1. Preface. This program is part of the solution to exercise 16 of D. E. Knuth, *The Art of Computer Programming, Fascicle 1*, p45. In this exercise the design and implementation is requested for a `PL/MMIX` language based on the PL/360 Language designed by N. Wirth.

2. Introduction. PL/MMIX is a very low level language that borrows part of its syntax from the C language. Most of the statements translate into one MMIX assembler instruction. The author intends to provide a language that is more readable than an assembly language and that relieves the programmer of the burden of managing register numbers explicitly, whilst maintaining all the facilities of an assembler language.

A program in PL/MMIX is composed of a sequence of statements. Where one would write `ADD a,b,c` in MMIXAL, the equivalent in PL/MMIX would be `a=b+c;`. The compiler assigns the register numbers automatically. Global registers may have to be renumbered when multiple object files are combined, so the programmer cannot rely on explicit numbering.

A comment is indicated by an exclamation mark and ends at the end of the line. The exclamation mark was chosen because it was unused in the syntax of PL/MMIX and because it looks like a pen for writing something important.

3. Like MMIXAL the PL/MMIX language also deals primarily with *symbols* and *constants*. PL/MMIX supports decimal, hexadecimal and floating point constants:

⟨ digit ⟩ ⟶ `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
⟨ hex digit ⟩ ⟶ ⟨ digit ⟩ | `a` | `b` | `c` | `d` | `e` | `f` | `A` | `B` | `C` | `D` | `E` | `F`
⟨ decimal ⟩ ⟶ ⟨ digit ⟩ | ⟨ decimal ⟩⟨ digit ⟩
⟨ hex ⟩ ⟶ `#`⟨ hex digit ⟩ | ⟨ hex ⟩⟨ hex digit ⟩
⟨ fraction ⟩ ⟶ `.`⟨ decimal ⟩
⟨ exp indicator ⟩ ⟶ `e` | `E`
⟨ sign ⟩ ⟶ `+` | `-`
⟨ exponent ⟩ ⟶ ⟨ exp indicator ⟩⟨ decimal ⟩ | ⟨ exp indicator ⟩⟨ sign ⟩⟨ decimal ⟩
⟨ float ⟩ ⟶ ⟨ decimal ⟩⟨ fraction ⟩ | ⟨ decimal ⟩⟨ exponent ⟩ |
⟨ decimal ⟩⟨ fraction ⟩⟨ exponent ⟩
⟨ constant ⟩ ⟶ ⟨ decimal ⟩ | ⟨ hex ⟩ | ⟨ float ⟩

As in MMIXAL decimal and hex constants whose value is 2^{64} or more are reduced modulo 2^{64}.

4. A *character constant* is a single character enclosed in single quotes, like in MMIXAL. PL/MMIX supports UTF-8 so the value of a character constant does not necessarily fit in one byte. Its value is its unicode representation.

5. A *symbol* is any sequence of UTF-8 letters and digits, starting with a letter. All characters beyond DEL are considered letters even if they represent a digit in another script. An underscore is also considered a letter.

⟨ letter ⟩ ⟶ `A` | `B` | ⋯ | `Z` | `_` | ⟨ character with code higher than 127 ⟩
⟨ symbol ⟩ ⟶ ⟨ letter ⟩ | ⟨ symbol ⟩⟨ letter ⟩ | ⟨ symbol ⟩⟨ digit ⟩

PL/MMIX does not use the prefix capability of MMIXAL. All names are considered fully qualified.

6. A symbol must be declared and be assigned a value before it can be used in statements and assembler instructions. Once the value is set it can not be changed anymore.

Constant and symbols can be used in *expressions* like in `MMIXAL` with a few differences.

⟨ atom ⟩ ⟶ ⟨ constant ⟩ | ⟨ symbol ⟩ | `(`⟨ expression ⟩`)`
⟨ unit ⟩ ⟶ ⟨ atom ⟩ | ⟨ unary operator ⟩⟨ unit ⟩
⟨ strong ⟩ ⟶ ⟨ unit ⟩ | ⟨ unit ⟩⟨ strong operator ⟩⟨ strong ⟩
⟨ weak ⟩ ⟶ ⟨ strong ⟩ | ⟨ strong ⟩⟨ weak operator ⟩⟨ weak ⟩
⟨ comparison ⟩ ⟶ ⟨ weak ⟩ | ⟨ weak ⟩⟨ comparison operator ⟩⟨ weak ⟩
⟨ ternary ⟩ ⟶ `?`⟨ weak ⟩`:`⟨ expression ⟩
⟨ expression ⟩ ⟶ ⟨ comparison ⟩⟨ ternary ⟩ | ⟨ comparison ⟩
⟨ unary operator ⟩ ⟶ `+` | `-` | `~` | `\/`
⟨ strong operator ⟩ ⟶ `*` | `/` | `%` | `<<` | `>>` | `&` | `&~`
⟨ weak operator ⟩ ⟶ `+` | `-` | `|` | `|~` | `^` | `$` | `--`
⟨ comparison operator ⟩ ⟶ `<` | `>` | `==` | `<=` | `>=` | `<>` | `%%`

The basic building blocks of an expression are numbers and symbols, they are called atoms. An expression between parentheses is treated as an atom. An atom may be preceded by one or more unary operators. The unary operators are `+`, `-`, `~` and `\/`. They mean respectively "do nothing", "change the sign", "invert the bits" and "take the square root". The result of taking the square root is always a floating point number, even if the result is an integer.

Binary operators come in three flavors, strong, weak and comparisons. The strong operators are `*`, `/`, `%`, `<<`, `>>`, `&` and `&~`. They stand for $(x \times y) \bmod 2^{64}$ (multiplication), $\lfloor x/y \rfloor$ (division), $x \bmod y$, (remainder), $(x \times 2^y) \bmod 2^{64}$ (left shift), $\lfloor x/2^y \rfloor$ (right shift), $x \,\&\, y$ (bitwise and) and $x \,\&\, \neg y$ (bitwise and-not).

Weak operators are `+`, `-`, `|`, `|~`, `^`, `$` and `--`. They stand for $(x + y) \bmod 2^{64}$ (addition), $(x - y) \bmod 2^{64}$ (subtraction), $x \mid y$ (bitwise or), $x \mid \neg y$ (bitwise or-not), $x \oplus y$ (bitwise exclusive-or), $\nu(x \setminus y)$ (bit count) and $S(x - y)$ (bytewise saturated difference). The last operator is implemented as `BDIF` instruction in the `MMIX` instruction set.

The weakest of all operators are the comparison operators, `<`, `>`, `==`, `<=`, `>=`, `<>` and `%%`. When used in calculations they result in a 0 when false and 1 when true. The `%%` followed by the number 0 is a test whether the left hand is even, if it is followed by a 1 it is a test whether the left hand is odd.

The `?` operator is somewhat special. It will compare two values and return -1, 0 or 1, when respectively the first value is smaller than, equal to or larger than the second value. It may also have a test on the left side and two expression on the right side separated by a colon. It will choose the first value if the test is true and the second value otherwise.

The result of an expression is a floating point number if either side of the operator is a floating point number. The result of an expression is an integer if both sides of the operator are integers.

7. Declarations define the variables used in the module. It is possible to define global and local variables for global and local registers respectively. Data and constants define data to be loaded in the data segment and the text segment after the program code. Definitions are used to declare values that are resolved during compilation that can be used in initialisations and immediate values in instructions.

Variables used in the program that are defined in another module are marked `extern`. Local variables and definitions cannot be marked extern. External variables cannot be initialised. Definitions declared with `define` must be initialised.

⟨ declaration ⟩ ⟶ ⟨ global ⟩ | ⟨ local ⟩ | ⟨ data ⟩ | ⟨ constant ⟩ | ⟨ define ⟩
⟨ global ⟩ ⟶ `global` ⟨ type specification ⟩ ⟨ initialisation list ⟩ `;`
⟨ local ⟩ ⟶ `local` ⟨ type specification ⟩ ⟨ symbol list ⟩ `;`
⟨ data ⟩ ⟶ `data` ⟨ type specification ⟩ ⟨ initialisation list ⟩ `;`
⟨ constant ⟩ ⟶ `constant` ⟨ type specification ⟩ ⟨ initialisation list ⟩ `;`
⟨ define ⟩ ⟶ `define` ⟨ type specification ⟩ ⟨ initialisation list ⟩ `;`
⟨ symbol list ⟩ ⟶ ⟨ symbol ⟩ | ⟨ symbol list ⟩ `,` ⟨ symbol ⟩
⟨ initialisation list ⟩ ⟶ ⟨ initialisation ⟩ | ⟨ initialisation list ⟩ `,` ⟨ initialisation ⟩
⟨ initialisation ⟩ ⟶ ⟨ symbol ⟩ | ⟨ symbol ⟩ `=` ⟨ expression ⟩ |
⟨ symbol ⟩ `=` `{` ⟨ expression list ⟩ `}`
⟨ expression list ⟩ ⟶ ⟨ expression ⟩ | ⟨ expression list ⟩ `,` ⟨ expression ⟩
⟨ type specification ⟩ ⟶ ⟨ type ⟩ ⟨ sign ⟩ | ⟨ type ⟩ | ⟨ sign ⟩
⟨ type ⟩ ⟶ `byte` | `wyde` | `tetra` | `octa` | `float` | `sfloat`
⟨ sign ⟩ ⟶ `signed` | `unsigned`

The value of a local or global is a register number which can not be used in calculations. The value of data and constants is the memory address where the initialised value is loaded.

Global, data, constant and define declarations are available on a global level, that is they can be used anywhere in the program. Local variables can only be declared inside procedures or in local blocks within a procedure. When the local block is left, the local variables declared in it, cease to exist. It is possible to use a local variable name in an inner block that already is used in an outer block. It then hides the variable in the outer block.

Here are some examples of declarations:

```
global octa eightbytes=8;
local unsigned byte a,b;
data float φ=1.618033988;
data octa fibonacci={1,1,2,3,5,8,13};
constant AnswerToTheUltimateQuestionOfLifeTheUniverseAndEverything=42;
define PAGESIZE=8192;
```

8. Statements are always enclosed in procedures. The entry point of the program is the procedure called "main". The compiler does not perform type checking, so it is the responsibility of the programmer to pass the right type of parameters. Because of the "PUSHJ" and "POP" semantics, procedures may pass and return multiple parameters on a procedure call. Input parameters are listed before the bar symbol, output parameters are listed after the bar symbol.

⟨ procedure ⟩ ⟶ ⟨ pheader ⟩⟨ block ⟩
⟨ pheader ⟩ ⟶ `procedure`⟨ symbol ⟩ | `procedure`⟨ symbol ⟩⟨ parameter list ⟩
⟨ parameter list ⟩ ⟶ `(`⟨ parameters ⟩`)` | `()`
⟨ parameters ⟩ ⟶ ⟨ symbol list ⟩ | ⟨ symbol list ⟩`|` | ⟨ symbol list ⟩`|`⟨ symbol list ⟩ | `|`⟨ symbol list ⟩

An example of an ordinary statement is `a=b+c;`. It translates into a single `MMIX` instruction `ADD a,b,c`, `ADDU a,b,c` or `FADD a,b,c` based on the declaration of a. a and b must be variables but c may be a variable or an expression, e.g. `MAXN-1`. When the target variable is the same as b, it is possible to use the variant `b+=c;`. The compiler then generates an instruction like `ADD b,b,c`. This can be done for all operations below.

Statement	**Signed Integer**	**Unsigned Integer**	**Float**
$a = b + c;$	ADD	ADDU	FADD
$a = b - c;$	SUB	SUBU	FSUB
$a = b * c;$	MUL	MULU	FMUL
$a = b/c;$	DIV	DIVU	FDIV
$a = b \,\%\, c;$	–	–	FREM
$a = \backslash/b;$	–	–	SQRT
$a = b \,?\, c;$	CMP	CMPU	FCMP
$a = b \;>>\; c;$	SHR	SHRU	–
$a = b \;<<\; c;$	SHL	SHLU	–
$a = b \,\&\, c;$	AND	AND	AND
$a = b \,\&\tilde{}\, c;$	ANDN	ANDN	ANDN
$a = b \,\vert\, c;$	OR	OR	OR
$a = b \,\vert\tilde{}\, c;$	ORN	ORN	ORN
$a = b \,\hat{}\, c;$	XOR	XOR	XOR
$a = b \,\$\, c;$	SADD	SADD	–

Straight assignments like `a=b;` are a bit different. They do not only take into account the type of the target variable but also the type of the source variable. If the types are both integer or both floating point an `ORI` instruction is generated, otherwise a `FIX` or `FLOT` instruction is generated.

The load and store operations are again a bit different because they do not only take into account the type but also the size of the target variable. The saturated difference operation is also size dependent.

Statement	**Byte**	**Wyde**	**Tetra**	**Octa**	**Float**	**SFloat**
$a = b \;--\; c;$	BDIF	WDIF	TDIF	ODIF	–	–
$a \;<-\; b + c;$	LDB(U)	LDW(U)	LDT(U)	LDO(U)	LDO	LDSF
$a \;->\; b + c;$	STB(U)	STW(U)	STT(U)	STO(U)	STO	STSF
$a \;<->\; b + c;$	CSWAP	CSWAP	CSWAP	CSWAP	–	–

There are also some more complex statements that nevertheless can be coded in one `MMIX` instruction. The first group are the statements that multiply the source and perform an add. These are encoded in the 2ADDU, 4ADDU, 8ADDU and 16ADDU instructions. The multiplier must be 2, 4, 8 or 16, otherwise an error is issued. The second group generates SETL, SETML, SETMH and SETH instructions and the related

INC, ANDN and OR instructions. The source value must be an immediate value possibly shifted by 16, 32, or 48 bits to the ML, MH and H version respectively.

Statement	Instruction
$a = 2 * b + c$	2ADDU
$a = 4 * b + c$	4ADDU
$a = 8 * b + c$	8ADDU
$a = 16 * b + c$	16ADDU
$a = 1 << 48;$	SETH
$a = 1 << 32;$	SETMH
$a = 1 << 16;$	SETML
$a = 256;$	SETL
$a+ = 1 << n;$	INCx
$a = a\|1 << n;$	ORx
$a\&\tilde{} = 1 << n;$	ANDNx

The conditional operations also generate a single instruction. The value to test against must be zero except for the test for being odd which allows a value of 1.

Statement	Instruction	Statement	Instruction
$a = b > 0?c;$	CSP	$a = b <= 0?c;$	CSNP
$a = b > 0?c : 0;$	ZSP	$a = b <= 0?c : 0;$	ZSNP
$a = b < 0?c;$	CSN	$a = b >= 0?c;$	CSNN
$a = b < 0?c : 0;$	ZSN	$a = b >= 0?c : 0;$	ZSNN
$a = b == 0?c;$	CSZ	$a = b <> 0?c;$	CSNZ
$a = b == 0?c : 0;$	ZSZ	$a = b <> 0?c : 0;$	ZSNZ
$a = b \,\%\% \, 0?c;$	CSEV	$a = b \,\%\% \, 1?c;$	CSOD
$a = b \,\%\% \, 0?c : 0;$	ZSEV	$a = b \,\%\% \, 1?c : 0;$	ZSOD

There are two more `PL/MMIX` instructions. Programmers will not use them very often but the author wanted to add them nonetheless.

Statement	Instruction
$a <<<$	SAVE
$a >>>$	UNSAVE

Assembler statements can be inserted intermixed with `PL/MMIX` statements, but this is not recommended. Some `MMIX` instructions have no corresponding `PL/MMIX` statement, so sometimes it is necessary:

```
SYNC 7;
SWYM 0;
```

The author cannot resist to smile seeing that SYNC and SWYM are located next to each other in the `MMIX` opcode table.

9. The flow control statements are "borrowed" from the C language. They include:

⟨ if ⟩ ⟶ `if` ⟨ comparison ⟩⟨ block ⟩ | `if` ⟨ comparison ⟩⟨ block ⟩`else` ⟨ block ⟩
⟨ while ⟩ ⟶ `while` ⟨ comparison ⟩⟨ block ⟩
⟨ until ⟩ ⟶ `do` ⟨ block ⟩`until` ⟨ comparison ⟩`;`
⟨ goto ⟩ ⟶ `goto` ⟨ symbol ⟩`;`
⟨ call ⟩ ⟶ `call` ⟨ symbol ⟩⟨ parameterlist ⟩`;` | `call` ⟨ symbol ⟩`;`
⟨ return ⟩ ⟶ `return;` | `return` ⟨ decimal ⟩`;`

The switch statement is missing since this statement cannot be translated into one MMIX instruction. The comparison must be a simple comparison of a variable against 0, e.g `a<0`. Testing against 1 is only possible in combination with `%%`. These flow control statements produce branch statements to skip forward or backward when required.

The call statement produces a `PUSHJ` instruction. The input variables are listed before the |-sign and the output variables are listed after the |-sign. Due to the nature of the `PUSH-POP` mechanism multiple input and output variables may be specified.

10. In some cases we must refer to global register 255. `PL/MMIX` offers the variable `r255` that always refers to this global register. It cannot be renumbered during linking. `PL/MMIX` also predefines the variables `ROUND` and `EPSILON` whose values are set to 4 and 0 respectively.

11. What is left is the "hello world" example:

```
extern data octa stdin;
constant unsigned byte message="Hello world!";

procedure octa main(octa argc,octa argv|octa rc) {
 call write(message,stdin);
 rc=0;
 return;
}
```

We assume here that a library is linked that supplies the `write` subroutine.

12. Main program.

```
#define panic(m)  fprintf(stderr, "plmmix: %s\n", m), exit(-1)
#define move(p, q)  p = q, q = Λ

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "mmix-arith64.h"
#include "mmix-elf64.h"

  extern octa val;

  ⟨ Type definitions 24 ⟩
  ⟨ Global variables 15 ⟩

  ⟨ Basic subroutines 13 ⟩
  ⟨ Symbol subroutines 20 ⟩
  ⟨ Lexical subroutines 30 ⟩
  ⟨ Error subroutines 40 ⟩
  ⟨ Syntactic subroutines 49 ⟩
  ⟨ Expression evaluation 92 ⟩
  ⟨ Semantic subroutines 104 ⟩
  ⟨ Code generation 162 ⟩

  int main(int argc, char **argv)
  {
    ⟨ Handle program arguments 14 ⟩
    ⟨ Initialise everything 16 ⟩
    ggroot = ggparse( );
    resolve(ggroot);
    ⟨ Perform all fixups 159 ⟩
    if (nxterr ≡ 0) {
      ⟨ Allocate section data 102 ⟩
      generate(ggroot);
      ⟨ Wrap up everything 103 ⟩
    }
    prterr( );
    return 0;
  }
```

13. The allocate function ensures that the returned pointer is not Λ.

```
⟨ Basic subroutines 13 ⟩ ≡
  void *allocate(int n, size_t s)
  {
    register void *p;
    if ((p = (void *) calloc(n, s)) ≡ Λ) panic("not enough memory");
    return p;
  }
```

See also section 18.

This code is used in section 12.

14. Determine the options passed at invocation of the compiler. The option "utf8" forces the input to be read as utf-8 encoded characters. The option "output=filename" changes the output filename.

```
⟨ Handle program arguments 14 ⟩ ≡
  {
    char **p = argv + 1;
    for (int n = argc − 1; n; n−−, p++) {
      if (**p ≡ '-') {
        char *q = strchr(*p, '=');
        if (q ≠ Λ) *q++ = '\0';
        if (¬q ∧ strncmp(*p, "-debug", strlen(*p)) ≡ 0) debug = true;
        else if (q ∧ strncmp(*p, "-output", strlen(*p)) ≡ 0) tgtname = strdup(q);
        else if (¬q ∧ strncmp(*p, "-utf8", strlen(*p)) ≡ 0) utf8 = true;
        else fprintf(stderr, "PL/MMIX␣invalid␣option␣%s\n", *p);
      }
      else if (n > 0) {
        if (srcname ≠ Λ) panic("more␣than␣one␣source␣file");
        srcname = *p;
      }
    }
    if (srcname ≡ Λ) panic("no␣source␣file␣specified");
  }
```

This code is used in section 12.

15.

```
⟨ Global variables 15 ⟩ ≡
  char *srcname;
  char *tgtname;
  bool utf8;
  bool debug = false;
```

See also sections 19, 25, 27, 29, 32, 38, 43, 45, 48, 51, 101, 109, 121, 160, 176, and 177.

This code is used in section 12.

16. The first step is to read the complete source into memory. The newline characters are replaced by `'\0'` characters later to create an array of source lines. This array can then be used to highlight where an error was discovered.

```
⟨ Initialise everything 16 ⟩ ≡
  if (¬tgtname) tgtname = setfext(srcname, ".obj");
  if ((srcfile = fopen(srcname, "r")) ≡ Λ) panic("can't␣open␣source␣file");
  fseek(srcfile, 0, SEEK_END);
  srcsize = ftell(srcfile);
  fseek(srcfile, 0, SEEK_SET);
  source = allocate(srcsize + 4, sizeof(char));
  fread(source, 1, srcsize, srcfile);
  ⟨ Check for utf-8 preamble 17 ⟩
  for (int i = 0; i < srcsize; i++)
    if (source[i] ≡ '\n') nrlines++;
  srcline = allocate(nrlines + 1, sizeof(char **));
  lbufptr = srcline[0] = source;
```

See also sections 23 and 98.

This code is used in section 12.

17. If a file contains utf-8 characters it should start with the sequence `#ef`, `#bb`, `#bf`. These characters are skipped and the source size is adapted accordingly, otherwise we might read beyond the end of the data. If the file does contain utf-8 characters but does not contain the start sequence, the utf flag can be set by passing the `utf8` argument on the command line.

```
⟨ Check for utf-8 preamble 17 ⟩ ≡
  if (((*source & #ff) ≡ #ef) ∧ ((source[1] & #ff) ≡ #bb) ∧ ((source[2] & #ff) ≡ #bf)) {
    source += 3;
    srcsize −= 3;
    utf8 = true;
  }
```

This code is used in section 16.

18. Here is a small routine to set the extension of a file.

```
⟨ Basic subroutines 13 ⟩ +≡
  char *setfext(char *fname, char *ext)
  {
    char *pdot, *pslash;
    char *result = allocate(strlen(fname) + 5, sizeof(char));
    strcpy(result, fname);
    pdot = strrchr(result, '.');
    pslash = strrchr(result, '/');
    if (pdot > pslash) *pdot = '\0';
    strcat(result, ext);
    return result;
  }
```

19. ⟨ Global variables 15 ⟩ +≡

```
  char *lbufptr;
  char *source;
  char **srcline;
  int nrlines = 0;
  int srcsize;
  FILE *srcfile;
  FILE *tgtfile;
```

20. Symbols. Symbols are as important to compilers as they are to humans. `PL/MMIX` stores symbols in hash tables. The slot number is calculated by the formula:

$$h(K) = (h_1(k_1) * 2^{2n-2} \oplus h_2(k_2) * 2^{2n-4} \oplus ... \oplus h_n(k_n)) \bmod N = (\bigoplus_{i=1}^{n} h_i(k_i) * 2^{2(n-i)}) \bmod N$$

where each k_n are the 8 bits of the n^{th} character and N is the size of the hash table. The $h_n(k_n)$ is found by looking it up in *hktable*. This table contains the 256 precalculated values determined by another hash function:

$$h(k) = \lfloor (M((\frac{A}{\omega}k) \bmod 1) \rfloor$$

where $M = \omega = 2^{32}$ and A=2654435761. This is a slight variation on the algorithm described in D. E. Knuth, *The Art of Computer Programming, Searching and Sorting*, 514-520. This variation ensures that repeating characters produce different hash values.

⟨ Symbol subroutines 20 ⟩ ≡

```
unsigned int hash(char *s, int n)
{
  unsigned int h = 0;
  for ( ; *s; s++) {
    h = (h ≪ 2) ⊕ hktable[(*s)];
  }
  return h % n;
}
```

See also sections 21 and 22.

This code is used in section 12.

21. Symbols are created by calling *mksym*. If a symbol with the same name already exists, a pointer to the existing symbol is returned and its *usage* is not updated, otherwise a new symbol is created.

