The `ldb` interface

Sukyoung Ryu and Norman Ramsey

February 2005

Abstract

We present a reusable *compile-time support library* which helps a compiler meet its obligations to our debugger `ldb` without exposing language-dependent details. Our library does not force the compiler's private data structures to fit the debugger's model, does not require changing the timing or ordering of the compiler's phases, and does not artificially prolong the lifetimes of the compiler's internal data structures. The library enables source-level debugging for multiple languages at the cost of only modest programming effort.

To be used with a particular compiler, the library must provide an interface in the implementation language of that compiler. We have therefore designed the interface in two layers: an abstract layer that is independent of implementation language, and instances for four different implementation languages (C, Java, Standard ML, and Objective Caml). Each instance is backed up by an implementation.

Contents

Chapter 1

Introduction

Source-level debugging is made possible by an agreement between debugger an compiler. Under such an agreement, a compiler provides information about each program it compiles, and the debugger uses this information to give users a source-level view at run time. Information about a program is highly structured and may describe such elements as source-language types, variables, statements, functions, methods, and so on. The effort required to pass such information from compiler to debugger is greatly affected by the interface between the two.

This technical report describes an interface that is designed to minimize a compiler writer's effort. The ideas behind the interface are described in a companion conference paper (Ryu and Ramsey, 2005); the purpose of this report is to reveal the details.

The idea behind the interface is that in any given compiler, information about a program has natural, native representations, which we call *language-level objects*. Our debugger, by contrast, works with *debug-level objects*. The interface enables a compiler to create debug-level objects from language-level objects, and to communicate debug-level objects to the debugger. Because the interface is an API, not a disk format, it has to be expressed in an implementation language suitable for the compiler at hand. We have therefore designed and implemented four instances of the interface: for C, Java, Standard ML, and Objective Caml. Each instance is described in one chapter of this report.

Chapter 2

The `ldb` interface in C

2.1 Legend

We use the following naming conventions for the `ldb` interface in C:

Token	Naming convention	Example
Types	Begin with `Ldb`. Use embedded capital letters for multiword names.	`LdbArray`
Procedures	Begin with `ldb` followed by a type name without the `Ldb` prefix, an underscore, an initial lower case, and embedded capital letters for multiword names.	`ldbArray_addHigh`
Macros	Begin with `LDB`, followed by an underscore, a capitalized type name without the `Ldb` prefix, an underscore, and all upper cases.	`LDB_ARRAY_MAX`

The procedures in the `ldb` interface are implemented as methods of an object; the first argument of each observer procedure has the type of the object. For example, `ldbArray_addHigh` is a method of an object `LdbArray` and the first argument of `ldbArray_addHigh` has the type `LdbArray *`. The only exception is `LdbCompUnit`. Because a compilation unit is the top-level information, the `ldb` interface makes it an implicit part of the interface's state and the observers of the compilation unit do not have the compilation unit as their first arguments. In addition, a conversion function is defined in a subtype. For example, a conversion function between `ldbArray` and its super type `ldbInfo` is defined as `ldbArray_asInfo` instead of `ldbInfo_ofArray`.

Abbreviations are used only for the followings:

`arg`	argument
`comp`	compilation
`env`	environment
`ext`	external
`float`	floating-point numbers
`hex`	hexadecimal
`info`	information
`int`	integer
`loc`	location
`loci`	stopping points
`locus`	a stopping point
`plcmt`	placement
`proc`	procedure
`ps`	PostScript
`ref`	reference
`reg`	register
`src`	source
`str`	string
`var`	variable

2.2 Preliminaries

This section describes the preliminary `ldb` interface which should be the source-language independent, the compiler independent but implementation-language (C) dependent.

2.2.1 Utility functions

8a ⟨*utils.h* 8a⟩≡ 9▷

```
#include <assert.h>
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define __signed__ signed /* needed for linux */
#include <sys/time.h>
#include "ldbinfo.h"

⟨macros 10a⟩
```

Defines:
`__signed__`, never used.

8b ⟨*type definitions in ldbinfo.h* 8b⟩≡ (13a) 14a▷

```
typedef struct ldblist      *LdbList;
```

Defines:
`LdbList`, used in chunks 9, 11, 63b, 64, 66, 83, 95, 97, and 99b.

9 ⟨*utils.h* 8a⟩+≡ ◁8a

```
struct ldblist {
        void    *x;
        LdbList  link;
};
typedef int (*LdbPredicate) (void *closure, void *data);
```

⟨*abstractions in utils.h* 14b⟩

```
/* list ***************************************************************/
extern LdbList ldbList_append(LdbList l, void *x);
extern LdbList ldbList_find  (LdbList l, LdbPredicate p, void *closure);
extern int     ldbList_length(LdbList l);

/* string utilities ***************************************************/
extern unsigned    ldbUtil_hashStr (const char *str);
extern const char *ldbUtil_intToStr(int i);

/* abbreviated conversion functions ***********************************/
extern LdbInfo  *null2i  ();
extern LdbInfo  *int2i   (int         i);
extern LdbInfo  *str2i   (const char *s);
extern LdbInfo  *name2i  (const char *s);
extern LdbInfo  *array2i (LdbArray   *a);
extern LdbInfo  *symbol2i(LdbSymbol  *s);
extern LdbInfo  *proc2i  (LdbProc    *p);
extern LdbInfo  *type2i  (LdbType    *t);
extern LdbInfo  *env2i   (LdbEnv     *e);
extern LdbTable *symbol2t(LdbSymbol  *s);
extern LdbTable *type2t  (LdbType    *t);

#define ldbUtil_baseToExtRef(BASE) \
(LdbExtRef *)ldbUtil_makeLabelNameFrom((const char *)ldbExtRef_fromBase(BASE))
```

⟨*immutable.h* 28⟩
⟨*mutable.h* 42⟩
⟨*lazy address management* 82⟩
⟨*external-reference.h* 20a⟩
⟨*file-handle.h in utils.h* 16⟩

Defines:
`LdbPredicate`, used in chunk 11.
`ldbUtil_baseToExtRef`, used in chunks 28, 42, 74, and 80.
Uses `array2i` 11, `env2i` 11, `int2i` 11, `LdbExtRef` 20b 179, `ldbExtRef_fromBase` 21, `LdbList` 8b, `ldbList_append` 11, `ldbList_find` 11, `ldbList_length` 11, `ldbUtil_hashStr` 11, `ldbUtil_intToStr` 11, `name2i` 11, `null2i` 11, `proc2i` 11, `str2i` 11, `symbol2i` 11, `symbol2t` 11, `type2i` 11, and `type2t` 11.

10a ⟨*macros* 10a⟩≡ (8a) 20c▷

```
#define LDB_UTIL_HASHSIZE       1024
#define LDB_UTIL_MAXDIGIT       50
```

Defines:
LDB_UTIL_HASHSIZE, used in chunks 11 and 28.
LDB_UTIL_MAXDIGIT, used in chunks 11 and 97.

10b ⟨*included files* 10b⟩≡ (11 17 21 25 29 31a 33a 34b 36a 37b 39 43 46 49 52a 54a 56a 58 61 66 71 74 77b 80 86 89 94c)

```
#include "utils.h"
```

11 ⟨*utils.c* 11⟩≡

⟨*included files* 10b⟩

```
static LdbList freenodes;          /* free list nodes */

/* ldbList_append - append x to list, return new list */
LdbList ldbList_append(LdbList list, void *x) {
        LdbList new;

        if ((new = freenodes) != NULL)
                freenodes = freenodes->link;
        else
                new = (LdbList)malloc(sizeof *new);
        if (list) {
                new->link = list->link;
                list->link = new;
        } else
                new->link = new;
        new->x = x;
        return new;
}

/* ldbList_find - search a list for an item matching a predicate */
LdbList ldbList_find(l, f, closure) LdbList l; LdbPredicate f; void *closure; {
        LdbList p = l;
        if (p)
                do {
                        if (f(closure,p->x)) return p;
                        else p = p->link;
                } while (p != l);
        return 0;
}

/* ldbList_length - # elements in list */
int ldbList_length(l) LdbList l; {
        int n = 0;

        if (l) {
                LdbList lp = l;
                do
                        n++;
                while ((lp = lp->link) != l);
        }
        return n;
}
```

```
/* ldbUtil_hashStr - hash a given string */
unsigned ldbUtil_hashStr(string) const char *string; {
        register unsigned int i = 0;
        while (*string) i = (i << 2) + *string++;
        return (i % LDB_UTIL_HASHSIZE);
}

/* ldbUtil_intToStr - return a string representaion of a given integer */
const char *ldbUtil_intToStr(i) int i; {
        char *s = malloc(LDB_UTIL_MAXDIGIT);
        sprintf(s, "%d", i);
        return s;
}

LdbInfo  *null2i  ()                 { return ldbNull_asInfo(ldbNull_make()) ;}
LdbInfo  *int2i   (i) int         i; { return ldbInt_asInfo (ldbInt_make (i));}
LdbInfo  *str2i   (s) const char *s; { return ldbStr_asInfo (ldbStr_make (s));}
LdbInfo  *name2i  (s) const char *s; { return ldbName_asInfo(ldbName_make(s));}
LdbInfo  *array2i (a) LdbArray   *a; { return ldbArray_asInfo(a);           }
LdbInfo  *proc2i  (p) LdbProc    *p; { return symbol2i(ldbProc_asSymbol(p));}
LdbInfo  *env2i   (e) LdbEnv     *e; { return ldbEnv_asInfo(e);             }
LdbTable *symbol2t(s) LdbSymbol  *s; { return ldbSymbol_asTable(s);         }
LdbTable *type2t  (t) LdbType    *t; { return ldbType_asTable(t);           }
#define ToI(TOT, I) if (I != NULL) return ldbTable_asInfo(TOT(I)); \
                    else           return null2i();
LdbInfo  *symbol2i(s) LdbSymbol  *s; { ToI(symbol2t, s); }
LdbInfo  *type2i  (t) LdbType    *t; { ToI(type2t  , t); }
```

Defines:
array2i, used in chunks 9, 66, 71, and 100a.
env2i, used in chunks 9, 66, and 74.
freenodes, never used.
int2i, used in chunks 9, 56a, and 66.
ldbList_append, used in chunks 9, 66, 83, and 100a.
ldbList_find, used in chunk 9.
ldbList_length, used in chunk 9.
ldbUtil_hashStr, used in chunks 9 and 28.
ldbUtil_intToStr, used in chunks 9, 46, and 49.
name2i, used in chunks 9, 42, 49, 74, and 97.
null2i, used in chunk 9.
proc2i, used in chunks 9, 95, and 100a.
str2i, used in chunks 9, 56a, 66, 77b, and 97.
symbol2i, used in chunks 9, 66, 71, 74, and 95.
symbol2t, used in chunks 9, 58, 61, 66, and 71.
ToI, never used.
type2i, used in chunks 9, 58, 61, 66, and 71.
type2t, used in chunk 9.

Uses LDB_UTIL_HASHSIZE 10a, LDB_UTIL_MAXDIGIT 10a, ldbArray_asInfo 46, ldbEnv_asInfo 74, ldbInt_asInfo 33a, ldbInt_make 33a, LdbList 8b, ldbName_asInfo 36a, ldbName_make 36a, ldbNull_asInfo 31a, ldbNull_make 31a, LdbPredicate 9, ldbProc_asSymbol 66, ldbStr_asInfo 34b, ldbStr_make 34b, ldbSymbol_asTable 56a, ldbTable_asInfo 49,

`ldbType_asTable` 77b, and `n` 21 179 186.

2.2.2 `ldb` information for clients

We provide a version for clients in which all the types are opaque.

13a ⟨*ldbinfo.h* 13a⟩≡

```
/** type definitions ******************************************************/
```

⟨*type definitions in ldbinfo.h* 8b⟩

⟨*procedure declarations in ldbinfo.h* 13b⟩

13b ⟨*procedure declarations in ldbinfo.h* 13b⟩≡ (13a)

⟨*file-handle.h* 15⟩
⟨*compiler-support.h* 22⟩
⟨*info.h* 24⟩
⟨*ldb-null.h* 30c⟩
⟨*integer.h* 32⟩
⟨*ldb-string.h* 34a⟩
⟨*name.h* 35c⟩
⟨*psproc.h* 37a⟩
⟨*pscode.h* 38c⟩
⟨*array.h* 45⟩
⟨*table.h* 48⟩
⟨*srcloc.h* 51⟩
⟨*value.h* 53⟩
⟨*symbol.h* 55b⟩
⟨*constant.h* 57b⟩
⟨*variable.h* 60⟩
⟨*procedure.h* 64⟩
⟨*type-symbol.h* 70⟩
⟨*env.h* 73⟩
⟨*type.h* 77a⟩
⟨*placement.h* 79⟩
⟨*label.h* 85⟩
⟨*locus.h* 88⟩
⟨*compilation-unit.h* 93⟩

2.2.3 File handle

- Abstraction : a file handle to control emitting debugging information
- Constructors:
  - `ldbFile_open`: open a specified file
- Observers :
  - `ldbFile_close`: close the open file
  - `ldbFile_write`: write a given string to the file
  - `ldbFile_push`: increase the current level
  - `ldbFile_pop`: decrease the current level
  - `ldbFile_assertTop`: make sure that the current level is 'top level'

`ldbFile_push`, `ldbFile_pop`, and `ldbFile_assertTop` are provided to maintain the current level of a file handle. So that the `ldb` interface can keep the following invariance:

- When a debugging information is to be emitted, the file handle is at 'top level'.

