶ `LZ` | ⋯ | `JMP`
⟨ alias operation ⟩ ⟶ `XXX` | ⋯ | `ZZZ`
⟨ pseudo-operation ⟩ ⟶ `IS` | `LOC` | `PREFIX` | `DATA` | `CODE` | `BSPEC` | `ESPEC` | `WYDE`

14. `ELTE RISC` operations like `ADD` require exactly two expressions as operands.

15. In all cases when the opcode corresponds to an `ELTE RISC` operation, the `ERISCAL` instruction tells the assembler to carry out three steps: (1) Define the equivalent of the label field to be the current location, if the label is nonempty; (2) Evaluate the operands and assemble the specified `ELTE RISC` instruction into the current location; (3) Increase the current location by 1.

16. Now let's consider the pseudo-operations, starting with the simplest cases.

• ⟨label⟩ `IS` ⟨expression⟩ defines the value of the label to be the value of the expression, which must not be a future reference. The expression may be either pure or a register number.

• ⟨label⟩ `LOC` ⟨expression⟩ first defines the label to be the value of the current location, if the label is nonempty. Then the current location is changed to the value of the expression, which must be pure.

For example, '`LOC #1000`' will start assembling subsequent instructions or data in location whose hexadecimal value is #`1000`. '`X LOC @+500`' defines `X` to be the address of the first of 500 bytes in memory; assembly will continue at location `X` + 500. The operation of aligning the current location to a multiple of 256, if it is not already aligned in that way, can be expressed as '`LOC @+(256-@)&255`'.

A less trivial example arises if we want to emit instructions and data into two separate areas of memory, but we want to intermix them in the `ERISCAL` source file. We could start by defining `8H` and `9H` to be the starting addresses of the instruction and data segments, respectively. Then, a sequence of instructions could be enclosed in '`LOC 8B`; ...; `8H IS @`'; a sequence of data could be enclosed in '`LOC 9B`; ...; `9H IS @`'. Any number of such sequences could then be combined. Instead of the two pseudo-instructions '`8H IS @; LOC 9B`' one could in fact write simply '`8H LOC 9B`' when switching from instructions to data.

• `PREFIX` ⟨symbol⟩ redefines the current prefix to be the given symbol (fully qualified). The label field should be blank.

17. The next pseudo-operations assemble wydes of data.

• ⟨label⟩ `WYDE` ⟨expression list⟩ defines the label to be the current location, if the label field is nonempty; then it assembles one wyde for each expression in the expression list, and advances the current location by the number of wydes. The expressions should all be pure numbers that fit in one wyde.

String constants are often used in such expression lists. For example, if the current location is `#1000`, the instruction `WYDE "Hello",0` assembles six wydes containing the constants `'H'`, `'e'`, `'l'`, `'l'`, `'o'`, and `0` into locations `#1000`, ..., `#1005`, and advances the current location to `#1006`.

18. Global registers are important by starting in `ELTE RISC` programs. We give starting values to these registers.

• ⟨label⟩ `GREG` ⟨expression⟩ allocates a new global register, and assigns its number as the equivalent of the label. At the beginning of assembly, the current global threshold G is $0. Each distinct `GREG` instruction increases G by 1.

The value of the expression will be loaded into the global register at the beginning of the program, except if the ABI of the given operating system and/or source language does not dictate otherwise. Register $0 is always has a defined starting value, the value of the label `:Main`.

When `ERISCAL` programs use subroutines with a memory stack in addition to the built-in register stack, they usually begin with the instructions '`sp GREG 0;fp GREG 0`'; these instructions allocate a *stack pointer* `sp=$1` and a *frame pointer* `fp=$2`. Usually with '`lp GREG0;`' we also give a name to a *link pointer* `lp=$3` for return addresses. However, subroutine libraries are free to implement any conventions for registers and stacks that they like.

19. If our program will run on an `ELTE RISC` processor embedded in hardware supporting Harvard architecture instead of von Neumann architecture, we need two more pseudo-instructions. (In this case we have to use the `-h` compiler option.)

• `CODE` ends generating data going to the data segment and begins generate code going to the code segment; it has no effect if `-h` option is not given.

• `DATA` ends generating code going to the code segment and begins generate data going to the data segment; it has no effect if `-h` option is not given.

Remark that by `-h` option there is the possibility also to generate code into the data segment, but cannot execute there (only read/write), and there is the possibility to generate data to the code segment but cannot read/write there, only execute. Nevertheless, there is one possibility to write the content of a register `$s` into the code segment by a '` PUSH $0,$s`' instruction and leter execute it; we does not suggest this trick to use in user programs, because some operating systems does not save the content of the code segment by pageing.

20. Finally, there are two pseudo-instructions to pass information and hints to the loading routine and/or to debuggers that will be using the assembled program.

• `BSPEC` ⟨expression⟩ begins "special mode"; the ⟨expression⟩ should have a value that fits in two bytes, and the label field should be blank.

• `ESPEC` ends "special mode"; the operand field is ignored, and the label field should be blank.

All material assembled between `BSPEC` and `ESPEC` is passed directly to the output, but not loaded as part of the assembled program. Ordinary `ELTE RISC` instructions cannot appear in special mode; only the pseudo-operations `IS`, `PREFIX`, `WYDE` are allowed. The operand of `BSPEC` should have a value that fits in a wyde; this value identifies the kind of data that follows. (For example, `BSPEC 0` might introduce information about subroutine calling conventions at the current location, and `BSPEC 1` might introduce line numbers from a high-level-language program that was compiled into the code at the current place. System routines often need to pass such information through an assembler to the operating system, hence `ERISCAL` provides a general-purpose conduit.)

21. A program should begin at the special symbolic location `Main` (more precisely, at the address corresponding to the fully qualified symbol `:Main`). This symbol always has serial number 1, and it must always be defined.

Locations should not receive assembled data more than once. (More precisely, the loader will load the bitwise xor of all the data assembled for each wyde position; but the general rule "do not load two things into the same wyde" is safest.) All locations that do not receive assembled data are initially zero, except that the the operating system may put command-line data and debugger data into data segment above the stack. (The rudimentary `ELTE RISC` operating system starts a program with the number of command-line arguments in and a pointer to the beginning of an array of argument pointers in stack pointed by $2.)

22. Binary ERO output. When the `ERISCAL` processor assembles a file called `foo.ers`, it produces a binary output file called `foo.ero`. (The suffix `ers` stands for "`ELTE RISC` symbolic," and `ero` stands for "`ELTE RISC` object.") Such `ero` files have a simple structure consisting of a sequence of wydes. Some of the wydes are instructions to a loading routine; others are data to be loaded.

Loader instructions are distinguished from wydes of data by their first three (most significant) nybble, which has the special escape-code value #`0e5`, called *ero* in the program below. This code value corresponds to `ELTE RISC`'s insruction `RESUME`, which is unlikely to occur in wydes of data. The last nybble of a loader instruction is the loader opcode, called the *lopcode*.

#define *ero* #`0e5`

23. When a wyde of the `ero` file does not begin with the escape code, it is loaded into the current location λ, and λ is increased by one. More exacly, there may be two current locations, one for code segment, and one for data segment and we may change between them. Both start with zero. The current line number is also increased by 1, if it is nonzero.

When a wyde does begin with the escape code, its last nybble is the lopcode defining a loader instruction. There are thirteen lopcodes:

- *lop_quote*: #`0`. Treat the next wyde as an ordinary wyde, even if it begins with the escape code.
- *lop_seg*: #`1`. Change between data and code segment.
- *lop_skip*: #`2`. Increase the current location by the next wyde.
- *lop_fixw*: #`3`. Load (by `XOR`) the value of the current location into wyde P, where P is the 16-bit address defined by the next wyde. (The wyde at P was previously assembled as zero because of a future reference.)
- *lop_fixr*: #`4`. Load (by `XOR`) the next wyde called δ into the `SRC` field of the wyde in location P, where P is the address that precedes the current location by δ. (This nybble was previously loaded by an `ELTE RISC` instruction with a relative address. Its `SRC` field was previously assembled as zero because of a future reference.)
- *lop_fixwx*: #`5`. Proceed as in *lop_fixw*, but load the current location to the other segment.
- *lop_fixrx*: #`6`. Proceed as in *lop_fixr*, but load the current location to the other segment.
- *lop_file*: #`9`. Set the current file number to the upper half of the next wyde and the current line number to zero. The lower half of the next wyde gives the length of the filename. The following wydes are the characters of the file name. If this file number has occurred previously, the file name has length zero.
- *lop_line*: #`a`. Set the current line number to the next wyde. If the line number is nonzero, the current file and current line should correspond to the source location that generated the next data to be loaded, for use in diagnostic messages. (The `ERISCAL` program gives precise line numbers to the sources of wydes in code segment, which tend to be instructions, but not to the sources of wydes assembled in data segments.)
- *lop_spec*: #`b`. Begin special data of type given by the next wyde. The subsequent wydes, continuing until the next loader operation other than *lop_quote*, comprise the special data. A *lop_quote* instruction allows wydes of special data to begin with the escape code.
- *lop_pre*: #`c`. A *lop_pre* instruction, which defines the "preamble," must be the first wyde of every `ero` file. The higher byte of the next wyde specifies the version number of `ero` format, currently 1; other version numbers may be defined later, but version 1 should always be supported as described in the present document. The lower byte of the next wyde specifies how many wydes following a *lop_pre* command provide additional information that might be of interest to system routines. If it is nonzero, the first two wydes of additional information in big endian order records the time that this `ero` file was created, measured in seconds since 00:00:00 Greenwich Mean Time on 1 Jan 1970.
- *lop_post*: #`d`. This instruction begins the *postamble*, which follows all instructions and data to be loaded. It causes \$0, 1, ..., \$15 initially set to the values of the next 16 wydes.
- *lop_stab*: #`e`. This instruction must appear immediately after the wydes following *lop_post*. It is followed by the symbol table, which lists the equivalents of all user-defined symbols in a compact form that will be described later.
- *lop_end*: #`f`. This instruction must be the very last two wydes of each `ero` file. The next wyde gives exactly, how many wydes must appear between it and the *lop_stab* command. (Therefore a program can easily find the symbol table without reading forward through the entire `ero` file.)

A separate routine called `EROtype` is available to translate binary `ero` files into human-readable form.

```
#define lop_quote #0   /* the quotation lopcode */
#define lop_seg #1   /* the segment change lopcode */
#define lop_skip #2   /* the skip lopcode */
#define lop_fixw #3   /* the wyde-fix lopcode */
#define lop_fixr #4   /* the relative-fix lopcode */
```

```
#define lop_fixwx #5 /* extended relative-fix lopcode */
#define lop_fixrx #6 /* extended relative-fix lopcode */
#define lop_file #9 /* the file name lopcode */
#define lop_line #a /* the file position lopcode */
#define lop_spec #b /* the special hook lopcode */
#define lop_pre #c /* the preamble lopcode */
#define lop_post #d /* the postamble lopcode */
#define lop_stab #e /* the symbol table lopcode */
#define lop_end #f /* the end-it-all lopcode */
```

24. Many readers will have noticed that ERISCAL has no facilities for relocatable output, nor does ero format support such features. Knuth's first drafts of MMIXAL and mmo did allow relocatable objects, with external linkages, but the rules were substantially more complicated and therefore inconsistent with the goals of *The Art of Computer Programming*. His MMIXAL design might actually prove to be superior to the current practice, now that computer memory is significantly cheaper than it used to be, because one-pass assembly and loading are extremely fast when relocatability and external linkages are disallowed. Different program modules can be assembled together about as fast as they could be linked together under a relocatable scheme, and they can communicate with each other in much more flexible ways. Debugging tools are enhanced when open-source libraries are combined with user programs, and such libraries will certainly improve in quality when their source form is accessible to a larger community of users.