There is a global symbol table for storing keywords, opcodes, special registers and all the global variables of the program. For each block a local symbol table is created for storing local variables defined in that block. The local symbol table points to the local symbol table of the parent block. Ultimately the local symbol tables point to the global table.

```
⟨ Symbol subroutines 20 ⟩ +≡
  symtype *mksym(char *name, int usage, symtabtype *symtbl, bool isextern)
  {
    symtype *p, *q;
    p = (symtype *) allocate(1, sizeof(symtype));
    p→name = (char *) allocate((strlen(name) + 1), sizeof(char));
    strcpy(p→name, name);
    p→usage = usage;
    p→extsym = isextern;
    q = symtbl→sym[hash(name, symtbl→size)];
    if (q ≡ Λ) symtbl→sym[hash(name, symtbl→size)] = p;
    else {
      for ( ; ; ) {
        if (strcmp(q→name, name) ≡ 0) {
          free(p→name); free(p);
          return q;
        }
        if (q→next ≡ Λ) break;
        q = q→next;
      }
      q→next = p;
    }
    return p;
  }
```

22. *Getsym* searches the local symbol table for the variable. If it is not found in the local symbol table it searches its parent. This will continue until the global table is reached which has no parent. If the variable is still not found it does not exist.

```
⟨ Symbol subroutines 20 ⟩ +≡
  symtype *getsym(char *name, bool onlylocal)
  {
    register symtype *p = Λ;
    for (symtabtype *currsym = lclsym; ¬p ∧ currsym; currsym = currsym→parent) {
      p = currsym→sym[hash(name, currsym→size)];
      while ((p ≠ Λ) ∧ (strcmp(p→name, name) ≠ 0)) p = p→next;
      if (onlylocal) break;
    }
    return p;
  }
```

23. The default size of the global hash table is $2 * \texttt{MMIX} + 1$, the default size of the local hash table is 101.

```
#define MAXGLBSYM 4019
#define MAXLCLSYM 101
⟨ Initialise everything 16 ⟩ +≡
  glbsymtabsize = MAXGLBSYM;
  glbsym = lclsym = (symtabtype *) allocate(1, sizeof(symtabtype));
  glbsym→size = glbsymtabsize;
  glbsym→sym = (symtype **) allocate(glbsymtabsize, sizeof(symtype *));
  for (int i = 0; i ≤ (SFLOAT − GLOBAL); i++) {
    mksym(inst[i], KEYWORD, glbsym, false)→var = (vartype) {GLOBAL + i, UOCTA};
  }
  for (int i = 0; i ≤ 31; i++) {
    mksym(sreg[i], SPECIAL, glbsym, false)→var = (vartype) {i, UBYTE};
  }
  for (int i = 0; i ≤ 147; i++) {
    mksym(mnem[i], OPCODE, glbsym, false)→var = (vartype) {opc2[i] & #ffffffff, UOCTA};
  }
  mksym("ROUND", IDENTIFIER, glbsym, false)→var = (vartype) {4, UBYTE};
  mksym("EPSILON", IDENTIFIER, glbsym, false)→var = (vartype) {0, UBYTE};
  mksym("r255", REGISTER, glbsym, false)→var = (vartype) {255, UBYTE};
```

24. ⟨ Type definitions 24 ⟩ ≡

```
  typedef struct varstruct {
    octa value;
    short int type;
  } vartype;
  typedef enum {
    SBYTE = 1, UBYTE, SWYDE, UWYDE, STETRA, UTETRA, SOCTA, UOCTA, TFLOAT, TSFLOAT
  } mmixtypes;
  typedef struct symstruct {
    struct symstruct *next;
    char *name;
    vartype var;
    int usage;
    int symidx;
    bool extsym;
  } symtype;
  typedef struct tabstruct {
    struct tabstruct *parent;
    int size;
    symtype **sym;
  } symtabtype;
```

See also sections 26, 28, 42, 44, 47, 100, 108, and 161.

This code is used in section 12.

25. ⟨ Global variables 15 ⟩ +≡

```
int glbsymtabsize = MAXGLBSYM;
int lclsymtabsize = MAXLCLSYM;
symtabtype *glbsym, *lclsym;
int size[] = {0, 1, 1, 2, 2, 4, 4, 8, 8, 8, 4};
char *sreg[] = {
"rB", "rD", "rE", "rH", "rJ", "rM", "rR", "rBB",
"rC", "rN", "rO", "rS", "rI", "rT", "rTT", "rK",
"rQ", "rU", "rV", "rG", "rL", "rA", "rF", "rP",
"rW", "rX", "rY", "rZ", "rWW", "rXX", "rYY", "rZZ"};
char *mnem[] = {
"TRAP", "FCMP", "FUN", "FEQL", "FADD", "FIX", "FSUB", "FIXU",
"FLOT", "FLOTU", "SFLOT", "SFLOTU",
"FMUL", "FCMPE", "FUNE", "FEQLE", "FDIV", "FSQRT", "FREM", "FINT",
"MUL", "MULU", "DIV", "DIVU",
"ADD", "ADDU", "SUB", "SUBU", "2ADDU", "4ADDU", "8ADDU", "16ADDU",
"CMP", "CMPU", "NEG", "NEGU", "SL", "SLU", "SR", "SRU",
"BN", "BZ", "BP", "BOD", "BNN", "BNZ", "BNP", "BEV",
"PBN", "PBZ", "PBP", "PBOD", "PBNN", "PBNZ", "PBNP", "PBEV",
"CSN", "CSZ", "CSP", "CSOD", "CSNN", "CSNZ", "CSNP", "CSEV",
"ZSN", "ZSZ", "ZSP", "ZSOD", "ZSNN", "ZSNZ", "ZSNP", "ZSEV",
"LDB", "LDBU", "LDW", "LDWU", "LDT", "LDTU", "LDO", "LDOU",
"LDSF", "LDHT", "CSWAP", "LDUNC", "LDVTS", "PRELD", "PREGO", "GO",
"STB", "STBU", "STW", "STWU", "STT", "STTU", "STO", "STOU",
"STSF", "STHT", "STCO", "STUNC", "SYNCD", "PREST", "SYNCID", "PUSHGO",
"OR", "ORN", "NOR", "XOR", "AND", "ANDN", "NAND", "NXOR",
"BDIF", "WDIF", "TDIF", "ODIF", "MUX", "SADD", "MOR", "MXOR",
"SETH", "SETMH", "SETML", "SETL", "INCH", "INCMH", "INCML", "INCL",
"ORH", "ORMH", "ORML", "ORL", "ANDNH", "ANDNMH", "ANDNML", "ANDNL",
"JMP", "PUSHJ", "GETA", "PUT", "POP", "RESUME", "SAVE", "UNSAVE", "SYNC", "SWYM", "GET", "TRIP"
};
unsigned int hktable[] = {
0, 2654435761, 1013904226, 3668339987, 2027808452, 387276917, 3041712678,
1401181143, 4055616904, 2415085369, 774553834, 3428989595, 1788458060, 147926525,
2802362286, 1161830751, 3816266512, 2175734977, 535203442, 3189639203, 1549107668,
4203543429, 2563011894, 922480359, 3576916120, 1936384585, 295853050, 2950288811,
1309757276, 3964193037, 2323661502, 683129967, 3337565728, 1697034193, 56502658,
2710938419, 1070406884, 3724842645, 2084311110, 443779575, 3098215336, 1457683801,
4112119562, 2471588027, 831056492, 3485492253, 1844960718, 204429183, 2858864944,
1218333409, 3872769170, 2232237635, 591706100, 3246141861, 1605610326, 4260046087,
2619514552, 978983017, 3633418778, 1992887243, 352355708, 3006791469, 1366259934,
4020695695, 2380164160, 739632625, 3394068386, 1753536851, 113005316, 2767441077,
1126909542, 3781345303, 2140813768, 500282233, 3154717994, 1514186459, 4168622220,
2528090685, 887559150, 3541994911, 1901463376, 260931841, 2915367602, 1274836067,
3929271828, 2288740293, 648208758, 3302644519, 1662112984, 21581449, 2676017210,
1035485675, 3689921436, 2049389901, 408858366, 3063294127, 1422762592, 4077198353,
2436666818, 796135283, 3450571044, 1810039509, 169507974, 2823943735, 1183412200,
3837847961, 2197316426, 556784891, 3211220652, 1570689117, 4225124878, 2584593343,
944061808, 3598497569, 1957966034, 317434499, 2971870260, 1331338725, 3985774486,
2345242951, 704711416, 3359147177, 1718615642, 78084107, 2732519868, 1091988333,
3746424094, 2105892559, 465361024, 3119796785, 1479265250, 4133701011, 2493169476,
```

852637941, 3507073702, 1866542167, 226010632, 2880446393, 1239914858, 3894350619, 2253819084, 613287549, 3267723310, 1627191775, 4281627536, 2641096001, 1000564466, 3655000227, 2014468692, 373937157, 3028372918, 1387841383, 4042277144, 2401745609, 761214074, 3415649835, 1775118300, 134586765, 2789022526, 1148490991, 3802926752, 2162395217, 521863682, 3176299443, 1535767908, 4190203669, 2549672134, 909140599, 3563576360, 1923044825, 282513290, 2936949051, 1296417516, 3950853277, 2310321742, 669790207, 3324225968, 1683694433, 43162898, 2697598659, 1057067124, 3711502885, 2070971350, 430439815, 3084875576, 1444344041, 4098779802, 2458248267, 817716732, 3472152493, 1831620958, 191089423, 2845525184, 1204993649, 3859429410, 2218897875, 578366340, 3232802101, 1592270566, 4246706327, 2606174792, 965643257, 3620079018, 1979547483, 339015948, 2993451709, 1352920174, 4007355935, 2366824400, 726292865, 3380728626, 1740197091, 99665556, 2754101317, 1113569782, 3768005543, 2127474008, 486942473, 3141378234, 1500846699, 4155282460, 2514750925, 874219390, 3528655151, 1888123616, 247592081, 2902027842, 1261496307, 3915932068, 2275400533, 634868998, 3289304759, 1648773224, 8241689, 2662677450, 1022145915, 3676581676, 2036050141, 395518606, 3049954367, 1409422832, 4063858593, 2423327058, 782795523, 3437231284, 1796699749, 156168214, 2810603975, 1170072440, 3824508201, 2183976666, 543445131, 3197880892, 1557349357, 4211785118, 2571253583};

26. ⟨ Type definitions 24 ⟩ +≡

```
typedef enum {
  EOS = 0, GLOBAL, LOCAL, DATA, CONSTANT, DEFINE, EXTERN, PROCEDURE, IF, ELSE, DO, WHILE, UNTIL, GOTO,
      CALL, RETURN, SIGNED, UNSIGNED, BYTE, WYDE, TETRA, OCTA, FLOAT, SFLOAT, PLUS, MINUS, DIFF, SADD,
      OR, ORN, XOR, TIMES, DIV, REM, SHL, SHR, AND, ANDN, CMP, SQRT, NOT, COLON, ASSIGN, LOAD, STORE,
      SWAP, SAVE, UNSAVE, LT, EQ, GT, ODD, GE, NE, LE, EVEN, LPRN, RPRN, LBRK, RBRK, LBRC, RBRC, COMMA,
      SEMICOLON, PLUSEQ, MINUSEQ, TIMESEQ, DIVEQ, REMEQ, DIFFEQ, SADDEQ, SHLEQ, SHREQ, CMPEQ, ANDEQ,
      ANDNEQ, OREQ, ORNEQ, XOREQ, IDENTIFIER, REGISTER, SPECIAL, NUMBER, FLTNUM, CHARACTER, STRING,
      KEYWORD, LABEL, OPCODE, EXPRESSION, ADDRESS, PUT, GET, PARAMETER, IN, OUT, ACTUAL, OFFSET,
      BLOCK, RELOCATION, ROOT, COMMENT, SPACE, INVALID, ERROR = 999
} lextype;
```

27. ⟨ Global variables 15 ⟩ +≡

```
char *inst[] = {
"global","local","data","constant","define","extern","procedure","if","else","do",
    "while","until","goto","call","return","signed","unsigned","byte","wyde","tetra",
    "octa","float","sfloat","+","-","--","$","|","|~","^","*","/","%","<<",">>","&","&~",
    "?","\\/","^",":","=","<-","->","<->","<<<",">>>","<","==",">","%%",">=","<>","<=",
    "%%","(",")","[","]","{","}",",",";","+=","-+","*=","/=","%=","--=","$=","<<=",">>=",
    "?","&=","&~=","|=","|~=","^=","identifier","register","special","number","fltnum",
    "character","string","keyword","label","opcode","expression","address","put","get",
    "parameter","in","out","actual","offset","block","relocation","root","comment",
    "space","invalid"};
```

28. Lexis. PL/MMIX uses a hand written lexer. The reason is that no tools like Lex are needed to build this compiler.

The lexer is a finite automaton. The automaton starts in state 1, the start state. The next state is determined by examining the next character. This continues until state 0 is reached, the dead state. During the scanning process the automaton checks whether the token read so far would be accepted.

See also Aho, Lam, Sethi and Ullman *Compilers, Principles, Techniques, &Tools; ISBN 0-321-48681-1*, Chapter 3.

```
⟨ Type definitions 24 ⟩ +≡
  typedef struct {
    char *name;
    int type;
    int linenr;
    int column;
  } tokentype;
```

29. The lexer variables have a prefix of "ll". "llptr" is the pointer to the next character to check, "lline" and "llcolumn" track the position in the text.

```
⟨ Global variables 15 ⟩ +≡
  int llline = 1;
  int llcolumn = 1;
  char *llptr;
  char *utfptr;
```

30. The lexer uses the *peek* and *fetch* routines to determine the type of the next character and to fetch it when it is needed by the lexer to complete the token.

```
⟨ Lexical subroutines 30 ⟩ ≡
  int peek( )
  {
    int c = ctype[*llptr & #ff];
    if (c < UTFC) return c;
    else if (c ≡ UTFC) panic("Invalid␣utf␣source");
    else
      for (int i = c − UTFC; i > 0; i−−)
        if (ctype[llptr[i] & #ff] ≠ UTFC) panic("Invalid␣utf␣source");
    return 3;
  }
  void fetch( )
  {
    int c = ctype[*llptr & #ff];
    llptr += (c < UTFC) ? 1 : c − UTFC + 1;
    llcolumn++;
    return;
  }
```

See also sections 31, 33, and 34.

This code is used in section 12.

31. We might as well directly define a routine to translate a utf-8 sequence into its character value.

```
⟨ Lexical subroutines 30 ⟩ +≡
  int utfval(char *c)
  {
    int v = 0;
    if ((*c & #ff) < #7f) {
      utfptr = c + 1;
      return (*c & #ff);
    }
    if ((*c & #e0) ≡ #c0) v = *c & #1f;
    else if ((*c & #f0) ≡ #e0) v = *c & #0f;
    else v = *c & #07;
    c++;
    while ((*c & #c0) ≡ #80) v = v * #40 + (*c & #3f), c++;
    utfptr = c;
    return v;
  }
```

32.

```
⟨ Global variables 15 ⟩ +≡
  const int UTFC = 48;
  const int ERRC = UTFC − 1;
  int ctype[ ] = {
  0, ERRC, ERRC, ERRC, ERRC, ERRC, ERRC, ERRC, ERRC, 1, 2, 1, 1, 1, ERRC, ERRC,
  ERRC, ERRC, ERRC, ERRC, ERRC, ERRC, ERRC, ERRC, ERRC, ERRC, ERRC, ERRC, ERRC, ERRC, ERRC, ERRC,
  1, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 19, 34, 20, 18, 35,
  14, 15, 16, 14, 16, 14, 17, 14, 16, 14, 36, 37, 21, 22, 23, 38,
  ERRC, 9, 12, 12, 10, 13, 12, 3, 3, 4, 3, 3, 3, 3, 7, 3,
  3, 3, 3, 3, 3, 11, 3, 3, 3, 3, 3, 39, 40, 41, 42, 3,
  ERRC, 8, 12, 12, 12, 13, 6, 3, 3, 3, 3, 3, 3, 3, 5, 3,
  3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 43, 44, 45, 46, ERRC,
  48, 48, 48, 48, 48, 48, 48, 48, 48, 48, 48, 48, 48, 48, 48, 48,
  48, 48, 48, 48, 48, 48, 48, 48, 48, 48, 48, 48, 48, 48, 48, 48,
  48, 48, 48, 48, 48, 48, 48, 48, 48, 48, 48, 48, 48, 48, 48, 48,
  48, 48, 48, 48, 48, 48, 48, 48, 48, 48, 48, 48, 48, 48, 48, 48,
  49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49,
  49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49,
  50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50,
  51, 51, 51, 51, 51, 51, 51, 51, ERRC, ERRC, ERRC, ERRC, ERRC, ERRC, ERRC, ERRC};
```

33. The following routine is copied from Aho, Lam, Sethi and Ullman *Compilers, Principles, Techniques &Tools; ISBN 0-321-48681-1*, paragraph 3.9.8, Trading Time for Space in DFA Simulation, except that `default` has been replaced by `deflt`.

```
⟨ Lexical subroutines 30 ⟩ +≡
  int nextstate(int s, int a)
  {
    if (check[base[s] + a] ≡ s) return next[base[s] + a];
    return nextstate(deflt[s], a);
  }
```

34. When a newline character is encountered it is replaced by a null character, splitting the input file into an array of lines.

```
⟨ Lexical subroutines 30 ⟩ +≡
  tokentype *lexer(void)
  {
    ⟨ Local variables for lexer 37 ⟩
    llptr = llaptr = lbufptr;
    ⟨ Scan a token 35 ⟩
    llptr = lbufptr;
    if (*lbufptr ≡ '\n') srcline[llline++] = llaptr, llcolumn = 1, *lbufptr = '\0';
    lbufptr = llaptr;
    ⟨ Wrap up and fill the lexer variables 36 ⟩
    return token;
  }
```

35. The start state of the lexer is 1. A token is scanned when the dead state is reached.

```
⟨ Scan a token 35 ⟩ ≡
  for (llstate = 1, llaccept = 0, llpos = llcolumn; llstate > 0; ) {
    llstate = nextstate(llstate, peek( ));
    if (llstate ∧ *llptr) fetch( );
    if (accept[llstate] > 0) llaccept = accept[llstate], llaptr = llptr;
  }
```

This code is used in section 34.

36.

```
⟨ Wrap up and fill the lexer variables 36 ⟩ ≡
  token = allocate(1, sizeof(tokentype));
  token→name = allocate(llaptr − llptr + 1, sizeof(char *));
  strncpy(token→name, llptr, llaptr − llptr);
  if (llaccept ≡ IDENTIFIER) {
    symtype *key = getsym(token→name, false);
    if (key ≠ Λ) llaccept = key→usage ≡ KEYWORD ? key→var.value : key→usage;
  }
  token→type = llaccept, token→linenr = llline, token→column = llpos;
```

This code is used in section 34.

37. "llaptr" is the ptr to the last character of an accepted token.

```
⟨ Local variables for lexer 37 ⟩ ≡
  tokentype *token;
  char *llaptr;
  int llaccept, llstate, llpos;
```

This code is used in section 34.

38. ⟨ Global variables 15 ⟩ +≡

int *base*[ ] = {0, 48, 0, 95, 0, 94, 107, 107, 0, 106, 108, 98, 103, 0, 108, 114, 118, 125, 0, 137, 0, 138, 147, 147, 148, 148, 0, 140, 142, 0, 0, 0, 145, 146, 149, 0, 0, 0, 0, 0, 150, 0, 151, 0, 153, 0, 0, 155, 0, 150, 0, 157, 0, 134, 159, 0, 160, 0, 137, 162, 0, 150, 0, 164, 0, 165, 0, 0, 0, 166, 0, 0, 167, 0, 0, 0, 168, 0, 0, 0, 169, 0, 0, 0, 0, 191, 238, 0, 285, 303, 0};

int *deflt*[ ] = {0, 0, 0, 0, 0, 0, 5, 5, 5, 5, 5, 5, 0, 12, 0, 0, 0, 0, 0, 0, 19, 14, 14, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 49, 0, 0, 0, 53, 0, 0, 0, 0, 58, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int *check*[ ] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 6, 7, 9, 10, 11, 12, 12, 12, 12, 14, 14, 14, 14, 14, 14, 15, 15, 15, 15, 15, 16, 16, 16, 16, 0, 16, 16, 17, 17, 17, 17, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 21, 22, 23, 24, 25, 27, 27, 27, 27, 28, 32, 32, 32, 32, 33, 34, 34, 40, 42, 42, 44, 44, 47, 49, 51, 53, 54, 56, 58, 59, 61, 63, 65, 69, 72, 76, 80, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 88, 89, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int *next*[ ] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 3, 4, 5, 6, 5, 5, 9, 5, 5, 5, 5, 5, 5, 14, 21, 22, 14, 5, 63, 27, 32, 40, 42, 85, 86, 19, 65, 47, 51, 88, 67, 68, 69, 71, 72, 74, 75, 76, 78, 61, 79, 80, 82, 56, 83, 84, 87, 3, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 7, 8, 10, 11, 12, 13, 13, 13, 13, 16, 14, 14, 14, 14, 15, 16, 15, 15, 15, 15, 18, 18, 18, 18, 0, 17, 17, 18, 18, 18, 18, 20, 0, 20, 20, 20, 0, 20, 20, 20, 20, 20, 20, 22, 23, 24, 25, 26, 28, 0, 29, 30, 31, 33, 34, 35, 36, 37, 38, 39, 41, 43, 44, 45, 46, 48, 50, 52, 54, 55, 57, 59, 60, 62, 64, 66, 70, 73, 77, 81, 85, 0, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 85, 86, 0, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 87, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 89, 0, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 89, 90};

int *accept*[ ] = {−1, −1, `EOS`, `SPACE`, `SPACE`, `IDENTIFIER`, `IDENTIFIER`, `IDENTIFIER`, `FLTNUM`, `IDENTIFIER`, `IDENTIFIER`, `FLTNUM`, −1, `FLTNUM`, `NUMBER`, `FLTNUM`, −1, −1, `FLTNUM`, 0, `NUMBER`, `NUMBER`, `NUMBER`, −1, −1, −1, `IDENTIFIER`, `MINUS`, `DIFF`, `MINUSEQ`, `STORE`, `DIFFEQ`, `LT`, `LOAD`, `SHL`, `LE`, `NE`, `SWAP`, `SAVE`, `SHLEQ`, `ASSIGN`, `EQ`, `GT`, `GE`, `SHR`, `SHREQ`, `UNSAVE`, `REM`, `REMEQ`, `REM`, `ODD`, `AND`, `ANDEQ`, `AND`, `ANDN`, `ANDNEQ`, `OR`, `OREQ`, `OR`, `ORN`, `ORNEQ`, `INVALID`, `SQRT`, `PLUS`, `PLUSEQ`, `SADD`, `SADDEQ`, `LPRN`, `RPRN`, `TIMES`, `TIMESEQ`, `COMMA`, `DIV`, `DIVEQ`, `COLON`, `SEMICOLON`, `CMP`, `CMPEQ`, `LBRK`, `RBRK`, `XOR`, `XOREQ`, `LBRC`, `RBRC`, `NOT`, `COMMENT`, `INVALID`, `STRING`, `INVALID`, `CHARACTER`, `CHARACTER`};

39. Errors. Error reporting in a compiler is always difficult. The error may be discovered much later than it occurred. Grammar and semantic routines may *report* an error when they discover one. At the end of the run all the errors found are presented along with the code where they were discovered.

```
#define MAXERR 25
```

40. ⟨ Error subroutines 40 ⟩ ≡

```
bool report(int linenr, int column, int errcl, int errnr)
{
  if (nxterr < MAXERR) {
    errpos[nxterr].linenr = linenr > 0 ? linenr - 1 : linenr;
    errpos[nxterr].column = column;
    errpos[nxterr].errcl = errcl;
    errpos[nxterr].errnr = errnr;
    nxterr++;
  }
  return true;
}
```

See also section 41.

This code is used in section 12.