14a ⟨*type definitions in ldbinfo.h* 8b⟩+≡ (13a) ◁8b 20b▷

```
typedef struct ldbfile       LdbFile;
```

14b ⟨*abstractions in utils.h* 14b⟩≡ (9) 23c▷

```
/** abstraction : file handle ****************************************/
/* a file handle to control emitting debugging information;
 * include an open file stream and its current level
 */
struct ldbfile {
        FILE *out;
        int   level; /* current level */
};
```

Uses `out` 94a.

15 ⟨*file-handle.h* 15⟩≡ (13b)

```
/** file handle ***********************************************************/
/** constructors **********************************************************/
/* open a specified file
 *
 * precondition : file is not open
 * postcondition: file is open and its default current level is 0
 */
extern LdbFile *ldbFile_open(const char *filename);

/** observers *************************************************************/
/* close the open file
 *
 * precondition : file is open
 * postcondition: file is closed
 */
extern void ldbFile_close(LdbFile *out);

/* write a string to the file
 *
 * precondition : file is open
 * postcondition: string is written to the file
 */
extern void ldbFile_write(LdbFile *out, const char *str, ...);
```

Uses `ldbFile_close` 17, `ldbFile_open` 17, `ldbFile_write` 17, and `out` 94a.

16 ⟨*file-handle.h in utils.h* 16⟩≡ (9)

```
/** file handle ***********************************************************/
/* increase current level
 *
 * precondition : current level is equal or greater than 0
 * postcondition: current level is increased and greater than 0
 */
extern void ldbFile_push(LdbFile *out);

/* decrease current level
 *
 * precondition : current level is greater than 0
 * postcondition: current level is decreased and equal or greater than 0
 */
extern void ldbFile_pop(LdbFile *out);

/* make sure that the current level is top level
 *
 * precondition : current level is equal or greater than 0
 * postcondition: if the current level is not 0, assertion error
 */
extern void ldbFile_assertTop(LdbFile *out);
```

Uses `equal` 100a, `ldbFile_assertTop` 17, `ldbFile_pop` 17, `ldbFile_push` 17, and `out` 94a.

17 ⟨*file-handle.c* 17⟩≡
⟨*included files* 10b⟩

```
/* open  a specified file */
LdbFile *ldbFile_open (filename) const char *filename; {
        LdbFile *new = (LdbFile *)malloc(sizeof *new);
        new->out     = fopen(filename, "w+rr");
        new->level   = 0;
        return new;
}
/* close a specified file */
void ldbFile_close(out) LdbFile *out; {
        fclose(out->out);
}

static char *outs(const char *str, FILE *f, char *bp) {
        if (f)
                fputs(str, f);
        else
                while ((*bp = *str++))
                        bp++;
        return bp;
}

static char *outd(long n, FILE *f, char *bp) {
        unsigned long m;
        char buf[25], *s = buf + sizeof buf;

        *--s = '\0';
        if (n < 0)
                m = -n;
        else
                m = n;
        do
                *--s = m%10 + '0';
        while ((m /= 10) != 0);
        if (n < 0)
                *--s = '-';
        return outs(s, f, bp);
}

static char *outu(unsigned long n, int base, FILE *f, char *bp) {
        char buf[25], *s = buf + sizeof buf;

        *--s = '\0';
        do
```

```
                *--s = "0123456789abcdef"[n%base];
        while ((n /= base) != 0);
        return outs(s, f, bp);
}

/* vfprint - formatted output to f or string bp */
static void vfprint(FILE *f, char *bp, const char *fmt, va_list ap) {
        for (; *fmt; fmt++)
                if (*fmt == '%')
                        switch (*++fmt) {
                        case 'd': bp = outd(va_arg(ap, int), f, bp); break;
                        case 'D': bp = outd(va_arg(ap, long), f, bp); break;
                        case 'U': bp = outu(va_arg(ap, unsigned long), 10, f, bp); br
                        case 'u': bp = outu(va_arg(ap, unsigned), 10, f, bp); break;
                        case 'o': bp = outu(va_arg(ap, unsigned), 8, f, bp); break;
                        case 'X': bp = outu(va_arg(ap, unsigned long), 16, f, bp); br
                        case 'x': bp = outu(va_arg(ap, unsigned), 16, f, bp); break;
                        case 'f': case 'e':
                        case 'g': {
                                        static char format[] = "%f";
                                        char buf[128];
                                        format[1] = *fmt;
                                        sprintf(buf, format, va_arg(ap, double));
                                        bp = outs(buf, f, bp);
                                  } break;
                        case 's': bp = outs(va_arg(ap, char *), f, bp); break;
                        case 'p': {
                                  void *p = va_arg(ap, void *);
                                  if (p)
                                          bp = outs("0x", f, bp);
                                  bp = outu((unsigned long)p, 16, f, bp);
                                  break;
                                  }
                        case 'c': if (f) fputc(va_arg(ap, int), f);
                                  else *bp++ = va_arg(ap, int);
                                  break;
                        case 'S': { char *s = va_arg(ap, char *);
                                    int n = va_arg(ap, int);
                                    if (s) {
                                        for ( ; n-- > 0; s++)
                                                if (f) (void)putc(*s, f);
                                                else *bp++ = *s;
                                    }
                                  } break;
                        case 'I': { int n = va_arg(ap, int);
                                    while (--n >= 0)
```

```
                                        if (f) (void)putc(' ', f);
                                        else *bp++ = ' ';
                                } break;
                        default:  if (f) (void)putc(*fmt, f);
                                  else *bp++ = *fmt; break;
                        }
                else if (f)
                        (void)putc(*fmt, f);
                else
                        *bp++ = *fmt;
        if (!f)
                *bp = '\0';
}
/* write a specified string to this file */
void ldbFile_write(LdbFile *out, const char *fmt, ...) {
        va_list ap;

        va_start(ap, fmt);
        vfprint(out->out, NULL, fmt, ap);
        va_end(ap);
}
/* increase current level */
void ldbFile_push(out) LdbFile *out; {
        if (out != NULL) out->level++;
}
/* decrease current level */
void ldbFile_pop(out) LdbFile *out; {
        if (out != NULL) out->level--;
}
/* make sure that the current level is top level */
void ldbFile_assertTop(out) LdbFile *out; {
        if (out != NULL) assert(out->level == 0);
}
```

Defines:
`ldbFile_assertTop`, used in chunks 16 and 25.
`ldbFile_close`, used in chunk 15.
`ldbFile_open`, used in chunk 15.
`ldbFile_pop`, used in chunks 16, 42, and 74.
`ldbFile_push`, used in chunks 16, 42, and 74.
`ldbFile_write`, used in chunks 15 and 94b.
`outd`, never used.
`outs`, never used.
`outu`, never used.
`vfprint`, never used.
Uses `n` 21 179 186 and `out` 94a.

2.2.4 External references

- Abstraction : a reference to an item that has been externalized
- Constructors:
  - `ldbExtRef_fromBase`: create an external reference from a base name and a generated number

External references satisfy the following invariances:

- An external reference of every immutable information is its string representation.
- An external reference of every mutable information is a uniquely generated name.

20a ⟨*external-reference.h* 20a⟩≡ (9)

```
/** external reference ************************************************/
/** constructors ******************************************************/
/* create an external reference from a base name
 *
 * postcondition: this external reference's string representation is
 *                its generated name which is the given base name
 *                appended by its unique index
 */
extern LdbExtRef *ldbExtRef_fromBase(const char *base);
```

Uses `LdbExtRef` 20b 179 and `ldbExtRef_fromBase` 21.

20b ⟨*type definitions in ldbinfo.h* 8b⟩+≡ (13a) ◁14a 23b▷

```
typedef const char *LdbExtRef;
```

Defines:
`LdbExtRef`, used in chunks 9, 20a, 21, 23–25, 27, 28, 31a, 33a, 34b, 36a, 37b, 39, 41–43, 46, 49, 54a, 72b, 74, 80, 184, and 185.

20c ⟨*macros* 10a⟩+≡ (8a) ◁10a 23a▷

```
#define LDB_EXTREF_MAX          300
```

Defines:
`LDB_EXTREF_MAX`, used in chunks 21, 28, 80, 83, and 97.

21 ⟨*external-reference.c* 21⟩≡
⟨*included files* 10b⟩

```
static int n = 0; /* integer value to form an external name */
LdbExtRef *ldbExtRef_fromBase(base) const char *base; {
        char *s = malloc(LDB_EXTREF_MAX);
        sprintf(s, "%s%d", base, ++n);
        return (LdbExtRef *)s;
}
```

Defines:
`ldbExtRef_fromBase`, used in chunks 9 and 20a.
`n`, used in chunks 11, 17, 30c, 31a, 44–47, 49, 95–97, 100a, and 200.
Uses `LDB_EXTREF_MAX` 20c and `LdbExtRef` 20b 179.

2.2.5 Compiler supports

Utility functions from a compiler for `ldb` interface.

22 ⟨*compiler-support.h* 22⟩≡ (13b)

```
/** compiler supports ***************************************************/
/* labels ***************************************************************/
/* return a fresh compiler/platform-specific label name */
extern const char *ldbUtil_makeLabelName();

/* return a fresh compiler/platform-specific label name from a given name */
extern const char *ldbUtil_makeLabelNameFrom(const char *name);

/* return a base name from a given compiler/platform-specific label name */
extern const char *ldbUtil_dropPrefixOfLabelName(const char *lab);

/* return a no-op format to tag a given stopping point's name in the
 * assembly-language output */
extern const char *ldbUtil_noOpFormat(const char *name);

/* assembly output *****************************************************/
/* emit a given label name to a given file handle */
extern void ldbUtil_emitLabel  (const char *lab , LdbFile *out);

/* emit a given string to a given file handle */
extern void ldbUtil_emitStr    (const char *str , LdbFile *out);

/* emit a given data name to a given file handle */
extern void ldbUtil_emitName   (const char *name, LdbFile *out);

/* export a given anchor symbol name to a given file handle */
extern void ldbUtil_exportName (const char *name, LdbFile *out);

/* emit a given section name to a given file handle */
extern void ldbUtil_emitSection(const char *secn, LdbFile *out);

/* emit a data section */
extern void ldbUtil_emitDataSection(LdbFile *out);

/* private data ********************************************************/
/* emit compiler's private data possibly for its expression server */
extern void ldbUtil_emitPrivate(LdbFile *out);
```

Uses `out` 94a.

23a ⟨*macros* 10a⟩+≡ (8a) ◁20c 44b▷

```
#define LDB_UTIL_MAXLABELNAME   50
```

Defines:
LDB_UTIL_MAXLABELNAME, used in chunk 83.

2.3 Generic PostScript data

This section includes the ldb interface for generic PostScript data which should be the source-language independent, the compiler independent, but implementation-language (C) dependent.

2.3.1 PostScript data (ldb information)

- Abstraction : the base information for all PostScript data
- Constructors:
  - ldbInfo_make: create a fresh information
- Observers :
  - ldbInfo_extern: externalize this information
  - ldbInfo_ref: return this information's external reference

An ldb information satisfies the following invariance:

* ldbInfo_extern is called after the information has been created.
* ldbInfo_ref is called after the information has been externalized.

For immutable information, ldbInfo_extern does nothing and the constructor of the information sets the external reference. Thus, ldbInfo_ref can be called right after the immutable information is created. For mutable information, ldbInfo_extern first sets the external reference, externalizes every element of it, and then emits itself. Thus, ldbInfo_ref can be called after ldbInfo_extern called and it's okay whether ldbInfo_extern has not been returned.

23b ⟨*type definitions in ldbinfo.h* 8b⟩+≡ (13a) ◁20b 26▷

```
typedef struct ldbinfo       LdbInfo;
```

23c ⟨*abstractions in utils.h* 14b⟩+≡ (9) ◁14b 27▷

```
/** abstraction : info ***************************************************/
/* the base type for all ldb information */
struct ldbinfo {
        LdbInfo   *i;
        void       (*ext)(LdbInfo *self, LdbFile *out);
        LdbExtRef *(*ref)(LdbInfo *self);
};
```

Uses LdbExtRef 20b 179 and out 94a.

24 ⟨*info.h* 24⟩≡ (13b)

```
/** info ***************************************************************/
/** constructors *******************************************************/
/* create a fresh information
 *
 * precondition : this information does not exist
 * postcondition: this information is created
 *
*/
extern LdbInfo *ldbInfo_make();

/** observers **********************************************************/
/* externalize a specified information
 *
 * precondition : file handle must be at top level
 * postcondition: file handle is still at top level,
 *                and this information has been externalized
 */
extern void ldbInfo_extern(LdbInfo *i, LdbFile *out);

/* return a specified information's external reference
 *
 * precondition : a given information has already been externalized
 * postcondition: this external reference can be written anywhere
 *                in a file handle
 */
extern LdbExtRef *ldbInfo_ref(LdbInfo *i);
```

Uses `LdbExtRef` 20b 179, `ldbInfo_extern` 25, `ldbInfo_make` 25, `ldbInfo_ref` 25, and `out` 94a.