25. Basic data types. This program for the 16-bit ELTE RISC architecture is based on 32-bit integer arithmetic, because it is essencial to be possible to rewrite to the `scc` compiler running on `ELTE RISC`. The definition of type **wyde** should be changed.

⟨ Type definitions 25 ⟩ ≡
```
typedef unsigned int wyde;   /* assumes that an int is at least 16 bits wide */
typedef unsigned int tetra;   /* assumes that an int is at exactly 32 bits wide */
typedef enum {
  false, true
} bool;
```
See also sections 31, 56, 60, 64, 68, and 81.

This code is used in section 137.

26. ⟨ Global variables 26 ⟩ ≡
```
wyde zero_wyde;   /* zero_wyde = 0 */
wyde neg_one = −1;   /* neg_one = −1 */
wyde aux;   /* auxiliary output of a subroutine */
bool overflow;   /* set by certain subroutines for signed arithmetic */
```
See also sections 34, 37, 38, 44, 47, 48, 53, 58, 62, 65, 69, 75, 82, 89, 104, 116, 134, 140, and 144.

This code is used in section 137.

27. Left and right shifts are not difficult.

⟨ Subroutines 27 ⟩ ≡
```
wyde shift_left ARGS((wyde, int));
wyde shift_left(y, s)   /* shift left by s bits, where 0 ≤ s ≤ 16 */
    wyde y;
    int s;
{
  while (s ≥ 8) y ≪= 8, s −= 8;
  y ≪= s;
  return y & #ffff;
}
wyde shift_right ARGS((wyde, int, int));
wyde shift_right(y, s, u)   /* shift right, arithmetically if u = 0 */
    wyde y;
    int s, u;
{
  while (s ≥ 8) y = (y ≫ 8) + (u ? 0 : −((y ≫ 7) & #ff)), s −= 8;
  if (s) y = (y ≫ s) + (u ? 0 : (−(y ≫ 7)) ≪ (8 − s));
  return y;
}
```
See also sections 28, 29, 42, 43, 45, 46, 49, 50, 51, 52, 54, 57, 59, 61, 73, and 74.

This code is used in section 137.

28. Multiplication. We need to multiply two unsigned 16-bit integers, obtaining an unsigned 32-bit product. It is easy to do this on a 16-bit machine by using Algorithm 4.3.1M of *Seminumerical Algorithms*, with $b = 2^8$.

The following subroutine returns the lower half of the product, and puts the upper half into a global tetrabyte called *aux*.

⟨ Subroutines 27 ⟩ +≡

```
wyde wmult ARGS((wyde, wyde));
wyde wmult(y, z)
    wyde y, z;
{
  wyde u, v, t;
  wyde acc;
  u = y & #ff;
  v = z & #ff;
  t = u * v;
  acc = t & #ff;
  t >>= 8;    /* low times low */
  y >>= 8;
  y &= #ff;
  v *= y;
  v += t;
  t = v & #ff;
  v >>= 8;    /* high times low */
  z >>= 8;
  z &= #ff;
  u *= z;
  u += t;
  t = u & #ff;
  u >>= 8;    /* low times high */
  aux = y * z;
  aux += u;
  acc += t;    /* high times high */
  return acc;
}
```

29. Division inputs the high half of a dividend in the global variable *aux* and returns the remainder in *aux*. Long division of an unsigned 32-bit integer by an unsigned 16-bit integer is, of course, one of the most challenging routines needed for `ELTE RISC` arithmetic. The following program, based on Algorithm 4.3.1D of *Seminumerical Algorithms*, computes wydes q and r such that $(2^{16}x + y) = qz + r$ and $0 \le r < z$, given wydes x, y, and z, assuming that $x < z$. (If $x \ge z$, it simply sets $q = x$ and $r = y$.) The quotient q is returned by the subroutine; the remainder r is stored in *aux*.

⟨ Subroutines 27 ⟩ +≡

```
wyde wdiv ARGS((wyde, wyde, wyde));
wyde wdiv(x, y, z)
    wyde x, y, z;
{
  int j, n;
  wyde zl, zh, c, q, m, t;
  x &= #ffff; y &= #ffff; z &= #ffff;
  if (x ≥ z) { aux = y & #ffff; return x & #ffff; }
  n = 0;
  while (¬(z & (1 ≪ 15))) {
    z ≪= 1;
    c = y ≫ 15;
    x ≪= 1;
    x += c;
    y ≪= 1;
    ++n;
  }
  zl = z & #ff;
  z −= zl;
  zh = z ≫ 8;
  aux = 0;
  for (j = 1; j ≥ 0; j−−) {
    if (x ≥ z) q = #ff;
    else q = x/zh;     /* approx q-digit */
    m = zl * q;
    t = (m & #ff) ≪ 8;
    m ≫ 8;     /* q times low part */
    c = 0;
    t = y − t;
    if (t > y) ++c;
    y = t;     /* multiple back */
    t = x − c;
    c = 0;
    if (t > x) ++c;
    x = t − m;
    if (x > t) ++c;
    t = x;
    x = t − zh * q;
    if (x > t) ++c;
    while (c) {     /* add back while carry */
      −−q;
      t = 0;
      y += (zl ≪ 8);
      if (y < (zl ≪ 8)) ++t;
      x += t;
```

```
        if (x < t) --c;
        x += zh;
        if (x < zh) --c;
      }
      aux <<= 8;
      aux += q;
      x <<= 8;
      x += y >> 8;
      y <<= 8;
    }
    x >>= n;
    return x;
  }
```

30. Here's a rudimentary check to see if arithmetic is in trouble.

31. Future versions of this program will work with symbols formed from Unicode characters, but the present code limits itself to an 8-bit subset. The type **Char** is defined here in order to ease the later transition: At present, **Char** is the same as **char**, but **Char** can be changed to a 16-bit type in the Unicode version.

Other changes will also be necessary when the transition to Unicode is made; for example, some calls of *fprintf* will become calls of *fwprintf*, and some occurrences of `%s` will become `%ls` in print formats. The switchable type name **Char** provides at least a first step towards a brighter future with Unicode.

⟨ Type definitions 25 ⟩ +≡

```
  typedef char Char;    /* bytes that will become wydes some day */
```

32. While we're talking about classic systems versus future systems, we might as well define the `ARGS` macro, which makes function prototypes available on ANSI C systems without making them uncompilable on older systems. Each subroutine below is declared first with a prototype, then with an old-style definition.

⟨ Preprocessor definitions 32 ⟩ ≡

```
#ifdef __STDC__
#define ARGS(list) list
#else
#define ARGS(list) ()
#endif
```

See also section 40.

This code is used in section 137.

33. Basic input and output. Input goes into a buffer that is normally limited to 72 characters. This limit can be raised, by using the `-b` option when invoking the assembler; but short buffers will keep listings from becoming unwieldy, because a symbolic listing adds 19 characters per line.

```
⟨ Initialize everything 33 ⟩ ≡
  if (buf_size < 72) buf_size = 72;
  buffer = (Char *) calloc(buf_size + 1, sizeof(Char));
  lab_field = (Char *) calloc(buf_size + 1, sizeof(Char));
  op_field = (Char *) calloc(buf_size, sizeof(Char));
  operand_list = (Char *) calloc(buf_size, sizeof(Char));
  err_buf = (Char *) calloc(buf_size + 60, sizeof(Char));
  if (¬buffer ∨ ¬lab_field ∨ ¬op_field ∨ ¬operand_list ∨ ¬err_buf) panic("No␣room␣for␣the␣buffers");
```

See also sections 63, 71, 83, 90, and 141.

This code is used in section 137.

34.

```
⟨ Global variables 26 ⟩ +≡
  Char *buffer;   /* raw input of the current line */
  Char *buf_ptr;   /* current position within buffer */
  Char *lab_field;   /* copy of the label field of the current instruction */
  Char *op_field;   /* copy of the opcode field of the current instruction */
  Char *operand_list;   /* copy of the operand field of the current instruction */
  Char *err_buf;   /* place where dynamic error messages are sprinted */
```

35.

```
⟨ Get the next line of input text, or break if the input has ended 35 ⟩ ≡
  if (¬fgets(buffer, buf_size + 1, src_file)) break;
  line_no++;
  line_listed = false;
  j = strlen(buffer);
  if (buffer[j − 1] ≡ '\n') buffer[j − 1] = '\0';   /* remove the newline */
  else if ((j = fgetc(src_file)) ≠ EOF) ⟨ Flush the excess part of an overlong line 36 ⟩;
  if (buffer[0] ≡ '#') ⟨ Check for a line directive 39 ⟩;
  buf_ptr = buffer;
```

This code is used in section 137.

36.

```
⟨ Flush the excess part of an overlong line 36 ⟩ ≡
  {
    while (j ≠ '\n' ∧ j ≠ EOF) j = fgetc(src_file);
    if (¬long_warning_given) {
      long_warning_given = true;
      err("*trailing␣characters␣of␣long␣input␣line␣have␣been␣dropped");
      fprintf(stderr, "(say␣`-b␣<number>'␣to␣increase␣the␣length␣of␣my␣input␣buffer)\n");
    } else err("*trailing␣characters␣dropped");
  }
```

This code is used in section 35.

37.

```
⟨ Global variables 26 ⟩ +≡
  int cur_file;   /* index of the current file in filename */
  int line_no;   /* current position in the file */
  bool line_listed;   /* have we listed the buffer contents? */
  bool long_warning_given;   /* have we given the hint about -b? */
```

38. We keep track of source file name and line number at all times, for error reporting and for synchronization data in the object file. Up to 256 different source file names can be remembered.

⟨ Global variables 26 ⟩ +≡
```
Char *filename[257];    /* source file names, including those in line directives */
int filename_count;     /* how many filename entries have we filled? */
```

39. If the current line is a line directive, it will also be treated as a comment by the assembler.

⟨ Check for a line directive 39 ⟩ ≡
```
{
  for (p = buffer + 1; isspace(*p); p++) ;
  for (j = 0; isdigit(*p); p++) j = 10 * j + *p - '0';
  for ( ; isspace(*p); p++) ;
  if (*p ≡ '\"') {
    if (¬filename[filename_count]) {
      filename[filename_count] = (Char *) calloc(FILENAME_MAX + 1, sizeof(Char));
      if (¬filename[filename_count]) panic("Capacity␣exceeded:␣Out␣of␣filename␣memory");
    }
    for (p++, k = 0; *p ∧ *p ≠ '\"' ∧ k < FILENAME_MAX; p++, k++) filename[filename_count][k] = *p;
    if (k ≡ FILENAME_MAX) panic("Capacity␣exceeded:␣File␣name␣too␣long");
    if (*p ≡ '\"' ∧ *(p − 1) ≠ '\"') {    /* yes, it's a line directive */
      filename[filename_count][k] = '\0';
      for (k = 0; strcmp(filename[k], filename[filename_count]) ≠ 0; k++) ;
      if (k ≡ filename_count) {
        if (filename_count ≡ 256) panic("Capacity␣exceeded:␣More␣than␣256␣file␣names");
        filename_count++;
      }
      cur_file = k;
      line_no = j − 1;
    }
  }
}
```
This code is used in section 35.

40. Archaic versions of the C library do not define `FILENAME_MAX`.

⟨ Preprocessor definitions 32 ⟩ +≡
```
#ifndef FILENAME_MAX
#define FILENAME_MAX 256
#endif
```

41. ⟨ Local variables 41 ⟩ ≡
```
register Char *p, *q;    /* the place where we're currently scanning */
```
See also section 67.

This code is used in section 137.

42. The next several subroutines are useful for preparing a listing of the assembled results. In such a listing, which the user can request with a command-line option, we fill the leftmost 19 columns with a representation of the output that has been assembled from the input in the buffer. Sometimes the assembled output requires more than one line, because we have room to output only a tetrabyte per line.

The *flush_listing_line* subroutine is called when we have finished generating one line's worth of assembled material. Its parameter is a string to be printed between the assembled material and the buffer contents, if the input line hasn't yet been echoed. The length of this string should be 19 minus the number of characters already printed on the current line of the listing.