41. Print the errors sorted by line and column.

⟨ Error subroutines 40 ⟩ +≡

```
void prterr()
{
  int i, j, m, v;
  for (i = 0; i < nxterr - 1; i++) {
    m = i;
    for (j = i + 1; j < nxterr; j++) {
      if ((errpos[j].linenr < errpos[m].linenr) ∨ ((errpos[j].linenr ≡
          errpos[m].linenr) ∧ (errpos[j].column < errpos[m].column))) m = j;
    }
    if (m ≠ i) {
      v = errpos[i].linenr; errpos[i].linenr = errpos[m].linenr; errpos[m].linenr = v;
      v = errpos[i].column; errpos[i].column = errpos[m].column; errpos[m].column = v;
      v = errpos[i].errcl; errpos[i].errcl = errpos[m].errcl; errpos[m].errcl = v;
      v = errpos[i].errnr; errpos[i].errnr = errpos[m].errnr; errpos[m].errnr = v;
    }
  }
  for (i = 0; i < nxterr; i++) {
    printf("%s\n", srcline[errpos[i].linenr]);
    for (int j = 1; j < errpos[i].column; j++) printf("-");
    printf("^\n");
    printf("%s%s␣at␣line␣%d\n\n", errclass[errpos[i].errcl], errdesc[errpos[i].errnr], errpos[i].linenr+1);
  }
}
```

42. ⟨ Type definitions 24 ⟩ +≡
```
typedef struct {
  int linenr;
  int column;
  int errcl;
  int errnr;
} errtype;
```

43. ⟨ Global variables 15 ⟩ +≡
```
errtype errpos[MAXERR];
int nxterr = 0;
```

44. ⟨ Type definitions 24 ⟩ +≡
```
typedef enum {
  ERRDUPL, ERRADDR, ERRCHAR, ERREXPR, ERRXTRN, ERRINIT, ERRNPAR, ERRINVR, ERRSTMT, ERRTYPE,
      ERRLBRC, ERRRBRC, ERRRPRN, ERRSCLN, ERRCOMP, ERRZERO, ERREVEN, ERRHLNM, ERRADDU, ERRIDRQ,
      ERRNVAL, ERRLBRQ, ERRNMRQ, ERRRNRQ, ERRPROC, ERRRGRQ, ERRZSYM, ERRSRRQ, ERRUNTL, SENTINEL
} errnrtype;
typedef enum {
  ERRLEX, ERRSYN, ERRSEM
} errcltype;
```

45. ⟨ Global variables 15 ⟩ +≡

```
char *errdesc[] = {
"already␣defined",
"invalid␣address",
"invalid␣character␣constant",
"invalid␣expression",
"invalid␣extern␣declaration",
"invalid␣initialisation",
"invalid␣parameter",
"invalid␣register",
"invalid␣statement",
"invalid␣type",
"missing␣'{'",
"missing␣'}'",
"missing␣')'",
"missing␣';'",
"missing␣'<',␣'<=',␣'<>',␣'=>',␣'>',␣'=='␣or␣'%%'",
"missing␣'0'",
"missing␣'0'␣or␣'1'",
"missing␣'16',␣'32'␣or␣'48'",
"missing␣'2',␣'4',␣'8'␣or␣'16'",
"missing␣identifier",
"missing␣initialisation",
"missing␣label",
"missing␣number",
"missing␣number␣or␣register",
"missing␣procedure",
"missing␣register",
"missing␣register␣or␣'0'",
"missing␣special␣register",
"missing␣'until'",
"Sentinel"
};
char *errclass[] = {
"Lexical␣error:␣",
"Syntax␣error:␣",
"Semantic␣error:␣",
};
```

46. Grammar. PL/MMIX not only uses a hand written lexer but also a hand written parser. Aho, Lam, Sethi and Ullman *Compilers, Principles, Techniques, &Tools; ISBN 0-321-48681-1*, Chapter 4 describes different types of parsers.

The parser builds a parse tree where each node represents a token and its function. A sequence of instructions is represented by nodes that are linked by the next pointer, where the children of the node (cl, cm and cr) are used to build the paramaters of the node.

Block nodes have an associated symbol table for the local variables. The local symbol table points to the local symbol table of the parent node. The global symbol table is the local symbol table of the root node.

47. ⟨ Type definitions 24 ⟩ +≡

```
typedef struct nodestruct {
  struct nodestruct *next, *cl, *cm, *cr, *rel;
  symtabtype *symtab;
  tokentype *lexeme;
  symtype *symbol;
  vartype var;
  int ntype, vtype;
  int inst1, inst2;
  bool hasvalue, isextern;
} node;
```

48. ⟨ Global variables 15 ⟩ +≡

```
tokentype *ggtoken = Λ;
node *ggroot = Λ;
```

49. If the previous token is not going to be used, for instance if it was a keyword, its memory is returned. The caller must copy *ggtoken* and set it to Λ if the token must be kept for later use.

⟨ Syntactic subroutines 49 ⟩ ≡

```
void lookahead( )
{
  if (ggtoken ≠ Λ) {
    if (ggtoken→name ≠ Λ) free(ggtoken→name);
    free(ggtoken);
  }
  while ((ggtoken = lexer( ))→type ≥ COMMENT) ;
}
```

See also sections 50, 52, 54, 55, 56, 58, 59, 60, 61, 62, 64, 65, 66, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 86, 87, 88, 89, 90, and 91.

This code is used in section 12.

50. A new node is created using *mknode*.

⟨ Syntactic subroutines 49 ⟩ +≡

```
node *mknode(node *l, node *m, node *r, int t)
{
  node *n = (node *) allocate(1, sizeof(node));
  n→cl = l; n→cm = m; n→cr = r;
  n→ntype = t;
  return n;
}
```

51. ⟨ Global variables 15 ⟩ +≡
```
tokentype lexeme0 = {"0", NUMBER, 0, 0};
tokentype lexemeR = {"ROUND", NUMBER, 0, 0};
node dummy0 = {Λ, Λ, Λ, Λ, Λ, Λ, &lexeme0, Λ, {0, UBYTE}, NUMBER, UBYTE, 0, 0, false};
node dummyR = {Λ, Λ, Λ, Λ, Λ, Λ, &lexemeR, Λ, {4, UBYTE}, NUMBER, UBYTE, 0, 0, false};
```

52. When a syntax error is discovered, the parser must forward to a location from where it is safe to continue parsing. The ";" is an obvious choice as it marks the end of an operation or statement. The parameter *movebeyond* indicates whether parsing continues at or just beyond this token.

⟨ Syntactic subroutines 49 ⟩ +≡
```
void recover( )
{
  for ( ; (ggtoken→type < LBRC) ∨ (ggtoken→type > SEMICOLON); lookahead( )) {
    if (ggtoken→type ≡ EOS) break;
  }
  return;
}
```

53. Most statements end with a semicolon. Here is some common code for checking the existence of the semicolon.

⟨ Check closing semicolon 53 ⟩ ≡
```
if (ggtoken→type ≡ SEMICOLON) lookahead( );
else {
  report(ggtoken→linenr, ggtoken→column, ERRSYN, ERRSCLN), recover( );
  lookahead( );
}
```
This code is used in sections 73, 76, 77, 79, 80, 81, 82, 87, and 90.

54. It is good tradition in computer science to draw trees with the root on top and the branches at the bottom. In line with this tradition we are going to describe how a tree looks by first recognising the leaves ending with recognising the root. Whether this should be considered bottom up or top down is left to the reader. The leaves of the tree are identifiers, numbers and special registers.

⟨ Syntactic subroutines 49 ⟩ +≡
```
node *ggid( )
{
  node *leaf = Λ;
  if (ggtoken→type ≡ IDENTIFIER) {
    leaf = mknode(Λ, Λ, Λ, IDENTIFIER);
    move(leaf→lexeme, ggtoken);
    lookahead( );
  }
  else report(ggtoken→linenr, ggtoken→column, ERRSYN, ERRIDRQ), recover( );
  return leaf;
}
```

55. ⟨ Syntactic subroutines 49 ⟩ +≡

```
node *ggreg( )
{
  node *leaf = Λ;
  if (ggtoken→type ≡ SPECIAL) {
    leaf = mknode(Λ, Λ, Λ, SPECIAL);
    leaf→symbol = getsym(ggtoken→name, false);
    move(leaf→lexeme, ggtoken);
    lookahead( );
  }
  else report(ggtoken→linenr, ggtoken→column, ERRSYN, ERRSRRQ), recover( );
  return leaf;
}
```

56. ⟨ Syntactic subroutines 49 ⟩ +≡

```
node *ggnum( )
{
  node *leaf = Λ;
  if ((ggtoken→type ≡ NUMBER) ∨ (ggtoken→type ≡ FLTNUM)) {
    ⟨ Recognise different types of numbers 57 ⟩
  }
  else if (ggtoken→type ≡ CHARACTER) {
    char *c = ggtoken→name + 1;
    leaf = mknode(Λ, Λ, Λ, NUMBER);
    leaf→var.value = utfval(c);
    c += (ctype[*c] < UTFC) ? 1 : ctype[*c] − UTFC + 1;
    if (*c ≠ '\'') report(ggtoken→linenr, ggtoken→column, ERRLEX, ERRCHAR), recover( );
  }
  else {
    report(ggtoken→linenr, ggtoken→column, ERRSYN, ERRNMRQ), recover( );
    return Λ;
  }
  leaf→hasvalue = true;
  if (¬leaf→var.type) {
    octa a = leaf→var.value;
    leaf→var.type = a < #100 ? UBYTE : a < #10000 ? UWYDE : a < #100000000 ? UTETRA : UOCTA;
  }
  move(leaf→lexeme, ggtoken);
  lookahead( );
  return leaf;
}
```

57. ⟨ Recognise different types of numbers 57 ⟩ ≡

```
if (ggtoken→name[0] ≡ '#') {
  leaf = mknode(Λ, Λ, Λ, NUMBER);
  for (char *p = ggtoken→name + 1; *p; p++)
    leaf→var.value = leaf→var.value * 16 + (*p ≥ 'a' ? *p − 'W' : *p ≥ 'A' ? *p − '7' : *p − '0');
}
else {
  leaf = mknode(Λ, Λ, Λ, NUMBER + scan_const(ggtoken→name));
  leaf→var.value = val, leaf→var.type = leaf→ntype ≡ NUMBER ? 0 : TFLOAT;
}
```

This code is used in section 56.

58. An atom is any leaf or an expression between braces.

$$\langle\,\text{atom}\,\rangle \longrightarrow \langle\,\text{constant}\,\rangle \mid \langle\,\text{symbol}\,\rangle \mid \texttt{(}\langle\,\text{expression}\,\rangle\texttt{)}$$

⟨ Syntactic subroutines 49 ⟩ +≡

```
node *ggexpr( );
node *ggatom( )
{
  int type = 0;
  tokentype *t;
  node *branch = Λ;
  switch (type = ggtoken→type) {
  case IDENTIFIER: return ggid( );
  case NUMBER: case FLTNUM: case CHARACTER: return ggnum( );
  case SPECIAL: return ggreg( );
  case LPRN: move(t, ggtoken);
    lookahead( );
    if ((branch = ggexpr( )) ≠ Λ) {
      branch = mknode(Λ, branch, Λ, EXPRESSION);
      move(branch→lexeme, t);
    }
    if (ggtoken→type ≡ RPRN) lookahead( );
    else report(ggtoken→linenr, ggtoken→column, ERRSYN, ERRRPRN), recover( );
    break;
  case SEMICOLON: case RPRN: case RBRC: case COMMA: break;
  default: report(ggtoken→linenr, ggtoken→column, ERRSYN, ERREXPR), recover( );
  }
  return branch;
}
```

59. A unit is an atom preceded with zero or more unary operators.

$$\langle\,\text{unit}\,\rangle \longrightarrow \langle\,\text{atom}\,\rangle \mid \langle\,\text{unary operator}\,\rangle\langle\,\text{unit}\,\rangle$$

This grammar rule is right recursive. The routine is therefore also recursive. A zero is inserted as a left leaf, which changes $-a$ into $0-a$ and $+a$ into $0+a$. For square root `ROUND` is inserted as a left hand instead of zero.

```
⟨ Syntactic subroutines 49 ⟩ +≡
  node *ggunit( )
  {
    int type = ggtoken→type;
    tokentype *t;
    node *leaf = Λ;
    switch (type) {
    case PLUS: case MINUS: case SQRT: case NOT: move(t, ggtoken);
      lookahead( );
      if ((leaf = ggunit( )) ≡ Λ) return Λ;
      leaf = mknode((type ≡ SQRT ? &dummyR : &dummy0), Λ, leaf, type);
      leaf→lexeme = t;
      return leaf;
    }
    return ggatom( );
  }
```

60. Strong and weak are binary expressions of strong (multiplicative) or weak (additive) operators.

$$\langle\,\text{strong}\,\rangle \longrightarrow \langle\,\text{unit}\,\rangle \mid \langle\,\text{strong}\,\rangle\langle\,\text{strong operator}\,\rangle\langle\,\text{unit}\,\rangle$$
$$\langle\,\text{weak}\,\rangle \longrightarrow \langle\,\text{strong}\,\rangle \mid \langle\,\text{weak}\,\rangle\langle\,\text{weak operator}\,\rangle\langle\,\text{strong}\,\rangle$$
$$\langle\,\text{strong operator}\,\rangle \longrightarrow \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{<<} \mid \texttt{>>} \mid \texttt{\&} \mid \texttt{\&\~{}}$$
$$\langle\,\text{weak operator}\,\rangle \longrightarrow \texttt{+} \mid \texttt{-} \mid \texttt{|} \mid \texttt{|\~{}} \mid \texttt{\^{}} \mid \texttt{\$} \mid \texttt{--}$$

```
⟨ Syntactic subroutines 49 ⟩ +≡
  node *ggstrong( )
  {
    tokentype *t;
    node *branch = Λ, *leaf = Λ;
    for (branch = ggunit( ); (ggtoken→type ≥ TIMES) ∧ (ggtoken→type ≤ ANDN); ) {
      move(t, ggtoken); lookahead( );
      leaf = ggunit( );
      if (leaf ≡ Λ) return &dummy0;
      else branch = mknode(branch, Λ, leaf, t→type); branch→lexeme = t;
    }
    return branch;
  }
```

61. ⟨ Syntactic subroutines 49 ⟩ +≡

```
node *ggweak( )
{
  int type;
  tokentype *t;
  node *branch = Λ, *leaf = Λ;
  for (branch = ggstrong( ); (ggtoken→type ≥ PLUS) ∧ (ggtoken→type ≤ XOR); ) {
    move(t, ggtoken); lookahead( );
    leaf = ggstrong( );
    if (leaf ≡ Λ) return &dummy0;
    else branch = mknode(branch, Λ, leaf, t→type); branch→lexeme = t;
  }
  return branch;
}
```

62. A comparison expression is a binary expression of a comparison between two weak expressions.

⟨ comparison ⟩ ⟶ ⟨ weak ⟩⟨ comparison operator ⟩⟨ weak ⟩
⟨ comparison operator ⟩ ⟶ `<` | `>` | `==` | `<=` | `>=` | `<>` | `%%`

⟨ Syntactic subroutines 49 ⟩ +≡

```
node *ggcomp( )
{
  int type;
  tokentype *t;
  node *branch = Λ, *leaf = Λ;
  branch = ggweak( );
  if (branch ≠ Λ) {
    type = ggtoken→type;
    switch (type) {
    case LT: case GT: case EQ: case LE: case GE: case NE: move(t, ggtoken); lookahead( );
      if ((leaf = ggweak( )) ≡ Λ) return &dummy0;
      break;
      ⟨ The odd case 63 ⟩
    default: return branch;
    }
    branch = mknode(branch, Λ, leaf, type); branch→lexeme = t;
  }
  return branch;
}
```

63. Odd or even are tested by `%%` which can be read as remainder when dividing by 2. It may only be followed by a literal 0 or 1.

```
⟨ The odd case 63 ⟩ ≡
case ODD: move(t, ggtoken); lookahead( );
  leaf = ggweak( );
  if ((leaf→ntype ≡ NUMBER) ∧ (leaf→var.value < 2)) type = leaf→var.value ? ODD : EVEN;
  else {
    report(leaf→lexeme→linenr, leaf→lexeme→column, ERRSYN, ERREVEN), recover( );
    return Λ;
  }
  break;
```

This code is used in section 62.

64. The *ggexpr* routine is the top of the expression subtree. It is used in initialisations, instructions and in assembler statements.

⟨ expression ⟩ ⟶ ⟨ comparison ⟩ | ⟨ comparison ⟩`?`⟨ weak ⟩ | ⟨ comparison ⟩`?`⟨ weak ⟩`:`⟨ expression ⟩

```
⟨ Syntactic subroutines 49 ⟩ +≡
  node *ggexpr( )
  {
    int type;
    tokentype *t;
    node *branch = Λ, *leaf = Λ;
    branch = ggcomp( );
    if (branch ≠ Λ) {
      if (ggtoken→type ≡ CMP) {
        move(t, ggtoken); lookahead( );
        branch = mknode(branch, Λ, ggweak( ), CMP);
        if (ggtoken→type ≡ COLON) {
          move(t, ggtoken); lookahead( );
          branch→cm = ggexpr( );
          branch→ntype = COLON;
        }
      }
    }
    return branch;
  }
```

65. The *ggdcl* routine scans a simple variable declaration. The variable is optionally preceded by a sign or a size or both. If a sign or size is found without a variable an error is flagged. The *ggidcl* routine is a declaration optionally followed by an initialisation. The initialisation can be a single value or a list of values.

⟨ initialisation ⟩ ⟶ ⟨ typed symbol ⟩ | ⟨ typed symbol ⟩`=`⟨ expression ⟩ |
⟨ typed symbol ⟩`={`⟨ expression list ⟩`}`
⟨ expression list ⟩ ⟶ ⟨ expression ⟩ ⟨ expression list ⟩ | ⟨ expression ⟩
⟨ typed symbol ⟩ ⟶ ⟨ type ⟩⟨ sign ⟩⟨ symbol ⟩ | ⟨ type ⟩⟨ symbol ⟩ | ⟨ sign ⟩⟨ symbol ⟩ | ⟨ symbol ⟩
⟨ type ⟩ ⟶ `byte` | `wyde` | `tetra` | `octa` | `float` | `sfloat`
⟨ sign ⟩ ⟶ `signed` | `unsigned`

```
⟨ Syntactic subroutines 49 ⟩ +≡
  node *ggdcl( )
  {
    if (ggtoken→type ≡ SPECIAL) return mknode(ggreg( ), Λ, Λ, 0);
    if (ggtoken→type ≡ IDENTIFIER) return mknode(ggid( ), Λ, Λ, 0);
    return Λ;
  }
```

66.
```
⟨ Syntactic subroutines 49 ⟩ +≡
  node *ggidcl( )
  {
    node *leaf = Λ;
    if (¬(leaf = ggdcl( ))) return Λ;
    if (ggtoken→type ≡ ASSIGN) {
      lookahead( );
      if (ggtoken→type ≡ LBRC) {
        ⟨ Handle list assignment 67 ⟩
      }
      else {
        if (ggtoken→type ≡ STRING) {
          leaf→cr = mknode(Λ, Λ, Λ, STRING);
          move(leaf→cr→lexeme, ggtoken);
          lookahead( );
        }
        else leaf→cr = ggexpr( );
        if (¬leaf→cr) report(ggtoken→linenr, ggtoken→column, ERRSYN, ERRINIT);
      }
    }
    return leaf;
  }
```

67. ⟨ Handle list assignment 67 ⟩ ≡

```
lookahead( );
if (ggtoken→type ≡ RBRC) {
  report(ggtoken→linenr, ggtoken→column, ERRSYN, ERRINIT);
  lookahead( );
}
else {
  node *branch = Λ, *twig = Λ, *last = Λ;
  if (¬(branch = twig = ggexpr( ))) report(ggtoken→linenr, ggtoken→column, ERRSYN, ERRINIT);
  while (ggtoken→type ≡ COMMA) {
    lookahead( );
    last = ggexpr( );
    if (¬last) report(ggtoken→linenr, ggtoken→column, ERRSYN, ERRINIT);
    if (¬branch) branch = twig = last;
    else if (last) twig→next = last, twig = twig→next;
  }
  if (ggtoken→type ≡ RBRC) lookahead( );
  else report(ggtoken→linenr, ggtoken→column, ERRSYN, ERRRBRC), recover( );
  leaf→cr = branch;
}
```

This code is used in section 66.

68. The routine ggtype returns an optional type as a mmixtype or 0 nothing is found. The calling routine must set the default if 0 is returned.

⟨ Syntactic subroutines 49 ⟩ +≡

```
int ggtype( )
{
  int sign = 0, size = 0;
  if ((ggtoken→type ≥ SIGNED) ∧ (ggtoken→type ≤ UNSIGNED)) {
    sign = ggtoken→type;
    lookahead( );
  }
  if ((ggtoken→type ≥ BYTE) ∧ (ggtoken→type ≤ SFLOAT)) {
    size = ggtoken→type;
    lookahead( );
  }
  if (sign ∨ size) {
    if (¬size) size = OCTA;
    if (¬sign) sign = (size ≤ OCTA) ? UNSIGNED : SIGNED;
    if (size ≤ OCTA) return (size − BYTE) * 2 + (sign − SIGNED) + 1;
    else return TFLOAT + size − FLOAT;
  }
  return 0;
}
```

69. The routines *ggdcls* and *ggidcls* scan a list of *ggdcl* or *ggidcl* declarations. Each declaration is separated by a comma.

⟨ symbol list ⟩ ⟶ ⟨ symbol ⟩ , ⟨ symbol list ⟩ | ⟨ symbol ⟩
⟨ initialisation list ⟩ ⟶ ⟨ initialisation ⟩ , ⟨ initialisation list ⟩ | ⟨ initialisation ⟩

```
⟨ Syntactic subroutines 49 ⟩ +≡
  node *ggdcls( )
  {
    node *branch = Λ, *leaf = Λ;
    int vtype = ggtype( );
    if (¬vtype ∧ ¬((ggtoken→type ≡ IDENTIFIER) ∨ (ggtoken→type ≡ SPECIAL))) return Λ;
    branch = leaf = ggdcl( );
    while (ggtoken→type ≡ COMMA) {
      lookahead( );
      if (ggtoken→type ≡ COMMA) report(ggtoken→linenr, ggtoken→column, ERRSYN, ERRIDRQ), recover( );
      else if (leaf) leaf→next = ggdcl( );
      else branch = leaf = ggdcl( );
      if (leaf ∧ leaf→next) leaf = leaf→next;
    }
    for (node *n = branch; n; n = n→next) n→vtype = vtype;
    return branch;
  }
```

70.
```
⟨ Syntactic subroutines 49 ⟩ +≡
  node *ggtdcls( )
  {
    node *branch = Λ, *leaf = Λ;
    if ((ggtoken→type ≡ OR) ∨ (ggtoken→type ≡ RPRN)) return Λ;
    int vtype = ggtype( );
    if (¬vtype) vtype = UOCTA;
    branch = leaf = mknode(ggid( ), Λ, Λ, LOCAL);
    if (leaf) leaf→vtype = vtype;
    while (ggtoken→type ≡ COMMA) {
      lookahead( );
      vtype = ggtype( );
      if (¬vtype) vtype = UOCTA;
      if (leaf) {
        leaf→next = mknode(ggid( ), Λ, Λ, LOCAL);
        if (leaf→next) leaf = leaf→next, leaf→vtype = vtype;
      }
      else branch = leaf = mknode(ggid( ), Λ, Λ, LOCAL);
    }
    return branch;
  }
```

71. ⟨ Syntactic subroutines 49 ⟩ +≡

```
node *ggidcls( )
{
  node *branch = Λ, *leaf = Λ;
  int vtype = ggtype( );
  if (¬vtype ∧ ¬((ggtoken→type ≡ IDENTIFIER) ∨ (ggtoken→type ≡ SPECIAL))) return Λ;
  branch = leaf = ggidcl( );
  if (vtype ∧ ¬branch) report(ggtoken→linenr, ggtoken→column, ERRSYN, ERRIDRQ), recover( );
  if (¬vtype) vtype = UOCTA;
  if (leaf) leaf→vtype = vtype;
  while (ggtoken→type ≡ COMMA) {
    lookahead( );
    if (ggtoken→type ≡ COMMA) report(ggtoken→linenr, ggtoken→column, ERRSYN, ERRIDRQ);
    else if (leaf) leaf→next = ggidcl( );
    else branch = leaf = ggidcl( );
    if (leaf ∧ leaf→next) leaf = leaf→next;
  }
  if (¬vtype) vtype = UOCTA;
  for (node *n = branch; n; n = n→next) n→vtype = vtype;
  return branch;
}
```

72. Scan the header of a procedure. A procedure may have input variables, output variables or both. These are local variables to the procedure.