25 ⟨*info.c* 25⟩≡

⟨*included files* 10b⟩

```
LdbInfo *ldbInfo_make() {
        LdbInfo *new = (LdbInfo *)malloc(sizeof *new);
        return new;
}
void ldbInfo_extern(i, out) LdbInfo *i; LdbFile *out; {
        ldbFile_assertTop(out);
        i->i->ext(i, out);
}
LdbExtRef *ldbInfo_ref(i) LdbInfo *i; { return i->i->ref(i); }
```

Defines:
`ldbInfo_extern`, used in chunks 24, 42, 46, 49, 74, 96c, and 97.
`ldbInfo_make`, used in chunks 24, 28, 42, 74, and 80.
`ldbInfo_ref`, used in chunks 24, 42, 46, 49, 74, and 97.
Uses `LdbExtRef` 20b 179, `ldbFile_assertTop` 17, and `out` 94a.

2.3.2 Immutable PostScript data

- Abstraction : an immutable PostScript data
- Constructors:
  - `ldbImmutable_make`: create a fresh immutable PostScript data
- Observers :
  - `ldbImmutable_extern`: externalize this immutable information
  - `ldbImmutable_ref`: return this immutable information's external reference
  - `ldbImmutable_asInfo` : return a given immutable information as an `ldb` information (subsumption)

Each immutable information is externalized by `ldbImmutable_extern` as follows. If it's not yet externalized, its label is emitted followed by `0` to denote that this information is immutable and then the label for the string representation of the immutable information is emitted. The string representation and its label are emitted before the immutable information is emitted.

For example, `(test.c)` is externalized as follows:

```
.LIg71:
.byte 40
.byte 116
.byte 101
.byte 115
.byte 116
.byte 46
.byte 99
.byte 41
.byte 0
.LIm12:
.long 0
.long .LIg71
```

Its label `.LIm12` is emitted followed by `0` and `.LIg71` which represents the string `(test.c)`.

26 ⟨*type definitions in ldbinfo.h* 8b⟩+≡ (13a) ◁23b 30a▷

```
typedef struct ldbimmutable  LdbImmutable;
```

27 ⟨*abstractions in utils.h* 14b⟩+≡ (9) ◁23c 30b▷

```
/** abstraction : immutable ***********************************************/
struct ldbimmutable {
        LdbExtRef  *ref;     /* external reference */
        const char *def;     /* definition */
        int externed; /* whether this immutable information is externalized */
};
```

Uses `LdbExtRef` 20b 179.

28 ⟨*immutable.h* 28⟩≡ (9)

```
/** immutable info ********************************************************/
struct entry {
        const char   *def;
        void         *imm;
        struct entry *link;
} *htab[LDB_UTIL_HASHSIZE];

/** constructors **********************************************************/
/* create a fresh immutable information
 *
 * postcondition: this immutable information is created
 */
#define ldbImmutable_make(IMMUTABLE,EXT,REF,FMT,ARG) \
        char *s = malloc(LDB_EXTREF_MAX); \
        sprintf(s, FMT, ARG); \
        { struct entry *p; \
          unsigned h = (ldbUtil_hashStr(s))&(LDB_UTIL_HASHSIZE-1); \
          for (p = htab[h]; p; p = p->link) \
                if (!(strcmp(p->def, s))) return (IMMUTABLE)p->imm; \
          { IMMUTABLE new   = (IMMUTABLE)     malloc(sizeof *new); \
            LdbImmutable *m = (LdbImmutable *)malloc(sizeof *m); \
            m->ref = ldbUtil_baseToExtRef("Im"); \
            m->def      = s; \
            m->externed = 0; \
            new->m      = m; \
            new->i      = ldbInfo_make(); \
            new->i->ext = EXT; \
            new->i->ref = REF; \
            p = (struct entry *)malloc(sizeof *p); \
            p->link = htab[h]; \
            htab[h] = p; \
            p->def  = s; \
            p->imm  = new; \
            return new; \
          } \
        }

/** observers **************************************************************/
/* externalize a specified immutable information
 *
 * precondition : file handle must be at top level
 * postcondition: file handle is still at top level,
 *                and this immutable information has been externalized
 */
#define ldbImmutable_extern(IMMUTABLE,OUT) \
```

```
        if (!IMMUTABLE->externed) { \
                LdbExtRef *defLabel = ldbUtil_baseToExtRef("Ig"); \
                ldbUtil_emitLabel((const char *)defLabel      , OUT); \
                ldbUtil_emitStr  (IMMUTABLE->def              , OUT); \
                ldbUtil_emitLabel((const char *)IMMUTABLE->ref, OUT); \
                ldbUtil_emitName ("0"                         , OUT); \
                ldbUtil_emitName ((const char *)defLabel      , OUT); \
                IMMUTABLE->externed = 1; \
        }

  /* return a specified immutable information's external reference
   *
   * precondition : a given immutable information has already been externalized
   * postcondition: this external reference can be written anywhere
   *                in a file handle
   */
  extern const char *ldbImmutable_ref(LdbImmutable *m);

  /* return a given immutable information as an ldb information (subsumption)
   *
   * precondition : true
   */
  #define ldbImmutable_asInfo(IMMUTABLE) return (LdbInfo *)IMMUTABLE;
```

Defines:
htab, never used.
ldbImmutable_asInfo, used in chunks 31a, 33a, 34b, 36a, 37b, 39, and 80.
ldbImmutable_extern, used in chunks 31a, 33a, 34b, 36a, 37b, 39, 54a, and 80.
ldbImmutable_make, used in chunks 31a, 33a, 34b, 36a, 37b, 39, and 54a.
Uses LDB_EXTREF_MAX 20c, LDB_UTIL_HASHSIZE 10a, LdbExtRef 20b 179, ldbImmutable_ref 29, ldbInfo_make 25, ldbUtil_baseToExtRef 9, and ldbUtil_hashStr 11.

29 ⟨*immutable.c* 29⟩≡
⟨*included files* 10b⟩

```
  const char *ldbImmutable_ref(m) LdbImmutable *m; { assert(m->externed == 1);
                                                      return m->ref; }
```

Defines:
ldbImmutable_ref, used in chunk 28.

2.3.3 PostScript null objects

- Abstraction : a PostScript null
- Constructors:
  - `ldbNull_make`: create a PostScript null object
- Observers :
  - `ldbNull_asInfo` : return a given PostScript null object as an `ldb` information (subsumption)

30a ⟨*type definitions in ldbinfo.h* 8b⟩+≡ (13a) ◁26 31b▷

```
typedef struct ldbnull       LdbNull;
```

30b ⟨*abstractions in utils.h* 14b⟩+≡ (9) ◁27 31c▷

```
/** abstraction : null ************************************************/
struct ldbnull {
        LdbInfo      *i; /* asInfo - subsumption */
        LdbImmutable *m;
};
```

30c ⟨*ldb-null.h* 30c⟩≡ (13b)

```
/** null ***************************************************************/
/** constructors *******************************************************/
/* create a PostScript null object
 *
 * precondition : true
 *
*/
extern LdbNull *ldbNull_make();

/** observers **********************************************************/
/* return a given PostScript null as an ldb information (subsumption)
 *
 * precondition : true
 */
extern LdbInfo *ldbNull_asInfo(LdbNull *n);
```

Uses `ldbNull_asInfo` 31a, `ldbNull_make` 31a, and `n` 21 179 186.

31a ⟨*ldb-null.c* 31a⟩≡
⟨*included files* 10b⟩

```
static void       null_extern(n, out) LdbNull *n; LdbFile *out; {
        ldbImmutable_extern(n->m, out);
}
static LdbExtRef *null_ref(n)         LdbNull *n; { return n->m->ref; }

LdbNull *ldbNull_make() {
        ldbImmutable_make(LdbNull *, &null_extern, &null_ref, "null%s", "");
}
LdbInfo *ldbNull_asInfo(n) LdbNull *n; { ldbImmutable_asInfo(n); }
```

Defines:
`ldbNull_asInfo`, used in chunks 11 and 30c.
`ldbNull_make`, used in chunks 11 and 30c.
`null_extern`, never used.
`null_ref`, never used.
Uses `LdbExtRef` 20b 179, `ldbImmutable_asInfo` 28, `ldbImmutable_extern` 28, `ldbImmutable_make` 28, `n` 21 179 186, and `out` 94a.

2.3.4 PostScript integers

- Abstraction : a PostScript integer
- Constructors:
  - `ldbInt_make`: create a fresh PostScript decimal integer
  - `ldbInt_makeHex`: create a fresh PostScript hexadecimal integer
- Observers :
  - `ldbInt_asInfo` : return a given PostScript integer as an `ldb` information (subsumption)

31b ⟨*type definitions in ldbinfo.h* 8b⟩+≡ (13a) ◁30a 33b▷

```
typedef struct ldbint        LdbInt;
```

31c ⟨*abstractions in utils.h* 14b⟩+≡ (9) ◁30b 33c▷

```
/** abstraction : integer ************************************************/
struct ldbint {
        LdbInfo      *i; /* asInfo - subsumption */
        LdbImmutable *m;
};
```

32 ⟨*integer.h* 32⟩≡ (13b)

```
/** integer ***********************************************************/
/** constructors ******************************************************/
/* create a fresh PostScript decimal integer
 *
 * precondition : this decimal integer does not exist
 * postcondition: this decimal integer is created
 *
 */
extern LdbInt *ldbInt_make(int i);

/* create a fresh PostScript hexadecimal integer
 *
 * precondition : this hexadecimal integer does not exist
 * postcondition: this hexadecimal integer is created
 *
 */
extern LdbInt *ldbInt_makeHex(int i);

/** observers *********************************************************/
/* return a given PostScript integer as an ldb information (subsumption)
 *
 * precondition : true
 */
extern LdbInfo *ldbInt_asInfo(LdbInt *i);
```

Uses `ldbInt_asInfo` 33a, `ldbInt_make` 33a, and `ldbInt_makeHex` 33a.

33a ⟨*integer.c* 33a⟩≡
⟨*included files* 10b⟩

```
static void       int_extern(i, out) LdbInt *i; LdbFile *out; {
        ldbImmutable_extern(i->m, out);
}
static LdbExtRef *int_ref(i)         LdbInt *i; { return i->m->ref; }

LdbInt *ldbInt_make(i) int i; {
        ldbImmutable_make(LdbInt *, &int_extern, &int_ref, "%d"   , i);
}
LdbInt *ldbInt_makeHex(i)     int i; {
        ldbImmutable_make(LdbInt *, &int_extern, &int_ref, "16#%x", i);
}
LdbInfo *ldbInt_asInfo(i) LdbInt *i; { ldbImmutable_asInfo(i); }
```

Defines:
`int_extern`, never used.
`int_ref`, never used.
`ldbInt_asInfo`, used in chunks 11, 32, and 66.
`ldbInt_make`, used in chunks 11 and 32.
`ldbInt_makeHex`, used in chunks 32 and 66.
Uses `LdbExtRef` 20b 179, `ldbImmutable_asInfo` 28, `ldbImmutable_extern` 28, `ldbImmutable_make` 28, and `out` 94a.

2.3.5 PostScript strings

- Abstraction : a PostScript string
- Constructors:
  - `ldbStr_make`: create a fresh PostScript string
- Observers :
  - `ldbStr_asInfo` : return a given PostScript string as an `ldb` information (subsumption)

33b ⟨*type definitions in ldbinfo.h* 8b⟩+≡ (13a) ◁31b 35a▷

```
typedef struct ldbstr        LdbStr;
```

33c ⟨*abstractions in utils.h* 14b⟩+≡ (9) ◁31c 35b▷

```
/** abstraction : string ***********************************************/
struct ldbstr {
        LdbInfo      *i; /* asInfo - subsumption */
        LdbImmutable *m;
};
```

34a ⟨*ldb-string.h* 34a⟩≡ (13b)

```
/** string ************************************************************/
/** constructors ******************************************************/
/* create a fresh PostScript string
 *
 * precondition : this string does not exist
 * postcondition: this string is created
 *
*/
extern LdbStr *ldbStr_make(const char *s);

/** observers *********************************************************/
/* return a given PostScript string as an ldb information (subsumption)
 *
 * precondition : true
 */
extern LdbInfo *ldbStr_asInfo(LdbStr *s);
```

Uses `ldbStr_asInfo` 34b and `ldbStr_make` 34b.

34b ⟨*ldb-string.c* 34b⟩≡

⟨*included files* 10b⟩

```
static void       str_extern(s, out) LdbStr *s; LdbFile *out; {
        ldbImmutable_extern(s->m, out);
}
static LdbExtRef *str_ref(s)         LdbStr *s; { return s->m->ref; }

LdbStr *ldbStr_make(str) const char *str; {
        ldbImmutable_make(LdbStr *, &str_extern, &str_ref, "(%s)", str);
}
LdbInfo *ldbStr_asInfo(s) LdbStr *s; { ldbImmutable_asInfo(s); }
```

Defines:
`ldbStr_asInfo`, used in chunks 11 and 34a.
`ldbStr_make`, used in chunks 11 and 34a.
`str_extern`, never used.
`str_ref`, never used.

Uses `LdbExtRef` 20b 179, `ldbImmutable_asInfo` 28, `ldbImmutable_extern` 28, `ldbImmutable_make` 28, and `out` 94a.