⟨ Subroutines 27 ⟩ +≡
```
void flush_listing_line ARGS((char *));
void flush_listing_line(s)
    char *s;
{
  if (line_listed) fprintf(listing_file, "\n");
  else {
    fprintf(listing_file, "%s%s\n", s, buffer);
    line_listed = true;
  }
}
```

43. Only the two least significant hex digits of a location are shown on the listing, unless the other digits have changed. The following subroutine prints an extra line when a change needs to be shown.

⟨ Subroutines 27 ⟩ +≡
```
void update_listing_loc ARGS((void));
void update_listing_loc()
{
  if (cur_seg ≠ listing_seg ∨ ((cur_loc ⊕ listing_loc) & #ff00)) {
    fprintf(listing_file, "%01x%04x:", cur_seg, cur_loc);
    flush_listing_line("␣␣");
  }
  listing_seg = cur_seg; listing_loc = cur_loc;
}
```

44. ⟨ Global variables 26 ⟩ +≡
```
wyde cur_loc;   /* current location of assembled output */
wyde cur_seg;   /* current segment of assembled output */
wyde cur_code_loc;   /* current location of assembled output */
wyde cur_data_loc;   /* current location of assembled output */
wyde listing_loc;   /* current location on the listing */
wyde listing_seg;   /* current segment on the listing */
unsigned char hold_buf[4];   /* assembled nybbles */
unsigned char held_bits;   /* which nybbles of hold_buf are active? */
unsigned char listing_bits;   /* which of them haven't been listed yet? */
bool spec_mode;   /* are we between BSPEC and ESPEC? */
wyde spec_mode_loc;   /* number of wydes in the current special output */
```

45. When nybbles are assembled, they are placed into the *hold_buf*. Furthermore, *listing_bits* is increased by #10 ≪ j if that nybble is a future reference to be resolved later.

The nybbles are held until we need to output them. The *listing_clear* routine lists any that have been held but not yet shown. It should be called only when *listing_bits* ≠ 0.

⟨ Subroutines 27 ⟩ +≡

```
void listing_clear ARGS((void));
void listing_clear()
{
  register int j;
  if (spec_mode) fprintf(listing_file, "         ");
  else {
    update_listing_loc();
    fprintf(listing_file, "%02x: ", listing_loc);
  }
  for (j = 0; j < 4; j++)
    if (listing_bits & (#10 << j)) fprintf(listing_file, "x");
    else fprintf(listing_file, "%01x", (j & 1) ? hold_buf[j >> 1] >> 4 : hold_buf[j >> 1] & #f);
  flush_listing_line("  ");
  listing_bits = 0;
}
```

46. Error messages are written to *stderr*. If the message begins with '`*`' it is merely a warning; if it begins with '`!`' it is fatal; otherwise the error is probably serious enough to make manual correction necessary, yet it is not tragic. Errors and warnings appear also on the optional listing file.

```
#define err(m)
        { report_error(m); if (m[0] ≠ '*') goto bypass; }
#define derr(m, p)
        { sprintf(err_buf, m, p);
          report_error(err_buf); if (err_buf[0] ≠ '*') goto bypass; }
#define dderr(m, p, q)
        { sprintf(err_buf, m, p, q);
          report_error(err_buf); if (err_buf[0] ≠ '*') goto bypass; }
#define panic(m)
        { sprintf(err_buf, "!%s", m); report_error(err_buf); }
#define dpanic(m, p)
        { err_buf[0] = '!'; sprintf(err_buf + 1, m, p); report_error(err_buf); }
```

```
⟨Subroutines 27⟩ +≡
  void report_error ARGS((char *));
  void report_error(message)
      char *message;
  {
    if (¬filename[cur_file]) filename[cur_file] = "(nofile)";
    if (message[0] ≡ '*')
      fprintf(stderr, "\"%s\",␣line␣%d␣warning:␣%s\n", filename[cur_file], line_no, message + 1);
    else if (message[0] ≡ '!')
      fprintf(stderr, "\"%s\",␣line␣%d␣fatal␣error:␣%s\n", filename[cur_file], line_no, message + 1);
    else {
      fprintf(stderr, "\"%s\",␣line␣%d:␣%s!\n", filename[cur_file], line_no, message);
      err_count++;
    }
    if (listing_file) {
      if (¬line_listed) flush_listing_line("******************␣");
      if (message[0] ≡ '*') fprintf(listing_file, "************␣warning:␣%s\n", message + 1);
      else if (message[0] ≡ '!') fprintf(listing_file, "********␣fatal␣error:␣%s!\n", message + 1);
      else fprintf(listing_file, "**********␣error:␣%s!\n", message);
    }
    if (message[0] ≡ '!') exit(−2);
  }
```

47. ⟨Global variables 26⟩ +≡

```
  int err_count;    /* this many errors were found */
```

48. Output to the binary *obj_file* occurs four nybbles at a time. The nybbles are assembled in small buffers, not output as single wyde, because we want the output to be big-endian even when the assembler is running on a little-endian machine.

```
#define ero_write(buf) if (fwrite(buf, 1, 2, obj_file) ≠ 2) dpanic("Can't␣write␣on␣%s", obj_file_name)
```

```
⟨Global variables 26⟩ +≡
  unsigned char lop_quote_command[2] = {(ero ≫ 4) & #ff, (ero ≪ 4) & #f0 + lop_quote};
  unsigned char ero_buf[2];
  int ero_ptr;
```

49. ⟨Subroutines 27⟩ +≡

```
void ero_out ARGS((void));
void ero_out_quote ARGS((void));
void ero_wyde ARGS((wyde));
void ero_lop ARGS((char, unsigned char, unsigned char));
void ero_lopp ARGS((char, wyde));

void ero_out()
{
  if (listing_file) listing_clear();
  ero_write(ero_buf);
}

void ero_out_quote()
{
  if ((ero_buf[0] ≡ (ero ≫ 4) & #ff) ∧ (ero_buf[1] & #f0 ≡ (ero ≪ 4) & #f0))
    ero_write(lop_quote_command);
  ero_out();
}

void ero_wyde(t)    /* output a wyde */
    wyde t;
{
  ero_buf[0] = (t ≫ 8) & #ff; ero_buf[1] = t & #ff;
  ero_out();
}

void ero_lop(x, y, z)    /* output a loader operation */
    char x;
    unsigned char y, z;
{
  ero_buf[0] = (ero ≫ 4) & #ff; ero_buf[1] = (ero ≪ 4) & #f0 + (x & #f);
  ero_out();
  ero_buf[0] = y & #ff; ero_buf[1] = z & #ff;
  ero_out();
}

void ero_lopp(x, yz)    /* output a loader operation with wyde operand */
    char x;
    wyde yz;
{
  ero_buf[0] = (ero ≫ 4) & #ff; ero_buf[1] = ((ero ≪ 4) & #f0) + (x & #f);
  ero_out();
  ero_buf[0] = (yz ≫ 8) & #ff; ero_buf[1] = yz & #ff;
  ero_out();
}
```

50. The *ero_seg* subroutine makes the current segment in the object file equal to *cur_seg*.

⟨Subroutines 27⟩ +≡

```
void ero_seg ARGS((void));
void ero_seg()
{
  if (ero_cur_seg ≠ cur_seg) ero_wyde((ero ≪ 4) + lop_seg);
  ero_cur_seg = cur_seg;
}
```

51. The *ero_loc* subroutine makes the current location in the object file equal to *cur_loc*.

⟨ Subroutines 27 ⟩ +≡

```
void ero_loc ARGS((void));
void ero_loc()
{
  wyde w;
  w = (cur_loc − ero_cur_loc) & #ffff;
  if (w) ero_lopp(lop_skip, w);
  ero_cur_loc = cur_loc;
}
```

52. Similarly, the *ero_sync* subroutine makes sure that the current file and line number in the output file agree with *cur_file* and *line_no*.

⟨ Subroutines 27 ⟩ +≡

```
void ero_sync ARGS((void));
void ero_sync()
{
  register int j;
  register unsigned char *p;
  if (cur_file ≠ ero_cur_file) {
    if (filename_passed[cur_file]) ero_lop(lop_file, cur_file, 0);
    else {
      ero_lop(lop_file, cur_file, strlen(filename[cur_file]));
      for (p = filename[cur_file]; *p; p++) {
        ero_buf[0] = (*p ≫ 8) & #ff;
        ero_buf[1] = *p & #ff;
        ero_out_quote();
      }
      filename_passed[cur_file] = 1;
    }
    ero_cur_file = cur_file;
    ero_line_no = 0;
  }
  if (line_no ≠ ero_line_no) {
    if (line_no ≥ #10000) panic("I␣can't␣deal␣with␣line␣numbers␣exceeding␣65535");
    ero_lopp(lop_line, line_no);
    ero_line_no = line_no;
  }
}
```

53. ⟨ Global variables 26 ⟩ +≡

```
wyde ero_cur_loc;   /* current location in the object file */
wyde ero_cur_seg;   /* current segment in the object file */
int ero_line_no;   /* current line number in the ero output so far */
int ero_cur_file;   /* index of the current file in the ero output so far */
char filename_passed[256];   /* has a filename been recorded in the output? */
```

54. Here is a basic subroutine that assembles a wyde starting at *cur_loc*. The *x_bits* parameter tells which wydes, if any, are part of a future reference.

⟨ Subroutines 27 ⟩ +≡
```
void assemble ARGS((wyde, unsigned char));
void assemble(dat, x_bits)
      wyde dat;
      unsigned char x_bits;     /* These nybbles will be listed as x */
{
   register int j, jj, l;
   if (spec_mode) l = spec_mode_loc;
   else {
      l = cur_loc;
      ⟨ Make sure cur_loc and ero_cur_loc refer to the same wyde 55 ⟩;
   }
   hold_buf[0] = (dat >> 8) & #ff;
   hold_buf[1] = dat & #ff;
   listing_bits |= x_bits;
   if (listing_file) listing_clear();
   ero_out();
   if (spec_mode) ++spec_mode_loc;
   else ++cur_loc;
}
```

55. ⟨ Make sure *cur_loc* and *ero_cur_loc* refer to the same wyde 55 ⟩ ≡
```
   if (cur_seg ≠ ero_cur_seg) ero_seg();
   if ((cur_loc ⊕ ero_cur_loc) & #ffff) ero_loc();
```
This code is used in sections 54 and 111.

56. The symbol table. Symbols are stored and retrieved by means of a *ternary search trie*, following ideas of Bentley and Sedgewick. (See *ACM–SIAM Symp. on Discrete Algorithms* **8** (1997), 360–369; R. Sedgewick, *Algorithms in C* (Reading, Mass.: Addison–Wesley, 1998), §15.4.) Each trie node stores a character, and there are branches to subtries for the cases where a given character is less than, equal to, or greater than the character in the trie. There also is a pointer to a symbol table entry if a symbol ends at the current node.