⟨ pheader ⟩ ⟶ `procedure` ⟨ symbol ⟩ | `procedure` ⟨ symbol ⟩ ⟨ parameter list ⟩
⟨ parameter list ⟩ ⟶ `(` ⟨ parameters ⟩ `)`
⟨ parameters ⟩ ⟶ ⟨ symbol list ⟩ | ⟨ symbol list ⟩ `|` | ⟨ symbol list ⟩ `|` ⟨ symbol list ⟩ | `|` ⟨ symbol list ⟩

⟨ Syntactic subroutines 49 ⟩ +≡

```
node *ggphdr( )
{
  node *branch = Λ, *ltwig = Λ, *rtwig = Λ, *leaf = Λ;
  leaf = ggid( );
  if (leaf) {
    if (ggtoken→type ≡ LPRN) {
      lookahead( );
      ltwig = ggtdcls( );
      if (ggtoken→type ≡ OR) {
        lookahead( );
        rtwig = ggtdcls( );
      }
      if (ggtoken→type ≡ RPRN) lookahead( );
      else report(ggtoken→linenr, ggtoken→column, ERRSYN, ERRRPRN), recover( );
    }
  }
  else {
    recover( );
    return Λ;
  }
  return mknode(leaf, mknode(ltwig, Λ, rtwig, PARAMETER), Λ, PROCEDURE);
}
```

73. Data and procedures in other modules are declared as extern.

⟨ extern ⟩ ⟶ `extern`⟨ linkable ⟩
⟨ linkable ⟩ ⟶ ⟨ global ⟩ | ⟨ data ⟩ | ⟨ constant ⟩ | ⟨ pheader ⟩`;`

```
⟨ Syntactic subroutines 49 ⟩ +≡
  node *ggextrn( )
  {
    node *branch;
    int type = 0;
    switch (type = ggtoken→type) {
    case GLOBAL: case DATA: case CONSTANT: lookahead( );
      branch = ggdcls( );
      for (node *n = branch; n; n = n→next) n→ntype = type, n→isextern = true, n→cr = &dummy0;
      break;
    case PROCEDURE: lookahead( );
      branch = ggphdr( );
      branch→isextern = true;
      break;
    default: report(ggtoken→linenr, ggtoken→column, ERRSYN, ERRXTRN), recover( );
      return Λ;
    }
    ⟨ Check closing semicolon 53 ⟩
    return branch;
  }
```

74. Scan an if, while, until, goto or return statement.

⟨ if ⟩ ⟶ `if`⟨ comparison ⟩⟨ block ⟩ | `if`⟨ comparison ⟩⟨ block ⟩`else`⟨ block ⟩
⟨ while ⟩ ⟶ `while`⟨ comparison ⟩⟨ block ⟩
⟨ until ⟩ ⟶ `do`⟨ block ⟩`until`⟨ comparison ⟩`;`
⟨ goto ⟩ ⟶ `goto`⟨ symbol ⟩`;`
⟨ call ⟩ ⟶ `call`⟨ symbol ⟩⟨ parameterlist ⟩`;` | `call`⟨ symbol ⟩`;`
⟨ return ⟩ ⟶ `return;` | `return`⟨ decimal ⟩`;`

```
⟨ Syntactic subroutines 49 ⟩ +≡
  node *ggblck( );
  node *ggifte( )
  {
    node *branch = Λ, *ltwig = Λ, *mtwig = Λ, *rtwig = Λ;
    lookahead( );
    ltwig = ggcomp( );
    mtwig = ggblck( );
    if (ggtoken→type ≡ ELSE) {
      lookahead( );
      rtwig = ggblck( );
    }
    if (ltwig ∧ mtwig) branch = mknode(ltwig, mtwig, rtwig, IF);
    return branch;
  }
```

75. ⟨ Syntactic subroutines 49 ⟩ +≡

```
node *ggwhile( )
{
  node *branch = Λ, *ltwig = Λ, *mtwig = Λ;
  lookahead( );
  ltwig = ggcomp( );
  mtwig = ggblck( );
  if (ltwig ∧ mtwig) branch = mknode(ltwig, mtwig, Λ, WHILE);
  return branch;
}
```

76. ⟨ Syntactic subroutines 49 ⟩ +≡

```
node *gguntil( )
{
  node *branch = Λ, *ltwig = Λ, *mtwig = Λ;
  lookahead( );
  mtwig = ggblck( );
  if (ggtoken→type ≡ UNTIL) {
    lookahead( );
    ltwig = ggcomp( );
    ⟨ Check closing semicolon 53 ⟩
  }
  else {
    report(ggtoken→linenr, ggtoken→column, ERRSYN, ERRUNTL), recover( );
  }
  if (ltwig ∧ mtwig) branch = mknode(ltwig, mtwig, Λ, UNTIL);
  return branch;
}
```

77. ⟨ Syntactic subroutines 49 ⟩ +≡

```
node *gggoto( )
{
  node *leaf = Λ;
  tokentype *t;
  move(t, ggtoken);
  lookahead( );
  if (ggtoken→type ≡ IDENTIFIER) {
    leaf = mknode(ggid( ), Λ, Λ, t→type);
    ⟨ Check closing semicolon 53 ⟩
  }
  else report(ggtoken→linenr, ggtoken→column, ERRSYN, ERRIDRQ), recover( );
  return leaf;
}
```

78. ⟨ Syntactic subroutines 49 ⟩ +≡

```
node *ggparm( )
{
  node *branch = Λ, *twig = Λ, *leaf = Λ;
  if (ggtoken→type ≡ IDENTIFIER) {
    branch = twig = mknode(ggid( ), Λ, Λ, 0);
    while (ggtoken→type ≡ COMMA) {
      lookahead( );
      leaf = mknode(ggid( ), Λ, Λ, 0);
      if (leaf) twig→next = leaf, twig = leaf;
      else recover( );
    }
  }
  return branch;
}
```

79. ⟨ Syntactic subroutines 49 ⟩ +≡

```
node *ggcall( )
{
  node *leaf = Λ, *ltwig = Λ, *rtwig = Λ, *branch = Λ;
  lookahead( );
  leaf = ggid( );
  if (leaf) {
    if (ggtoken→type ≡ LPRN) {
      lookahead( );
      ltwig = ggparm( );
      for (node *n = ltwig; n; n = n→next) n→ntype = IN;
      if (ggtoken→type ≡ OR) {
        lookahead( );
        rtwig = ggparm( );
        for (node *n = rtwig; n; n = n→next) n→ntype = OUT;
      }
      if (ggtoken→type ≡ RPRN) lookahead( );
      else {
        report(ggtoken→linenr, ggtoken→column, ERRSYN, ERRRPRN), recover( );
        return Λ;
      }
    }
    branch = mknode(leaf, mknode(ltwig, Λ, rtwig, ACTUAL), Λ, CALL);
    ⟨ Check closing semicolon 53 ⟩
  }
  else recover( );
  return branch;
}
```

80. ⟨ Syntactic subroutines 49 ⟩ +≡

```
node *ggrtrn( )
{
  node *branch = Λ, *leaf = Λ;
  lookahead( );
  if (ggtoken→type ≡ NUMBER) leaf = ggnum( );
  branch = mknode(leaf, Λ, Λ, RETURN);
  ⟨ Check closing semicolon 53 ⟩
  return branch;
}
```

81. The PL/MMIX language is built around assembler instructions and ordinary PL/MMIX instructions. They are parsed by *ggopc* and *ggins*.

⟨ assembler ⟩ ⟶ ⟨ opcode ⟩⟨ xyz ⟩
⟨ xyz ⟩ ⟶ ⟨ expression ⟩ | ⟨ expression ⟩, ⟨ expression ⟩ | ⟨ expression ⟩, ⟨ expression ⟩, ⟨ expression ⟩ |
⟨ simple ⟩ ⟶ ⟨ assignment ⟩ | ⟨ load ⟩ | ⟨ store ⟩ | ⟨ swap ⟩ | ⟨ save ⟩ | ⟨ unsave ⟩
⟨ assignment ⟩ ⟶ ⟨ symbol ⟩=⟨ expression ⟩;
⟨ load ⟩ ⟶ ⟨ symbol ⟩<-⟨ expression ⟩;
⟨ store ⟩ ⟶ ⟨ expression ⟩->⟨ expression ⟩;
⟨ swap ⟩ ⟶ ⟨ symbol ⟩<->⟨ expression ⟩;
⟨ save ⟩ ⟶ ⟨ symbol ⟩<<<;
⟨ unsave ⟩ ⟶ >>>⟨ symbol ⟩;

⟨ Syntactic subroutines 49 ⟩ +≡

```
node *ggopc( )
{
  node *twig = Λ;
  twig = mknode(Λ, Λ, Λ, OPCODE);
  twig→symbol = getsym(ggtoken→name, false);
  move(twig→lexeme, ggtoken);
  lookahead( );
  twig→cl = ggexpr( );
  if (ggtoken→type ≡ COMMA) {
    lookahead( );
    twig→cm = ggexpr( );
    if (ggtoken→type ≡ COMMA) {
      lookahead( );
      twig→cr = ggexpr( );
    }
  }
  ⟨ Check closing semicolon 53 ⟩
  return twig;
}
```

82. ⟨ Syntactic subroutines 49 ⟩ +≡

```
node *ggins( )
{
  node *branch = 0, *twig = Λ;
  twig = ggexpr( );
  ⟨ Handle parsing SAVE, UNSAVE, ASSIGN, LOAD, STORE, SWAP 83 ⟩
  ⟨ Handle parsing += ... ⊕= 84 ⟩
  ⟨ Handle unrecognised statement 85 ⟩
  if (branch→ntype ≠ LABEL) {
    ⟨ Check closing semicolon 53 ⟩
  }
  if (branch ∧ branch→cr ≡ Λ) branch = Λ;
  return branch;
}
```

83. ⟨ Handle parsing SAVE, UNSAVE, ASSIGN, LOAD, STORE, SWAP 83 ⟩ ≡

```
if ((ggtoken→type ≡ SAVE) ∨ (ggtoken→type ≡ UNSAVE)) {
  branch = mknode(twig, Λ, &dummy0, ggtoken→type);
  move(branch→lexeme, ggtoken);
  lookahead( );
}
else if (ggtoken→type ≡ COLON) {
  branch = mknode(twig, Λ, &dummy0, LABEL);
  move(branch→lexeme, ggtoken);
  lookahead( );
}
else if ((ggtoken→type ≥ ASSIGN) ∧ (ggtoken→type ≤ SWAP)) {
  branch = mknode(twig, Λ, Λ, ggtoken→type);
  move(branch→lexeme, ggtoken);
  lookahead( );
  branch→cr = ggexpr( );
}
```

This code is used in section 82.

84. ⟨ Handle parsing += ... ⊕= 84 ⟩ ≡

```
else
  if ((ggtoken→type ≥ PLUSEQ) ∧ (ggtoken→type ≤ XOREQ)) {
    branch = mknode(twig, Λ, mknode(twig, Λ, Λ, ggtoken→type − PLUSEQ + PLUS), ASSIGN);
    move(branch→lexeme, ggtoken);
    lookahead( );
    branch→cr→cr = ggexpr( );
  }
```

This code is used in section 82.

85. ⟨ Handle unrecognised statement 85 ⟩ ≡

```
else {
  branch = mknode(twig, Λ, Λ, ERROR);
}
```

This code is used in section 82.

86. A block contains a series of statements and statements can contain blocks. So this grammar has indirect recursion too.

⟨ statement ⟩ ⟶ ⟨ assembler ⟩ | ⟨ if ⟩ | ⟨ while ⟩ | ⟨ until ⟩ | ⟨ label ⟩ | ⟨ goto ⟩ | ⟨ call ⟩ | ⟨ return ⟩ | ⟨ simple ⟩
⟨ statements ⟩ ⟶ ⟨ statement ⟩⟨ statements ⟩ | ⟨ statement ⟩
⟨ block ⟩ ⟶ `{`⟨ local ⟩⟨ statements ⟩`}`

```
⟨ Syntactic subroutines 49 ⟩ +≡
  node *ggstmnt( )
  {
    switch (ggtoken→type) {
    case IF: return ggifte( );
    case WHILE: return ggwhile( );
    case DO: return gguntil( );
    case GOTO: return gggoto( );
    case CALL: return ggcall( );
    case RETURN: return ggrtrn( );
    case LBRC: return ggblck( );
    case OPCODE: return ggopc( );
    default: return ggins( );
    }
    return Λ;
  }
```

87. The routine *gglcl* scans a list of declarations and sets the nodes in the subtree to `LOCAL`.

⟨ local ⟩ ⟶ `local`⟨ symbol list ⟩`;`

```
⟨ Syntactic subroutines 49 ⟩ +≡
  node *gglcl( )
  {
    node *branch = Λ, *twig = Λ, *leaf = Λ;
    while (ggtoken→type ≡ LOCAL) {
      lookahead( );
      int vtype = ggtype( );
      leaf = ggdcls( );
      if (¬vtype ∧ ¬leaf) report(ggtoken→linenr, ggtoken→column, ERRSYN, ERRIDRQ), recover( );
      if (¬vtype) vtype = UOCTA;
      ⟨ Check closing semicolon 53 ⟩
      if (twig) twig→next = leaf, twig = twig→next;
      else branch = twig = leaf;
      for (node *n = twig; n; n = n→next) n→ntype = LOCAL, n→vtype = vtype, twig = n;
    }
    return branch;
  }
```

88. A block starts with declaring local variables followed by statements.

```
⟨ Syntactic subroutines 49 ⟩ +≡
  node *ggblck( )
  {
    node *branch = Λ, *ltwig = Λ, *rtwig = Λ, *leaf = Λ;
    if (ggtoken→type ≡ LBRC) {
      lookahead( );
      ltwig = gglcl( );
      while ((ggtoken→type ≠ RBRC) ∧ (ggtoken→type ≠ PROCEDURE) ∧ (ggtoken→type ≠ EOS)) {
        leaf = ggstmnt( );
        if (rtwig) rtwig→next = leaf, rtwig = (leaf) ? leaf : rtwig;
        else branch = rtwig = leaf;
      }
      if (ggtoken→type ≡ RBRC) {
        lookahead( );
        branch = mknode(ltwig, Λ, branch, BLOCK);
      }
      else report(ggtoken→linenr, ggtoken→column, ERRSYN, ERRRBRC), recover( );
    }
    else report(ggtoken→linenr, ggtoken→column, ERRSYN, ERRLBRC), recover( );
    return branch;
  }
```

89. Procedures are built from a header followed by a block of statements.

$$\langle\,\text{procedure}\,\rangle \longrightarrow \langle\,\text{pheader}\,\rangle\langle\,\text{block}\,\rangle$$

```
⟨ Syntactic subroutines 49 ⟩ +≡
  node *ggproc( )
  {
    node *branch;
    branch = ggphdr( );
    if (branch) {
      branch→cr = ggblck( );
    }
    return branch;
  }
```

90. We almost there. Here is the routine for parsing declarations on the top level. These include external declarations, variable declarations of different sorts and procedure declarations.

```
⟨ Syntactic subroutines 49 ⟩ +≡
  node *ggtopdcl( )
  {
    node *branch = Λ;
    int type = 0;
    switch (type = ggtoken→type) {
    case GLOBAL: case DATA: case CONSTANT: case DEFINE: lookahead( );
      branch = ggidcls( );
      for (node *n = branch; n; n = n→next) n→ntype = type;
      ⟨ Check closing semicolon 53 ⟩
      return branch;
    case EXTERN: lookahead( );
      return ggextrn( );
    case PROCEDURE: lookahead( );
      return ggproc( );
    case SEMICOLON: lookahead( );
      break;
    default: report(ggtoken→linenr, ggtoken→column, ERRSYN, ERRSTMT), recover( );
      lookahead( );
    }
    return Λ;
  }
```

91.

```
⟨ Syntactic subroutines 49 ⟩ +≡
  node *ggparse( )
  {
    node *branch = Λ, *twig = Λ;
    lookahead( );
    branch = twig = mknode(Λ, Λ, Λ, ROOT);
    while (ggtoken→type ≠ EOS) {
      twig→next = ggtopdcl( );
      while (twig→next ≠ Λ) twig = twig→next;
    }
    return branch;
  }
```

92. Expressions.

```
⟨ Expression evaluation 92 ⟩ ≡
  vartype evaluate(node *n)
  {
    if (n ≡ Λ) return (vartype){0, 0};
    else if (n→ntype ≡ NUMBER) return n→var;
    else if (n→ntype ≡ FLTNUM) return n→var;
    else if (n→ntype ≡ IDENTIFIER) ⟨ Evaluate identifier 93 ⟩
    else if (n→ntype ≡ EXPRESSION) n→var = evaluate(n→cm);
    else if (n→ntype ≡ SPECIAL) report(n→lexeme→linenr, n→lexeme→column, ERRSEM, ERRINVR);
    else if (¬n→cl ∨ ¬n→cl) return (vartype){0, 0};
    else {
      vartype a = evaluate(n→cl);
      vartype b = evaluate(n→cr);
      if (¬a.type ∨ ¬b.type) return (vartype){0, 0};
      int t = n→ntype;
      if ((t ≡ AND) ∨ (t ≡ ANDN) ∨ (t ≡ OR) ∨ (t ≡ ORN) ∨ (t ≡ XOR) ∨ (t ≡ NOT))
        ⟨ Evaluate boolean expression 94 ⟩
      else if (n→ntype ≡ SQRT) ⟨ Evaluate SQRT 95 ⟩
      else if ((n→cl→var.type ≤ UOCTA) ∧ (n→cr→var.type ≤ UOCTA)) ⟨ Evaluate integer expression 96 ⟩
      else ⟨ Evaluate floating point expression 97 ⟩
    }
    return n→var;
  }
```

This code is used in section 12.

93. It is not possible to perform calculations on register numbers, any attempt to do so will result in an error being posted.

```
⟨ Evaluate identifier 93 ⟩ ≡
  {
    n→symbol = getsym(n→lexeme→name, false);
    if (n→symbol ∧ (n→symbol→usage ≡ REGISTER)) {
      report(n→lexeme→linenr, n→lexeme→column, ERRSEM, ERRINVR);
      return (vartype){0, 0};
    }
    else if (¬n→symbol) {
      report(n→lexeme→linenr, n→lexeme→column, ERRSEM, ERRNVAL);
      return (vartype){0, 0};
    }
    else n→var = n→symbol→var, n→hasvalue = true;
  }
```

This code is used in section 92.

94. ⟨ Evaluate boolean expression 94 ⟩ ≡

```
{
  if ((n→cl→var.type > UOCTA) ∧ (n→cr→var.type > UOCTA)) {
    report(n→lexeme→linenr, n→lexeme→column, ERRSEM, ERRTYPE);
    return (vartype) {0, 0};
  }
  switch (n→ntype) {
  case AND: n→var.value = n→cl→var.value & n→cr→var.value; break;
  case ANDN: n→var.value = n→cl→var.value & ∼n→cr→var.value; break;
  case OR: n→var.value = n→cl→var.value | n→cr→var.value; break;
  case ORN: n→var.value = n→cl→var.value | ∼n→cr→var.value; break;
  case XOR: n→var.value = n→cl→var.value ⊕ n→cr→var.value; break;
  case NOT: n→var.value = ∼n→cr→var.value; break;
  }
  n→var.type = n→cl→var.type > n→cr→var.type ? n→cl→var.type : n→cr→var.type;
}
```

This code is used in section 92.

95. ⟨ Evaluate SQRT 95 ⟩ ≡

```
{
  if (n→cr→var.type ≤ UOCTA) n→cr→var.value = floatit(n→cr→var.value, 0, 0, 0);
  n→var.value = froot(n→cr→var.value, n→cl→var.value);
  n→var.type = TFLOAT;
}
```

This code is used in section 92.

96. ⟨ Evaluate integer expression 96 ⟩ ≡

```
{
  switch (n→ntype) {
  case PLUS: n→var.value = n→cl→var.value + n→cr→var.value; break;
  case MINUS: n→var.value = n→cl→var.value − n→cr→var.value; break;
  case TIMES: n→var.value = n→cl→var.value ∗ n→cr→var.value; break;
  case DIV: n→var.value = signed_odiv(n→cl→var.value, n→cr→var.value); break;
  case REM: n→var.value = signed_odiv(n→cl→var.value, n→cr→var.value);
    n→var.value = aux; break;
  case SHR: n→var.value = n→cl→var.value ≫ n→cr→var.value; break;
  case SHL: n→var.value = n→cl→var.value ≪ n→cr→var.value; break;
  case SADD: n→var.value = count_bits(n→cl→var.value & ∼n→cr→var.value); break;
  case DIFF: n→var.value = octa_bdif(n→cl→var.value, n→cr→var.value); break;
  case EQ: n→var.value = (n→cl→var.value ≡ n→cr→var.value); break;
  case LT: n→var.value = (n→cl→var.value < n→cr→var.value); break;
  case GT: n→var.value = (n→cl→var.value > n→cr→var.value); break;
  case NE: n→var.value = (n→cl→var.value ≠ n→cr→var.value); break;
  case LE: n→var.value = (n→cl→var.value ≤ n→cr→var.value); break;
  case GE: n→var.value = (n→cl→var.value ≥ n→cr→var.value); break;
  case ODD: n→var.value = ((n→cl→var.value & 1) ≡ 1); break;
  case EVEN: n→var.value = ((n→cl→var.value & 1) ≡ 0); break;
  default: return (vartype) {0, 0};
  }
  n→var.type = n→cl→var.type > n→cr→var.type ? n→cl→var.type : n→cr→var.type;
}
```

This code is used in section 92.

97. ⟨ Evaluate floating point expression 97 ⟩ ≡

```
{
  if (n→cr→var.type ≤ UOCTA) n→cr→var.value = floatit(n→cr→var.value, 0, 0, 0);
  if (n→cl→var.type ≤ UOCTA) n→cl→var.value = floatit(n→cl→var.value, 0, 0, 0);
  int order = fcomp(n→cl→var.value, n→cr→var.value);
  switch (n→ntype) {
  case PLUS: n→var.value = fplus(n→cl→var.value, n→cr→var.value); break;
  case MINUS: n→var.value = fplus(n→cl→var.value, n→cr→var.value ⊕ #8000000000000000LL); break;
  case TIMES: n→var.value = fmult(n→cl→var.value, n→cr→var.value); break;
  case DIV: n→var.value = fdivide(n→cl→var.value, n→cr→var.value); break;
  case REM: n→var.value = fremstep(n→cl→var.value, n→cr→var.value, 9999); break;
  case EQ: n→var.value = (order ≡ 0); break;
  case LT: n→var.value = (order ≡ −1); break;
  case GT: n→var.value = (order ≡ 1); break;
  case NE: n→var.value = (order ≠ 0); break;
  case LE: n→var.value = (order < 1); break;
  case GE: n→var.value = ((order & 2) ≡ 0); break;
  default: return (vartype) {0, 0};
  }
  n→var.type = (n→ntype ≥ LT) ? UBYTE : TFLOAT;
}
```

This code is used in section 92.

98. The object file. An Elf file always has a header indicating the type of the Elf file and for what system it is intended. The compiler will build an object file with sections containing code, data and relocation information.

```
⟨ Initialise everything 16 ⟩ +≡
  ehdr = buildEhdr( );
  ehdr→e_type = ET_REL;
  ehdr→e_shentsize = sizeof (Elf64_Shdr);
  ehdr→e_shnum = 1;
  ehdr→e_shstrndx = 1;
  ⟨ Build the sections 99 ⟩
  plan(sct[STR], sizeof(char), false);
  plan(sct[SYM], sizeof (Elf64_Sym), false);
  if (debug) {
    plan(sct[STAB], sizeof (Elf64_Stab), false);
    plan(sct[STABSTR], strlen(srcname) * sizeof(char) + 1, false);
  }
```

99. ⟨ Build the sections 99 ⟩ ≡

```
  sct = allocate (SCTSNT, sizeof ( Elf64_Section * ) ) ;
  sct[UND] = buildSection("", 0, 0, 0, 0, 0);
  sct[STR] = buildSection(".strtab", SHT_STRTAB, 0, 0, 1, 1);
  sct[SYM] = buildSection(".symtab", SHT_SYMTAB, 0, 0, 2, 8);
  sct[TEXT] = buildSection(".text", SHT_PROGBITS, SHF_ALLOC + SHF_EXECINSTR, 0, 0, 4);
  sct[RTEXT] = buildSection(".rela.text", SHT_RELA, 0, 0, 0, 8);
  sct[RO] = buildSection(".rodata", SHT_PROGBITS, SHF_ALLOC, 0, 0, 8);
  sct[RRO] = buildSection(".rela.rodata", SHT_RELA, 0, 0, 0, 8);
  sct[RW] = buildSection(".data", SHT_PROGBITS, SHF_WRITE + SHF_ALLOC, 0, 0, 8);
  sct[RRW] = buildSection(".rela.data", SHT_RELA, 0, 0, 0, 8);
  sct[BSS] = buildSection(".bss", SHT_NOBITS, SHF_WRITE + SHF_ALLOC, 0, 0, 8);
  sct[STK] = buildSection(".mmix.stack", SHT_PROGBITS, SHF_WRITE + SHF_ALLOC, 0, 0, 8);
  sct[RSTK] = buildSection(".rela.mmix.stack", SHT_RELA, 0, 0, 0, 8);
  sct[REG] = buildSection(".mmix.register", SHT_PROGBITS, SHF_WRITE + SHF_ALLOC, 0, 0, 8);
  sct[RREG] = buildSection(".rela.mmix.register", SHT_RELA, 0, 0, 0, 8);
  sct[STAB] = buildSection(".stab", SHT_PROGBITS, 0, 0, 0, 8);
  sct[STABSTR] = buildSection(".stabstr", SHT_STRTAB, 0, 0, 0, 1);
```

This code is used in section 98.