2.3.6 PostScript names

- Abstraction : a PostScript name
- Constructors:
  - `ldbName_make`: create a fresh PostScript name
- Observers :
  - `ldbName_asInfo` : return a given PostScript name as an `ldb` information (subsumption)

35a ⟨*type definitions in ldbinfo.h* 8b⟩+≡ (13a) ◁33b 36b▷

```
typedef struct ldbname       LdbName;
```

35b ⟨*abstractions in utils.h* 14b⟩+≡ (9) ◁33c 36c▷

```
/** abstraction : name ************************************************/
struct ldbname {
        LdbInfo      *i; /* asInfo - subsumption */
        LdbImmutable *m;
};
```

35c ⟨*name.h* 35c⟩≡ (13b)

```
/** name ***************************************************************/
/** constructors *******************************************************/
/* create a fresh PostScript name
 *
 * precondition : this name does not exist
 * postcondition: this name is created
 *
*/
extern LdbName *ldbName_make(const char *s);

/** observers **********************************************************/
/* return a given PostScript name as an ldb information (subsumption)
 *
 * precondition : true
 */
extern LdbInfo *ldbName_asInfo(LdbName *s);
```

Uses `ldbName_asInfo` 36a and `ldbName_make` 36a.

36a ⟨*name.c* 36a⟩≡
⟨*included files* 10b⟩

```
static void       name_extern(s, out) LdbName *s; LdbFile *out; {
        ldbImmutable_extern(s->m, out);
}
static LdbExtRef *name_ref(s)         LdbName *s; { return s->m->ref; }

LdbName *ldbName_make(str) const char *str; {
        ldbImmutable_make(LdbName *, &name_extern, &name_ref, "%s", str);
}
LdbInfo *ldbName_asInfo(s) LdbName *s; { ldbImmutable_asInfo(s); }
```

Defines:
`ldbName_asInfo`, used in chunks 11, 35c, and 49.
`ldbName_make`, used in chunks 11, 35c, and 49.
`name_extern`, never used.
`name_ref`, never used.
Uses `LdbExtRef` 20b 179, `ldbImmutable_asInfo` 28, `ldbImmutable_extern` 28, `ldbImmutable_make` 28, and `out` 94a.

2.3.7 PostScript procedures

- Abstraction : a PostScript procedure
- Constructors:
  - `ldbPsProc_make`: create a fresh PostScript procedure
- Observers :
  - `ldbPsProc_asInfo` : return a given PostScript procedure as an `ldb` information (subsumption)

36b ⟨*type definitions in ldbinfo.h* 8b⟩+≡ (13a) ◁35a 38a▷

```
typedef struct ldbpsproc     LdbPsProc;
```

36c ⟨*abstractions in utils.h* 14b⟩+≡ (9) ◁35b 38b▷

```
/** abstraction : PostScript code ***************************************/
struct ldbpsproc {
        LdbInfo      *i; /* asInfo - subsumption */
        LdbImmutable *m;
};
```

37a ⟨*psproc.h* 37a⟩≡ (13b)

```
/** PostScript code ***********************************************************/
/** constructors **************************************************************/
/* create a fresh PostScript code
 *
 * precondition : this PostScript code does not exist
 * postcondition: this PostScript code is created
 *
*/
extern LdbPsProc *ldbPsProc_make(const char *s);

/** observers *****************************************************************/
/* return a given PostScript code as an ldb information (subsumption)
 *
 * precondition : true
 */
extern LdbInfo *ldbPsProc_asInfo(LdbPsProc *s);
```

Uses `ldbPsProc_asInfo` 37b and `ldbPsProc_make` 37b.

37b ⟨*psproc.c* 37b⟩≡

⟨*included files* 10b⟩

```
static void       psproc_extern(s, out) LdbPsProc *s; LdbFile *out; {
        ldbImmutable_extern(s->m, out);
}
static LdbExtRef *psproc_ref(s)         LdbPsProc *s; { return s->m->ref; }

LdbPsProc *ldbPsProc_make(str) const char *str; {
        ldbImmutable_make(LdbPsProc *,&psproc_extern,&psproc_ref,"{%s}",str);
}
LdbInfo *ldbPsProc_asInfo(s) LdbPsProc *s; { ldbImmutable_asInfo(s); }
```

Defines:
`ldbPsProc_asInfo`, used in chunks 37a and 77b.
`ldbPsProc_make`, used in chunk 37a.
`psproc_extern`, never used.
`psproc_ref`, never used.
Uses `LdbExtRef` 20b 179, `ldbImmutable_asInfo` 28, `ldbImmutable_extern` 28, `ldbImmutable_make` 28, and `out` 94a.

2.3.8 PostScript codes

- Abstraction : a PostScript code
- Constructors:
  - `ldbPsCode_make`: create a fresh PostScript code
- Observers :
  - `ldbPsCode_asInfo` : return a given PostScript code as an `ldb` information (subsumption)

38a ⟨*type definitions in ldbinfo.h* 8b⟩+≡ (13a) ◁36b 41a▷

```
typedef struct ldbpscode     LdbPsCode;
```

38b ⟨*abstractions in utils.h* 14b⟩+≡ (9) ◁36c 41b▷

```
/** abstraction : PostScript code ****************************************/
struct ldbpscode {
        LdbInfo      *i; /* asInfo - subsumption */
        LdbImmutable *m;
};
```

38c ⟨*pscode.h* 38c⟩≡ (13b)

```
/** PostScript code ******************************************************/
/** constructors *********************************************************/
/* create a fresh PostScript code
 *
 * precondition : this PostScript code does not exist
 * postcondition: this PostScript code is created
 *
*/
extern LdbPsCode *ldbPsCode_make(const char *s);

/** observers ************************************************************/
/* return a given PostScript code as an ldb information (subsumption)
 *
 * precondition : true
 */
extern LdbInfo *ldbPsCode_asInfo(LdbPsCode *s);
```

Uses `ldbPsCode_asInfo` 39 and `ldbPsCode_make` 39.

39 ⟨*pscode.c* 39⟩≡
⟨*included files* 10b⟩

```
static void       pscode_extern(s, out) LdbPsCode *s; LdbFile *out; {
        ldbImmutable_extern(s->m, out);
}
static LdbExtRef *pscode_ref(s)         LdbPsCode *s; { return s->m->ref; }

LdbPsCode *ldbPsCode_make(str) const char *str; {
        ldbImmutable_make(LdbPsCode *, &pscode_extern, &pscode_ref, "%s", str);
}
LdbInfo *ldbPsCode_asInfo(s) LdbPsCode *s; { ldbImmutable_asInfo(s); }
```

Defines:
`ldbPsCode_asInfo`, used in chunk 38c.
`ldbPsCode_make`, used in chunk 38c.
`pscode_extern`, never used.
`pscode_ref`, never used.
Uses `LdbExtRef` 20b 179, `ldbImmutable_asInfo` 28, `ldbImmutable_extern` 28, `ldbImmutable_make` 28, and `out` 94a.

2.3.9 Mutable PostScript data

- Abstraction : a mutable PostScript data
- Constructors:
  - `ldbMutable_make`: create a fresh mutable PostScript data
- Observers :
  - `ldbMutable_extern`: externalize this mutable information
  - `ldbMutable_ref`: return this mutable information's external reference
  - `ldbMutable_mutate`: make sure that a given mutable information has not been externalized yet
  - `ldbMutable_asInfo` : return a given mutable information as an `ldb` information (subsumption)

A mutable information satisfies the following invariances:

* `ldbMutable_extern` is called in any states.

* `ldbMutable_ref` is called only in the `EMITTED` state.

* `ldbMutable_mutate` is called only in the `NEW` state.

Each mutable information is externalized by `ldbMutable_extern` as follows. If it's not yet externalized, its label is emitted followed by the number of its elements, its external reference, and every element in it.

For example, the following type dictionary

```
/T45 <<
        /decl    (type-decl)
        /printer {psproc}
        /symbol  S23
>> def
```

is externalized as follows:

```
.LT45:
.long 3
.long .LIm139   ;; T45
.long .LIm141   ;; decl
.long .LIm46    ;; (type-decl)
.long .LIm144   ;; printer
.long .LIm4     ;; {psproc}
.long .LIm146   ;; symbol
.long .LS23     ;; S23
```

Its label `.LT45` is emitted followed by the number of its elements `3`, its external reference `T45` denoted by `.LIm139`, and every key-value pair in the type dictionary.

41a ⟨*type definitions in ldbinfo.h* 8b⟩+≡ (13a) ◁38a 44a▷

```
typedef struct ldbmutable    LdbMutable;
```

41b ⟨*abstractions in utils.h* 14b⟩+≡ (9) ◁38b 44c▷

```
/** abstraction : mutable ************************************************/
struct ldbmutable {
        enum { NEW, EMITTED } state;  /* state */
        LdbExtRef *ref;               /* external reference */
};
```

Uses `LdbExtRef` 20b 179.

42 ⟨*mutable.h* 42⟩≡ (9)

```
/** mutable info ************************************************************/
/** constructors ************************************************************/
/* create a fresh mutable information
 *
 * precondition : this mutable information does not exist
 * postcondition: this mutable information is created
 */
#define ldbMutable_make(MUTABLE,BASE,EXT,REF) \
        MUTABLE     new = (MUTABLE)     malloc(sizeof *new); \
        LdbMutable *m   = (LdbMutable *)malloc(sizeof *m); \
        m->state        = NEW; \
        m->ref          = ldbUtil_baseToExtRef(BASE); \
        new->m          = m; \
        new->i          = ldbInfo_make(); \
        new->i->ext     = EXT; \
        new->i->ref     = REF;

/** observers ***************************************************************/
/* externalize a specified mutable information
 *
 * precondition : file handle must be at top level
 * postcondition: file handle is still at top level,
 *                and this information has been externalized
 */
#define ldbMutable_extern(MUTABLE,OUT,VISITELEMENTS,EMITDEFINITION, SIZE) \
        switch(MUTABLE->state) { \
                case NEW:     MUTABLE->state = EMITTED; \
                              VISITELEMENTS; \
                              switch(MUTABLE->state) { \
                                case EMITTED: { \
                                        LdbInfo *info = \
          name2i(ldbUtil_dropPrefixOfLabelName((const char *)MUTABLE->ref)); \
                                        ldbInfo_extern   (info, OUT); \
                                        ldbUtil_emitLabel((const char *)MUTABLE->ref , OUT); \
                                        ldbUtil_emitName (SIZE, OUT); \
                                        ldbUtil_emitName ((const char *)ldbInfo_ref(info), OUT)
                                        ldbFile_push(OUT); \
                                        EMITDEFINITION; \
                                        ldbFile_pop(OUT); \
                                        MUTABLE->state = EMITTED; \
                                        break; } \
                                default     : assert(0); \
                              } \
                              break; \
                case EMITTED: break; /* if already emitted then return */ \
```

```
        }

/* return a specified mutable information's external reference
 *
 * precondition : a given mutable information has already been externalized
 * postcondition: this external reference can be written anywhere
 *                in a file handle
 */
extern LdbExtRef *ldbMutable_ref(LdbMutable *m);

/* make sure that the state of a given mutable information is NEW
 *
 * precondition : this mutable information has not yet been externalized
 * postcondition: this mutable information has not yet been externalized
 */
extern void ldbMutable_mutate(LdbMutable *m);

/* return a given mutable information as an ldb information (subsumption)
 *
 * precondition : true
 */
#define ldbMutable_asInfo(MUTABLE) return (LdbInfo *)MUTABLE;
```

Defines:
ldbMutable_asInfo, used in chunks 46 and 49.
ldbMutable_extern, used in chunks 46 and 49.
ldbMutable_make, used in chunks 46 and 49.
Uses LdbExtRef 20b 179, ldbFile_pop 17, ldbFile_push 17, ldbInfo_extern 25, ldbInfo_make 25, ldbInfo_ref 25, ldbMutable_mutate 43, ldbMutable_ref 43, ldbUtil_baseToExtRef 9, and name2i 11.

43 ⟨*mutable.c* 43⟩≡
⟨*included files* 10b⟩

```
LdbExtRef *ldbMutable_ref   (m) LdbMutable *m; { assert(m->state == EMITTED);
                                                 return m->ref; }
void       ldbMutable_mutate(m) LdbMutable *m; { assert(m->state == NEW); }
```

Defines:
ldbMutable_mutate, used in chunks 42, 46, and 49.
ldbMutable_ref, used in chunk 42.
Uses LdbExtRef 20b 179.

2.3.10 PostScript arrays (lists)

- Abstraction : a PostScript array whose elements are PostScript data
- Constructors:
  - `ldbArray_make` : create a fresh PostScript array
  - `ldbArray_addHigh` : put a given value to the end of this PostScript array
- Observers :
  - `ldbArray_get` : get the value of the indexed component in this array
  - `ldbArray_asInfo` : return a given PostScript array as an `ldb` information (subsumption)

44a ⟨*type definitions in ldbinfo.h* 8b⟩+≡ (13a) ◁41a 47a▷

```
typedef struct ldbarray      LdbArray;
```

44b ⟨*macros* 10a⟩+≡ (8a) ◁23a 47b▷

```
#define LDB_ARRAY_MAX           300
```

Defines:
`LDB_ARRAY_MAX`, used in chunks 44c, 46, and 96a.