⟨ Type definitions 25 ⟩ +≡

```
typedef struct ternary_trie_struct {
  unsigned short ch;    /* the (possibly wyde) character stored here */
  struct ternary_trie_struct *left, *mid, *right;    /* downward in the ternary trie */
  struct sym_tab_struct *sym;    /* equivalents of symbols */
} trie_node;
```

57. We allocate trie nodes in chunks of 1000 at a time.

⟨ Subroutines 27 ⟩ +≡

```
trie_node *new_trie_node ARGS((void));
trie_node *new_trie_node()
{
  register trie_node *t = next_trie_node;
  if (t ≡ last_trie_node) {
    t = (trie_node *) calloc(1000, sizeof(trie_node));
    if (¬t) panic("Capacity␣exceeded:␣Out␣of␣trie␣memory");
    last_trie_node = t + 1000;
  }
  next_trie_node = t + 1;
  return t;
}
```

58. ⟨ Global variables 26 ⟩ +≡

```
trie_node *trie_root;    /* root of the trie */
trie_node *op_root;    /* root of subtrie for opcodes */
trie_node *next_trie_node, *last_trie_node;    /* allocation control */
trie_node *cur_prefix;    /* root of subtrie for unqualified symbols */
```

59. The *trie_search* subroutine starts at a given node of the trie and finds a given string in its middle subtrie, inserting new nodes if necessary. The string ends with the first nonletter or nondigit; the location of the terminating character is stored in global variable *terminator*.

```
#define isletter(c) (isalpha(c) ∨ c ≡ '_' ∨ c ≡ ':' ∨ (unsigned int)(c) > 126)
⟨ Subroutines 27 ⟩ +≡
  trie_node *trie_search ARGS((trie_node *, Char *));
  Char *terminator;    /* where the search ended */

  trie_node *trie_search(t, s)
      trie_node *t;
      Char *s;
  {
    register trie_node *tt = t;
    register Char *p = s;

    while (1) {
      if (¬isletter(*p) ∧ ¬isdigit(*p)) {
        terminator = p; return tt;
      }
      if (tt→mid) {
        tt = tt→mid;
        while (*p ≠ tt→ch) {
          if (*p < tt→ch) {
            if (tt→left) tt = tt→left;
            else {
              tt→left = new_trie_node( ); tt = tt→left; goto store_new_char;
            }
          } else {
            if (tt→right) tt = tt→right;
            else {
              tt→right = new_trie_node( ); tt = tt→right; goto store_new_char;
            }
          }
        }
        p++;
      } else {
        tt→mid = new_trie_node( ); tt = tt→mid;
      store_new_char: tt→ch = *p++;
      }
    }
  }
```

60. Symbol table nodes hold the serial numbers and equivalents of defined symbols. They also hold "fixup information" for undefined symbols; this will allow the loader to correct any previously assembled instructions that refer to such symbols when they are eventually defined.

In the symbol table node for a defined symbol, the *link* field has one of the special codes `DEFINED` or `REGISTER` or `PREDEFINED`, and the *equiv* field holds the defined value. The *serial* number is a unique identifier for all user-defined symbols.

In the symbol table node for an undefined symbol, the *equiv* field is ignored. The *link* field points to the first node of fixup information; that node is, in turn, a symbol table node that might link to other fixups. The *serial* number in a fixup node is either 0 or 1 or 2, meaning respectively "fixup the wyde pointed to by *equiv*" or "fixup the relative address in the YZ field of the instruction pointed to by *equiv*" or "fixup the relative address in the XYZ field of the instruction pointed to by *equiv*."

```
#define DEFINED (sym_node *) 1   /* code value for wyde equivalents */
#define REGISTER (sym_node *) 2   /* code value for register-number equivalents */
#define PREDEFINED (sym_node *) 3   /* code value for not-yet-used equivalents */
#define seg_bit 1   /* serial code bit for data segment */
#define nyb_bit 2   /* serial code bit for signed nybble fixup */
#define rel_bit 4   /* serial code bit for relative fixup */
```

⟨ Type definitions 25 ⟩ +≡

```
typedef struct sym_tab_struct {
  int serial;   /* serial number of symbol; type number for fixups */
  struct sym_tab_struct *link;   /* DEFINED status or link to fixup */
  wyde equiv;   /* the equivalent value */
  wyde seg;   /* the segment: 0 for code, 1 for data, if it is */
} sym_node;
```

61. The allocation of new symbol table nodes proceeds in chunks, like the allocation of trie nodes. But in this case we also have the possibility of reusing old fixup nodes that are no longer needed.

```
#define recycle_fixup(pp) pp->link = sym_avail, sym_avail = pp
```

⟨ Subroutines 27 ⟩ +≡

```
sym_node *new_sym_node ARGS((bool));
sym_node *new_sym_node(serialize)
    bool serialize;   /* should the new node receive a unique serial number? */
{
  register sym_node *p = sym_avail;
  if (p) {
    sym_avail = p->link; p->link = Λ; p->serial = 0; p->equiv = zero_wyde; p->seg = zero_wyde;
  } else {
    p = next_sym_node;
    if (p ≡ last_sym_node) {
      p = (sym_node *) calloc(1000, sizeof(sym_node));
      if (¬p) panic("Capacity␣exceeded:␣Out␣of␣symbol␣memory");
      last_sym_node = p + 1000;
    }
    next_sym_node = p + 1;
  }
  if (serialize) p->serial = ++serial_number;
  return p;
}
```

62. ⟨ Global variables 26 ⟩ +≡

```
int serial_number;
sym_node *sym_root;   /* root of the sym */
sym_node *next_sym_node, *last_sym_node;   /* allocation control */
sym_node *sym_avail;   /* stack of recycled symbol table nodes */
```

63. We initialize the trie by inserting all the predefined symbols. Opcodes are given the prefix ^, to distinguish them from ordinary symbols; this character nicely divides uppercase letters from lowercase letters.

⟨ Initialize everything 33 ⟩ +≡

```
trie_root = new_trie_node();
cur_prefix = trie_root;
op_root = new_trie_node();
trie_root→mid = op_root;
trie_root→ch = ':';
op_root→ch = '^';
⟨ Put the ELTE RISC opcodes and ERISCAL pseudo-ops into the trie 66 ⟩;
⟨ Put other predefined symbols into the trie 70 ⟩;
```

64. Most of the assembly work can be table driven, based on bits that are stored as the "equivalents" of opcode symbols like ^ADD.

```
#define arg_num_bits #3   /* number of arguments: 0,1,2,¿=3? */
#define immed_bit #4   /* immediate addressing is allowed? */
#define dest_bit #8   /* destination modified addressing is allowed? */
#define reg_bit #10   /* register addressing is allowed? */
#define indir_bit #20   /* indirect addressings is allowed? */
#define no_label_bit #40   /* should the label be blank? */
#define spec_bit #80   /* is this opcode allowed in SPEC mode? */
```

⟨ Type definitions 25 ⟩ +≡

```
typedef struct {
  Char *name;   /* symbolic opcode */
  int bits;   /* treatment of operands */
} op_spec;

typedef enum {
  IS = #01, LOC, PREFIX, BSPEC = #11, ESPEC, WYDE, GREG = #21, CODE, DATA
} pseudo_op;
```

65. ⟨ Global variables 26 ⟩ +≡

```
op_spec op_init_table[] = {
{"LZ", #003e}, {"JMP", #f43e}, {"IS", (IS ≪ 8) + #81}, {"LOC", (LOC ≪ 8) + #01}, {"PREFIX",
    (PREFIX ≪ 8) + #c1}, {"WYDE", (WYDE ≪ 8) + #83},
{"GREG", (GREG ≪ 8) + #81}, {"CODE", (CODE ≪ 8) + #00}, {"DATA", (DATA ≪ 8) + #00}, {"BSPEC",
    (BSPEC ≪ 8) + #41}, {"ESPEC", (ESPEC ≪ 8) + #c0}};
int op_init_size;   /* the number of items in op_init_table */
```

66. ⟨ Put the ELTE RISC opcodes and ERISCAL pseudo-ops into the trie 66 ⟩ ≡

```
op_init_size = (sizeof op_init_table)/sizeof(op_spec);
for (j = 0; j < op_init_size; j++) {
  tt = trie_search(op_root, op_init_table[j].name);
  pp = tt->sym = new_sym_node(false);
  pp->link = PREDEFINED;
  pp->equiv = op_init_table[j].bits;
  pp->seg = 0;
}
```

This code is used in section 63.

67. ⟨ Local variables 41 ⟩ +≡

```
register trie_node *tt;
register sym_node *pp, *qq;
```

68. ⟨ Type definitions 25 ⟩ +≡

```
typedef struct {
  Char *name;
  wyde h, l;
} predef_spec;
```

69. ⟨ Global variables 26 ⟩ +≡

```
predef_spec predefs[] = {{"Inf", 1, #ff00},
{"StdIn", 0, 0}, {"StdOut", 0, 1}, {"StdErr", 0, 2},
{"TextRead", 0, 0}, {"TextWrite", 0, 1}, {"BinaryRead", 0, 2}, {"BinaryWrite", 0, 3},
    {"BinaryReadWrite", 0, 4},
{"Halt", 0, 0}, {"Fopen", 0, 1}, {"Fclose", 0, 2}, {"Fread", 0, 3}, {"Fgets", 0, 4}, {"Fgetws", 0, 5},
    {"Fwrite", 0, 6}, {"Fputs", 0, 7}, {"Fputws", 0, 8}, {"Fseek", 0, 9}, {"Ftell", 0, 10}};
int predef_size;
```

70. ⟨ Put other predefined symbols into the trie 70 ⟩ ≡

```
predef_size = (sizeof predefs)/sizeof(predef_spec);
for (j = 0; j < predef_size; j++) {
  tt = trie_search(trie_root, predefs[j].name);
  pp = tt->sym = new_sym_node(false);
  pp->link = PREDEFINED;
  pp->seg = harvard & predefs[j].h, pp->equiv = predefs[j].l;
}
```

This code is used in section 63.

71. We place Main into the trie at the beginning of assembly, so that it will show up as an undefined symbol if the user specifies no starting point.

⟨ Initialize everything 33 ⟩ +≡

```
trie_search(trie_root, "Main")->sym = new_sym_node(true);
```

72. At the end of assembly we traverse the entire symbol table, visiting each symbol in lexicographic order and transmitting the trie structure to the output file. We detect any undefined future references at this time.

The order of traversal has a simple recursive pattern: To traverse the subtrie rooted at t, we

traverse $t\to left$, if the left subtrie is nonempty;
visit $t\to sym$, if this symbol table entry is present;
traverse $t\to mid$, if the middle subtrie is nonempty;
traverse $t\to right$, if the right subtrie is nonempty.

This pattern leads to a compact representation in the `ero` file, usually requiring fewer than two wydes per trie node plus the wydes needed to encode the equivalents and serial numbers. Each node of the trie is encoded as a "master wyde" followed by the encodings of the left subtrie, character, equivalent, middle subtrie, and right subtrie. If possible, we put the character *ch* and part or all of the equivalent into the master wyde. The master wyde is the sum of

#`8000`, if the left subtrie is nonempty;
#`4000`, if the middle subtrie is nonempty;
#`2000`, if the right subtrie is nonempty;
  and one of the following values:
#`0xyz`, if the symbol's equivalent is \$0 plus x
  and the character code is yz;
#`1xyz`, if $xyz = (s \ll 10) + ch$, where $s$is the symbol's segment and the
  character code at most 10 bits (so most significant bit of x is 0);
#`1xyz`, if the symbol is nondefined and $xyz = (1 \ll 11) + ch$, where the
  character code at most 10 bits (so most significant bits of x are 10);
#`1c0z`, if the symbol's equivalent is \$0 plus z, and ch is in a
  separate wyde (so most significant bits of the second wyde are 110);
#`1e0z`, if the symbol's segment is z, and ch is in separate wyde;
  (so bits of the second wyde are 1110);
#`1f0z`, if the symbol is nondefined, and ch is in separate wyde;
  (so bits of the second wyde are 1111);

the character is omitted if the middle subtrie and the equivalent are both empty. Symbol equivalents are followed by the serial number, represented as a wyde.