100. ⟨ Type definitions 24 ⟩ +≡

```
  typedef enum {
    UND, STR, TEXT, RTEXT, RW, RRW, BSS, RO, RRO, STK, RSTK, REG, RREG, SYM, STAB, STABSTR, SCTSNT
  } scttype;
```

101. ⟨ Global variables 15 ⟩ +≡

```
  Elf64_Ehdr * ehdr;
  Elf64_Section * *sct;
```

102. ⟨ Allocate section data 102 ⟩ ≡

```
int index = 1;
for (int i = 1; i < SCTSNT; i++) {
  if (sct[i]→ptr > 0) {
    sct[i]→index = index++;
    sct[STR]→ptr += strlen(sct[i]→name) + 1;
    ehdr→e_shnum++;
  }
  else sct[i]→index = −1;
}
Elf64_Off offset = sizeof (Elf64_Ehdr);
for (int i = 1; i < SCTSNT; i++) {
  if (sct[i]→ptr > 0) {
    align(sct[i], sct[i]→shdr→sh_addralign);
    sct[i]→data = allocate(sct[i]→ptr, sizeof(unsigned char));
    sct[i]→shdr→sh_offset = offset;
    offset += sct[i]→shdr→sh_size = sct[i]→ptr;
    sct[i]→ptr = 0;
  }
}
ehdr→e_shoff = offset;
sct[STR]→ptr = 1;
sct[SYM]→ptr = sizeof (Elf64_Sym);
for (int i = 0; i < SCTSNT; i++) {
  if (sct[i]→index > 0) {
    sct[i]→shdr→sh_name = sct[STR]→ptr;
    setStr(sct[STR], sct[i]→name);
  }
}
if (debug) {
  sct[STABSTR]→ptr = 1;
  setStab(sct[STAB], sct[STABSTR]→ptr, N_UNDF, 0, 0, 0);
  setStr(sct[STABSTR], srcname);
}
```

This code is used in section 12.

103. ⟨ Wrap up everything 103 ⟩ ≡

```
if ((tgtfile = fopen(tgtname, "w")) ≡ Λ) panic("can't␣open␣object␣file");
sct[RTEXT]→shdr→sh_link = sct[SYM]→index;
sct[RTEXT]→shdr→sh_info = sct[TEXT]→index;
sct[RRW]→shdr→sh_link = sct[SYM]→index;
sct[RRW]→shdr→sh_info = sct[RW]→index;
sct[RRO]→shdr→sh_link = sct[SYM]→index;
sct[RRO]→shdr→sh_info = sct[RO]→index;
sct[RREG]→shdr→sh_link = sct[SYM]→index;
sct[RREG]→shdr→sh_info = sct[REG]→index;
sct[SYM]→shdr→sh_link = sct[STR]→index;
sct[SYM]→shdr→sh_info = 1;
writeEhdr(ehdr, tgtfile);
for (int i = 0; i < SCTSNT; i++) {
  if (sct[i]→index ≥ 0) {
    fwrite(sct[i]→data, sct[i]→shdr→sh_size, 1, tgtfile);
  }
}
for (int i = 0; i < SCTSNT; i++) {
  if (sct[i]→index ≥ 0) {
    writeShdr(sct[i]→shdr, tgtfile);
  }
}
fclose(tgtfile);
```

This code is used in section 12.

104. Semantics. Once the parse tree has been built, it is checked for internal consistency. Variables need to be declared before they are used, but they may not be declared twice. Syntactic rules are not capable of performing such checks. During the semantic check space is allocated for the sections in the object file and where possible the instructions are built.

```
⟨ Semantic subroutines 104 ⟩ ≡
  void resolve(node *n)
  {
    octa skip;
    node *lhs = Λ, *mhs = Λ, *rhs = Λ;
    for ( ; n; n = n→next) {
      lhs = n→cl; mhs = n→cm; rhs = n→cr;
      switch (n→ntype) {
        ⟨ Resolve cases 105 ⟩
      default: break;
      }
    }
  }
```

This code is used in section 12.

105. A global register must be initialised and it may not be declared twice.

```
⟨ Resolve cases 105 ⟩ ≡
case GLOBAL:
  if (rhs ≡ Λ) rhs = n→cr = &dummy0;
  if (rhs→next ≠ Λ) report(rhs→lexeme→linenr, rhs→lexeme→column, ERRSEM, ERRINIT);
  else {
    evaluate(rhs);
    if (lhs→ntype ≡ SPECIAL) {
      ⟨ resolve special register 106 ⟩
    }
    else if (getsym(lhs→lexeme→name, false))
      report(lhs→lexeme→linenr, lhs→lexeme→column, ERRSEM, ERRDUPL);
    else ⟨ resolve ordinary global 107 ⟩
  }
  break;
```

See also sections 110, 114, 117, 118, 120, 122, 123, 124, 128, 129, 130, 131, 132, 133, and 143.

This code is used in section 104.

106. Some special registers cannot be initialised.

```
⟨ resolve special register 106 ⟩ ≡
  {
    int r = lhs→symbol→var.value;
    if (rstk[r] < 0) report(lhs→lexeme→linenr, lhs→lexeme→column, ERRSEM, ERRINIT);
    sct[REG]→ptr = 13 * 8;
    if ((rhs→ntype ≡ IDENTIFIER) ∧ (rhs→symbol→usage ≡ ADDRESS))
      plan(sct[RREG], sizeof (Elf64_Rela), false);
  }
```

This code is used in section 105.

107. ⟨ resolve ordinary global 107 ⟩ ≡

```
{
  lhs→symbol = mksym(lhs→lexeme→name, REGISTER, glbsym, n→isextern);
  lhs→symbol→var.type = n→vtype;
  if (¬lhs→symbol→extsym) {
    if ((rhs→ntype ≡ IDENTIFIER) ∧ (rhs→symbol→usage ≡ ADDRESS)) {
      greg[gregcntr].g = lhs→symbol;
      greg[gregcntr].v = rhs→symbol;
      plan(sct[RSTK], sizeof (Elf64_Rela), false);
    }
    lhs→symbol→var.value = gregcntr−−;
    plan(sct[STK], 8, true);
  }
  plan(sct[SYM], sizeof (Elf64_Sym), false);
  plan(sct[STR], strlen(lhs→symbol→name) + 1, false);
  if (debug ∧ ¬lhs→symbol→extsym) plan(sct[STAB], sizeof (Elf64_Stab), false);
}
```

This code is used in section 105.

108. ⟨ Type definitions 24 ⟩ +≡

```
typedef struct {
  symtype *g;
  symtype *v;
} gtype;
```

109. ⟨ Global variables 15 ⟩ +≡

```
int rstk[] = {0, 1, 2, 3, 4, 5, 6, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, 12, −1, 7, 8, 9, 10, 11,
    −1, −1, −1, −1};
int gregcntr = 254;
gtype greg[256];
```

110. Variables may not be declared twice. Variables may be declared external which implies they are not initialised. Initialised variables will be allocated to the . *data* seiction and unitialised data will be allocated to the . *bss* section.

⟨ Resolve cases 105 ⟩ +≡

```
case DATA:
  if (getsym(lhs→lexeme→name, false)) report(lhs→lexeme→linenr, lhs→lexeme→column, ERRSEM, ERRDUPL);
  else {
    lhs→symbol = mksym(lhs→lexeme→name, ADDRESS, glbsym, n→isextern);
    lhs→symbol→var.type = n→vtype;
    if (lhs→symbol→extsym) lhs→symbol→var.value = 0;
    else if (rhs) ⟨ Resolve initialised data 111 ⟩
    else ⟨ Resolve uninitialised data 112 ⟩
    plan(sct[SYM], sizeof (Elf64_Sym), false);
    plan(sct[STR], strlen(lhs→symbol→name) + 1, false);
    if (debug) plan(sct[STAB], sizeof (Elf64_Stab), false);
  }
  break;
```

111. ⟨ Resolve initialised data 111 ⟩ ≡
```
{ align(sct[RW], size[lhs→symbol→var.type]);
lhs→symbol→var.value = sct[RW]→ptr; if (rhs→ntype ≡ STRING) ⟨ Resolve data string 113 ⟩
else {
  for (node *p = rhs; p; p = p→next) {
    evaluate(p);
    plan(sct[RW], size[lhs→symbol→var.type], true);
    if ((p→ntype ≡ IDENTIFIER) ∧ (p→symbol→usage ≡ ADDRESS)) {
      plan(sct[RRW], sizeof (Elf64_Rela), false);
    }
  }
}
}
```
This code is used in section 110.

112. ⟨ Resolve uninitialised data 112 ⟩ ≡
```
{
  align(sct[BSS], size[lhs→symbol→var.type]);
  lhs→symbol→var.value = sct[BSS]→ptr;
  plan(sct[BSS], size[lhs→symbol→var.type], true);
}
```
This code is used in section 110.

113. ⟨ Resolve data string 113 ⟩ ≡
```
{
  char *p = rhs→lexeme→name;
  if (lhs→symbol→var.type ≤ UBYTE) {
    plan(sct[RW], strlen(p) − 2, false);
  }
  else
    for (utfval(p), p = utfptr; *p ∧ (*p ≠ '"'); ) {
      utfval(p), p = utfptr;
      plan(sct[RW], size[lhs→symbol→var.type], true);
    }
}
```
This code is used in section 111.

114. ⟨ Resolve cases 105 ⟩ +≡
```
case CONSTANT:
  if (rhs ≡ Λ) rhs = n→cr = &dummy0;
  if (getsym(lhs→lexeme→name, false)) report(lhs→lexeme→linenr, lhs→lexeme→column, ERRSEM, ERRDUPL);
  else {
    lhs→symbol = mksym(lhs→lexeme→name, ADDRESS, glbsym, n→isextern);
    lhs→symbol→var.type = n→vtype;
    if (lhs→symbol→extsym) lhs→symbol→var.value = 0;
    else ⟨ Resolve constant data 115 ⟩
    plan(sct[SYM], sizeof (Elf64_Sym), false);
    plan(sct[STR], strlen(lhs→symbol→name) + 1, false);
    if (debug) plan(sct[STAB], sizeof (Elf64_Stab), false);
  }
  break;
```

115. ⟨ Resolve constant data 115 ⟩ ≡

```
{ align(sct[RO], size[lhs→symbol→var.type]);
lhs→symbol→var.value = sct[RO]→ptr; if (rhs→ntype ≡ STRING) ⟨ Resolve constant string 116 ⟩
else {
  for (node *p = rhs; p; p = p→next) {
    evaluate(p);
    plan(sct[RO], size[lhs→symbol→var.type], true);
    if ((p→ntype ≡ IDENTIFIER) ∧ (p→symbol→usage ≡ ADDRESS)) {
      plan(sct[RRO], sizeof (Elf64_Rela), false);
    }
  }
}
}
```

This code is used in section 114.

116. ⟨ Resolve constant string 116 ⟩ ≡

```
{
  char *p = rhs→lexeme→name;
  if (lhs→symbol→var.type ≡ UBYTE) {
    plan(sct[RO], strlen(p), false);
  }
  else
    for (utfval(p), p = utfptr; *p ∧ (*p ≠ '"'); ) {
      utfval(p), p = utfptr;
      plan(sct[RO], size[lhs→symbol→var.type], true);
    }
}
```

This code is used in section 115.

117. Definitions are used as variables during the translation process, they may not be declared twice and must be initialised with a single value.

⟨ Resolve cases 105 ⟩ +≡

```
case DEFINE:
  if (getsym(lhs→lexeme→name, false)) report(lhs→lexeme→linenr, lhs→lexeme→column, ERRSEM, ERRDUPL);
  else if (rhs ≡ Λ) report(lhs→lexeme→linenr, lhs→lexeme→column, ERRSEM, ERRNVAL);
  else if (rhs→next ≠ Λ) report(rhs→lexeme→linenr, rhs→lexeme→column, ERRSEM, ERRINIT);
  else {
    lhs→symbol = mksym(lhs→lexeme→name, NUMBER, glbsym, false);
    lhs→symbol→var = evaluate(rhs);
  }
  break;
```

118. ⟨ Resolve cases 105 ⟩ +≡
```
case PROCEDURE:
  if (getsym(lhs→lexeme→name, false)) report(lhs→lexeme→linenr, lhs→lexeme→column, ERRSEM, ERRDUPL);
  lhs→symbol = mksym(lhs→lexeme→name, PROCEDURE, glbsym, n→isextern);
  lregcntr = 0;
  if (¬lhs→symbol→extsym) {
    lhs→symbol→var.value = sct[TEXT]→ptr;
    ⟨ Make a local symbol table 119 ⟩
    n→cr→symtab = lclsym;
    resolve(n→cm);
    resolve(n→cr);
    lclsym = glbsym;
  }
  plan(sct[SYM], sizeof (Elf64_Sym), false);
  plan(sct[STR], strlen(lhs→symbol→name) + 1, false);
  if (debug) {
    plan(sct[STAB], sizeof (Elf64_Stab), false);
    plan(sct[STABSTR], strlen(lhs→symbol→name) * sizeof(char) + 1, false);
  }
  break;
```

119. ⟨ Make a local symbol table 119 ⟩ ≡
```
  symtabtype *parent = lclsym;
  lclsym = (symtabtype *) allocate(1, sizeof(symtabtype));
  lclsym→parent = parent;
  lclsym→size = lclsymtabsize;
  lclsym→sym = (symtype **) allocate(lclsymtabsize, sizeof(symtype *));
```
This code is used in sections 118 and 123.

120. ⟨ Resolve cases 105 ⟩ +≡
```
case PARAMETER: resolve(n→cl);
  bigL = lregcntr, lregcntr = 0;
  resolve(n→cr);
  npop = lregcntr;
  if (bigL > lregcntr) lregcntr = bigL;
  break;
```

121. ⟨ Global variables 15 ⟩ +≡
```
  int lregcntr = 0;
  int npop, bigL, regcntr;
```

122. ⟨ Resolve cases 105 ⟩ +≡
```
case LOCAL:
  if (getsym(lhs→lexeme→name, true)) report(lhs→lexeme→linenr, lhs→lexeme→column, ERRSEM, ERRDUPL);
  else {
    n→symbol = mksym(lhs→lexeme→name, REGISTER, lclsym, false);
    n→symbol→var.value = lregcntr++;
    n→symbol→var.type = n→vtype;
  }
  break;
```

123.

```
#define isregister(n) ((n→ntype ≡ IDENTIFIER) ∧ n→symbol ∧ (n→symbol→usage ≡ REGISTER))
#define isnumber(n) ((n→ntype ≡ NUMBER) ∨ (n→ntype ≡ EXPRESSION) ∨ ((n→ntype ≡
        IDENTIFIER) ∧ (n→symbol ∧ n→symbol→usage ≡ NUMBER)))
#define isspecial(n) (n→ntype ≡ SPECIAL)
#define isaddress(n) ((n→ntype ≡ IDENTIFIER) ∧ (n→symbol→usage ≡ ADDRESS))
#define islabel(n) ((n→ntype ≡ IDENTIFIER) ∧ (n→symbol→usage ≡ LABEL))
#define isfloat(n) (n→ntype ≡ FLTNUM)
⟨ Resolve cases 105 ⟩ +≡
case BLOCK:
  {
    int savelregcntr = lregcntr;
    if (n→symtab) lclsym = n→symtab;
    else {
      ⟨ Make a local symbol table 119 ⟩
      n→symtab = lclsym;
    }
    resolve(n→cl);
    resolve(n→cr);
    lclsym = lclsym→parent;
    lregcntr = savelregcntr;
  }
  break;
```

124. ⟨ Resolve cases 105 ⟩ +≡

```
case IF: case WHILE: case UNTIL:
  while (lhs→ntype ≡ EXPRESSION) lhs = lhs→cm;
  if (lhs→cl) {
    if ((lhs→ntype ≥ LT) ∧ (lhs→ntype ≤ EVEN)) {
      node *ll = lhs→cl, *rl = lhs→cr;
      if (ll→ntype ≡ IDENTIFIER) ll→symbol = getsym(ll→lexeme→name, false);
      if (isregister(ll)) {
        if (isnumber(rl)) {
          int op, x, yz;
          op = #40 + ((lhs→ntype − LT) * 2) ⊕ 8;
          x = ll→symbol→var.value;
          if ((x > gregcntr) ∧ (x < 255)) plan(sct[RTEXT], sizeof (Elf64_Rela), false), x = 0;
          if (n→ntype ≡ IF) ⟨ Check IF jumps 125 ⟩
          else if (n→ntype ≡ WHILE) ⟨ Check WHILE jumps 126 ⟩
          else if (n→ntype ≡ UNTIL) ⟨ Check UNTIL jumps 127 ⟩
        }
        else report(rl→lexeme→linenr, rl→lexeme→column, ERRSEM, ERREVEN);
      }
      else report(ll→lexeme→linenr, ll→lexeme→column, ERRSEM, ERRRGRQ);
    }
    else report(n→cl→lexeme→linenr, n→cl→lexeme→column, ERRSEM, ERRCOMP);
  }
  break;
```

125. ⟨ Check IF jumps 125 ⟩ ≡

```
{
  plan(sct[TEXT], 4, true);
  int s = sct[TEXT]→ptr;
  resolve(mhs);
  if (rhs) plan(sct[TEXT], 4, true);
  yz = (sct[TEXT]→ptr − s)/4;
  n→inst1 = (op ≪ 24) + (x ≪ 16) + yz;
  if (debug) plan(sct[STAB], sizeof (Elf64_Stab), false);
  if (rhs) {
    resolve(rhs);
    yz = (sct[TEXT]→ptr − s)/4 − yz;
    n→inst2 = #f0000000 + yz;
  }
}
```

This code is used in section 124.

126. ⟨ Check WHILE jumps 126 ⟩ ≡

```
{
  plan(sct[TEXT], 4, true);
  int s = sct[TEXT]→ptr;
  resolve(mhs);
  plan(sct[TEXT], 4, true);
  yz = (sct[TEXT]→ptr − s)/4;
  n→inst1 = (op ≪ 24) + (x ≪ 16) + yz;
  if (debug) plan(sct[STAB], sizeof (Elf64_Stab), false);
  n→inst2 = ((op ⊕ 24) ≪ 24) + (x ≪ 16) + yz;
}
```

This code is used in section 124.

127. ⟨ Check UNTIL jumps 127 ⟩ ≡

```
{
  int s = sct[TEXT]→ptr;
  resolve(mhs);
  plan(sct[TEXT], 4, true);
  yz = (sct[TEXT]→ptr − s)/4;
  n→inst1 = ((op ⊕ 24) ≪ 24) + (x ≪ 16) + yz;
  if (debug) plan(sct[STAB], sizeof (Elf64_Stab), false);
}
```

This code is used in section 124.

128. ⟨ Resolve cases 105 ⟩ +≡

```
case GOTO:
  if (lhs→ntype ≡ IDENTIFIER) lhs→symbol = getsym(lhs→lexeme→name, false);
  if (lhs→symbol ∧ islabel(lhs)) n→inst1 = #f1000000 + ((sct[TEXT]→ptr + 4 − lhs→symbol→var.value) ≫ 2);
  else { n→inst1 = #f0000000; fixtype ∗ fixup = ( fixtype ∗ ) allocate(1, sizeof (fixtype));
  fixup→next = ggfix, fixup→loc = sct[TEXT]→ptr, fixup→ins = n;
  ggfix = fixup; }
  if (debug) plan(sct[STAB], sizeof (Elf64_Stab), false);
  plan(sct[TEXT], 4, true);
  break;
```

129. ⟨ Resolve cases 105 ⟩ +≡

```
case CALL: regcntr = 1;
  if (n→cm) resolve(n→cm→cl);
  if (lhs→ntype ≡ IDENTIFIER) lhs→symbol = getsym(lhs→lexeme→name, false);
  if (debug) plan(sct[STAB], sizeof (Elf64_Stab), false);
  plan(sct[TEXT], 4, true);
  if (lhs→symbol→extsym) {
    n→inst1 = #f2000000 + (lregcntr ≪ 16);
    plan(sct[RTEXT], sizeof (Elf64_Rela), false);
  }
  else if ((lhs→ntype ≡ IDENTIFIER) ∧ (lhs→symbol→usage ≡ PROCEDURE)) {
    n→inst1 = #f3000000 + (lregcntr ≪ 16) + (((lhs→symbol→var.value − sct[TEXT]→ptr) ≫ 2) & #ffff);
  }
  else report(lhs→lexeme→linenr, lhs→lexeme→column, ERRSEM, ERRPROC);
  regcntr = 1;
  if (n→cm) resolve(n→cm→cr);
  break;
case IN: lhs→symbol = getsym(lhs→lexeme→name, false);
  if (isregister(lhs)) {
    n→inst1 = #c1000000 + ((lregcntr + regcntr++) ≪ 16) + (lhs→symbol→var.value ≪ 8);
    if (debug) plan(sct[STAB], sizeof (Elf64_Stab), false);
    plan(sct[TEXT], 4, true);
  }
  else report(lhs→lexeme→linenr, lhs→lexeme→column, ERRSEM, ERRRGRQ);
  break;
case OUT: lhs→symbol = getsym(lhs→lexeme→name, false);
  if (isregister(lhs)) {
    n→inst1 = #c1000000 + (lhs→symbol→var.value ≪ 16) + ((lregcntr + regcntr++) ≪ 8);
    if (debug) plan(sct[STAB], sizeof (Elf64_Stab), false);
    plan(sct[TEXT], 4, true);
  }
  else report(lhs→lexeme→linenr, lhs→lexeme→column, ERRSEM, ERRRGRQ);
  break;
case RETURN: skip = evaluate(lhs).value & #ffff;
  n→inst1 = #f8000000 + (npop ≪ 16) + skip;
  if (debug) plan(sct[STAB], sizeof (Elf64_Stab), false);
  plan(sct[TEXT], 4, true);
  break;
```

130. ⟨ Resolve cases 105 ⟩ +≡

```
case LABEL:
  if (lhs→symbol) report(lhs→lexeme→linenr, lhs→lexeme→column, ERRSEM, ERRDUPL);
  else {
    lhs→symbol = mksym(lhs→lexeme→name, LABEL, glbsym, false);
    lhs→symbol→var.value = sct[TEXT]→ptr;
  }
  break;
```

131. ⟨ Resolve cases 105 ⟩ +≡

```
case SAVE: case UNSAVE: lhs→symbol = getsym(lhs→lexeme→name, false);
  if (¬isregister(lhs)) report(lhs→lexeme→linenr, lhs→lexeme→column, ERRSEM, ERRRGRQ);
  else if (n→ntype ≡ SAVE) {
    n→inst1 = #fa000000 + (lhs→symbol→var.value ≪ 16);
  }
  else {
    n→inst1 = #fb000000 + lhs→symbol→var.value;
  }
  plan(sct[TEXT], 4, true);
  if (debug) plan(sct[STAB], sizeof (Elf64_Stab), false);
  break;
```

132. ⟨ Resolve cases 105 ⟩ +≡

```
case LOAD: case STORE: case SWAP:
  {
    int op = -1, x = 0, y = 0, z = 0, yz = 0;
    bool useyz = false;
    node *rela = mknode(Λ, Λ, Λ, RELOCATION);
    if (lhs→ntype ≡ IDENTIFIER) lhs→symbol = getsym(lhs→lexeme→name, false);
    if (isspecial(lhs)) report(lhs→lexeme→linenr, lhs→lexeme→column, ERRSEM, ERRRGRQ);
    else if (isnumber(lhs)) {
      if (n→ntype ≡ STORE) op = #b4;
      else report(lhs→lexeme→linenr, lhs→lexeme→column, ERRSEM, ERRRGRQ);
    }
    else if (isregister(lhs)) op = opc1[(n→ntype − PLUS) * 10 + lhs→symbol→var.type − 1];
    else report(lhs→lexeme→linenr, lhs→lexeme→column, ERRSEM, ERRRGRQ);
    if (rhs→ntype ≡ PLUS) {
      if (rhs→cl→ntype ≡ IDENTIFIER) rhs→cl→symbol = getsym(rhs→cl→lexeme→name, false);
      if (rhs→cr→ntype ≡ IDENTIFIER) rhs→cr→symbol = getsym(rhs→cr→lexeme→name, false);
      else evaluate(rhs→cr);
    }
    else if (rhs→ntype ≡ IDENTIFIER) {
      rhs→symbol = getsym(rhs→lexeme→name, false);
      if (isaddress(rhs)) {
        for (int i = 254; i > gregcntr; i--) {
          if ((greg[i].g) ∧ ((rhs→symbol→var.value − greg[i].v→var.value) < 256)) {
            node *n1 = mknode(Λ, Λ, Λ, IDENTIFIER);
            node *n2 = mknode(Λ, Λ, Λ, NUMBER);
            n1→symbol = greg[i].g;
            n1→var = (vartype){i, UBYTE};
            n2→var = (vartype){rhs→symbol→var.value − greg[i].v→var.value, UBYTE};
            n→cr = mknode(n1, rhs, n2, PLUS);
            break;
          }
        }
      }
      else n→cr = mknode(n→cr, Λ, &dummy0, PLUS);
    }
    else op = -1, report(n→lexeme→linenr, n→lexeme→column + 2, ERRSEM, ERRADDR);
    if (op ≥ 0) {
      ⟨ Handle x,y and z 141 ⟩
      ⟨ Build the instruction 142 ⟩
    }
  }
  break;
```

133. ⟨ Resolve cases 105 ⟩ +≡

```
case ASSIGN:
  {
    int op = -1, x = 0, y = 0, z = 0, yz = 0;
    bool useyz = false;
    node *rela = mknode(Λ, Λ, Λ, RELOCATION);
    if (lhs→ntype ≡ IDENTIFIER) lhs→symbol = getsym(lhs→lexeme→name, false);
    if (rhs→ntype ≡ IDENTIFIER) rhs→symbol = getsym(rhs→lexeme→name, false);
    ⟨ Handle PUT and GET 134 ⟩
    else
      if (¬isregister(lhs)) report(lhs→lexeme→linenr, lhs→lexeme→column, ERRSEM, ERRINVR);
    ⟨ Handle straight assignment 135 ⟩
    ⟨ Handle conditionals 136 ⟩
    ⟨ Handle ADDUs 137 ⟩
    ⟨ Handle other instructions 138 ⟩
    if (op ≥ 0) {
      ⟨ Handle x,y and z 141 ⟩
      ⟨ Build the instruction 142 ⟩
    }
  }
  break;
```

134. `PUT` and `GET` transfer data between a special register and a global or local register, it is also possible to put an immediate value into a special register.