44c ⟨*abstractions in utils.h* 14b⟩+≡ (9) ◁41b 47c▷

```
/** abstraction : array ************************************************/
struct ldbarray {
        LdbInfo    *i;   /* asInfo - subsumption */
        LdbMutable *m;
        LdbInfo    *array[LDB_ARRAY_MAX];
        int         n;
};
```

Uses `LDB_ARRAY_MAX` 44b and `n` 21 179 186.

45 ⟨*array.h* 45⟩≡ (13b)

```
/** array ************************************************************/
/** constructors *****************************************************/
/* create a fresh PostScript array
 *
 * precondition : this array does not exist
 * postcondition: this array is created
 *
*/
extern LdbArray *ldbArray_make(const char *s);

/* put a given value to the end of this PostScript array
 *
 * precondition : ldbArray_addHigh([a,..,n],i)
 * postcondition: [a,..,n,i]
 */
extern void ldbArray_addHigh(LdbArray *a, LdbInfo *i);

/** observers ********************************************************/
/* return the value of the indexed component in this PostScript array
 *
 * precondition : true
 */
extern LdbInfo *ldbArray_get(LdbArray *a, int i);

/* return a given PostScript array as an ldb information (subsumption)
 *
 * precondition : true
 */
extern LdbInfo *ldbArray_asInfo(LdbArray *a);
```

Uses `ldbArray_addHigh` 46, `ldbArray_asInfo` 46, `ldbArray_get` 46, `ldbArray_make` 46, and `n` 21 179 186.

46 ⟨*array.c* 46⟩≡
⟨*included files* 10b⟩

```
#define forall(ACTION) for (i=0; i<a->n; i++) ACTION
static void array_extern(a, out) LdbArray *a; LdbFile *out; {
        int i;
        ldbMutable_extern(a->m, out,
                          forall(ldbInfo_extern(a->array[i], out)),
                          forall(ldbUtil_emitName((const char *)ldbInfo_ref(a->array[i]), out)),
                          ldbUtil_intToStr(a->n));
}
static LdbExtRef *array_ref(a) LdbArray *a; { return a->m->ref; }

LdbArray *ldbArray_make(b) const char *b; {
        ldbMutable_make(LdbArray *, b, &array_extern, &array_ref);
        *new->array = malloc(sizeof(LdbInfo *) * LDB_ARRAY_MAX);
        return new;
}

void ldbArray_addHigh(a, i) LdbArray *a; LdbInfo *i; {
        ldbMutable_mutate(a->m);
        a->array[a->n++] = i;
}

LdbInfo *ldbArray_get(a, i) LdbArray *a; int i; {
        return a->array[i];
}

LdbInfo *ldbArray_asInfo(a) LdbArray *a; { ldbMutable_asInfo(a); }
```

Defines:
`array_extern`, never used.
`array_ref`, never used.
`forall`, never used.
`ldbArray_addHigh`, used in chunks 45, 66, 71, 95, and 100a.
`ldbArray_asInfo`, used in chunks 11 and 45.
`ldbArray_get`, used in chunks 45, 95, and 100a.
`ldbArray_make`, used in chunks 45, 66, 71, 95, and 100a.
Uses `LDB_ARRAY_MAX` 44b, `LdbExtRef` 20b 179, `ldbInfo_extern` 25, `ldbInfo_ref` 25, `ldbMutable_asInfo` 42, `ldbMutable_extern` 42, `ldbMutable_make` 42, `ldbMutable_mutate` 43, `ldbUtil_intToStr` 11, `n` 21 179 186, and `out` 94a.

2.3.11 PostScript dictionaries (tables)

- Abstraction : a PostScript dictionary with string keys and PostScript values
- Constructors:
  - `ldbTable_make` : create a fresh PostScript dictionary
  - `ldbTable_put`: add a given key-value pair to this table; overwrite any existing entry with the same key
- Observers :
  - `ldbTable_get`: find the value bound to a given key in this table
  - `ldbTable_asInfo` : return a given table as an `ldb` information (subsumption)

47a ⟨*type definitions in ldbinfo.h* 8b⟩+≡ (13a) ◁44a 50a▷

```
typedef struct ldbtable      LdbTable;
```

47b ⟨*macros* 10a⟩+≡ (8a) ◁44b

```
#define LDB_TABLE_MAX           20
```

Defines:
`LDB_TABLE_MAX`, used in chunks 47c and 49.

47c ⟨*abstractions in utils.h* 14b⟩+≡ (9) ◁44c 50b▷

```
/** abstraction : table ************************************************/
typedef struct ldbentry {
        const char *key;
        LdbInfo    *info;
} LdbEntry;
struct ldbtable {
        LdbInfo    *i;   /* asInfo - subsumption */
        LdbMutable *m;
        LdbEntry   *table[LDB_TABLE_MAX];
        int         n;
};
```

Defines:
`LdbEntry`, used in chunk 49.
Uses `LDB_TABLE_MAX` 47b and `n` 21 179 186.

48 ⟨*table.h* 48⟩≡ (13b)

```
/** table ***********************************************************/
/** constructors ****************************************************/
/* create a fresh PostScript table
 *
 * precondition : this table does not exist
 * postcondition: this table is created
 *
*/
extern LdbTable *ldbTable_make(const char *base);

/* add a given key-value pair to this table
 *
 * precondition : ldbTable_put <<...(k,v)...>> k i
 * postcondition: <<...(k,i)...>>
 */
extern void ldbTable_put(LdbTable *t, const char *k, LdbInfo *i);

/** observers *******************************************************/
/* return the value associated with the specified key in this table
 *
 * precondition : table might be empty
 * postcondition: return a value associated with a given key, if any
 */
extern LdbInfo *ldbTable_get(LdbTable *t, const char *k);

/* return a given table as an ldb information (subsumption)
 *
 * precondition : true
 */
extern LdbInfo *ldbTable_asInfo(LdbTable *t);
```

Uses ldbTable_asInfo 49, ldbTable_get 49, ldbTable_make 49, and ldbTable_put 49.

49 ⟨*table.c* 49⟩≡
⟨*included files* 10b⟩

```
#define forall(ACTION) for (i=0; i<t->n; i++) ACTION
#define emitkey(KEY)   (const char *)ldbInfo_ref(name2i(KEY))
static void table_extern(t, out) LdbTable *t; LdbFile *out; {
        int i;
        ldbMutable_extern(t->m, out,
                          forall({LdbName *n = ldbName_make(t->table[i]->key);
                                  ldbInfo_extern(t->table[i]->info, out);
                                  ldbInfo_extern(ldbName_asInfo(n), out);}),
                          forall({ldbUtil_emitName(emitkey(t->table[i]->key),out);
                                  ldbUtil_emitName((const char *)ldbInfo_ref(t->table
                          ldbUtil_intToStr(t->n));
}
static LdbExtRef *table_ref(t) LdbTable *t; { return t->m->ref; }

LdbTable *ldbTable_make(b) const char *b; {
        ldbMutable_make(LdbTable *, b, &table_extern, &table_ref);
        *new->table = malloc(sizeof(LdbEntry *) * LDB_TABLE_MAX);
        return new;
}

void ldbTable_put(t, k, i) LdbTable *t; const char *k; LdbInfo *i; {
        int j;
        LdbEntry *e = malloc(sizeof(LdbEntry *));
        ldbMutable_mutate(t->m);
        for (j=0; j<t->n; j++)
                if (t->table[j]->key == k) { t->table[j]->info = i; return; }
        e->key           = k;
        e->info          = i;
        t->table[t->n++] = e;
}

LdbInfo *ldbTable_get(t, k) LdbTable *t; const char *k; {
      int i;
      for (i=0; i<t->n; i++)
              if (!strcmp(t->table[i]->key, k)) return t->table[i]->info;
      return 0;
}

LdbInfo *ldbTable_asInfo(t) LdbTable *t; { ldbMutable_asInfo(t); }
```

Defines:
`emitkey`, never used.
`forall`, never used.
`ldbTable_asInfo`, used in chunks 11 and 48.

ldbTable_get, used in chunks 48, 97, and 100a.
ldbTable_make, used in chunks 48, 56a, 77b, and 95.
ldbTable_put, used in chunks 48, 56a, 58, 61, 66, 71, 77b, and 100a.
table_extern, never used.
table_ref, never used.
Uses LDB_TABLE_MAX 47b, LdbEntry 47c, LdbExtRef 20b 179, ldbInfo_extern 25, ldbInfo_ref 25, ldbMutable_asInfo 42, ldbMutable_extern 42, ldbMutable_make 42, ldbMutable_mutate 43, ldbName_asInfo 36a, ldbName_make 36a, ldbUtil_intToStr 11, n 21 179 186, name2i 11, and out 94a.

2.4 ldb data

This section includes the ldb interface for ldb data which might be the source-language dependent but the compiler independent. All the compiler-dependent procedures used in this section are defined in Section 2.2.5.

2.4.1 Source-code locations

- Abstraction : a source-code location
- Constructors:
  - ldbSrcLoc_make : create a source-code location
- Observers :
  - ldbSrcLoc_file : return this source-code location's file name
  - ldbSrcLoc_line : return this source-code location's line number
  - ldbSrcLoc_column : return this source-code location's column number

50a ⟨*type definitions in ldbinfo.h* 8b⟩+≡ (13a) ◁47a 52b▷

```
typedef struct ldbsrcloc     LdbSrcLoc;
```

50b ⟨*abstractions in utils.h* 14b⟩+≡ (9) ◁47c 52c▷

```
/** abstraction : value ************************************************/
struct ldbsrcloc {
        const char *file;
        int         line;
        int         column;
};
```

51 ⟨*srcloc.h* 51⟩≡ (13b)

```
/** source-code location ********************************************/
/** constructors ****************************************************/
/* create a source-code location
 *
 * precondition : true
 */
extern LdbSrcLoc *ldbSrcLoc_make(const char *file, int line, int column);

/** observers *******************************************************/
/* return a given source-code location's file name
 *
 * precondition : true
 */
extern const char *ldbSrcLoc_file(LdbSrcLoc *srcloc);

/* return a given source-code location's line number
 *
 * precondition : true
 */
extern int ldbSrcLoc_line(LdbSrcLoc *srcloc);

/* return a given source-code location's column number
 *
 * precondition : true
 */
extern int ldbSrcLoc_column(LdbSrcLoc *srcloc);
```

Uses ldbSrcLoc_column 52a, ldbSrcLoc_file 52a, ldbSrcLoc_line 52a, and ldbSrcLoc_make 52a.

52a ⟨*srcloc.c* 52a⟩≡
⟨*included files* 10b⟩

```
LdbSrcLoc *ldbSrcLoc_make(file, line, column) const char *file;
                                              int line, column; {
        LdbSrcLoc *new = (LdbSrcLoc *)malloc(sizeof *new);
        new->file      = file;
        new->line      = line;
        new->column    = column;
        return new;
}

const char *ldbSrcLoc_file  (src) LdbSrcLoc *src; { return src->file;   }
int         ldbSrcLoc_line  (src) LdbSrcLoc *src; { return src->line;   }
int         ldbSrcLoc_column(src) LdbSrcLoc *src; { return src->column; }
```

Defines:
`ldbSrcLoc_column`, used in chunks 51, 56a, and 89.
`ldbSrcLoc_file`, used in chunks 51, 56a, 66, and 89.
`ldbSrcLoc_line`, used in chunks 51, 56a, and 89.
`ldbSrcLoc_make`, used in chunk 51.

2.4.2 Values of machine-level constants

- Abstraction : a machine-level value which is either an integer or a floating-point value
- Constructors:
  - `ldbValue_makeInt` : create a PostScript code whose evaluation results in an integer value
  - `ldbValue_makeFloat` : create a PostScript code whose evaluation results in a floating-point value
- Observers :
  - `ldbValue_asInfo` : return a given value as an `ldb` information (subsumption)

52b ⟨*type definitions in ldbinfo.h* 8b⟩+≡ (13a) ◁50a 54b▷

```
typedef struct ldbvalue      LdbValue;
```

52c ⟨*abstractions in utils.h* 14b⟩+≡ (9) ◁50b 55a▷

```
/** abstraction : value ***********************************************/
struct ldbvalue {
        LdbInfo      *i;   /* asInfo - subsumption */
        LdbImmutable *m;
};
```

53 ⟨*value.h* 53⟩≡ (13b)

```
/** value ************************************************************/
/** constructors *****************************************************/
/* create a PostScript code whose evaluation results in an integer value
 *
 * postcondition: evaluation of the PostScript code results in a value
 */
extern LdbValue *ldbValue_makeInt(int i);

/* create a PostScript code whose evaluation results in a floating-point value
 *
 * postcondition: evaluation of the PostScript code results in a value
 */
extern LdbValue *ldbValue_makeFloat(float f);

/** observers ********************************************************/
/* return a given value as an ldb information (subsumption)
 *
 * precondition : true
 */
extern LdbInfo *ldbValue_asInfo(LdbValue *v);
```

Uses `ldbValue_asInfo` 54a, `ldbValue_makeFloat` 54a, and `ldbValue_makeInt` 54a.