73. First we prune the trie by removing all predefined symbols that the user did not redefine.

⟨ Subroutines 27 ⟩ +≡

```
trie_node *prune ARGS((trie_node *));
trie_node *prune(t)
    trie_node *t;
{
  register int useful = 0;
  if (t→sym) {
    if (t→sym→serial) useful = 1;
    else t→sym = Λ;
  }
  if (t→left) {
    t→left = prune(t→left);
    if (t→left) useful = 1;
  }
  if (t→mid) {
    t→mid = prune(t→mid);
    if (t→mid) useful = 1;
  }
  if (t→right) {
    t→right = prune(t→right);
    if (t→right) useful = 1;
  }
  if (useful) return t;
  else return Λ;
}
```

74. Then we output the trie by following the recursive traversal pattern.

⟨ Subroutines 27 ⟩ +≡
```
void out_stab ARGS((trie_node *));
void out_stab(t)
    trie_node *t;
{
  register int m = 0, j;
  register sym_node *pp;
  bool c = 1;    /* out character in separate wyde? */
  bool s = 1;    /* defined symbol? */
  if (t→ch > #3ff) m += #1f00;
  else m += t→ch, c = 0;
  if (t→left) m += #8000;
  if (t→mid) m += #4000;
  if (t→right) m += #2000;
  if (t→sym) {
    if (t→sym→link ≡ REGISTER)
      if (t→ch < #ff) m += #1000 + t→ch + (((t→sym→equiv) & #f) ≪ 7), c = 0;
      else m += #1c00 + ((t→sym→equiv) & #f);
    else if (t→sym→link ≡ DEFINED)
      if (t→ch < #3ff) m += #1800 + (t→sym→seg ≪ 10) + t→ch, c = 0;
      else m += #1e00 + t→sym→seg;
    else if (t→sym→link ∨ t→sym→serial ≡ 1) ⟨ Report an undefined symbol 78 ⟩;
  }
  ero_wyde(m);
  if (t→left) out_stab(t→left);
  if (m & #4000 ∨ c ∨ s) ⟨ Visit t and traverse t→mid 76 ⟩;
  if (t→right) out_stab(t→right);
}
```

75. We make room for symbols up to 999 bytes long. Strictly speaking, the program should check if this limit is exceeded; but really!

⟨ Global variables 26 ⟩ +≡
```
Char sym_buf[1000];
Char *sym_ptr;
```

76. A global variable called *sym_buf* holds all characters on middle branches to the current trie node; *sym_ptr* is the first currently unused character in *sym_buf*.

⟨ Visit t and traverse t→*mid* 76 ⟩ ≡
```
{
  if (c) ero_wyde(t→ch);
  *sym_ptr++ = t→ch;
  if (s ∧ t→sym→link) {
    if (listing_file) ⟨ Print symbol sym_buf and its equivalent 77 ⟩;
    ero_wyde(t→sym→serial);
  }
  if (t→mid) out_stab(t→mid);
  sym_ptr−−;
}
```
This code is used in section 74.

77. The initial ‘`:`’ of each fully qualified symbol is omitted here, since most users of `ERISCAL` will probably not need the `PREFIX` feature. One consequence of this omission is that the one-character symbol ‘`:`’ itself, which is allowed by the rules of `ERISCAL`, is printed as the null string.

⟨ Print symbol *sym_buf* and its equivalent 77 ⟩ ≡
```
{
  *sym_ptr = '\0';
  fprintf(listing_file, " %s = ", sym_buf + 1);
  pp = t→sym;
  if (pp→link ≡ DEFINED) fprintf(listing_file, "#%01x%04x", pp→seg, pp→equiv);
  else if (pp→link ≡ REGISTER) fprintf(listing_file, "$%02d", pp→equiv);
  else fprintf(listing_file, "?");
  fprintf(listing_file, " (%d)\n", pp→serial);
}
```
This code is used in section 76.

78. ⟨ Report an undefined symbol 78 ⟩ ≡
```
{
  *sym_ptr = t→ch;
  *(sym_ptr + 1) = '\0';
  fprintf(stderr, "undefined symbol: %s\n", sym_buf + 1);
  err_count++, s = 0;
  if (t→ch < #3ff) m += #1800 + t→ch, c = 1;
  else m += #1fff;
}
```
This code is used in section 74.

79. ⟨ Check and output the trie 79 ⟩ ≡
```
op_root→mid = Λ;    /* annihilate all the opcodes */
prune(trie_root);
sym_ptr = sym_buf;
if (listing_file) fprintf(listing_file, "\nSymbol table:\n");
ero_lop(lop_stab, 0, 0);
out_stab(trie_root);
ero_lopp(lop_end, ero_ptr);
```
This code is used in section 143.

80. Expressions. The most intricate part of the assembly process is the task of scanning and evaluating expressions in the operand field. Fortunately, `ERISCAL`'s expressions have a simple structure that can be handled easily with a stack-based approach.

Two stacks hold pending data as the operand field is scanned and evaluated. The *op_stack* contains operators that have not yet been performed; the *val_stack* contains values that have not yet been used. After an entire operand list has been scanned, the *op_stack* will be empty and the *val_stack* will hold the operand values needed to assemble the current instruction.

81. Entries on *op_stack* have one of the constant values defined here, and they have one of the precedence levels defined here.

Entries on *val_stack* have *equiv*, *link*, and *status* fields; the *link* points to a trie node if the expression is a symbol that has not yet been subjected to any operations.

```
⟨ Type definitions 25 ⟩ +≡
  typedef enum {
    indirectize, relativize, negate, serialize, complement, registerize,
        plus, minus, times, over, frac, mod, shl, shr, and, or, xor,
        outer_lp, outer_rp, inner_lp, inner_rp
  } stack_op;
  typedef enum {
    zero, weak, strong, unary
  } prec;
  typedef enum {
    pure, reg_val, undefined, rel_undefined,
        ind_pure, ind_reg_val, ind_undefined, ind_rel_undefined
  } stat;
  typedef struct {
    wyde equiv;   /* current value */
    trie_node *link;   /* trie reference for symbol */
    stat status;   /* pure, reg_val, undefined, ... */
  } val_node;
```

```
82. #define top_op op_stack[op_ptr − 1]   /* top entry on the operator stack */
#define top_val val_stack[val_ptr − 1]   /* top entry on the value stack */
#define next_val val_stack[val_ptr − 2]   /* next-to-top entry of the value stack */
⟨ Global variables 26 ⟩ +≡
  stack_op *op_stack;   /* stack for pending operators */
  int op_ptr;   /* number of items on op_stack */
  val_node *val_stack;   /* stack for pending operands */
  int val_ptr;   /* number of items on val_stack */
  prec precedence[ ] = {unary, unary, unary, unary, unary, unary,
      weak, weak, strong, strong, strong, strong, strong, strong, strong, weak, weak,
      zero, zero, zero, zero};   /* precedences of the respective stack_op values */
  stack_op rt_op;   /* newly scanned operator */
  wyde acc;   /* temporary accumulator */
```

```
83. ⟨ Initialize everything 33 ⟩ +≡
  op_stack = (stack_op *) calloc(buf_size, sizeof(stack_op));
  val_stack = (val_node *) calloc(buf_size, sizeof(val_node));
  if (¬op_stack ∨ ¬val_stack) panic("No␣room␣for␣the␣stacks");
```

84. The operand field of an instruction will have been copied into a separate **Char** array called *operand_list* when we reach this part of the program.

⟨ Scan the operand field 84 ⟩ ≡
  p = *operand_list*;
  val_ptr = 0;   /* *val_stack* is empty */
  op_stack[0] = *outer_lp*, *op_ptr* = 1;   /* *op_stack* contains an "outer left parenthesis" */
  while (1) {
    ⟨ Scan opening tokens until putting something on *val_stack* 85 ⟩;
  scan_close: ⟨ Scan a binary operator or closing token, *rt_op* 96 ⟩;
    while (*precedence*[*top_op*] ≥ *precedence*[*rt_op*]) ⟨ Perform the top operation on *op_stack* 97 ⟩;
  hold_op: *op_stack*[*op_ptr*++] = *rt_op*;
  }
operands_done:

This code is used in section 101.

85. A comment that follows an empty operand list needs to be detected here.

⟨ Scan opening tokens until putting something on *val_stack* 85 ⟩ ≡
scan_open: **if** (*isletter*(*p)) ⟨ Scan a symbol 86 ⟩
  else if (*isdigit*(*p)) {
    if (*(*p* + 1) ≡ `'F'`) ⟨ Scan a forward local 87 ⟩
    else if (*(*p* + 1) ≡ `'B'`) ⟨ Scan a backward local 88 ⟩
    else ⟨ Scan a decimal constant 93 ⟩;
  } **else switch** (*p++) {
    case `'#'`: ⟨ Scan a hexadecimal constant 94 ⟩; **break**;
    case `'\''`: ⟨ Scan a character constant 91 ⟩; **break**;
    case `'\"'`: ⟨ Scan a string constant 92 ⟩; **break**;
    case `'@'`: ⟨ Scan the current location 95 ⟩; **break**;
    case `'*'`: *op_stack*[*op_ptr*++] = *indirectize*; **goto** *scan_open*;
    case `'+'`: *op_stack*[*op_ptr*++] = *relativize*; **goto** *scan_open*;
    case `'-'`: *op_stack*[*op_ptr*++] = *negate*; **goto** *scan_open*;
    case `'&'`: *op_stack*[*op_ptr*++] = *serialize*; **goto** *scan_open*;
    case `'~'`: *op_stack*[*op_ptr*++] = *complement*; **goto** *scan_open*;
    case `'$'`: *op_stack*[*op_ptr*++] = *registerize*; **goto** *scan_open*;
    case `'('`: *op_stack*[*op_ptr*++] = *inner_lp*; **goto** *scan_open*;
    default:
      if (*p* ≡ *operand_list* + 1) {   /* treat operand list as empty */
        operand_list[0] = `'0'`, *operand_list*[1] = `'\0'`, *p* = *operand_list*;
        goto *scan_open*;
      }
      if (*(*p* − 1)) *derr*(`"syntax␣error␣at␣character␣`%c'"`,
          *(*p* − 1))*derr*(`"syntax␣error␣after␣character␣`%c'"`, *(*p* − 2))
    }

This code is used in section 84.

86. ⟨ Scan a symbol 86 ⟩ ≡

```
{
  if (*p ≡ ':') tt = trie_search(trie_root, p + 1);
  else tt = trie_search(cur_prefix, p);
  p = terminator;
symbol_found: val_ptr++;
  pp = tt→sym;
  if (¬pp) pp = tt→sym = new_sym_node(true);
  top_val.link = tt, top_val.equiv = pp→equiv;
  if (pp→link ≡ PREDEFINED) pp→link = DEFINED;
  top_val.status = (pp→link ≡ DEFINED ? pure : pp→link ≡ REGISTER ? reg_val : undefined);
}
```

This code is used in section 85.

87. ⟨ Scan a forward local 87 ⟩ ≡

```
{
  tt = &forward_local_host[*p − '0']; p += 2; goto symbol_found;
}
```

This code is used in section 85.

88. ⟨ Scan a backward local 88 ⟩ ≡

```
{
  tt = &backward_local_host[*p − '0']; p += 2; goto symbol_found;
}
```

This code is used in section 85.

89. Statically allocated variables *forward_local_host*[j] and *backward_local_host*[j] masquerade as nodes of the trie.

⟨ Global variables 26 ⟩ +≡

```
trie_node forward_local_host[10], backward_local_host[10];
sym_node forward_local[10], backward_local[10];
```

90. Initially `0H`, `1H`, ..., `9H` are defined to be zero.

⟨ Initialize everything 33 ⟩ +≡

```
for (j = 0; j < 10; j++) {
  forward_local_host[j].sym = &forward_local[j];
  backward_local_host[j].sym = &backward_local[j];
  backward_local[j].link = DEFINED;
}
```

91. We have already checked to make sure that the character constant is legal.

⟨ Scan a character constant 91 ⟩ ≡

```
acc = *p;
p += 2;
goto constant_found;
```

This code is used in section 85.

92. ⟨ Scan a string constant 92 ⟩ ≡

```
acc = *p;
if (*p ≡ '\"') {
  p++;
  acc = 0;
  err("*null␣string␣is␣treated␣as␣zero")
} else if (*(p + 1) ≡ '\"') p += 2;
else *p = '\"', *−−p = ',';
goto constant_found;
```

This code is used in section 85.