⟨ Handle PUT and GET 134 ⟩ ≡

```
if (isspecial(lhs)) {
  if (isspecial(rhs) ∨ isfloat(rhs)) report(rhs→lexeme→linenr, rhs→lexeme→column, ERRSEM, ERRINVR);
  else n→cr = rhs = mknode(&dummy0, Λ, rhs, PUT), op = #f6;
}
else if (isspecial(rhs)) {
  if (¬isregister(lhs)) report(lhs→lexeme→linenr, lhs→lexeme→column, ERRSEM, ERRRGRQ);
  else n→cr = rhs = mknode(&dummy0, Λ, rhs, GET), op = #fe;
}
```

This code is used in section 133.

135. ⟨ Handle straight assignment 135 ⟩ ≡

```
else
  if (isregister(rhs)) {
    if ((lhs→symbol→var.type ≤ UOCTA) ∧ (rhs→symbol→var.type > UOCTA))
      n→cr = rhs = mknode(&dummy0, Λ, rhs, PLUS), op = #07 − 2 * (lhs→symbol→var.type & 1);
    else if ((lhs→symbol→var.type > UOCTA) ∧ (rhs→symbol→var.type ≤ UOCTA)) {
      op = #0a + 4 * (lhs→symbol→var.type − TFLOAT) − 2 * (rhs→symbol→var.type & 1);
      n→cr = rhs = mknode(&dummy0, Λ, rhs, PLUS);
    }
    else n→cr = rhs = mknode(rhs, Λ, &dummy0, OR), op = #c0;
  }
```

This code is used in section 133.

136. ⟨ Handle conditionals 136 ⟩ ≡

```
else
  if (rhs→ntype ≡ COLON) {
    node *rl = rhs→cl;
    while (rl→ntype ≡ EXPRESSION) rl = rl→cm;
    if ((rl→ntype < LT ∨ rl→ntype > EVEN) ∨ (rl→cr→ntype ≠ NUMBER))
      report(rl→lexeme→linenr, rl→lexeme→column, ERRSEM, ERRCOMP);
    else if ((rl→ntype ≠ ODD) ∧ (rl→cr→var.value ≠ 0))
      report(rl→cr→lexeme→linenr, rl→cr→lexeme→column, ERRSEM, ERRZERO);
    else {
      if (rhs→cm→ntype ≡ IDENTIFIER) rhs→cm→symbol = getsym(rhs→cm→lexeme→name, false);
      if (rhs→cr→ntype ≡ IDENTIFIER) rhs→cr→symbol = getsym(rhs→cr→lexeme→name, false);
      if ((rhs→cm→ntype ≡ IDENTIFIER) ∧ (rhs→cm→symbol ≡ lhs→symbol)) op = #60;
      else if ((rhs→cm→ntype ≡ NUMBER) ∧ (rhs→cm→var.value ≡ 0)) op = #70;
      else report(rhs→cm→lexeme→linenr, rhs→cm→lexeme→column, ERRSEM, ERRZSYM);
      if (op ≥ 0) op += (rl→ntype − LT) * 2;
      rhs→cl = rl→cl;
      rhs→cl→symbol = getsym(rhs→cl→lexeme→name, false);
    }
  }
```

This code is used in section 133.

137. ⟨ Handle ADDUs 137 ⟩ ≡

```
else
  if ((rhs→ntype ≡ PLUS) ∧ (rhs→cl→ntype ≡ TIMES)) {
    node *ll, *lr;
    if (rhs→cr→ntype ≡ IDENTIFIER) rhs→cr→symbol = getsym(rhs→cr→lexeme→name, false);
    if (rhs→cl→cl→ntype ≡ NUMBER) ll = rhs→cl→cr, lr = rhs→cl→cl;
    else ll = rhs→cl→cl, lr = rhs→cl→cr;
    if (ll→ntype ≡ IDENTIFIER) ll→symbol = getsym(ll→lexeme→name, false);
    if (¬isregister(ll)) report(ll→lexeme→linenr, ll→lexeme→column, ERRSEM, ERRRGRQ);
    if ((lr→ntype ≠ NUMBER) ∨ (count_bits(lr→var.value) ≠ 1) ∨ (lr→var.value > 16))
      report(lr→lexeme→linenr, lr→lexeme→column, ERRSEM, ERRADDU);
    op = #26 + count_bits(lr→var.value − 1) * 2;
    rhs→cl = ll;
  }
```

This code is used in section 133.

138. ⟨ Handle other instructions 138 ⟩ ≡

```
else
  if ((rhs→ntype ≥ PLUS) ∧ (rhs→ntype ≤ SQRT)) {
    op = opc1[(rhs→ntype − PLUS) * 10 + lhs→symbol→var.type − 1];
    if (rhs→cl→ntype ≡ IDENTIFIER) rhs→cl→symbol = getsym(rhs→cl→lexeme→name, false);
    if (rhs→cr→ntype ≡ IDENTIFIER) rhs→cr→symbol = getsym(rhs→cr→lexeme→name, false);
    ⟨ Handle NEG 139 ⟩
    ⟨ Handle yz immediate 140 ⟩
  }
  else op = opc1[159 + lhs→symbol→var.type], yz = evaluate(rhs).value, useyz = true;
```

This code is used in section 133.

139. ⟨ Handle NEG 139 ⟩ ≡

```
if ((rhs→ntype ≡ MINUS) ∧ isnumber(rhs→cl)) {
  if (evaluate(rhs→cl).value > 255) yz = evaluate(rhs).value, useyz = true, op = #e3;
  else op = #36 − (lhs→symbol→var.type & 1) ∗ 2;
}
```

This code is used in section 138.

140. ⟨ Handle yz immediate 140 ⟩ ≡

```
else
  if ((rhs→ntype ≡ PLUS) ∨ (rhs→ntype ≡ OR) ∨ (rhs→ntype ≡ SHL) ∨ (rhs→ntype ≡ ANDN)) {
    node ∗a, ∗b, ∗c;
    octa n;
    int newop = −1;
    if ((rhs→ntype ≡ PLUS) ∨ (rhs→ntype ≡ OR)) {
      if (rhs→cr→ntype ≡ SHL) a = rhs→cl, b = rhs→cr→cl, c = rhs→cr→cr;
      else if (rhs→cl→symbol ≡ lhs→symbol) a = rhs→cl, b = rhs→cr, c = &dummy0;
      if (isnumber(b)) newop = #e3 + (rhs→ntype ≡ PLUS ? 4 : 8);
    }
    else if (rhs→ntype ≡ SHL) {
      if (rhs→cl→ntype ≡ ANDN) {
        a = rhs→cl→cl, b = rhs→cl→cr, c = rhs→cr;
        newop = #ef;
      }
      else if (isnumber(rhs→cl)) {
        a = lhs, b = rhs→cl, c = rhs→cr;
        newop = #e3;
      }
    }
    else if (rhs→ntype ≡ ANDN) {
      if (rhs→cl→symbol ≡ lhs→symbol) {
        a = rhs→cl, b = rhs→cr, c = &dummy0;
        newop = #ef;
      }
    }
    if ((newop > 0) ∧ isnumber(b)) {
      useyz = true;
      n = evaluate(c).value;
      if (n & ∼48) report(c→lexeme→linenr, c→lexeme→column, ERRSEM, ERRHLNM);
      op = newop − (n ≫ 4);
      yz = evaluate(b).value;
    }
  }
```

This code is used in section 138.

141. ⟨ Handle x,y and z 141 ⟩ ≡

```
node *lr = n→cr→cl, *rr = n→cr→cr;
if (isregister(lhs)) {
  x = lhs→symbol→var.value;
  if ((x ≥ gregcntr) ∧ (x < 255)) rela→cl = lhs;
}
else if (isspecial(lhs)) x = lhs→symbol→var.value;
else x = evaluate(lhs).value;
if (useyz) y = (yz ≫ 8) & #ff, z = yz & #ff;
else {
  if (isregister(lr)) {
    y = lr→symbol→var.value;
    if ((y ≥ gregcntr) ∧ (y < 255)) rela→cm = lr;
  }
  else y = evaluate(lr).value;
  if (isregister(rr)) {
    z = rr→symbol→var.value;
    if ((z > gregcntr) ∧ (z < 255)) rela→cr = rr;
  }
  else if (isspecial(rr)) z = rr→symbol→var.value;
  else z = evaluate(rr).value, op++;
}
```

This code is used in sections 132 and 133.

142. ⟨ Build the instruction 142 ⟩ ≡

```
plan(sct[TEXT], 4, true);
if (rela→cl ∨ rela→cm ∨ rela→cr) {
  if (rela→cl) plan(sct[RTEXT], sizeof (Elf64_Rela), false), x = 0;
  if (rela→cm) plan(sct[RTEXT], sizeof (Elf64_Rela), false), y = 0;
  if (rela→cr) plan(sct[RTEXT], sizeof (Elf64_Rela), false), z = 0;
  n→rel = rela;
}
else free(rela);
if (debug) plan(sct[STAB], sizeof (Elf64_Stab), false);
n→inst1 = (op ≪ 24) + (x ≪ 16) + (y ≪ 8) + z;
```

This code is used in sections 132, 133, and 143.

143. ⟨ Resolve cases 105 ⟩ +≡

```
case OPCODE:
  {
    int op = −1, x = 0, y = 0, z = 0, yz = 0;
    node *rela = mknode(Λ, Λ, Λ, RELOCATION);
    op = n→symbol→var.value ≫ 24;
    ⟨ Check X 144 ⟩
    ⟨ Check Y 149 ⟩
    ⟨ Check Z 155 ⟩
    ⟨ Build the instruction 142 ⟩
  }
  break;
```

144. ⟨ Check X 144 ⟩ ≡

```
{
  if (¬lhs) {     /* TODO XYZO */
  }
  else {
    ⟨ Check for X as a register 145 ⟩
    ⟨ Check for X as immediate 146 ⟩
    ⟨ Check for X as optional 147 ⟩
    ⟨ Check for X as relative 148 ⟩
  }
}
```

This code is used in section 143.

145. ⟨ Check for X as a register 145 ⟩ ≡

```
if (lhs→ntype ≡ IDENTIFIER) lhs→symbol = getsym(lhs→lexeme→name, false);
if (n→symbol→var.value & XR) {
  if (¬isregister(lhs)) report(lhs→lexeme→linenr, lhs→lexeme→column, ERRSEM, ERRRGRQ);
  else {
    x = lhs→symbol→var.value;
    if ((x ≥ gregcntr) ∧ (x < 255)) rela→cl = lhs;
  }
}
else if (n→symbol→var.value & XS) {
  if (¬isspecial(lhs)) report(lhs→lexeme→linenr, lhs→lexeme→column, ERRSEM, ERRSRRQ);
  else x = lhs→symbol→var.value;
}
```

This code is used in section 144.

146. ⟨ Check for X as immediate 146 ⟩ ≡

```
else
  if (n→symbol→var.value & (XI + XYZI)) {
    if (¬isnumber(lhs)) report(lhs→lexeme→linenr, lhs→lexeme→column, ERRSEM, ERRNMRQ);
    else {
      if (lhs) x = evaluate(lhs).value;
      if ((n→symbol→var.value & XYZI) ∧ ¬mhs) z = x & #ff, y = (x & #ff00) ≫ 8, x = (x ≫ 16) & #ff;
    }
  }
```

This code is used in section 144.

147. ⟨ Check for X as optional 147 ⟩ ≡

```
else
  if (n→symbol→var.value & XO) {
    rhs = mhs, mhs = lhs, lhs = Λ, x = 0;
  }
```

This code is used in section 144.

148. ⟨ Check for X as relative 148 ⟩ ≡
```
else if (n→symbol→var.value & XYZR) {
if (lhs→symbol ∧ islabel(lhs)) {
  x = (sct[TEXT]→ptr + 4 − lhs→symbol→var.value) ≫ 2;
  z = x & #ff, y = (x ≫ 8) & #ff, x = (x ≫ 16) & #ff, op++;
}
else if ((lhs→ntype ≡ IDENTIFIER) ∧ ¬lhs→symbol) { fixtype * fixup = ( fixtype * ) allocate(1, sizeof
    (fixtype));
fixup→next = ggfix, fixup→loc = sct[TEXT]→ptr, fixup→ins = n;
ggfix = fixup; }
else report(lhs→lexeme→linenr, lhs→lexeme→column, ERRSEM, ERRLBRQ);
}
```
This code is used in section 144.

149. ⟨ Check Y 149 ⟩ ≡
```
{
  if (mhs) {
    ⟨ Check for Y as a register 150 ⟩
    ⟨ Check for Y as immediate 151 ⟩
    ⟨ Check for Y as optional 152 ⟩
    ⟨ Check for Y as relative 153 ⟩
    ⟨ Check for Y as memory 154 ⟩
  }
}
```
This code is used in section 143.

150. ⟨ Check for Y as a register 150 ⟩ ≡
```
if (mhs→ntype ≡ IDENTIFIER) mhs→symbol = getsym(mhs→lexeme→name, false);
if (n→symbol→var.value & YR) {
  if (¬isregister(mhs)) report(mhs→lexeme→linenr, mhs→lexeme→column, ERRSEM, ERRRGRQ);
  else {
    y = mhs→symbol→var.value;
    if ((y ≥ gregcntr) ∧ (y < 255)) rela→cm = mhs;
  }
}
```
This code is used in section 149.

151. ⟨ Check for Y as immediate 151 ⟩ ≡
```
else
  if (n→symbol→var.value & (YI + YZI + XYZI)) {
    if (¬isnumber(mhs)) report(mhs→lexeme→linenr, mhs→lexeme→column, ERRSEM, ERRNMRQ);
    else {
      if (mhs) y = evaluate(mhs).value;
      if ((n→symbol→var.value & (YZI + XYZI)) ∧ ¬rhs) z = y & #ff, y = (y & #ff00) ≫ 8;
    }
  }
```
This code is used in section 149.

152. ⟨ Check for Y as optional 152 ⟩ ≡

```
else
  if (n→symbol→var.value & YO) {
    if (¬rhs ∨ op > #17) rhs = mhs, mhs = Λ;
    else {
      if (¬isnumber(mhs)) report(mhs→lexeme→linenr, mhs→lexeme→column, ERRSEM, ERRNMRQ);
      else y = evaluate(mhs).value;
    }
  }
```

This code is used in section 149.

153. ⟨ Check for Y as relative 153 ⟩ ≡

```
else if (n→symbol→var.value & YZR) {
if (mhs→symbol ∧ islabel(mhs)) {
  y = (sct[TEXT]→ptr + 4 − mhs→symbol→var.value) ≫ 2;
  z = y & #ff, y = (y ≫ 8) & #ff, op++;
}
else if ((mhs→ntype ≡ IDENTIFIER) ∧ (¬mhs→symbol)) { fixtype * fixup = ( fixtype * ) allocate(1, sizeof
    (fixtype));
fixup→next = ggfix, fixup→loc = sct[TEXT]→ptr, fixup→ins = n;
ggfix = fixup; }
else report(mhs→lexeme→linenr, mhs→lexeme→column, ERRSEM, ERRLBRQ);
if (rhs) report(rhs→lexeme→linenr, rhs→lexeme→column, ERRSEM, ERRNPAR);
}
```

This code is used in section 149.

154. ⟨ Check for Y as memory 154 ⟩ ≡

```
else
  if (n→symbol→var.value & YZM) {
    if (mhs→symbol ∧ isaddress(mhs)) {
      for (int i = 254; i > gregcntr; i--) {
        if ((greg[i].g) ∧ ((mhs→symbol→var.value − greg[i].v→var.value) < 256)) {
          y = greg[i].v→var.value;
          z = mhs→symbol→var.value − greg[i].v→var.value;
          if ((y ≥ gregcntr) ∧ (y < 255)) rela→cm = mhs;
          break;
        }
      }
    }
    else if (isregister(mhs)) {
      y = mhs→symbol→var.value;
      if ((y ≥ gregcntr) ∧ (y < 255)) rela→cm = mhs;
      if (rhs→ntype ≡ IDENTIFIER) rhs→symbol = getsym(rhs→lexeme→name, false);
      if (rhs→symbol ∧ isregister(rhs)) {
        z = rhs→symbol→var.value;
        if ((z ≥ gregcntr) ∧ (z < 255)) rela→cr = rhs;
      }
      else if (isnumber(rhs)) {
        z = evaluate(rhs).value, op++;
      }
      else report(rhs→lexeme→linenr, rhs→lexeme→column, ERRSEM, ERRNMRQ);
    }
    else report(mhs→lexeme→linenr, mhs→lexeme→column, ERRSEM, ERRRGRQ);
  }
```

This code is used in section 149.

155. ⟨ Check Z 155 ⟩ ≡

```
{
  if (rhs) {
    ⟨ Check for Z as a register 156 ⟩
    ⟨ Check for Z as immediate 157 ⟩
    ⟨ Check for Z as register/immediate 158 ⟩
  }
}
```

This code is used in section 143.

156. ⟨ Check for Z as a register 156 ⟩ ≡
```
if (rhs→ntype ≡ IDENTIFIER) rhs→symbol = getsym(rhs→lexeme→name, false);
if (n→symbol→var.value & ZR) {
  if (¬isregister(rhs)) report(rhs→lexeme→linenr, rhs→lexeme→column, ERRSEM, ERRRGRQ);
  else {
    z = rhs→symbol→var.value;
    if ((z ≥ gregcntr) ∧ (z < 255)) rela→cr = rhs;
  }
}
else if (n→symbol→var.value & ZS) {
  if (¬isspecial(rhs)) report(rhs→lexeme→linenr, rhs→lexeme→column, ERRSEM, ERRSRRQ);
  else z = rhs→symbol→var.value;
}
```
This code is used in section 155.

157. ⟨ Check for Z as immediate 157 ⟩ ≡
```
else
  if (n→symbol→var.value & (ZI + YZI + XYZI)) {
    if (¬isnumber(rhs)) report(rhs→lexeme→linenr, rhs→lexeme→column, ERRSEM, ERRNMRQ);
    else if (rhs) z = evaluate(rhs).value;
  }
```
This code is used in section 155.

158. ⟨ Check for Z as register/immediate 158 ⟩ ≡
```
else
  if (n→symbol→var.value & ZRI) {
    if (isregister(rhs)) {
      z = (rhs→symbol→var.value > 255) ? bigL : rhs→symbol→var.value;
      if ((z ≥ gregcntr) ∧ (z < 255)) rela→cr = rhs;
    }
    else if (¬isnumber(rhs)) report(rhs→lexeme→linenr, rhs→lexeme→column, ERRSEM, ERRRNRQ);
    else z = evaluate(rhs).value, op++;
  }
```
This code is used in section 155.

159. ⟨ Perform all fixups 159 ⟩ ≡
```
for ( ; ggfix; ggfix = ggfix→next) {
  node *lf = Λ;
  if ((ggfix→ins→inst1 & #ff000000) ≡ #f0000000) lf = ggfix→ins→cl;
  else lf = ggfix→ins→cm;
  lf→symbol = getsym(lf→lexeme→name, false);
  if (lf→symbol→usage ≠ LABEL) report(lf→lexeme→linenr, lf→lexeme→column, ERRSEM, ERRLBRQ);
  else ggfix→ins→inst1 += (lf→symbol→var.value − ggfix→loc − 4) ≫ 2;
}
```
This code is used in section 12.

160. ⟨ Global variables 15 ⟩ +≡
```
fixtype * ggfix = Λ;
```

161. ⟨ Type definitions 24 ⟩ +≡

```
typedef struct fixstruct {
  struct fixstruct *next;
  octa loc;
  node *ins;
} fixtype;
```

162. Code generation.

```
⟨ Code generation 162 ⟩ ≡
  void generate(node *n)
  {
    node *lhs = Λ, *mhs = Λ, *rhs = Λ;
    for ( ; n; n = n→next) {
      Elf64_Off base;
      lhs = n→cl; mhs = n→cm; rhs = n→cr;
      switch (n→ntype) {
        ⟨ Generate cases 163 ⟩
      default: break;
      }
    }
  }
```

This code is used in section 12.

163. ⟨ Generate cases 163 ⟩ ≡

```
case GLOBAL:
  {
    if (lhs→ntype ≡ SPECIAL) {
      ⟨ Generate special registers 164 ⟩
    }
    else if (¬lhs→symbol→extsym) {
      ⟨ Generate global registers 165 ⟩
    }
    else {
      ⟨ Generate external global registers 166 ⟩
    }
  }
  break;
```

See also sections 167, 169, 171, 172, 173, 174, and 175.

This code is used in section 162.

164. ⟨ Generate special registers 164 ⟩ ≡

```
  int r = lhs→symbol→var.value;
  setptr(sct[REG], 8 * rstk[r]);
  evaluate(rhs);
  if (rstk[r] < 99) {
    if ((rhs→ntype ≡ IDENTIFIER) & (rhs→symbol→usage ≡ ADDRESS))
      setRela(sct[RREG], sct[REG]→ptr, rhs→symbol→symidx, R_MMIX_A64, 0);
    else store(sct[REG], rhs→var.value, 8, false);
  }
```

This code is used in section 163.