54a ⟨*value.c* 54a⟩≡
⟨*included files* 10b⟩

```
static void       value_extern(v, out) LdbValue *v; LdbFile *out; {
        ldbImmutable_extern(v->m, out);
}
static LdbExtRef *value_ref(v)         LdbValue *v; { return v->m->ref; }

LdbValue *ldbValue_makeInt  (i) int   i; {
        ldbImmutable_make(LdbValue *, &value_extern, &value_ref,
                          "%d Immediate", i);
}
LdbValue *ldbValue_makeFloat(f) float f; {
        ldbImmutable_make(LdbValue *, &value_extern, &value_ref,
                          "%f Immediate", f);
}

LdbInfo *ldbValue_asInfo(v) LdbValue *v; { return (LdbInfo *)v; }
```

Defines:
`ldbValue_asInfo`, used in chunks 53 and 58.
`ldbValue_makeFloat`, used in chunk 53.
`ldbValue_makeInt`, used in chunk 53.
`value_extern`, never used.
`value_ref`, never used.
Uses `LdbExtRef` 20b 179, `ldbImmutable_extern` 28, `ldbImmutable_make` 28, and `out` 94a.

2.4.3 Symbols

- Abstraction : a symbol; every symbol is one of a variable, a procedure, a constant, and a type.
- Constructors:
  - `ldbSymbol_make` : create a symbol and initialize its name and source location
- Observers :
  - `ldbSymbol_name` : return this symbol's name
  - `ldbSymbol_asTable` : return a given symbol as a table (subsumption)

54b ⟨*type definitions in ldbinfo.h* 8b⟩+≡ (13a) ◁52b 56b▷

```
typedef struct ldbsymbol     LdbSymbol;
```

55a ⟨*abstractions in utils.h* 14b⟩+≡ (9) ◁52c 57a▷

```
/** abstraction : symbol ***************************************************/
/* the base type for all the symbols */
struct ldbsymbol {
        LdbTable   *t;
        const char *name;
};
```

55b ⟨*symbol.h* 55b⟩≡ (13b)

```
/** symbol *****************************************************************/
/** constructors ***********************************************************/
/* create a fresh symbol
 *
 * precondition : this symbol does not exist
 * postcondition: this symbol is created
 */
extern LdbSymbol *ldbSymbol_make(const char *name, const char *kind,
                                 LdbSrcLoc *src);

/** observers **************************************************************/
/* return a given symbol's name
 *
 * precondition : true
 */
extern const char *ldbSymbol_name(LdbSymbol *s);

/* return a given symbol as a table (subsumption)
 *
 * precondition : true
 */
extern LdbTable *ldbSymbol_asTable(LdbSymbol *s);
```

Uses `ldbSymbol_asTable` 56a, `ldbSymbol_make` 56a, and `ldbSymbol_name` 56a.

56a ⟨*symbol.c* 56a⟩≡
⟨*included files* 10b⟩

```
LdbSymbol *ldbSymbol_make(name, kind, src) const char *name, *kind;
                                           LdbSrcLoc *src; {
        /* every symbol's external name starts with "S" */
        LdbSymbol *new = (LdbSymbol *)malloc(sizeof *new);
        LdbTable  *t   = ldbTable_make("S");
        ldbTable_put(t, "name",       str2i(name));
        ldbTable_put(t, "sourcefile", str2i(ldbSrcLoc_file  (src)));
        ldbTable_put(t, "sourcey",    int2i(ldbSrcLoc_line  (src)));
        ldbTable_put(t, "sourcex",    int2i(ldbSrcLoc_column(src)));
        ldbTable_put(t, "kind",       str2i(kind));
        new->t    = t;
        new->name = name;
        return new;
}

const char *ldbSymbol_name   (sym) LdbSymbol *sym; { return sym->name; }
LdbTable   *ldbSymbol_asTable(sym) LdbSymbol *sym; { return sym->t; }
```

Defines:
`ldbSymbol_asTable`, used in chunks 11 and 55b.
`ldbSymbol_make`, used in chunks 55b, 58, 61, 66, and 71.
`ldbSymbol_name`, used in chunk 55b.
Uses `int2i` 11, `ldbSrcLoc_column` 52a, `ldbSrcLoc_file` 52a, `ldbSrcLoc_line` 52a, `ldbTable_make` 49, `ldbTable_put` 49, and `str2i` 11.

Constants

- Abstraction : a constant symbol
- Constructors:
  - `ldbConstant_make` : create a constant symbol
  - `ldbConstant_putType` : initialize this constant's type
  - `ldbConstant_putValue` : initialize this constant's value
- Observers :
  - `ldbConstant_asSymbol` : return a given constant as a symbol (subsumption)

When a constant is created, its name and source location are given but its type is usually not yet provided. Thus, a constant's type is initialized when the type is avaiable by `ldbConstant_putType`. A constant's value is either an integer or a floating-point value and this information is initialized by `ldbConstant_putValue`.

56b ⟨*type definitions in ldbinfo.h* 8b⟩+≡ (13a) ◁54b 59a▷

```
typedef struct ldbconstant   LdbConstant;
```

57a ⟨*abstractions in utils.h* 14b⟩+≡ (9) ◁55a 59b▷

```
/** abstraction : constant ********************************************/
struct ldbconstant {
        LdbSymbol *s;
};
```

57b ⟨*constant.h* 57b⟩≡ (13b)

```
/** constant **********************************************************/
/** constructors ******************************************************/
/* create a fresh constant
 *
 * precondition : this constant does not exist
 * postcondition: this constant is created
 */
extern LdbConstant *ldbConstant_make(const char *name, LdbSrcLoc *src);

/* initialize a given constant's type
 *
 * precondition : after type checking
 */
extern void ldbConstant_putType(LdbConstant *c, LdbType *ty);

/* initialize a given constant's value
 *
 * precondition : after lexing
 */
extern void ldbConstant_putValue(LdbConstant *c, LdbValue *value);

/** observers *********************************************************/
/* return a given constant as a symbol (subsumption)
 *
 * precondition : true
 */
extern LdbSymbol *ldbConstant_asSymbol(LdbConstant *c);
```

Uses ldbConstant_asSymbol 58, ldbConstant_make 58, ldbConstant_putType 58, and ldbConstant_putValue 58.

58 ⟨*constant.c* 58⟩≡
⟨*included files* 10b⟩

```
LdbConstant *ldbConstant_make(name, src) const char *name; LdbSrcLoc *src; {
        LdbConstant *new = (LdbConstant *)malloc(sizeof *new);
        LdbSymbol   *s   = ldbSymbol_make(name, "constant", src);
        new->s = s;
        return new;
}

void ldbConstant_putType(c, ty) LdbConstant *c; LdbType *ty; {
        ldbTable_put(symbol2t(ldbConstant_asSymbol(c)), "type", type2i(ty));
}

void ldbConstant_putValue(c, v) LdbConstant *c; LdbValue *v; {
        ldbTable_put(symbol2t(ldbConstant_asSymbol(c)), "value",
                     ldbValue_asInfo(v));
}

LdbSymbol *ldbConstant_asSymbol(c) LdbConstant *c; { return c->s; }
```

Defines:
`ldbConstant_asSymbol`, used in chunk 57b.
`ldbConstant_make`, used in chunk 57b.
`ldbConstant_putType`, used in chunk 57b.
`ldbConstant_putValue`, used in chunk 57b.
Uses `ldbSymbol_make` 56a, `ldbTable_put` 49, `ldbValue_asInfo` 54a, `symbol2t` 11, and `type2i` 11.

Variables

- Abstraction : a variable symbol
- Constructors:
  - `ldbVar_make` : create a variable symbol
  - `ldbVar_putType` : initialize this variable's type
  - `ldbVar_putWhere` : initialize this variable's placement
  - `ldbVar_putLabel` : set this variable's assembly-language label
- Observers :
  - `ldbVar_label` : return this variable's assembly-language label
  - `ldbVar_asSymbol` : return a given variable as a symbol (subsumption)

When a variable is created, its name and source location are given but its type is usually not yet provided. Thus, a variable's type is initialized when the type is avaiable by `ldbVar_putType`. A variable's placement is either in a stack frame or in a register and this information is initialized by `ldbVar_putWhere` after the register allocation phase. A variable's assembly-language label is used to refer the variable in the executable program. `ldbVar_putLabel` records the variable's label when it's available and `ldbVar_label` returns the label. The naming convention for labels is compiler dependent and labels are described in section 2.4.7.

59a ⟨*type definitions in ldbinfo.h* 8b⟩+≡ (13a) ◁56b 63a▷

```
typedef struct ldblabel      LdbLabel;
typedef struct ldbvar        LdbVar;
```

59b ⟨*abstractions in utils.h* 14b⟩+≡ (9) ◁57a 63b▷

```
/** abstraction : label ************************************************/
struct ldblabel {
        const char *name;
        const char *space;
};

/** abstraction : variable *********************************************/
struct ldbvar {
        LdbSymbol *s;
        LdbLabel  *label;
};
```

60 ⟨*variable.h* 60⟩≡ (13b)

```
/** variable ***********************************************************/
/** constructors *******************************************************/
/* create a fresh variable
 *
 * precondition : this variable does not exist
 * postcondition: this variable is created
 */
extern LdbVar *ldbVar_make(const char *name, LdbSrcLoc *src);

/* initialize a given variable's type
 *
 * precondition : after type checking
 */
extern void ldbVar_putType(LdbVar *v, LdbType *ty);

/* initialize a given variable's placement
 *
 * precondition : after register allocation
 */
extern void ldbVar_putWhere(LdbVar *v, LdbPlcmt *loc);

/* set a given variable's assembly-language label
 *
 * precondition : after the variable's label is known
 */
extern void ldbVar_putLabel(LdbVar *v, LdbLabel *lab);

/** observers **********************************************************/
/* returns a given variable's assembly-language label
 *
 * precondition : true
 */
extern LdbLabel *ldbVar_label(LdbVar *v);

/* return a given variable as a symbol (subsumption)
 *
 * precondition : true
 */
extern LdbSymbol *ldbVar_asSymbol(LdbVar *v);
```

Uses ldbVar_asSymbol 61, ldbVar_label 61, ldbVar_make 61, ldbVar_putLabel 61, ldbVar_putType 61, and ldbVar_putWhere 61.

61 ⟨*variable.c* 61⟩≡
⟨*included files* 10b⟩

```
LdbVar *ldbVar_make(name, src) const char *name; LdbSrcLoc *src; {
        LdbVar    *new = (LdbVar *)malloc(sizeof *new);
        LdbSymbol *s   = ldbSymbol_make(name, "variable", src);
        new->s = s;
        return new;
}

void ldbVar_putType(v, ty) LdbVar *v; LdbType *ty; {
        ldbTable_put(symbol2t(ldbVar_asSymbol(v)), "type", type2i(ty));
}

void ldbVar_putWhere(v, l) LdbVar *v; LdbPlcmt *l; {
        ldbTable_put(symbol2t(ldbVar_asSymbol(v)),"where",ldbPlcmt_asInfo(l));
}

void       ldbVar_putLabel(v, l) LdbVar *v; LdbLabel *l; { v->label = l; }
LdbLabel  *ldbVar_label   (v)    LdbVar *v; { return v->label; }
LdbSymbol *ldbVar_asSymbol(v)    LdbVar *v; { return v->s; }
```

Defines:
`ldbVar_asSymbol`, used in chunks 60 and 71.
`ldbVar_label`, used in chunk 60.
`ldbVar_make`, used in chunk 60.
`ldbVar_putLabel`, used in chunk 60.
`ldbVar_putType`, used in chunk 60.
`ldbVar_putWhere`, used in chunk 60.
Uses `ldbPlcmt_asInfo` 80, `ldbSymbol_make` 56a, `ldbTable_put` 49, `symbol2t` 11, and `type2i` 11.

Procedures

- Abstraction : a procedure symbol
- Constructors:
  - `ldbProc_make` : create a procedure symbol
  - `ldbProc_putType` : initialize this procedure's type
  - `ldbProc_putFormal` : set this procedure's last formal parameter
  - `ldbProc_putArgEnv` : set this procedure's arguments environment
  - `ldbProc_putReturnType` : set this procedure's return type
  - `ldbProc_putFrameSize` : set this procedure's frame size
  - `ldbProc_putRegInfo` : set this procedure's callee-saved registers information
  - `ldbProc_putLabel` : set this procedure's assembly-language label
  - `ldbProc_addLocus` : add a stopping point in this procedure
  - `ldbProc_addLocal` : add a local symbol in this procedure
- Observers :
  - `ldbProc_file` : return this procedure's file name
  - `ldbProc_locals` : return this procedure's local symbols
  - `ldbProc_asSymbol` : return a given procedure as a symbol (subsumption)

When a procedure is created, its name and source location are given but its type is usually not yet provided. Thus, a procedure's type is initialized when the type is avaiable by `ldbProc_putType`. `ldbProc_putFormal` sets a procedure's the last formal parameter and `ldbProc_putArgEnv` sets the procedure's arguments environment during the semantic analysis phase and `ldbProc_putReturnType` sets the procedure's return type after the type-checking phase. `ldbProc_putFrameSize` sets a procedure's stack frame size and `ldbProc_putRegInfo` sets a procedure's callee-saved registers information. Callee-saved registers information is given as a tuple of an integer array whose elements are the register-save mask in a decreasing order and the distance of least significant register from the heavy pointer in turn and the size of the array.

`ldb` assumes that each stack frame can be identified by a single pointer, called the *heavy pointer*, the value of which remains unchanged for the lifetime of the frame. Since the frame and stack pointers are used in different ways on different architectures, `ldb` uses the term "heavy" pointer, not frame or stack pointer. The heavy pointer is realized as either the frame pointer or the stack pointer.