93. ⟨ Scan a decimal constant 93 ⟩ ≡

```
acc = *p − '0';
for (p++; isdigit(*p); p++) {
  acc += (acc ≪ 2);
  acc = (acc ≪ 1) + (*p − '0');
}
constant_found: val_ptr++;
  top_val.link = Λ;
  top_val.equiv = acc;
  top_val.status = pure;
```

This code is used in section 85.

94. ⟨ Scan a hexadecimal constant 94 ⟩ ≡

```
if (¬isxdigit(*p)) err("illegal␣hexadecimal␣constant");
acc = 0;
for ( ; isxdigit(*p); p++) {
  acc = (acc ≪ 4) + (*p − '0');
  if (*p ≥ 'a') acc += '0' − 'a' + 10;
  else if (*p ≥ 'A') acc += '0' − 'A' + 10;
}
goto constant_found;
```

This code is used in section 85.

95. ⟨ Scan the current location 95 ⟩ ≡

```
acc = cur_loc;
goto constant_found;
```

This code is used in section 85.

96. ⟨ Scan a binary operator or closing token, *rt_op* 96 ⟩ ≡

```
switch (*p++) {
case '+': rt_op = plus; break;
case '-': rt_op = minus; break;
case '*': rt_op = times; break;
case '/': if (*p ≠ '/') rt_op = over;
  else p++, rt_op = frac; break;
case '%': rt_op = mod; break;
case '<': rt_op = shl; goto sh_check;
case '>': rt_op = shr;
sh_check: p++; if (*(p - 1) ≡ *(p - 2)) break;
  derr("syntax error at '%c'", *(p - 2));
case '&': rt_op = and; break;
case '|': rt_op = or; break;
case '^': rt_op = xor; break;
case ')': rt_op = inner_rp; break;
case '\0': case ',': rt_op = outer_rp; break;
default: derr("syntax error at '%c'", *(p - 1));
}
```

This code is used in section 84.

97. ⟨ Perform the top operation on *op_stack* 97 ⟩ ≡

```
switch (op_stack[--op_ptr]) {
case inner_lp: if (rt_op ≡ inner_rp) goto scan_close;
  err("*missing right parenthesis"); break;
case outer_lp: if (rt_op ≡ outer_rp) {
    if ((top_val.status ≡ reg_val ∨ top_val.status ≡ ind_reg_val) ∧
          top_val.equiv > #f) {
      err("*register number too large, will be reduced mod 16");
      top_val.equiv &= #f;
    }
    if (¬*(p - 1)) goto operands_done;
    else rt_op = outer_lp; goto hold_op;    /* comma */
  } else {
    op_ptr++;
    err("*missing left parenthesis");
    goto scan_close;
  }
⟨ Cases for unary operators 99 ⟩
⟨ Cases for binary operators 98 ⟩
}
```

This code is used in section 84.

98. Now we come to the part where equivalents are changed by unary or binary operators found in the expression being scanned.

The most typical operator, and in some ways the fussiest one to deal with, is binary addition. Once we've written the code for this case, the other cases almost take care of themselves.

⟨ Cases for binary operators 98 ⟩ ≡
case *plus*: **if** (*top_val.status* ≥ *ind_pure*) *err*("cannot␣add␣an␣indirect␣quantity");
  if (*next_val.status* ≥ *ind_pure*) *err*("cannot␣add␣to␣an␣indirect␣quantity");
  if (*top_val.status* ≥ *undefined*) *err*("cannot␣add␣an␣undefined␣quantity");
  if (*next_val.status* ≥ *undefined*) *err*("cannot␣add␣to␣an␣undefined␣quantity");
  if (*top_val.status* ≡ *reg_val* ∧ *next_val.status* ≡ *reg_val*) *err*("cannot␣add␣two␣register␣numbers");
  next_val.equiv += *top_val.equiv*;
fin_bin: *next_val.status* = (*top_val.status* ≡ *next_val.status* ? *pure* : *reg_val*);
  val_ptr −−;
delink: *top_val.link* = Λ; **break**;

See also section 100.

This code is used in section 97.

99. **#define** *unary_check*(*verb*) **if** (*top_val.status* ≠ *pure*) *derr*("can␣%s␣pure␣values␣only", *verb*)

⟨ Cases for unary operators 99 ⟩ ≡
case *indirectize*: **if** (*top_val.status* < *ind_pure*) *top_val.status* += *ind_pure* − *pure*; **goto** *delink*;
case *relativize*: **if** (¬(*top_val.status* ≡ *pure* ∨ *top_val.status* ≡ *undefined*))
    err("can␣relativise␣only␣pure␣values␣and␣forward␣references");
  if (*top_val.status* ≡ *pure*) *top_val.equiv* −= *cur_loc*;
  else *top_val.status* = *rel_undefined*; **goto** *delink*;
case *negate*: *unary_check*("negate");
  top_val.equiv = *zero_wyde* − *top_val.equiv*; **goto** *delink*;
case *complement*: *unary_check*("complement");
  top_val.equiv = ∼*top_val.equiv*;
  goto *delink*;
case *registerize*: *unary_check*("registerize");
  top_val.status = *reg_val*; **goto** *delink*;
case *serialize*: **if** (¬*top_val.link*) *err*("can␣take␣serial␣number␣of␣symbol␣only");
  top_val.equiv = *top_val.link*→*sym*→*serial*;
  top_val.status = *pure*; **goto** *delink*;

This code is used in section 97.

100. **#define** *binary_check*(*verb*)
    if (*top_val.status* ≠ *pure* ∨ *next_val.status* ≠ *pure*) *derr*("can␣%s␣pure␣values␣only", *verb*)

⟨ Cases for binary operators 98 ⟩ +≡

```
case minus: if (top_val.status ≥ ind_pure) err("cannot␣subtract␣an␣indirect␣quantity");
  if (top_val.status ≥ undefined) err("cannot␣subtract␣an␣undefined␣quantity");
  if (next_val.status ≥ ind_pure) err("cannot␣subtract␣from␣an␣indirect␣quantity");
  if (next_val.status ≥ undefined) err("cannot␣subtract␣from␣an␣undefined␣quantity");
  if (top_val.status ≡ reg_val ∧ next_val.status ≠ reg_val)
    err("cannot␣subtract␣register␣number␣from␣pure␣value");
  next_val.equiv −= top_val.equiv; goto fin_bin;
case times: binary_check("multiply");
  next_val.equiv = wmult(next_val.equiv, top_val.equiv); goto fin_bin;
case over: case mod: binary_check("divide");
  if (top_val.equiv ≡ 0) err("*division␣by␣zero");
  next_val.equiv = wdiv(zero_wyde, next_val.equiv, top_val.equiv);
  if (op_stack[op_ptr] ≡ mod) next_val.equiv = aux;
  goto fin_bin;
case frac: binary_check("compute␣a␣ratio␣of");
  if (next_val.equiv ≥ top_val.equiv) err("*illegal␣fraction");
  next_val.equiv = wdiv(next_val.equiv, zero_wyde, top_val.equiv); goto fin_bin;
case shl: case shr: binary_check("compute␣a␣bitwise␣shift␣of");
  if (top_val.equiv > 15) next_val.equiv = zero_wyde;
  else if (op_stack[op_ptr] ≡ shl) next_val.equiv ≪= top_val.equiv;
  else next_val.equiv ≫= top_val.equiv;
  goto fin_bin;
case and: binary_check("compute␣bitwise␣and␣of");
  next_val.equiv &= top_val.equiv;
  goto fin_bin;
case or: binary_check("compute␣bitwise␣or␣of");
  next_val.equiv |= top_val.equiv;
  goto fin_bin;
case xor: binary_check("compute␣bitwise␣xor␣of");
  next_val.equiv ⊕= top_val.equiv;
  goto fin_bin;
```

101. Assembling an instruction. Now let's move up from the expression level to the instruction level. We get to this part of the program at the beginning of a line, or after a semicolon at the end of an instruction earlier on the current line. Our current position in the buffer is the value of *buf_ptr*.

```
⟨ Process the next ERISCAL instruction or comment 101 ⟩ ≡
  p = buf_ptr; buf_ptr = "";
  ⟨ Scan the label field; goto bypass if there is none 102 ⟩;
  ⟨ Scan the opcode field; goto bypass if there is none 103 ⟩;
  ⟨ Copy the operand field 105 ⟩;
  buf_ptr = p;
  if (spec_mode ∧ ¬(op_bits & spec_bit)) derr("cannot␣use␣`%s'␣in␣special␣mode", op_field);
  if ((op_bits & no_label_bit) ∧ lab_field[0]) {
    derr("*label␣field␣of␣`%s'␣instruction␣is␣ignored", op_field);
    lab_field[0] = '\0';
  }
  ⟨ Scan the operand field 84 ⟩;
  if (opcode ≡ GREG) ⟨ Allocate a global register 106 ⟩;
  if (lab_field[0]) ⟨ Define the label 107 ⟩;
  ⟨ Do the operation 113 ⟩;
bypass:
```

This code is used in section 137.

```
102. ⟨ Scan the label field; goto bypass if there is none 102 ⟩ ≡
  if (¬*p) goto bypass;
  q = lab_field;
  if (¬isspace(*p)) {
    if (¬isdigit(*p) ∧ ¬isletter(*p)) goto bypass;     /* comment */
    for (*q++ = *p++; isdigit(*p) ∨ isletter(*p); p++, q++) *q = *p;
    if (*p ∧ ¬isspace(*p)) derr("label␣syntax␣error␣at␣`%c'", *p);
  }
  *q = '\0';
  if (isdigit(lab_field[0]) ∧ (lab_field[1] ≠ 'H' ∨ lab_field[2]))
    derr("improper␣local␣label␣`%s'", lab_field);
  for (p++; isspace(*p); p++) ;
```

This code is used in section 101.

103. We copy the opcode field to a special buffer because we might want to refer to the symbolic opcode in error messages.

```
⟨ Scan the opcode field; goto bypass if there is none 103 ⟩ ≡
  q = op_field; while (isletter(*p) ∨ isdigit(*p)) *q++ = *p++;
  *q = '\0';
  if (¬isspace(*p) ∧ *p ∧ op_field[0]) derr("opcode␣syntax␣error␣at␣`%c'", *p);
  pp = trie_search(op_root, op_field)→sym;
  if (¬pp) {
    if (op_field[0]) derr("unknown␣operation␣code␣`%s'", op_field);
    if (lab_field[0]) derr("*no␣opcode;␣label␣`%s'␣will␣be␣ignored", lab_field);
    goto bypass;
  }
  opcode = (pp→equiv ≫ 8) & #ff, op_bits = pp→equiv & #ff;
  while (isspace(*p)) p++;
```

This code is used in section 101.

104. ⟨ Global variables 26 ⟩ +≡

```
wyde opcode;     /* numeric code for ELTE RISC operation or ERISCAL pseudo-op */
wyde op_bits;    /* flags describing an operator's special characteristics */
wyde arg_num;    /* number of arguments: 0,1,2,¿=3 */
```

105. We copy the operand field to a special buffer so that we can change string constants while scanning them later.

⟨ Copy the operand field 105 ⟩ ≡

```
q = operand_list;
while (*p) {
  if (*p ≡ ';') break;
  if (*p ≡ '\'') {
    *q++ = *p++;
    if (¬*p) err("incomplete␣character␣constant");
    *q++ = *p++;
    if (*p ≠ '\'') err("illegal␣character␣constant");
  } else if (*p ≡ '\"') {
    for (*q++ = *p++; *p ∧ *p ≠ '\"'; p++, q++) *q = *p;
    if (¬*p) err("incomplete␣string␣constant");
  }
  *q++ = *p++;
  if (isspace(*p)) break;
}
while (isspace(*p)) p++;
if (*p ≡ ';') p++;
else p = "";     /* if not followed by semicolon, rest of the line is a comment */
if (q ≡ operand_list) *q++ = '0';     /* change empty operand field to '0' */
*q = '\0';
```

This code is used in section 101.