165. ⟨ Generate global registers 165 ⟩ ≡
```
evaluate(rhs);
if ((rhs→ntype ≡ IDENTIFIER) & (rhs→symbol→usage ≡ ADDRESS)) {
  setRela(sct[RSTK], sct[STK]→ptr, rhs→symbol→symidx, R_MMIX_A64, 0);
  store(sct[STK], 0, 8, true);
}
else store(sct[STK], rhs→var.value, 8, true);
lhs→symbol→symidx = sct[SYM]→ptr/sizeof (Elf64_Sym);
setSym(sct[SYM], sct[STR]→ptr, STB_GLOBAL, STT_OBJECT, 0, sct[STK]→index,
    (254 − lhs→symbol→var.value) ∗ 8, 8);
setStr(sct[STR], lhs→symbol→name);
if (debug) setStab(sct[STAB], 0, N_DSLINE, 0, lhs→lexeme→linenr, lhs→symbol→var.value);
```
This code is used in section 163.

166. ⟨ Generate external global registers 166 ⟩ ≡
```
lhs→symbol→symidx = sct[SYM]→ptr/sizeof (Elf64_Sym);
setSym(sct[SYM], sct[STR]→ptr, STB_GLOBAL, STT_OBJECT, 0, 0, 0, 8);
setStr(sct[STR], lhs→symbol→name);
```
This code is used in section 163.

167. ⟨ Generate cases 163 ⟩ +≡
```
case DATA:
  {
    if (lhs→symbol→extsym) {
      lhs→symbol→symidx = sct[SYM]→ptr/sizeof (Elf64_Sym);
      setSym(sct[SYM], sct[STR]→ptr, STB_GLOBAL, STT_OBJECT, 0, 0, 0, 0);
      setStr(sct[STR], lhs→symbol→name);
    }
    else if (rhs) ⟨ Generate data rhs 168 ⟩
    else {
      lhs→symbol→symidx = sct[SYM]→ptr/sizeof (Elf64_Sym);
      setSym(sct[SYM], sct[STR]→ptr, STB_GLOBAL, STT_OBJECT, 0, sct[BSS]→index, lhs→symbol→var.value,
          size[lhs→symbol→var.type]);
      setStr(sct[STR], lhs→symbol→name);
      if (debug) setStab(sct[STAB], 0, N_DSLINE, 0, lhs→lexeme→linenr, lhs→symbol→var.value);
    }
  }
  break;
```

168. ⟨ Generate data rhs 168 ⟩ ≡

```
{
  align(sct[RW], size[lhs→symbol→var.type]);
  int base = sct[RW]→ptr;
  for (node *p = rhs; p; p = p→next) {
    evaluate(p);
    if ((p→ntype ≡ IDENTIFIER) & (p→symbol→usage ≡ ADDRESS)) {
      setRela(sct[RRW], sct[RW]→ptr, p→symbol→symidx, R_MMIX_A64, 0);
      store(sct[RW], 0, size[lhs→symbol→var.type], true);
    }
    else if (p→ntype ≡ STRING) {
      char *q = p→lexeme→name + 1;
      if (lhs→symbol→var.type ≡ UBYTE) {
        for ( ; *q ∧ (*q ≠ '"'); q++) store(sct[RW], *q, 1, false);
      }
      else
        for ( ; *q ∧ (*q ≠ '"'); q = utfptr) store(sct[RW], utfval(q), size[lhs→symbol→var.type], true);
    }
    else if (lhs→symbol→var.type ≡ TSFLOAT) {
      octa f; int e; char s;
      funpack(p→var.value, &f, &e, &s);
      store(sct[RW], sfpack(f, e, s, 4), 4, true);
    }
    else store(sct[RW], p→var.value, size[lhs→symbol→var.type], true);
  }
  lhs→symbol→symidx = sct[SYM]→ptr/sizeof (Elf64_Sym);
  setSym(sct[SYM], sct[STR]→ptr, STB_GLOBAL, STT_OBJECT, 0, sct[RW]→index, lhs→symbol→var.value,
      sct[RW]→ptr − base);
  setStr(sct[STR], lhs→symbol→name);
  if (debug) setStab(sct[STAB], 0, N_DSLINE, 0, lhs→lexeme→linenr, lhs→symbol→var.value);
}
```

This code is used in section 167.

169. ⟨ Generate cases 163 ⟩ +≡

```
case CONSTANT:
  {
    if (lhs→symbol→extsym) {
      lhs→symbol→symidx = sct[SYM]→ptr/sizeof (Elf64_Sym);
      setSym(sct[SYM], sct[STR]→ptr, STB_GLOBAL, STT_OBJECT, 0, 0, 0, 0);
      setStr(sct[STR], lhs→symbol→name);
    }
    else if (rhs) ⟨ Generate constant rhs 170 ⟩
  }
  break;
```

170. ⟨ Generate constant rhs 170 ⟩ ≡

```
{
  align(sct[RO], size[lhs→symbol→var.type]);
  int base = sct[RO]→ptr;
  for (node *p = rhs; p; p = p→next) {
    evaluate(p);
    if ((p→ntype ≡ IDENTIFIER) & (p→symbol→usage ≡ ADDRESS)) {
      setRela(sct[RRO], sct[RO]→ptr, p→symbol→symidx, R_MMIX_A64, 0);
      store(sct[RO], 0, size[lhs→symbol→var.type], true);
    }
    else if (p→ntype ≡ STRING) {
      char *q = p→lexeme→name + 1;
      if (lhs→symbol→var.type ≡ UBYTE) {
        for ( ; *q ∧ (*q ≠ '"'); q++) store(sct[RO], *q, 1, false);
      }
      else
        for ( ; *q ∧ (*q ≠ '"'); q = utfptr) store(sct[RO], utfval(q), size[lhs→symbol→var.type], true);
    }
    else if (lhs→symbol→var.type ≡ TSFLOAT) {
      octa f; int e; char s;
      funpack(p→var.value, &f, &e, &s);
      store(sct[RO], sfpack(f, e, s, 4), 4, true);
    }
    else store(sct[RO], p→var.value, size[lhs→symbol→var.type], true);
  }
  lhs→symbol→symidx = sct[SYM]→ptr/sizeof (Elf64_Sym);
  setSym(sct[SYM], sct[STR]→ptr, STB_GLOBAL, STT_OBJECT, 0, sct[RO]→index, lhs→symbol→var.value,
      sct[RO]→ptr − base);
  setStr(sct[STR], lhs→symbol→name);
  if (debug) setStab(sct[STAB], 0, N_DSLINE, 0, lhs→lexeme→linenr, lhs→symbol→var.value);
}
```

This code is used in section 169.

171. ⟨ Generate cases 163 ⟩ +≡

```
case PROCEDURE:
  {
    base = sct[TEXT]→ptr;
    generate(rhs);
    lhs→symbol→symidx = sct[SYM]→ptr/sizeof (Elf64_Sym);
    if (lhs→symbol→extsym) setSym(sct[SYM], sct[STR]→ptr, STB_GLOBAL, STT_FUNC, 0, 0, 0, 0);
    else {
      setSym(sct[SYM], sct[STR]→ptr, STB_GLOBAL, STT_FUNC, 0, sct[TEXT]→index, base, sct[TEXT]→ptr − base);
    }
    setStr(sct[STR], lhs→symbol→name);
    if (debug) {
      setStab(sct[STAB], sct[STABSTR]→ptr, N_FUN, 0, 0, base);
      setStr(sct[STABSTR], lhs→symbol→name);
    }
  }
  break;
```

172. ⟨ Generate cases 163 ⟩ +≡
```
case BLOCK: generate(n→cr);
  if (n→inst1) store(sct[TEXT], n→inst1, 4, true);
  break;
```

173. ⟨ Generate cases 163 ⟩ +≡
```
case CALL:
  if (n→cm) generate(n→cm→cl);
  if (debug) setStab(sct[STAB], 0, N_SLINE, 0, lhs→lexeme→linenr, sct[TEXT]→ptr);
  store(sct[TEXT], n→inst1, 4, true);
  if (n→cm) generate(n→cm→cr);
  break;
case RETURN: store(sct[TEXT], n→inst1, 4, true);
  break;
```

174. ⟨ Generate cases 163 ⟩ +≡
```
case IN: case OUT: case ASSIGN: case SAVE: case UNSAVE: case LOAD: case STORE: case SWAP:
  if ((n→rel) ∧ (n→rel→cl)) setRela(sct[RTEXT], sct[TEXT]→ptr, n→rel→cl→symbol→symidx, R_MMIX_XR, 0);
  if ((n→rel) ∧ (n→rel→cm)) setRela(sct[RTEXT], sct[TEXT]→ptr, n→rel→cm→symbol→symidx, R_MMIX_YR, 0);
  if ((n→rel) ∧ (n→rel→cr)) setRela(sct[RTEXT], sct[TEXT]→ptr, n→rel→cr→symbol→symidx, R_MMIX_ZR, 0);
  if (debug) setStab(sct[STAB], 0, N_SLINE, 0, lhs→lexeme→linenr, sct[TEXT]→ptr);
  store(sct[TEXT], n→inst1, 4, true);
  break;
```

175. ⟨ Generate cases 163 ⟩ +≡
```
case OPCODE:
  if ((n→rel) ∧ (n→rel→cl)) setRela(sct[RTEXT], sct[TEXT]→ptr, n→rel→cl→symbol→symidx, R_MMIX_XR, 0);
  if ((n→rel) ∧ (n→rel→cm)) setRela(sct[RTEXT], sct[TEXT]→ptr, n→rel→cm→symbol→symidx, R_MMIX_YR, 0);
  if ((n→rel) ∧ (n→rel→cr)) setRela(sct[RTEXT], sct[TEXT]→ptr, n→rel→cr→symbol→symidx, R_MMIX_ZR, 0);
  if (debug) setStab(sct[STAB], 0, N_SLINE, 0, n→lexeme→linenr, sct[TEXT]→ptr);
  store(sct[TEXT], n→inst1, 4, true);
  break;
```

176. ⟨ Global variables 15 ⟩ +≡
```
int opc1[] = {#20, #22, #20, #22, #20, #22, #20, #22, #04, #04, #24, #26, #24, #26, #24, #26, #24, #26, #06,
    #06, #d0, #d0, #d2, #d2, #d4, #d4, #d6, #d6, −1, −1, #da, #da, #da, #da, #da, #da, #da, #da, −1, −1,
    #c0, #c0, #c0, #c0, #c0, #c0, #c0, #c0, #c0, #c0, #c2, #c2, #c2, #c2, #c2, #c2, #c2, #c2, #c2, #c2, #c6,
    #c6, #c6, #c6, #c6, #c6, #c6, #c6, #c6, #c6, #18, #1a, #18, #1a, #18, #1a, #18, #1a, #10, #10, #1c, #1e,
    #1c, #1e, #1c, #1e, #1c, #1e, #14, #14, −1, −1, −1, −1, −1, −1, −1, −1, #16, #16, #38, #3a, #38, #3a,
    #38, #3a, #38, #3a, −1, −1, #3c, #3e, #3c, #3e, #3c, #3e, #3c, #3e, −1, −1, #c8, #c8, #c8, #c8, #c8,
    #c8, #c8, #c8, #c8, #c8, #ca, #ca, #ca, #ca, #ca, #ca, #ca, #ca, #ca, #ca, #30, #32, #30, #32, #30, #32,
    #30, #32, #01, #01, −1, −1, −1, −1, −1, −1, −1, −1, #15, #15, #e3, #e3, #e3, #e3, #e3, #e3, #e3, #e3,
    #09, #0d, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, −1, #80, #82,
    #84, #86, #88, #8a, #8c, #8e, #8e, #90, #a0, #a2, #a4, #a6, #a8, #aa, #ac, #ae, #ae, #b0, #94, #94, #94,
    #94, #94, #94, #94, #94, −1, −1, −1};
```

177.

```
#define XS 8
#define XR 4
#define YR 2
#define ZR 1
#define ZS #80
#define XI #40
#define YI #20
#define ZI #10
#define XO #400
#define YO #200
#define ZO #100
#define ZRI #8000
#define YZM #4000
#define YZR #2000
#define YZI #1000
#define XYZR #20000
#define XYZI #10000
```

⟨ Global variables 15 ⟩ +≡

```
int opc2[] = {#00010000, #01000007, #02000007, #03000007, #04000007, #05000205, #06000007,
    #07000205, #08008204, #0a008204, #0c008204, #0e008204, #10000007, #11000007, #12000007,
    #13000007, #14000007, #15000205, #16000007, #17000205, #18008006, #1a008006, #1c008006,
    #1e008006, #20008006, #22008006, #24008006, #26008006, #28008006, #2a008006, #2c008006,
    #2e008006, #30008006, #32008006, #34008024, #36008024, #38008006, #3a008006, #3c008006,
    #3e008006, #40002004, #42002004, #44002004, #46002004, #48002004, #4a002004, #4c002004,
    #4e002004, #50002004, #52002004, #54002004, #56002004, #58002004, #5a002004, #5c002004,
    #5e002004, #60008006, #62008006, #64008006, #66008006, #68008006, #6a008006, #6c008006,
    #6e008006, #70008006, #72008006, #74008006, #76008006, #78008006, #7a008006, #7c008006,
    #7e008006, #80004004, #82004004, #84004004, #86004004, #88004004, #8a004004, #8c004004,
    #8e004004, #90004004, #92004004, #94004004, #96004004, #98004004, #9a004040, #9c004040,
    #9e004004, #a0004004, #a2004004, #a4004004, #a6004004, #a8004004, #aa004004, #ac004004,
    #ae004004, #b0004004, #b2004004, #b4004040, #b6004004, #b8004040, #ba004040, #bc004040,
    #be004004, #c0008006, #c2008006, #c4008006, #c6008006, #c8008006, #ca008006, #cc008006,
    #ce008006, #d0008006, #d2008006, #d4008006, #d6008006, #d8008006, #da008006, #dc008006,
    #de008006, #e0001004, #e1001004, #e2001004, #e3001004, #e4001004, #e5001004, #e6001004,
    #e7001004, #e8001004, #e9001004, #ea001004, #eb001004, #ec001004, #ed001004, #ee001004,
    #ef001004, #f0020000, #f2002004, #f4002004, #f6008208, #f8001040, #f9000610, #fa000004,
    #fb000601, #fc010000, #fd010000, #fe000284, #ff010000};
```


a: 33, 56, 92, 140.
accept: 35, 38.
ACTUAL: 26, 79.
ADDRESS: 26, 106, 107, 110, 111, 114, 115, 123, 164, 165, 168, 170.
Aho, A.V.: 28.
align: 102, 111, 112, 115, 168, 170.
allocate: 13, 16, 18, 21, 23, 36, 50, 99, 102, 119, 128, 148, 153.
AND: 26, 38, 92, 94.
ANDEQ: 26, 38.
ANDN: 26, 38, 60, 92, 94, 140.
ANDNEQ: 26, 38.
argc: 12, 14.
argv: 12, 14.
ASSIGN: 26, 38, 66, 83, 84, 133, 174.
aux: 96.
b: 92, 140.
base: 33, 38, 162, 168, 170, 171.
bigL: 120, 121, 158.
BLOCK: 26, 88, 123, 172.
branch: 58, 60, 61, 62, 64, 67, 69, 70, 71, 72, 73, 74, 75, 76, 78, 79, 80, 82, 83, 84, 85, 87, 88, 89, 90, 91.
BSS: 99, 100, 112, 167.
bss: 110.

buildEhdr: 98.
buildSection: 99.
BYTE: 26, 68.
c: 30, 31, 56, 140.
CALL: 26, 79, 86, 129, 173.
calloc: 13.
CHARACTER: 26, 38, 56, 58.
check: 33, 38.
cl: 47, 50, 81, 92, 94, 95, 96, 97, 104, 120, 123, 124, 129, 132, 136, 137, 138, 139, 140, 141, 142, 145, 159, 162, 173, 174, 175.
cm: 47, 50, 64, 81, 92, 104, 118, 124, 129, 136, 141, 142, 150, 154, 159, 162, 173, 174, 175.
CMP: 26, 38, 64.
CMPEQ: 26, 38.
COLON: 26, 38, 64, 83, 136.
column: 28, 36, 40, 41, 42, 53, 54, 55, 56, 58, 63, 66, 67, 69, 71, 72, 73, 76, 77, 79, 87, 88, 90, 92, 93, 94, 105, 106, 110, 114, 117, 118, 122, 124, 129, 130, 131, 132, 133, 134, 136, 137, 140, 145, 146, 148, 150, 151, 152, 153, 154, 156, 157, 158, 159.
COMMA: 26, 38, 58, 67, 69, 70, 71, 78, 81.
COMMENT: 26, 38, 49.
CONSTANT: 26, 73, 90, 114, 169.
count_bits: 96, 137.
cr: 47, 50, 66, 67, 73, 81, 82, 83, 84, 89, 92, 94, 95, 96, 97, 104, 105, 114, 118, 120, 123, 124, 129, 132, 134, 135, 136, 137, 138, 140, 141, 142, 154, 156, 158, 162, 172, 173, 174, 175.
ctype: 30, 32, 56.
currsym: 22.
data: 102, 103, 110.
DATA: 26, 73, 90, 110, 167.
debug: 14, 15, 98, 102, 107, 110, 114, 118, 125, 126, 127, 128, 129, 131, 142, 165, 167, 168, 170, 171, 173, 174, 175.
DEFINE: 26, 90, 117.
deflt: 33, 38.
DIFF: 26, 38, 96.
DIFFEQ: 26, 38.
DIV: 26, 38, 96, 97.
DIVEQ: 26, 38.
DO: 26, 86.
dummyR: 51, 59.
dummy0: 51, 59, 60, 61, 62, 73, 83, 105, 114, 132, 134, 135, 140.
e: 168, 170.
e_shentsize: 98.
e_shnum: 98, 102.
e_shoff: 102.
e_shstrndx: 98.
e_type: 98.
ehdr: 98, 101, 102, 103.
Elf64_Ehdr: 101, 102.
Elf64_Off: 102, 162.
Elf64_Rela: 106, 107, 111, 115, 124, 129, 142.
Elf64_Section: 99, 101.
Elf64_Shdr: 98.
Elf64_Stab: 98, 107, 110, 114, 118, 125, 126, 127, 128, 129, 131, 142.
Elf64_Sym: 98, 102, 107, 110, 114, 118, 165, 166, 167, 168, 169, 170, 171.
ELSE: 26, 74.
EOS: 26, 38, 52, 88, 91.
EPSILON: 10.
EQ: 26, 38, 62, 96, 97.
ERRADDR: 44, 132.
ERRADDU: 44, 137.
ERRC: 32.
ERRCHAR: 44, 56.
errcl: 40, 41, 42.
errclass: 41, 45.
errcltype: 44.
ERRCOMP: 44, 124, 136.
errdesc: 41, 45.
ERRDUPL: 44, 105, 110, 114, 117, 118, 122, 130.
ERREVEN: 44, 63, 124.
ERREXPR: 44, 58.
ERRHLNM: 44, 140.
ERRIDRQ: 44, 54, 69, 71, 77, 87.
ERRINIT: 44, 66, 67, 105, 106, 117.
ERRINVR: 44, 92, 93, 133, 134.
ERRLBRC: 44, 88.
ERRLBRQ: 44, 148, 153, 159.
ERRLEX: 44, 56.
ERRNMRQ: 44, 56, 146, 151, 152, 154, 157.
ERRNPAR: 44, 153.
errnr: 40, 41, 42.
errnrtype: 44.
ERRNVAL: 44, 93, 117.
ERROR: 26, 85.
errpos: 40, 41, 43.
ERRPROC: 44, 129.
ERRRBRC: 44, 67, 88.
ERRRGRQ: 44, 124, 129, 131, 132, 134, 137, 145, 150, 154, 156.
ERRRNRQ: 44, 158.
ERRRPRN: 44, 58, 72, 79.
ERRSCLN: 44, 53.
ERRSEM: 44, 92, 93, 94, 105, 106, 110, 114, 117, 118, 122, 124, 129, 130, 131, 132, 133, 134, 136, 137, 140, 145, 146, 148, 150, 151, 152, 153, 154, 156, 157, 158, 159.

ERRSRRQ: 44, 55, 145, 156.
ERRSTMT: 44, 90.
ERRSYN: 44, 53, 54, 55, 56, 58, 63, 66, 67, 69, 71, 72, 73, 76, 77, 79, 87, 88, 90.
ERRTYPE: 44, 94.
errtype: 42, 43.
ERRUNTL: 44, 76.
ERRXTRN: 44, 73.
ERRZERO: 44, 136.
ERRZSYM: 44, 136.
ET_REL: 98.
evaluate: 92, 105, 111, 115, 117, 129, 132, 138, 139, 140, 141, 146, 151, 152, 154, 157, 158, 164, 165, 168, 170.
EVEN: 26, 63, 96, 124, 136.
exit: 12.
EXPRESSION: 26, 58, 92, 123, 124, 136.
ext: 18.
EXTERN: 26, 90.
extsym: 21, 24, 107, 110, 114, 118, 129, 163, 167, 169, 171.
f: 168, 170.
false: 15, 23, 36, 51, 55, 81, 93, 98, 105, 106, 107, 110, 111, 113, 114, 115, 116, 117, 118, 122, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 136, 137, 138, 142, 145, 150, 154, 156, 159, 164, 168, 170.
fclose: 103.
fcomp: 97.
fdivide: 97.
fetch: 30, 35.
fixstruct: 161.
fixtype: 128, 148, 153, 160, 161.
fixup: 128, 148, 153.
FLOAT: 26, 68.
floatit: 95, 97.
FLTNUM: 26, 38, 56, 58, 92, 123.
fmult: 97.
fname: 18.
fopen: 16, 103.
fplus: 97.
fprintf: 12, 14.
fread: 16.
free: 21, 49, 142.
fremstep: 97.
froot: 95.
fseek: 16.
ftell: 16.
funpack: 168, 170.
fwrite: 103.
g: 108.
GE: 26, 38, 62, 96, 97.
generate: 12, 162, 171, 172, 173.
GET: 26, 134.
getsym: 22, 36, 55, 81, 93, 105, 110, 114, 117, 118, 122, 124, 128, 129, 131, 132, 133, 136, 137, 138, 145, 150, 154, 156, 159.
Getsym: 22.
ggatom: 58, 59.
ggblck: 74, 75, 76, 86, 88, 89.
ggcall: 79, 86.
ggcomp: 62, 64, 74, 75, 76.
ggdcl: 65, 66, 69.
ggdcls: 69, 73, 87.
ggexpr: 58, 64, 66, 67, 81, 82, 83, 84.
ggextrn: 73, 90.
ggfix: 128, 148, 153, 159, 160.
gggoto: 77, 86.
ggid: 54, 58, 65, 70, 72, 77, 78, 79.
ggidcl: 65, 66, 69, 71.
ggidcls: 69, 71, 90.
ggifte: 74, 86.
ggins: 81, 82, 86.
gglcl: 87, 88.
ggnum: 56, 58, 80.
ggopc: 81, 86.
ggparm: 78, 79.
ggparse: 12, 91.
ggphdr: 72, 73, 89.
ggproc: 89, 90.
ggreg: 55, 58, 65.
ggroot: 12, 48.
ggrtrn: 80, 86.
ggstmnt: 86, 88.
ggstrong: 60, 61.
ggtdcls: 70, 72.
ggtoken: 48, 49, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 76, 77, 78, 79, 80, 81, 83, 84, 86, 87, 88, 90, 91.
ggtopdcl: 90, 91.
ggtype: 68, 69, 70, 71, 87.
ggunit: 59, 60.
gguntil: 76, 86.
ggweak: 61, 62, 63, 64.
ggwhile: 75, 86.
glbsym: 23, 25, 107, 110, 114, 117, 118, 130.
glbsymtabsize: 23, 25.
GLOBAL: 23, 26, 73, 90, 105, 163.
GOTO: 26, 86, 128.
greg: 107, 109, 132, 154.
gregcntr: 107, 109, 124, 132, 141, 145, 150, 154, 156, 158.
GT: 26, 38, 62, 96, 97.