A procedure's assembly-language label is used to refer the procedure in the executable program. `ldbProc_putLabel` records the procedure's label when it's

available. The naming convention for labels is compiler dependent and labels are described in section 2.4.7. A procedure records all the stopping points in it and each stopping point is added to the procedure by `ldbProc_addLocus` when the stopping point is created. Compilers usually have several intermediate representations for a program and in which representation to put stopping points are compiler dependent. Stopping points are described in section 2.4.8. All the local symbols in a procedure is recorded by `ldbProc_addLocal`. `ldbProc_file` returns the procedure's file name to make a source map for the current compilation unit and `ldbProc_locals` returns all the local symbols in the procedure.

63a ⟨*type definitions in ldbinfo.h* 8b⟩+≡ (13a) ◁59a 69a▷

```
typedef struct ldbloci       LdbLoci;
typedef struct ldbproc       LdbProc;
```

63b ⟨*abstractions in utils.h* 14b⟩+≡ (9) ◁59b 69b▷

```
/** abstraction : loci ***************************************************/
struct ldbloci {
        LdbArray *a;
        LdbArray *envs;
        LdbArray *myloci;
};
/** abstraction : procedure **********************************************/
struct ldbproc {
        LdbSymbol  *s;
        LdbLoci    *loci;
        const char *file;
        LdbList     locals;
};
```

Uses `LdbList` 8b.

64 ⟨*procedure.h* 64⟩≡ (13b)

```
/** procedure ***************************************************************/
/** constructors ************************************************************/
/* create a fresh procedure
 *
 * precondition : this procedure does not exist
 * postcondition: this procedure is created
 */
extern LdbProc *ldbProc_make(const char *name, LdbSrcLoc *src);

/* initialize a given procedure's type
 *
 * precondition : after type checking
 */
extern void ldbProc_putType(LdbProc *p, LdbType *ty);

/* initialize a given procedure's the last formal parameter
 *
 * precondition : during semantic analysis
 */
extern void ldbProc_putFormal(LdbProc *p, LdbSymbol *formal);

/* initialize a given procedure's arguments environment
 *
 * precondition : during semantic analysis
 */
extern void ldbProc_putArgEnv(LdbProc *p, LdbEnv *argenv);

/* initialize a given procedure's return type
 *
 * precondition : after type checking
 */
extern void ldbProc_putReturnType(LdbProc *p, const char *returnty);

/* set a given procedure's frame size
 *
 * precondition : after register allocation
 */
extern void ldbProc_putFrameSize(LdbProc *p, int framesize);

/* set a given procedure's callee-saved registers information
 *
 * precondition : after register allocation
 */
extern void ldbProc_putRegInfo(LdbProc *p, int size, int regs[]);
```

```
/* set a given procedure's assembly-language label and
 * initialize a given procedure's placement
 *
 * precondition : after the procedure's label is known
 */
extern void ldbProc_putLabel(LdbProc *p, LdbLabel *lab);

/* add a specified stopping point to a given procedure's stopping points
 *
 * precondition : after a given stopping point is created
 */
extern void ldbProc_addLocus(LdbProc *p, LdbLocus *locus);

/* add a local symbol in this procedure
 *
 * precondition : symbol is not   in this procedure
 * postcondition: symbol is added to this procedure
 */
extern void ldbProc_addLocal(LdbProc *p, LdbSymbol *s);

/** observers ***********************************************************/
/* returns a given procedure's file name
 *
 * precondition : true
 */
extern const char *ldbProc_file(LdbProc *p);

/* returns a given procedure's local symbols
 *
 * precondition : true
 */
extern LdbList ldbProc_locals(LdbProc *p);

/* return a given procedure as a symbol (subsumption)
 *
 * precondition : true
 */
extern LdbSymbol *ldbProc_asSymbol(LdbProc *p);
```

Uses LdbList 8b, ldbProc_addLocal 66, ldbProc_addLocus 66, ldbProc_asSymbol 66, ldbProc_file 66, ldbProc_locals 66, ldbProc_make 66, ldbProc_putArgEnv 66, ldbProc_putFormal 66, ldbProc_putFrameSize 66, ldbProc_putLabel 66, ldbProc_putRegInfo 66, ldbProc_putReturnType 66, and ldbProc_putType 66.

66 ⟨*procedure.c* 66⟩≡
⟨*included files* 10b⟩

```
static LdbTable  *proc2t (p) LdbProc  *p; {
        return symbol2t(ldbProc_asSymbol(p));
}
static LdbInfo   *label2i(l) LdbLabel *l; {
        return ldbPlcmt_asInfo(ldbPlcmt_makeAtLabel(l));
}
static LdbLoci *ldbLoci_make() {
        LdbLoci  *new    = (LdbLoci *)malloc(sizeof *new);
        LdbArray *loci   = ldbArray_make("L");
        LdbArray *envs   = ldbArray_make("I");
        LdbArray *myloci = ldbArray_make("I");
        ldbArray_addHigh(loci, array2i(envs));
        ldbArray_addHigh(loci, array2i(myloci));
        new->a      = loci;
        new->envs   = envs;
        new->myloci = myloci;
        return new;
}
LdbProc *ldbProc_make(name, src) const char *name; LdbSrcLoc *src; {
        LdbProc   *new = (LdbProc *)malloc(sizeof *new);
        LdbSymbol *s   = ldbSymbol_make(name, "procedure", src);
        new->s         = s;
        new->loci      = ldbLoci_make();
        new->file      = ldbSrcLoc_file(src);
        new->locals    = 0;
        ldbTable_put(proc2t(new), "loci", array2i(new->loci->a));
        return new;
}

void ldbProc_putType(p, ty) LdbProc *p; LdbType *ty; {
        ldbTable_put(symbol2t(ldbProc_asSymbol(p)), "type", type2i(ty));
}

void ldbProc_putFormal    (p, f) LdbProc *p; LdbSymbol *f; {
        ldbTable_put(proc2t(p), "formals", symbol2i(f));
}

void ldbProc_putArgEnv    (p, e) LdbProc *p; LdbEnv *e; {
        ldbTable_put(proc2t(p), "argenv",     env2i(e));
}

void ldbProc_putReturnType(p, s) LdbProc *p; const char *s; {
        ldbTable_put(proc2t(p), "returntype", str2i(s));
```

```
}

void ldbProc_putFrameSize (p, i) LdbProc *p; int i; {
        ldbTable_put(proc2t(p), "framesize",  int2i(i));
}

typedef struct regi {
        int       size;
        int      *regs;
        LdbArray *reginfo;
} Regi;
static Regi *cachedRegInfo;
/* the argument is (size, regs)
 * where size is the size of regs
 *   and regs is an integer array whose elements are the register usage masks
 *            and the distances of least significant register from hp in turn
 */
static LdbArray *makeRegInfo(size, regs) int size, regs[]; {
        int i;
        LdbArray *regI = ldbArray_make("I");
        for (i = 0; i < size; i++) {
                ldbArray_addHigh(regI,ldbInt_asInfo(ldbInt_makeHex(regs[i++])));
                ldbArray_addHigh(regI,int2i(regs[i]));
        }
        return regI;
}
void ldbProc_putRegInfo   (p, size, regs) LdbProc *p; int size, regs[]; {
        LdbArray *reginfo;
        if (cachedRegInfo == NULL) {
                reginfo = makeRegInfo(size, regs);
                cachedRegInfo = (Regi *)malloc(sizeof *cachedRegInfo);
                cachedRegInfo->size    = size;
                cachedRegInfo->regs    = regs;
                cachedRegInfo->reginfo = reginfo;
        }
        else {
                if (size == cachedRegInfo->size &&
                    regs == cachedRegInfo->regs)
                        reginfo = cachedRegInfo->reginfo;
        }
        ldbTable_put(proc2t(p), "reginfo", array2i(reginfo));
}

void ldbProc_putLabel (p, l) LdbProc *p; LdbLabel *l; {
        ldbTable_put(proc2t(p), "where", label2i(l));
}
```

```
void ldbProc_addLocus (p, locus) LdbProc *p; LdbLocus *locus; {
        LdbArray *mylocus = ldbArray_make("I");
        ldbArray_addHigh(p->loci->envs  , env2i  (ldbLocus_env   (locus)));
        ldbArray_addHigh(mylocus        , str2i  (ldbLocus_file  (locus)));
        ldbArray_addHigh(mylocus        , int2i  (ldbLocus_line  (locus)));
        ldbArray_addHigh(mylocus        , int2i  (ldbLocus_column(locus)));
        ldbArray_addHigh(mylocus        , label2i(ldbLocus_label (locus)));
        ldbArray_addHigh(p->loci->myloci, array2i(mylocus));
}

void ldbProc_addLocal (p, s) LdbProc *p; LdbSymbol *s; {
        p->locals = ldbList_append(p->locals, s);
}

const char *ldbProc_file    (p) LdbProc *p; { return p->file; }

LdbList     ldbProc_locals  (p) LdbProc *p; { return p->locals; }

LdbSymbol  *ldbProc_asSymbol(p) LdbProc *p; { return p->s; }
```

Defines:
`cachedRegInfo`, never used.
`label2i`, never used.
`ldbLoci_make`, never used.
`ldbProc_addLocal`, used in chunk 64.
`ldbProc_addLocus`, used in chunk 64.
`ldbProc_asSymbol`, used in chunks 11, 64, and 71.
`ldbProc_file`, used in chunks 64 and 95.
`ldbProc_locals`, used in chunks 64 and 95.
`ldbProc_make`, used in chunk 64.
`ldbProc_putArgEnv`, used in chunk 64.
`ldbProc_putFormal`, used in chunk 64.
`ldbProc_putFrameSize`, used in chunk 64.
`ldbProc_putLabel`, used in chunk 64.
`ldbProc_putRegInfo`, used in chunk 64.
`ldbProc_putReturnType`, used in chunk 64.
`ldbProc_putType`, used in chunk 64.
`makeRegInfo`, never used.
`proc2t`, never used.
`Regi`, never used.
Uses `array2i` 11, `env2i` 11, `int2i` 11, `ldbArray_addHigh` 46, `ldbArray_make` 46, `ldbInt_asInfo` 33a, `ldbInt_makeHex` 33a, `LdbList` 8b, `ldbList_append` 11, `ldbLocus_column` 89, `ldbLocus_env` 89, `ldbLocus_file` 89, `ldbLocus_label` 89, `ldbLocus_line` 89, `ldbPlcmt_asInfo` 80, `ldbPlcmt_makeAtLabel` 83, `ldbSrcLoc_file` 52a, `ldbSymbol_make` 56a, `ldbTable_put` 49, `str2i` 11, `symbol2i` 11, `symbol2t` 11, and `type2i` 11.

Type symbols

- Abstraction : a type symbol
- Constructors:
  - `ldbTypeSymbol_make` : create a type symbol
  - `ldbTypeSymbol_putType` : initialize this type symbol's type
  - `ldbTypeSymbol_putVar` : record every variable in this type symbol
  - `ldbTypeSymbol_putProc` : record every procedure in this type symbol
- Observers :
  - `ldbTypeSymbol_asSymbol` : return a given type symbol as a symbol (subsumption)

When a type symbol is created, its name and source location are given but its type is usually not yet provided. Thus, a type symbol's type is initialized when the type is avaiable by `ldbTypeSymbol_putType`.

69a ⟨*type definitions in ldbinfo.h* 8b⟩+≡ (13a) ◁63a 72a▷

```
typedef struct ldbtypesymbol LdbTypeSymbol;
```

69b ⟨*abstractions in utils.h* 14b⟩+≡ (9) ◁63b 72b▷

```
/** abstraction : type symbol ********************************************/
struct ldbtypesymbol {
        LdbSymbol *s;
        LdbArray  *vars;
        LdbArray  *procs;
};
```

70 ⟨*type-symbol.h* 70⟩≡ (13b)

```
/** type symbol *************************************************************/
/** constructors ************************************************************/
/* create a fresh type symbol
 *
 * precondition : this type symbol does not exist
 * postcondition: this type symbol is created
 */
extern LdbTypeSymbol *ldbTypeSymbol_make(const char *name, LdbSrcLoc *src);

/* initialize a given type symbol's type
 *
 * precondition : after type checking
 */
extern void ldbTypeSymbol_putType(LdbTypeSymbol *t, LdbType *ty);

/* add a given variable to the set of variables in a given type symbol
 *
 * precondition : true
 */
extern void ldbTypeSymbol_putVar(LdbTypeSymbol *t, LdbVar *v);

/* add a given procedure to the set of procedures in a given type symbol
 *
 * precondition : true
 */
extern void ldbTypeSymbol_putProc(LdbTypeSymbol *t, LdbProc *p);

/** observers ***************************************************************/
/* return a given type symbol as a symbol (subsumption)
 *
 * precondition : true
 */
extern LdbSymbol *ldbTypeSymbol_asSymbol(LdbTypeSymbol *t);
```

Uses `ldbTypeSymbol_asSymbol` 71, `ldbTypeSymbol_make` 71, `ldbTypeSymbol_putProc` 71, `ldbTypeSymbol_putType` 71, and `ldbTypeSymbol_putVar` 71.