106. ⟨ Allocate a global register 106 ⟩ ≡

```
{ if (greg ≡ 15) err("too␣many␣global␣registers")
  else {
    ++greg;
    greg_val[greg] = val_stack[0].equiv;
  }
}
```

This code is used in section 101.

107. If the label is, say 2H, we will already have used the old value of 2B when evaluating the operands. Furthermore, an operand of 2F will have been treated as undefined, which it still is.

Symbols can be defined more than once, but only if each definition gives them the same equivalent value.

A warning message is given when a predefined symbol is being redefined, if its predefined value has already been used.

```
⟨Define the label 107⟩ ≡
  {
    sym_node *new_link = DEFINED;
    acc = cur_loc;
    if (opcode ≡ IS) {
      cur_loc = val_stack[0].equiv;
      if (val_stack[0].status ≡ reg_val) new_link = REGISTER;
    } else if (opcode ≡ GREG) cur_loc = greg, new_link = REGISTER;
    ⟨Find the symbol table node, pp 109⟩;
    if (pp→link ≡ DEFINED ∨ pp→link ≡ REGISTER) {
      if (pp→seg ≠ cur_seg ∨ pp→equiv ≠ cur_loc ∨ pp→link ≠ new_link) {
        if (pp→serial) derr("symbol␣'%s'␣is␣already␣defined", lab_field);
        pp→serial = ++serial_number;
        derr("*redefinition␣of␣predefined␣symbol␣'%s'", lab_field);
      }
    } else if (pp→link ≡ PREDEFINED) pp→serial = ++serial_number;
    else if (pp→link) {
      if (new_link ≡ REGISTER) err("future␣reference␣cannot␣be␣to␣a␣register");
      do ⟨Fix prior references to this label 110⟩ while (pp→link);
    }
    if (isdigit(lab_field[0])) pp = &backward_local[lab_field[0] − '0'];
    pp→equiv = cur_loc; pp→seg = cur_seg; pp→link = new_link;
    ⟨Fix references that might be in the val_stack 108⟩;
    if (listing_file ∧ (opcode ≡ IS ∨ opcode ≡ LOC)) ⟨Make special listing to show the label equivalent 112⟩;
    cur_loc = acc;
  }
```

This code is used in section 101.

108. ⟨Fix references that might be in the *val_stack* 108⟩ ≡

```
  if (¬isdigit(lab_field[0]))
    for (j = 0; j < val_ptr; j++)
      if (val_stack[j].status ≡ undefined ∧ val_stack[j].link→sym ≡ pp) {
        val_stack[j].status = (new_link ≡ REGISTER ? reg_val : pure);
        val_stack[j].equiv = cur_loc;
      }
```

This code is used in section 107.

109. ⟨Find the symbol table node, *pp* 109⟩ ≡

```
  if (isdigit(lab_field[0])) pp = &forward_local[lab_field[0] − '0'];
  else {
    if (lab_field[0] ≡ ':') tt = trie_search(trie_root, lab_field + 1);
    else tt = trie_search(cur_prefix, lab_field);
    pp = tt→sym;
    if (¬pp) pp = tt→sym = new_sym_node(true);
  }
```

This code is used in section 107.

110. ⟨ Fix prior references to this label 110 ⟩ ≡

```
{
  qq = pp→link;
  pp→link = qq→link;
  ⟨ Fix a future reference 111 ⟩
  recycle_fixup(qq);
}
```

This code is used in section 107.

111. ⟨ Fix a future reference 111 ⟩ ≡

```
{ wyde w;
int s;
s = qq→serial; if (s & seg_bit ≠ cur_seg)     /* different segment */
dderr("location␣#%01x%04x␣is␣in␣a␣different␣segment", qq→seg, qq→equiv)
else {
  ⟨ Make sure cur_loc and ero_cur_loc refer to the same wyde 55 ⟩
  if (s & rel_bit) {
    k = 0;
    w = cur_loc − qq→equiv;
    if (s & nyb_bit) {
      if (¬(w & #8000))
        if (w < #8) ero_lopp(lop_fixr, w);
        else k = 1;
      else if (w ≥ #fff8) ero_lopp(lop_fixr, w & #f);
      else k = 1;
    } else ero_lopp(lop_fixr, w);
    if (k) dderr("relative␣address␣in␣location␣#%01x%04x␣is␣too␣far␣away", qq→seg, qq→equiv);
  }
  else {
    k = 0;
    w = qq→equiv;
    if (s & nyb_bit) {
      if (¬(w & #8000))
        if (w < #8) ero_lopp(lop_fixw, w);
        else k = 1;
      else if (w ≥ #fff8) ero_lopp(lop_fixw, w & #f);
      else k = 1;
    } else ero_lopp(lop_fixr, w);
    if (k) dderr("defined␣nybble␣in␣location␣#%01x%04x␣is␣too␣large", qq→seg, qq→equiv);
  }
}
}
```

This code is used in section 110.

112. ⟨ Make special listing to show the label equivalent 112 ⟩ ≡

```
if (new_link ≡ DEFINED) {
  fprintf(listing_file, "(%04x)", cur_loc);
  flush_listing_line("␣");
} else {
  fprintf(listing_file, "($%02d)", cur_loc & #f);
  flush_listing_line("␣␣␣␣␣␣");
}
```

This code is used in section 107.

113. ⟨ Do the operation 113 ⟩ ≡

```
future_bits = 0;
arg_num = op_bits & arg_num_bits;
if (arg_num ≡ 3) ⟨ Do a many-operand operation 114 ⟩
else switch (arg_num) {
  case 0: if (¬(val_ptr ≤ 1)) derr("opcode␣`%s'␣needs␣no␣operand", op_field);
    ⟨ Do a one-operand operation 133 ⟩;
    break;
  case 1: if (¬(val_ptr ≤ 1)) derr("opcode␣`%s'␣needs␣one␣operand", op_field);
    ⟨ Do a one-operand operation 133 ⟩;
    break;
  case 2: if (¬(val_ptr ≡ 2)) derr("opcode␣`%s'␣must␣have␣two␣operands", op_field)
    ⟨ Do a two-operand operation 117 ⟩;
    break;
  default: derr("too␣many␣operands␣for␣opcode␣`%s'", op_field);
  }
```

This code is used in section 101.

114. The many-operand operator is `WYDE`.

⟨ Do a many-operand operation 114 ⟩ ≡

```
for (j = 0; j < val_ptr; j++) {
  ⟨ Deal with cases where val_stack[j] is impure 115 ⟩;
  if (val_stack[j].status ≡ undefined ∨ val_stack[j].status ≡ rel_undefined) assemble(0, #f0);
  else assemble(val_stack[j].equiv, 0);
}
```

This code is used in section 113.

115. ⟨ Deal with cases where *val_stack*[*j*] is impure 115 ⟩ ≡

```
if (val_stack[j].status ≥ ind_pure) {
  err("*indirect␣number␣used␣as␣a␣constant")
  val_stack[j].status −= ind_pure;
}
if (val_stack[j].status ≡ reg_val) err("*register␣number␣used␣as␣a␣constant")
else if (val_stack[j].status ≡ undefined) {
  pp = val_stack[j].link→sym;
  qq = new_sym_node(false);
  qq→link = pp→link;
  pp→link = qq;
  qq→serial = cur_seg;
  qq→equiv = cur_loc;
}
else if (val_stack[j].status ≡ rel_undefined) {
  pp = val_stack[j].link→sym;
  qq = new_sym_node(false);
  qq→link = pp→link;
  pp→link = qq;
  qq→serial = cur_seg + rel_bit;
  qq→equiv = cur_loc;
}
```

This code is used in section 114.

116. Individual fields of an instruction are placed into global variables x, y, z.

⟨ Global variables 26 ⟩ +≡

```
wyde x, y, z;   /* pieces for assembly */
int future_bits;   /* places where there are future references */
char addr_mode;   /* places where there are the addressing mode bits */
```

117. ⟨ Do a two-operand operation 117 ⟩ ≡

```
z = 0;   /* Presuppose one-wyde code */
⟨ Find the addressing mode 125 ⟩;
switch (addr_mode) {
case 0: ⟨ Check if this case is allowed by this instruction 118 ⟩
   ⟨ Handle exceptions for 0 addr_mode 120 ⟩
   ⟨ Do the SRC field, it have to be a constant 123 ⟩;
   break;
case 1: ⟨ Check if this case is allowed by this instruction 118 ⟩
   ⟨ Change the X and Y fields 119 ⟩
   ⟨ Do the SRC field, it have to be a constant 123 ⟩
   break;
case 2: ⟨ Check if this case is allowed by this instruction 118 ⟩
   ⟨ Handle exceptions for 2 addr_mode 121 ⟩
   ⟨ Do the SRC field, it have to be a register 124 ⟩
   break;
case 3: ⟨ Check if this case is allowed by this instruction 118 ⟩
   ⟨ Handle exceptions for 3 addr_mode 122 ⟩
   ⟨ Do the SRC field, it is indirect 126 ⟩
   break;
}
assemble_DST: ⟨ Do the DST field 127 ⟩;
assemble_inst: assemble((x ≪ 12) + ((opcode + addr_mode) ≪ 4) + y, future_bits);
if (z) ⟨ Do the second wyde 130 ⟩;
```

This code is used in section 113.

118. ⟨ Check if this case is allowed by this instruction 118 ⟩ ≡

```
if (¬((1 ≪ (2 + addr_mode)) & op_bits))
   dderr("addressing␣mode␣%01d␣is␣not␣allowed␣by␣'%s'", addr_mode, op_field);
```

This code is used in section 117.

119. ⟨ Change the X and Y fields 119 ⟩ ≡

```
acc = val_stack[1].equiv;
val_stack[1].equiv = val_stack[0].equiv;
val_stack[0].equiv = acc;
acc = val_stack[0].status;
val_stack[1].status = val_stack[0].status;
val_stack[0].status = acc;
```

This code is used in section 117.

120. ⟨ Handle exceptions for 0 addr_mode 120 ⟩ ≡

```
if (opcode = #e5) {   /* TRAP */
   if (val_stack[0].status ≠ reg_val ∨ val_stack[0].equiv ≡ 0)
      derr("X␣field␣of␣'%s'␣should␣be␣a␣nonzero␣register", op_field);
   if (val_stack[1].status ≠ pure) derr("Y␣field␣of␣'%s'␣should␣be␣a␣number", op_field);
   if (val_stack[1].equiv > #f) err("Y␣field␣doesn't␣fit␣in␣one␣unsigned␣nybble");
   y = val_stack[1].equiv & #f; break;
}
```

This code is used in section 117.

121. ⟨ Handle exceptions for 2 *addr_mode* 121 ⟩ ≡

```
if (opcode ≡ #e5) {    /* RESUME */
  if (val_stack[0].status ≠ reg_val ∨ val_stack[0].equiv)
    derr("DSC␣field␣of␣'%s'␣should␣be␣the␣zero␣register", op_field);
  y = 0, addr_mode = 0; goto assemble_DST;
}
else if (opcode ≡ #c5)    /* MOR */
  addr_mode = 0;
```

This code is used in section 117.

122. ⟨ Handle exceptions for 3 *addr_mode* 122 ⟩ ≡

```
{
  if (opcode ≡ #d5) {    /* PUSH */
    if (val_stack[1].status ≠ reg_val) derr("SRC␣field␣of␣'%s'␣should␣be␣a␣register", op_field);
    if (val_stack[1].equiv > #f) err("*SRC␣field␣doesn't␣fit␣in␣one␣unsigned␣nybble");
    y = val_stack[1].equiv & #f, addr_mode = 0; goto assemble_DST;
  }
  else if (opcode ≡ #e7)    /* POP */
    addr_mode = 0;
}
```

This code is used in section 117.