gtype: 108, 109.
h: 20.
hash: 20, 21, 22.
hasvalue: 47, 56, 93.
hktable: 20, 25.
i: 16, 23, 30, 41, 102, 103, 132, 154.
IDENTIFIER: 23, 26, 36, 38, 54, 58, 65, 69, 71, 77, 78, 92, 106, 107, 111, 115, 123, 124, 128, 129, 132, 133, 136, 137, 138, 145, 148, 150, 153, 154, 156, 164, 165, 168, 170.
IF: 26, 74, 86, 124.
IN: 26, 79, 129, 174.
index: 102, 103, 165, 167, 168, 170, 171.
ins: 128, 148, 153, 159, 161.
inst: 23, 27.
inst1: 47, 125, 126, 127, 128, 129, 131, 142, 159, 172, 173, 174, 175.
inst2: 47, 125, 126.
INVALID: 26, 38.
isaddress: 123, 132, 154.
isextern: 21, 47, 73, 107, 110, 114, 118.
isfloat: 123, 134.
islabel: 123, 128, 148, 153.
isnumber: 123, 124, 132, 139, 140, 146, 151, 152, 154, 157, 158.
isregister: 123, 124, 129, 131, 132, 133, 134, 135, 137, 141, 145, 150, 154, 156, 158.
isspecial: 123, 132, 134, 141, 145, 156.
j: 41.
key: 36.
KEYWORD: 23, 26, 36.
Knuth, D. E.: 1, 20.
l: 50.
LABEL: 26, 82, 83, 123, 130, 159.
Lam, M.S.: 28.
last: 67.
LBRC: 26, 38, 52, 66, 86, 88.
LBRK: 26, 38.
lbufptr: 16, 19, 34.
lclsym: 22, 23, 25, 118, 119, 122, 123.
lclsymtabsize: 25, 119.
LE: 26, 38, 62, 96, 97.
leaf: 54, 55, 56, 57, 59, 60, 61, 62, 63, 64, 66, 67, 69, 70, 71, 72, 77, 78, 79, 80, 87, 88.
lexeme: 47, 54, 55, 56, 58, 59, 60, 61, 62, 63, 66, 81, 83, 84, 92, 93, 94, 105, 106, 107, 110, 113, 114, 116, 117, 118, 122, 124, 128, 129, 130, 131, 132, 133, 134, 136, 137, 138, 140, 145, 146, 148, 150, 151, 152, 153, 154, 156, 157, 158, 159, 165, 167, 168, 170, 173, 174, 175.
lexemeR: 51.
lexeme0: 51.
lexer: 34, 49.
lextype: 26.
lf: 159.
lhs: 104, 105, 106, 107, 110, 111, 112, 113, 114, 115, 116, 117, 118, 122, 124, 128, 129, 130, 131, 132, 133, 134, 135, 136, 138, 139, 140, 141, 144, 145, 146, 147, 148, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 173, 174.
linenr: 28, 36, 40, 41, 42, 53, 54, 55, 56, 58, 63, 66, 67, 69, 71, 72, 73, 76, 77, 79, 87, 88, 90, 92, 93, 94, 105, 106, 110, 114, 117, 118, 122, 124, 129, 130, 131, 132, 133, 134, 136, 137, 140, 145, 146, 148, 150, 151, 152, 153, 154, 156, 157, 158, 159, 165, 167, 168, 170, 173, 174, 175.
ll: 124, 137.
llaccept: 35, 36, 37.
llaptr: 34, 35, 36, 37.
llcolumn: 29, 30, 34, 35.
llline: 29, 34, 36.
llpos: 35, 36, 37.
llptr: 29, 30, 34, 35, 36.
llstate: 35, 37.
LOAD: 26, 38, 132, 174.
loc: 128, 148, 153, 159, 161.
LOCAL: 26, 70, 87, 122.
lookahead: 49, 52, 53, 54, 55, 56, 58, 59, 60, 61, 62, 63, 64, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 83, 84, 87, 88, 90, 91.
LPRN: 26, 38, 58, 72, 79.
lr: 137, 141.
lregcntr: 118, 120, 121, 122, 123, 129.
LT: 26, 38, 62, 96, 97, 124, 136.
ltwig: 72, 74, 75, 76, 79, 88.
m: 41, 50.
main: 12.
MAXERR: 39, 40, 43.
MAXGLBSYM: 23, 25.
MAXLCLSYM: 23, 25.
mhs: 104, 125, 126, 127, 146, 147, 149, 150, 151, 152, 153, 154, 162.
MINUS: 26, 38, 59, 96, 97, 139.
MINUSEQ: 26, 38.
mknode: 50, 54, 55, 56, 57, 58, 59, 60, 61, 62, 64, 65, 66, 70, 72, 74, 75, 76, 77, 78, 79, 80, 81, 83, 84, 85, 88, 91, 132, 133, 134, 135, 143.
mksym: 21, 23, 107, 110, 114, 117, 118, 122, 130.
MMIX: 23.
mmixtypes: 24.
mnem: 23, 25.
move: 12, 54, 55, 56, 58, 59, 60, 61, 62, 63, 64, 66, 77, 81, 83, 84.
movebeyond: 52.

mtwig: 74, 75, 76.
n: 13, 14, 20, 50, 69, 71, 73, 79, 87, 90, 92, 104, 140, 162.
N_DSLINE: 165, 167, 168, 170.
N_FUN: 171.
N_SLINE: 173, 174, 175.
N_UNDF: 102.
name: 21, 22, 24, 28, 36, 49, 55, 56, 57, 81, 93, 102, 105, 107, 110, 113, 114, 116, 117, 118, 122, 124, 128, 129, 130, 131, 132, 133, 136, 137, 138, 145, 150, 154, 156, 159, 165, 166, 167, 168, 169, 170, 171.
NE: 26, 38, 62, 96, 97.
newop: 140.
next: 21, 22, 24, 33, 38, 47, 67, 69, 70, 71, 73, 78, 79, 87, 88, 90, 91, 104, 105, 111, 115, 117, 128, 148, 153, 159, 161, 162, 168, 170.
nextstate: 33, 35.
node: 47, 48, 50, 51, 54, 55, 56, 58, 59, 60, 61, 62, 64, 65, 66, 67, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 86, 87, 88, 89, 90, 91, 92, 104, 111, 115, 124, 132, 133, 136, 137, 140, 141, 143, 159, 161, 162, 168, 170.
nodestruct: 47.
NOT: 26, 38, 59, 92, 94.
npop: 120, 121, 129.
nrlines: 16, 19.
ntype: 47, 50, 57, 63, 64, 73, 79, 82, 87, 90, 92, 94, 96, 97, 104, 105, 106, 107, 111, 115, 123, 124, 128, 129, 131, 132, 133, 136, 137, 138, 139, 140, 145, 148, 150, 153, 154, 156, 162, 163, 164, 165, 168, 170.
NUMBER: 26, 38, 51, 56, 57, 58, 63, 80, 92, 117, 123, 132, 136, 137.
nxterr: 12, 40, 41, 43.
n1: 132.
n2: 132.
octa: 12, 24, 56, 104, 140, 161, 168, 170.
OCTA: 26, 68.
octa_bdif: 96.
ODD: 26, 38, 63, 96, 136.
offset: 102.
OFFSET: 26.
onlylocal: 22.
op: 124, 125, 126, 127, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 148, 152, 153, 154, 158.
OPCODE: 23, 26, 81, 86, 143, 175.
opc1: 132, 138, 176.
opc2: 23, 177.
OR: 26, 38, 70, 72, 79, 92, 94, 135, 140.
order: 97.
OREQ: 26, 38.
ORN: 26, 38, 92, 94.
ORNEQ: 26, 38.
OUT: 26, 79, 129, 174.
p: 13, 14, 21, 22, 57, 111, 113, 115, 116, 168, 170.
panic: 12, 13, 14, 16, 30, 103.
PARAMETER: 26, 72, 120.
parent: 22, 24, 119, 123.
pdot: 18.
peek: 30, 35.
PL/360: 1.
plan: 98, 106, 107, 110, 111, 112, 113, 114, 115, 116, 118, 124, 125, 126, 127, 128, 129, 131, 142.
PLUS: 26, 38, 59, 61, 84, 96, 97, 132, 135, 137, 138, 140.
PLUSEQ: 26, 38, 84.
printf: 41.
PROCEDURE: 26, 72, 73, 88, 90, 118, 129, 171.
prterr: 12, 41.
pslash: 18.
ptr: 102, 106, 111, 112, 115, 118, 125, 126, 127, 128, 129, 130, 148, 153, 164, 165, 166, 167, 168, 169, 170, 171, 173, 174, 175.
PUT: 26, 134.
q: 14, 21, 168, 170.
r: 50, 106, 164.
R_MMIX_A64: 164, 165, 168, 170.
R_MMIX_XR: 174, 175.
R_MMIX_YR: 174, 175.
R_MMIX_ZR: 174, 175.
RBRC: 26, 38, 58, 67, 88.
RBRK: 26, 38.
recover: 52, 53, 54, 55, 56, 58, 63, 67, 69, 71, 72, 73, 76, 77, 78, 79, 87, 88, 90.
REG: 99, 100, 103, 106, 164.
regcntr: 121, 129.
REGISTER: 23, 26, 93, 107, 122, 123.
rel: 47, 142, 174, 175.
rela: 132, 133, 141, 142, 143, 145, 150, 154, 156, 158.
RELOCATION: 26, 132, 133, 143.
REM: 26, 38, 96, 97.
REMEQ: 26, 38.
report: 39, 40, 53, 54, 55, 56, 58, 63, 66, 67, 69, 71, 72, 73, 76, 77, 79, 87, 88, 90, 92, 93, 94, 105, 106, 110, 114, 117, 118, 122, 124, 129, 130, 131, 132, 133, 134, 136, 137, 140, 145, 146, 148, 150, 151, 152, 153, 154, 156, 157, 158, 159.
resolve: 12, 104, 118, 120, 123, 125, 126, 127, 129.
result: 18.
RETURN: 26, 80, 86, 129, 173.

rhs: 104, 105, 106, 107, 110, 111, 113, 114, 115, 116, 117, 125, 132, 133, 134, 135, 136, 137, 138, 139, 140, 147, 151, 152, 153, 154, 155, 156, 157, 158, 162, 164, 165, 167, 168, 169, 170, 171.
rl: 124, 136.
RO: 99, 100, 103, 115, 116, 170.
ROOT: 26, 91.
ROUND: 10, 59.
RPRN: 26, 38, 58, 70, 72, 79.
rr: 141.
RREG: 99, 100, 103, 106, 164.
RRO: 99, 100, 103, 115, 170.
RRW: 99, 100, 103, 111, 168.
rstk: 106, 109, 164.
RSTK: 99, 100, 107, 165.
RTEXT: 99, 100, 103, 124, 129, 142, 174, 175.
rtwig: 72, 74, 79, 88.
RW: 99, 100, 103, 111, 113, 168.
s: 13, 20, 33, 125, 126, 127, 168, 170.
SADD: 26, 38, 96.
SADDEQ: 26, 38.
SAVE: 26, 38, 83, 131, 174.
savelregcntr: 123.
SBYTE: 24.
scan_const: 57.
sct: 98, 99, 101, 102, 103, 106, 107, 110, 111, 112, 113, 114, 115, 116, 118, 124, 125, 126, 127, 128, 129, 130, 131, 142, 148, 153, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175.
SCTSNT: 99, 100, 102, 103.
scttype: 100.
SEEK_END: 16.
SEEK_SET: 16.
SEMICOLON: 26, 38, 52, 53, 58, 90.
SENTINEL: 44.
setfext: 16, 18.
Sethi, R.: 28.
setptr: 164.
setRela: 164, 165, 168, 170, 174, 175.
setStab: 102, 165, 167, 168, 170, 171, 173, 174, 175.
setStr: 102, 165, 166, 167, 168, 169, 170, 171.
setSym: 165, 166, 167, 168, 169, 170, 171.
SFLOAT: 23, 26, 68.
sfpack: 168, 170.
sh_addralign: 102.
sh_info: 103.
sh_link: 103.
sh_name: 102.
sh_offset: 102.
sh_size: 102, 103.
shdr: 102, 103.
SHF_ALLOC: 99.
SHF_EXECINSTR: 99.
SHF_WRITE: 99.
SHL: 26, 38, 96, 140.
SHLEQ: 26, 38.
SHR: 26, 38, 96.
SHREQ: 26, 38.
SHT_NOBITS: 99.
SHT_PROGBITS: 99.
SHT_RELA: 99.
SHT_STRTAB: 99.
SHT_SYMTAB: 99.
sign: 68.
SIGNED: 26, 68.
signed_odiv: 96.
size: 21, 22, 23, 24, 25, 68, 111, 112, 113, 115, 116, 119, 167, 168, 170.
skip: 104, 129.
SOCTA: 24.
source: 16, 17, 19.
SPACE: 26, 38.
SPECIAL: 23, 26, 55, 58, 65, 69, 71, 92, 105, 123, 163.
SQRT: 26, 38, 59, 92, 138.
srcfile: 16, 19.
srcline: 16, 19, 34, 41.
srcname: 14, 15, 16, 98, 102.
srcsize: 16, 17, 19.
sreg: 23, 25.
STAB: 98, 99, 100, 102, 107, 110, 114, 118, 125, 126, 127, 128, 129, 131, 142, 165, 167, 168, 170, 171, 173, 174, 175.
STABSTR: 98, 99, 100, 102, 118, 171.
STB_GLOBAL: 165, 166, 167, 168, 169, 170, 171.
stderr: 12, 14.
STETRA: 24.
STK: 99, 100, 107, 165.
STORE: 26, 38, 132, 174.
store: 164, 165, 168, 170, 172, 173, 174, 175.
STR: 98, 99, 100, 102, 103, 107, 110, 114, 118, 165, 166, 167, 168, 169, 170, 171.
strcat: 18.
strchr: 14.
strcmp: 21, 22.
strcpy: 18, 21.
strdup: 14.
STRING: 26, 38, 66, 111, 115, 168, 170.
strlen: 14, 18, 21, 98, 102, 107, 110, 113, 114, 116, 118.
strncmp: 14.
strncpy: 36.
strrchr: 18.
STT_FUNC: 171.

STT_OBJECT: 165, 166, 167, 168, 169, 170.
SWAP: 26, 38, 83, 132, 174.
SWYDE: 24.
SYM: 98, 99, 100, 102, 103, 107, 110, 114, 118, 165, 166, 167, 168, 169, 170, 171.
sym: 21, 22, 23, 24, 119.
symbol: 47, 55, 81, 93, 106, 107, 110, 111, 112, 113, 114, 115, 116, 117, 118, 122, 123, 124, 128, 129, 130, 131, 132, 133, 135, 136, 137, 138, 139, 140, 141, 143, 145, 146, 147, 148, 150, 151, 152, 153, 154, 156, 157, 158, 159, 163, 164, 165, 166, 167, 168, 169, 170, 171, 174, 175.
symidx: 24, 164, 165, 166, 167, 168, 169, 170, 171, 174, 175.
symstruct: 24.
symtab: 47, 118, 123.
symtabtype: 21, 22, 23, 24, 25, 47, 119.
symtbl: 21.
symtype: 21, 22, 23, 24, 36, 47, 108, 119.
t: 50, 58, 59, 60, 61, 62, 64, 77, 92.
tabstruct: 24.
TETRA: 26.
TEXT: 99, 100, 103, 118, 125, 126, 127, 128, 129, 130, 131, 142, 148, 153, 171, 172, 173, 174, 175.
TFLOAT: 24, 57, 68, 95, 97, 135.
tgtfile: 19, 103.
tgtname: 14, 15, 16, 103.
TIMES: 26, 38, 60, 96, 97, 137.
TIMESEQ: 26, 38.
token: 34, 36, 37.
tokentype: 28, 34, 36, 37, 47, 48, 51, 58, 59, 60, 61, 62, 64, 77.
true: 14, 17, 40, 56, 73, 93, 107, 111, 112, 113, 115, 116, 122, 125, 126, 127, 128, 129, 131, 138, 139, 140, 142, 165, 168, 170, 172, 173, 174, 175.
TSFLOAT: 24, 168, 170.
twig: 67, 78, 81, 82, 83, 84, 85, 87, 91.
type: 24, 28, 36, 49, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 76, 77, 78, 79, 80, 81, 83, 84, 86, 87, 88, 90, 91, 92, 94, 95, 96, 97, 107, 110, 111, 112, 113, 114, 115, 116, 122, 132, 135, 138, 139, 167, 168, 170.
UBYTE: 23, 24, 51, 56, 97, 113, 116, 132, 168, 170.
Ullman, J.D.: 28.
UND: 99, 100.
UNSAVE: 26, 38, 83, 131, 174.
UNSIGNED: 26, 68.
UNTIL: 26, 76, 124.
UOCTA: 23, 24, 56, 70, 71, 87, 92, 94, 95, 97, 135.
usage: 21, 24, 36, 93, 106, 107, 111, 115, 123, 129, 159, 164, 165, 168, 170.
useyz: 132, 133, 138, 139, 140, 141.
UTETRA: 24, 56.
UTFC: 30, 32, 56.
utfptr: 29, 31, 113, 116, 168, 170.
utfval: 31, 56, 113, 116, 168, 170.
utf8: 14, 15, 17.
UWYDE: 24, 56.
v: 31, 41, 108.
val: 12, 57.
value: 24, 36, 56, 57, 63, 94, 95, 96, 97, 106, 107, 110, 111, 112, 114, 115, 118, 122, 124, 128, 129, 130, 131, 132, 136, 137, 138, 139, 140, 141, 143, 145, 146, 147, 148, 150, 151, 152, 153, 154, 156, 157, 158, 159, 164, 165, 167, 168, 170.
var: 23, 24, 36, 47, 56, 57, 63, 92, 93, 94, 95, 96, 97, 106, 107, 110, 111, 112, 113, 114, 115, 116, 117, 118, 122, 124, 128, 129, 130, 131, 132, 135, 136, 137, 138, 139, 141, 143, 145, 146, 147, 148, 150, 151, 152, 153, 154, 156, 157, 158, 159, 164, 165, 167, 168, 170.
varstruct: 24.
vartype: 23, 24, 47, 92, 93, 94, 96, 97, 132.
vtype: 47, 69, 70, 71, 87, 107, 110, 114, 122.
WHILE: 26, 75, 86, 124.
Wirth, N.: 1.
writeEhdr: 103.
writeShdr: 103.
WYDE: 26.
x: 124, 132, 133, 143.
XI: 146, 177.
XO: 147, 177.
XOR: 26, 38, 61, 92, 94.
XOREQ: 26, 38, 84.
XR: 145, 177.
XS: 145, 177.
XYZI: 146, 151, 157, 177.
XYZR: 148, 177.
y: 132, 133, 143.
YI: 151, 177.
YO: 152, 177.
YR: 150, 177.
yz: 124, 125, 126, 127, 132, 133, 138, 139, 140, 141, 143.
YZI: 151, 157, 177.
YZM: 154, 177.
YZR: 153, 177.
z: 132, 133, 143.
ZI: 157, 177.
ZO: 177.
ZR: 156, 177.
ZRI: 158, 177.
ZS: 156, 177.

⟨ Allocate section data 102 ⟩ Used in section 12.
⟨ Basic subroutines 13, 18 ⟩ Used in section 12.
⟨ Build the instruction 142 ⟩ Used in sections 132, 133, and 143.
⟨ Build the sections 99 ⟩ Used in section 98.
⟨ Check IF jumps 125 ⟩ Used in section 124.
⟨ Check UNTIL jumps 127 ⟩ Used in section 124.
⟨ Check WHILE jumps 126 ⟩ Used in section 124.
⟨ Check X 144 ⟩ Used in section 143.
⟨ Check Y 149 ⟩ Used in section 143.
⟨ Check Z 155 ⟩ Used in section 143.
⟨ Check closing semicolon 53 ⟩ Used in sections 73, 76, 77, 79, 80, 81, 82, 87, and 90.
⟨ Check for X as a register 145 ⟩ Used in section 144.
⟨ Check for X as immediate 146 ⟩ Used in section 144.
⟨ Check for X as optional 147 ⟩ Used in section 144.
⟨ Check for X as relative 148 ⟩ Used in section 144.
⟨ Check for Y as a register 150 ⟩ Used in section 149.
⟨ Check for Y as immediate 151 ⟩ Used in section 149.
⟨ Check for Y as memory 154 ⟩ Used in section 149.
⟨ Check for Y as optional 152 ⟩ Used in section 149.
⟨ Check for Y as relative 153 ⟩ Used in section 149.
⟨ Check for Z as a register 156 ⟩ Used in section 155.
⟨ Check for Z as immediate 157 ⟩ Used in section 155.
⟨ Check for Z as register/immediate 158 ⟩ Used in section 155.
⟨ Check for utf-8 preamble 17 ⟩ Used in section 16.
⟨ Code generation 162 ⟩ Used in section 12.
⟨ Error subroutines 40, 41 ⟩ Used in section 12.
⟨ Evaluate SQRT 95 ⟩ Used in section 92.
⟨ Evaluate boolean expression 94 ⟩ Used in section 92.
⟨ Evaluate floating point expression 97 ⟩ Used in section 92.
⟨ Evaluate identifier 93 ⟩ Used in section 92.
⟨ Evaluate integer expression 96 ⟩ Used in section 92.
⟨ Expression evaluation 92 ⟩ Used in section 12.
⟨ Generate cases 163, 167, 169, 171, 172, 173, 174, 175 ⟩ Used in section 162.
⟨ Generate constant rhs 170 ⟩ Used in section 169.
⟨ Generate data rhs 168 ⟩ Used in section 167.
⟨ Generate external global registers 166 ⟩ Used in section 163.
⟨ Generate global registers 165 ⟩ Used in section 163.
⟨ Generate special registers 164 ⟩ Used in section 163.
⟨ Global variables 15, 19, 25, 27, 29, 32, 38, 43, 45, 48, 51, 101, 109, 121, 160, 176, 177 ⟩ Used in section 12.
⟨ Handle ADDUs 137 ⟩ Used in section 133.
⟨ Handle NEG 139 ⟩ Used in section 138.
⟨ Handle PUT and GET 134 ⟩ Used in section 133.
⟨ Handle conditionals 136 ⟩ Used in section 133.
⟨ Handle list assignment 67 ⟩ Used in section 66.
⟨ Handle other instructions 138 ⟩ Used in section 133.
⟨ Handle parsing SAVE, UNSAVE, ASSIGN, LOAD, STORE, SWAP 83 ⟩ Used in section 82.
⟨ Handle parsing += ... ⊕= 84 ⟩ Used in section 82.
⟨ Handle program arguments 14 ⟩ Used in section 12.
⟨ Handle straight assignment 135 ⟩ Used in section 133.
⟨ Handle unrecognised statement 85 ⟩ Used in section 82.
⟨ Handle x,y and z 141 ⟩ Used in sections 132 and 133.
⟨ Handle yz immediate 140 ⟩ Used in section 138.

⟨ Initialise everything 16, 23, 98 ⟩ Used in section 12.
⟨ Lexical subroutines 30, 31, 33, 34 ⟩ Used in section 12.
⟨ Local variables for lexer 37 ⟩ Used in section 34.
⟨ Make a local symbol table 119 ⟩ Used in sections 118 and 123.
⟨ Perform all fixups 159 ⟩ Used in section 12.
⟨ Recognise different types of numbers 57 ⟩ Used in section 56.
⟨ Resolve cases 105, 110, 114, 117, 118, 120, 122, 123, 124, 128, 129, 130, 131, 132, 133, 143 ⟩ Used in section 104.
⟨ Resolve constant data 115 ⟩ Used in section 114.
⟨ Resolve constant string 116 ⟩ Used in section 115.
⟨ Resolve data string 113 ⟩ Used in section 111.
⟨ Resolve initialised data 111 ⟩ Used in section 110.
⟨ Resolve uninitialised data 112 ⟩ Used in section 110.
⟨ Scan a token 35 ⟩ Used in section 34.
⟨ Semantic subroutines 104 ⟩ Used in section 12.
⟨ Symbol subroutines 20, 21, 22 ⟩ Used in section 12.
⟨ Syntactic subroutines 49, 50, 52, 54, 55, 56, 58, 59, 60, 61, 62, 64, 65, 66, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 86, 87, 88, 89, 90, 91 ⟩ Used in section 12.
⟨ The odd case 63 ⟩ Used in section 62.
⟨ Type definitions 24, 26, 28, 42, 44, 47, 100, 108, 161 ⟩ Used in section 12.
⟨ Wrap up and fill the lexer variables 36 ⟩ Used in section 34.
⟨ Wrap up everything 103 ⟩ Used in section 12.
⟨ resolve ordinary global 107 ⟩ Used in section 105.
⟨ resolve special register 106 ⟩ Used in section 105.

PL/MMIX

	Section	Page
Preface	1	1
Introduction	2	2
Main program	12	8
Symbols	20	11
Lexis	28	16
Errors	39	20
Grammar	46	23
Expressions	92	42
The object file	98	45
Semantics	104	48
Code generation	162	68