123. ⟨ Do the SRC field, it have to be a constant 123 ⟩ ≡

```
{
if (val_stack[1].status ≡ undefined) ⟨ Assemble SRC as a future reference 128 ⟩
else if (val_stack[1].status ≡ rel_undefined) ⟨ Assemble SRC as a relative future reference 129 ⟩
else if (val_stack[1].status ≡ reg_val) derr("*SRC␣field␣of␣'%s'␣should␣not␣be␣a␣register␣number",
      op_field)
else {
  if (val_stack[1].equiv ≤ #fff8 ∧ val_stack[1].equiv > #7)
    err("*SRC␣field␣doesn't␣fit␣in␣one␣signed␣nybble");
  y = val_stack[1].equiv & #f;
}
}
```

This code is used in section 117.

124. ⟨ Do the SRC field, it have to be a register 124 ⟩ ≡

```
if (val_stack[1].status ≠ reg_val)
  derr("*SRC␣field␣of␣'%s'␣should␣be␣a␣register␣number", op_field);
if (val_stack[1].equiv > #f) err("*SRC␣field␣doesn't␣fit␣in␣one␣unsigned␣nybble");
y = val_stack[1].equiv & #f;
```

This code is used in sections 117 and 126.

125. ⟨ Find the addressing mode 125 ⟩ ≡

```
if (val_stack[1].status ≥ ind_pure) addr_mode = 3;
else if (val_stack[1].status = reg_val)
  if (val_stack[0].status = reg_val) addr_mode = 1;
  else addr_mode = 2;
else addr_mode = 0;
```

This code is used in section 117.

126. ⟨ Do the SRC field, it is indirect 126 ⟩ ≡
```
val_stack[1].status −= ind_pure;
if (val_stack[1].status ≡ reg_val) {
  if (¬val_stack[1].equiv) derr("*SRC␣field␣of␣`%s'␣should␣not␣be␣*$0", op_field);
  ⟨ Do the SRC field, it have to be a register 124 ⟩;
}
else y = 0, z = 1;
```
This code is used in section 117.

127. ⟨ Do the DST field 127 ⟩ ≡
```
if (val_stack[0].status ≠ reg_val)
  derr("*DST␣field␣of␣`%s'␣should␣be␣a␣register␣number", op_field);
if (val_stack[0].equiv > #f) err("*DST␣field␣doesn't␣fit␣in␣one␣nybble");
x = val_stack[0].equiv & #f;
```
This code is used in section 117.

128. ⟨ Assemble SRC as a future reference 128 ⟩ ≡
```
{
  pp = val_stack[0].link→sym;
  qq = new_sym_node(false);
  qq→link = pp→link;
  pp→link = qq;
  qq→serial = cur_seg + nyb_bit;
  qq→equiv = cur_loc;
  y = 0;
  future_bits = #80;
  goto assemble_DST;
}
```
This code is used in section 123.

129. ⟨ Assemble SRC as a relative future reference 129 ⟩ ≡
```
{
  pp = val_stack[0].link→sym;
  qq = new_sym_node(false);
  qq→link = pp→link;
  pp→link = qq;
  qq→serial = cur_seg + nyb_bit + rel_bit;
  qq→equiv = cur_loc;
  y = 0;
  future_bits = #80;
  goto assemble_DST;
}
```
This code is used in section 123.

130. ⟨ Do the second wyde 130 ⟩ ≡
```
if (val_stack[1].status ≡ pure) assemble(val_stack[1].equiv, 0);
else if (val_stack[1].status ≡ undefined) ⟨ Assemble YZ as a future reference 131 ⟩
else   /* val_stack[1].status ≡ rel_undefined */
  ⟨ Assemble YZ as a relative future reference 132 ⟩;
```
This code is used in section 117.

131. ⟨ Assemble YZ as a future reference 131 ⟩ ≡

```
{
  pp = val_stack[1].link→sym;
  qq = new_sym_node(false);
  qq→link = pp→link;
  pp→link = qq;
  qq→serial = cur_seg;
  qq→equiv = cur_loc;
  assemble(0, #f0);
}
```

This code is used in section 130.

132. ⟨ Assemble YZ as a relative future reference 132 ⟩ ≡

```
{
  pp = val_stack[1].link→sym;
  qq = new_sym_node(false);
  qq→link = pp→link;
  pp→link = qq;
  qq→serial = cur_seg + rel_bit;
  qq→equiv = cur_loc;
  assemble(0, #f0);
}
```

This code is used in section 130.

133. ⟨ Do a one-operand operation 133 ⟩ ≡

```
switch (opcode) {    /* Pseudo operations */
case CODE:
  if (harvard ∧ cur_seg) {
    cur_data_loc = cur_loc;
    cur_loc = cur_code_loc;
    cur_seg = 0;
  } goto bypass;
case DATA:
  if (harvard ∧ ¬cur_seg) {
    cur_code_loc = cur_loc;
    cur_loc = cur_data_loc;
    cur_seg = harvard;
  } goto bypass;
case LOC: cur_loc = val_stack[0].equiv;
case IS: goto bypass;
case PREFIX: if (¬val_stack[0].link) err("not␣a␣valid␣prefix");
  cur_prefix = val_stack[0].link; goto bypass;
case GREG: if (listing_file) ⟨ Make listing for GREG 135 ⟩;
  goto bypass;
case BSPEC: if (val_stack[0].equiv > #ffff) err("*operand␣of␣‘BSPEC’␣doesn’t␣fit␣in␣a␣wyde");
  ero_loc(); ero_sync();
  ero_lopp(lop_spec, val_stack[0].equiv);
  spec_mode = true; spec_mode_loc = 0; goto bypass;
case ESPEC: spec_mode = false; goto bypass;
}
```

This code is used in section 113.

134. ⟨ Global variables 26 ⟩ +≡

wyde *greg_val*[256]; /* initial values of global registers */

135. ⟨ Make listing for `GREG` 135 ⟩ ≡

fprintf (*listing_file*, `"($%02d=#%04x"`, *greg*, *val_stack*[0].*equiv*);
flush_listing_line(`"␣"`);

This code is used in section 133.

136. Running the program. On a UNIX-like system, the command

```
eriscal [options] sourcefilename
```

will assemble the `ERISCAL` program in file `sourcefilename`, writing any error messages on the standard error file. (Nothing is written to the standard output.) The options, which may appear in any order, are:

- `-o objectfilename` Send the output to a binary file called `objectfilename`. If no `-o` specification is given, the object file name is obtained from the input file name by changing the final letter from '`s`' to '`o`', or by appending '`.ero`' if `sourcefilename` doesn't end with `s`.
- `-l listingname` Output a listing of the assembled input and output to a text file called `listingname`.
- `-h` Allow Harward type architecture, assembling data to a separate space from instructions.
- `-b bufsize` Allow up to `bufsize` characters per line of input.

137. Here, finally, is the overall structure of this program.

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <time.h>
  ⟨ Preprocessor definitions 32 ⟩
  ⟨ Type definitions 25 ⟩
  ⟨ Global variables 26 ⟩
  ⟨ Subroutines 27 ⟩
  int main(argc, argv)
      int argc; char *argv[];
  {
    register int j, k;      /* all-purpose integers */
    ⟨ Local variables 41 ⟩;
    ⟨ Process the command line 138 ⟩;
    ⟨ Initialize everything 33 ⟩;
    while (1) {
      ⟨ Get the next line of input text, or break if the input has ended 35 ⟩;
      while (1) {
        ⟨ Process the next ERISCAL instruction or comment 101 ⟩;
        if (¬*buf_ptr) break;
      }
      if (listing_file) {
        if (listing_bits) listing_clear();
        else if (¬line_listed) flush_listing_line("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣");
      }
    }
    ⟨ Finish the assembly 143 ⟩;
  }
```

138. ⟨ Process the command line 138 ⟩ ≡

```
for (j = 1; j < argc − 1 ∧ argv[j][0] ≡ '-'; j++)
  if (¬argv[j][2]) {
    if (argv[j][1] ≡ 'h') harvard = true;
    else if (argv[j][1] ≡ 'o') j++, strcpy(obj_file_name, argv[j]);
    else if (argv[j][1] ≡ 'l') j++, strcpy(listing_name, argv[j]);
    else if (argv[j][1] ≡ 'b' ∧ sscanf(argv[j + 1], "%d", &buf_size) ≡ 1) j++;
    else break;
  } else if (argv[j][1] ≠ 'b' ∨ sscanf(argv[j] + 2, "%d", &buf_size) ≠ 1) break;
if (j ≠ argc − 1) {
  fprintf(stderr, "Usage:␣%s␣%s␣sourcefilename\n", argv[0],
      "[-l␣listingname]␣[-b#]␣[-o␣objectfilename]");
  exit(−1);
}
src_file_name = argv[j];
```

This code is used in section 137.

139. ⟨ Open the files 139 ⟩ ≡

```
src_file = fopen(src_file_name, "r");
if (¬src_file) dpanic("Can't␣open␣the␣source␣file␣%s", src_file_name);
if (¬obj_file_name[0]) {
  j = strlen(src_file_name);
  if (src_file_name[j − 1] ≡ 's') {
    strcpy(obj_file_name, src_file_name); obj_file_name[j − 1] = 'o';
  }
  else sprintf(obj_file_name, "%s.ero", src_file_name);
}
obj_file = fopen(obj_file_name, "wb");
if (¬obj_file) dpanic("Can't␣open␣the␣object␣file␣%s", obj_file_name);
if (listing_name[0]) {
  listing_file = fopen(listing_name, "w");
  if (¬listing_file) dpanic("Can't␣open␣the␣listing␣file␣%s", listing_name);
}
```

This code is used in section 141.

140. ⟨ Global variables 26 ⟩ +≡

```
char *src_file_name;   /* name of the ERISCAL input file */
char obj_file_name[FILENAME_MAX + 1];   /* name of the binary output file */
char listing_name[FILENAME_MAX + 1];   /* name of the optional listing file */
FILE *src_file, *obj_file, *listing_file;
bool harvard;   /* 1 if separate data memory, else 0 */
int buf_size;   /* maximum number of characters per line of input */
tetra present_time;   /* THE time */
```

141. ⟨ Initialize everything 33 ⟩ +≡

```
⟨ Open the files 139 ⟩;
filename[0] = src_file_name;
filename_count = 1;
⟨ Output the preamble 142 ⟩;
```

142. ⟨ Output the preamble 142 ⟩ ≡

```
ero_lopp(lop_pre, 3);
ero_wyde(#101);
present_time = time(Λ);
ero_wyde(present_time ≫ 16);
ero_wyde(present_time & #ffff);
ero_cur_file = −1;
```

This code is used in section 141.

143. ⟨ Finish the assembly 143 ⟩ ≡

```
⟨ Output the postamble 145 ⟩;
⟨ Check and output the trie 79 ⟩;
⟨ Report any undefined local symbols 146 ⟩;
if (err_count)
  if (err_count > 1) fprintf(stderr, "(%d errors were found.)\n", err_count);
  else fprintf(stderr, "(One error was found.)\n");
exit(err_count);
```

This code is used in section 137.

144. ⟨ Global variables 26 ⟩ +≡

```
int greg = 0;    /* global register allocator */
```

145. ⟨ Output the postamble 145 ⟩ ≡

```
ero_lopp(lop_post, greg + 1);
greg_val[0] = trie_search(trie_root, "Main")→sym→equiv;
for (j = 0; j ≤ greg; j++) ero_wyde(greg_val[j]);
```

This code is used in section 143.

146. ⟨ Report any undefined local symbols 146 ⟩ ≡

```
for (j = 0; j < 10; j++)
  if (forward_local[j].link) ++err_count, fprintf(stderr, "undefined local symbol %dF\n", j);
```

This code is used in section 143.

147. Index.

March 31, 2014 at 15:50

ERISCAL

Preliminary work on the Stanford GraphBase project was supported in part by National Science Foundation grant CCR-86-10181.