71 ⟨*type-symbol.c* 71⟩≡
⟨*included files* 10b⟩

```
LdbTypeSymbol *ldbTypeSymbol_make(name, src) const char *name;
                                             LdbSrcLoc  *src; {
        LdbTypeSymbol *new = (LdbTypeSymbol *)malloc(sizeof *new);
        LdbSymbol     *s   = ldbSymbol_make(name, "type", src);
        new->s     = s;
        new->vars  = ldbArray_make("I");
        new->procs = ldbArray_make("I");
        ldbTable_put(symbol2t(ldbTypeSymbol_asSymbol(new)), "vars",
                     array2i(new->vars));
        ldbTable_put(symbol2t(ldbTypeSymbol_asSymbol(new)), "procs",
                     array2i(new->procs));
        return new;
}

void ldbTypeSymbol_putType(t, ty) LdbTypeSymbol *t; LdbType *ty; {
        ldbTable_put(symbol2t(ldbTypeSymbol_asSymbol(t)), "type", type2i(ty));
}

void ldbTypeSymbol_putVar(t, v)  LdbTypeSymbol *t; LdbVar *v; {
        ldbArray_addHigh(t->vars , symbol2i(ldbVar_asSymbol (v)));
}

void ldbTypeSymbol_putProc(t, p) LdbTypeSymbol *t; LdbProc *p; {
        ldbArray_addHigh(t->procs, symbol2i(ldbProc_asSymbol(p)));
}

LdbSymbol *ldbTypeSymbol_asSymbol(t) LdbTypeSymbol *t; { return t->s; }
```

Defines:
ldbTypeSymbol_asSymbol, used in chunk 70.
ldbTypeSymbol_make, used in chunk 70.
ldbTypeSymbol_putProc, used in chunk 70.
ldbTypeSymbol_putType, used in chunk 70.
ldbTypeSymbol_putVar, used in chunk 70.
Uses array2i 11, ldbArray_addHigh 46, ldbArray_make 46, ldbProc_asSymbol 66, ldbSymbol_make 56a, ldbTable_put 49, ldbVar_asSymbol 61, symbol2i 11, symbol2t 11, and type2i 11.

2.4.4 Environments

- Abstraction : an environment
- Constructors:
  - `ldbEnv_makeEmpty` : create an empty environment
  - `ldbEnv_bind` : bind a given symbol to a given environment
- Observers :
  - `ldbEnv_asInfo` : return a given environment as an `ldb` information (subsumption)

72a ⟨*type definitions in ldbinfo.h* 8b⟩+≡ (13a) ◁69a 76a▷

```
typedef struct ldbenv        LdbEnv;
```

72b ⟨*abstractions in utils.h* 14b⟩+≡ (9) ◁69b 76b▷

```
/** abstraction : environment ***************************************/
struct ldbenv {
        LdbInfo   *i;          /* asInfo - subsumption */
        LdbExtRef *ref;        /* external reference */
        int        isExterned; /* if this environment has been externalized */
        int        isEmpty;    /* if this environment is empty */
        union {
          struct {
                LdbEnv    *env;
                LdbSymbol *symbol;
          } t;
        } u;
};
```

Uses `LdbExtRef` 20b 179.

73 ⟨*env.h* 73⟩≡ (13b)

```
/** environment ************************************************************/
/** constructors ***********************************************************/
/* create an empty environment
 *
 * precondition : true
 */
extern LdbEnv *ldbEnv_makeEmpty();

/* bind a given symbol to a given environment
 *
 * precondition : during semantic analysis
 */
extern LdbEnv *ldbEnv_bind(LdbEnv *env, LdbSymbol *sym);

/** observers **************************************************************/
/* return a given environment as an ldb information (subsumption)
 *
 * precondition : true
 */
extern LdbInfo *ldbEnv_asInfo(LdbEnv *env);
```

Uses `ldbEnv_asInfo` 74, `ldbEnv_bind` 74, and `ldbEnv_makeEmpty` 74.

74 ⟨*env.c* 74⟩≡
⟨*included files* 10b⟩

```
static void env_emptyExtern(e, out) LdbEnv *e; LdbFile *out; {
        if (!e->isExterned) {
                LdbInfo *info =
                name2i(ldbUtil_dropPrefixOfLabelName((const char *)e->ref));
                ldbInfo_extern   (info                          , out);
                ldbUtil_emitLabel((const char *)e->ref          , out);
                ldbUtil_emitName ("0"                           , out);
                ldbUtil_emitName ((const char *)ldbInfo_ref(info), out);
                e->isExterned = 1;
      }
}
static void env_nonemptyExtern(e, out) LdbEnv *e; LdbFile *out; {
        if (!e->isExterned) {
                LdbInfo *info =
                name2i(ldbUtil_dropPrefixOfLabelName((const char *)e->ref));
                ldbInfo_extern(env2i        (e->u.t.env)        , out);
                ldbInfo_extern(symbol2i     (e->u.t.symbol)     , out);
                ldbInfo_extern(info                             , out);
                ldbUtil_emitLabel((const char *)e->ref          , out);
                ldbUtil_emitName ("2"                           , out);
                ldbUtil_emitName ((const char *)ldbInfo_ref(info), out);
                ldbFile_push(out);
                ldbUtil_emitName ((const char *)ldbInfo_ref(env2i(e->u.t.env)), out);
                ldbUtil_emitName ((const char *)ldbInfo_ref(symbol2i(e->u.t.symbol)), out);
                ldbFile_pop(out);
                e->isExterned = 1;
        }
}
static LdbExtRef *env_ref(e) LdbEnv *e; { return e->ref; }

#define env_make(EXT, ISEMPTY) \
        LdbEnv *new    = (LdbEnv *)malloc(sizeof *new); \
        new->i          = ldbInfo_make(); \
        new->i->ext     = EXT; \
        new->i->ref     = &env_ref; \
        new->ref        = ldbUtil_baseToExtRef("E"); \
        new->isExterned = 0; \
        new->isEmpty    = ISEMPTY;

LdbEnv  *ldbEnv_makeEmpty() {
      env_make(&env_emptyExtern, 0);
      return new;
}
```

```
LdbEnv  *ldbEnv_bind(e, s) LdbEnv *e; LdbSymbol *s; {
        env_make(&env_nonemptyExtern, 1);
        new->u.t.env    = e;
        new->u.t.symbol = s;
      return new;
}

LdbInfo *ldbEnv_asInfo(e) LdbEnv *e; { return (LdbInfo *)e; }
```

Defines:
env_emptyExtern, never used.
env_make, never used.
env_nonemptyExtern, never used.
env_ref, never used.
ldbEnv_asInfo, used in chunks 11 and 73.
ldbEnv_bind, used in chunk 73.
ldbEnv_makeEmpty, used in chunk 73.

Uses env2i 11, LdbExtRef 20b 179, ldbFile_pop 17, ldbFile_push 17, ldbInfo_extern 25, ldbInfo_make 25, ldbInfo_ref 25, ldbUtil_baseToExtRef 9, name2i 11, out 94a, and symbol2i 11.

2.4.5 Types

- Abstraction : a type
- Constructors:
  - `ldbType_make` : create a type
- Observers :
  - `ldbType_asTable` : return a given type as a table (subsumption)

When a type is created, how to print a declaration of the type `decl` and how to print a value of the type `printer` are provided. This information is compiler dependent. For example, let's consider the following structure `Fib` and its type `FIB`:

```
/S4 <<>> def
/T5 << /decl       (%s : FIB)
       /printer    {STRUCT}
       /symbol     S4          >> def
%% Backpatches
S4 /name       (Fib)     put
S4 /sourcefile (Fib.c) put
S4 /sourcey    1         put
S4 /sourcex    6         put
S4 /kind       (type)    put
S4 /type       T5        put
S4 /vars       I6        put
S4 /procs      I7        put
```

In order to print the structure `Fib` denoted by `S4` here, the printing method `STRUCT` in `FIB` which is `T5` needs to know which variables and functions are in `S4`. Thus, the `symbol` key in `T5` is bound to `S4` and now `STRUCT` can access the variables and functions in `S4` through the `symbol` key.

76a ⟨*type definitions in ldbinfo.h* 8b⟩+≡ (13a) ◁72a 78a▷

```
typedef struct ldbtype       LdbType;
```

76b ⟨*abstractions in utils.h* 14b⟩+≡ (9) ◁72b 78b▷

```
/** abstraction : type ***************************************************/
struct ldbtype {
        LdbTable   *t;
        const char *decl;
};
```

77a ⟨*type.h* 77a⟩≡ (13b)

```
/** type ***********************************************************************/
/** constructors ***************************************************************/
/* create a fresh type
 *
 * precondition : this type does not exist
 * postcondition: this type is created
 */
extern LdbType *ldbType_make(const char *decl, LdbPsProc *printer);

/** observers ******************************************************************/
/* return a given type as a PostScript dictionary (subsumption)
 *
 * precondition : true
 */
extern LdbTable *ldbType_asTable(LdbType *ty);
```

Uses `ldbType_asTable` 77b and `ldbType_make` 77b.

77b ⟨*type.c* 77b⟩≡

⟨*included files* 10b⟩

```
LdbType *ldbType_make(decl, printer) const char *decl; LdbPsProc *printer; {
        /* every type's external name starts with "T" */
        LdbType  *new = (LdbType *)malloc(sizeof *new);
        LdbTable *t   = ldbTable_make("T");
        /* how to print a declartion of this type */
        ldbTable_put(t, "decl",    str2i(decl));
        /* how to print a value      of this type */
        ldbTable_put(t, "printer", ldbPsProc_asInfo(printer));
        new->t    = t;
        new->decl = decl;
        return new;
}

LdbTable *ldbType_asTable(ty) LdbType *ty; { return ty->t; }
```

Defines:
`ldbType_asTable`, used in chunks 11 and 77a.
`ldbType_make`, used in chunk 77a.
Uses `ldbPsProc_asInfo` 37b, `ldbTable_make` 49, `ldbTable_put` 49, and `str2i` 11.

2.4.6 Placements

- Abstraction : an object location
- Constructors:
  - `ldbPlcmt_makeAbsolute` : create a fundamental kind of location
    `ldbPlcmt_makeAbsolute(s, o) = s[o]`
  - `ldbPlcmt_makeShifted` : create a location relative to some other location
    `ldbPlcmt_makeShifted(s[o'], o) = s[o'+o]`
  - `ldbPlcmt_makeIndirect` : create one level of indirection, with displacement
    `ldbPlcmt_makeIndirect(s, s'[o'], o) = s[s'[o']+o]`
  - `ldbPlcmt_makeAtLabel` : create a placement of a given label
- Observers :
  - `ldbPlcmt_asInfo` : return a given placement as an `ldb` information (subsumption)

78a ⟨*type definitions in ldbinfo.h* 8b⟩+≡ (13a) ◁76a 87a▷

```
typedef struct ldbplcmt      LdbPlcmt;
```

78b ⟨*abstractions in utils.h* 14b⟩+≡ (9) ◁76b 87b▷

```
/** abstraction : placement ********************************************/
struct ldbplcmt {
        LdbInfo      *i; /* asInfo - subsumption */
        LdbImmutable *m;
        LdbPlcmt     *addr;
};
```

79 ⟨*placement.h* 79⟩≡ (13b)

```
/** placement ***************************************************************/
/** constructors ************************************************************/
/* create a fundamental kind of location
 *
 * postcondition: ldbPlcmt_makeAbsolute s o = s[o]
 */
extern LdbPlcmt *ldbPlcmt_makeAbsolute(char space, int offset);

/* create a location relative to some other location
 *
 * postcondition: ldbPlcmt_makeShifted s[o] o' = s[o+o']
 */
extern LdbPlcmt *ldbPlcmt_makeShifted(LdbPlcmt *loc, int offset);

/* create one level of indirection, with displacement
 *
 * postcondition: ldbPlcmt_makeIndirect s s[o] o' = s[s[o]+o']
 */
extern LdbPlcmt *ldbPlcmt_makeIndirect(char space, LdbPlcmt *loc, int offset);

/* create a placement of a given label
 *
 * postcondition: label gets its placement
 */
extern LdbPlcmt *ldbPlcmt_makeAtLabel(LdbLabel *label);

/** observers ***************************************************************/
/* return a given placement as an ldb information (subsumption)
 *
 * precondition : true
 */
extern LdbInfo *ldbPlcmt_asInfo(LdbPlcmt *loc);
```

Uses ldbPlcmt_asInfo 80, ldbPlcmt_makeAbsolute 80, ldbPlcmt_makeAtLabel 83, ldbPlcmt_makeIndirect 80, ldbPlcmt_makeShifted 80, and offset 83 241.

80 ⟨*placement.c* 80⟩≡
⟨*included files* 10b⟩

```
static void plcmt_extern(a, out) LdbPlcmt *a; LdbFile *out; {
        LdbPlcmt *addr = a->addr;
        if (addr != NULL)
                do {
                        ldbImmutable_extern(addr->m, out);
                        addr = addr->addr;
                } while (addr != NULL);
        ldbImmutable_extern(a->m, out);
}
static LdbExtRef  *plcmt_ref(a) LdbPlcmt *a; { return a->m->ref; }
static const char *plcmt_def(a) LdbPlcmt *a; { return a->m->def; }
#define plcmt_make(ADDR,FMT,ARG1,ARG2,ARG3) \
        LdbPlcmt     *new = (LdbPlcmt     *)malloc(sizeof *new); \
        LdbImmutable *m   = (LdbImmutable *)malloc(sizeof *m); \
        char         *s   = malloc(LDB_EXTREF_MAX); \
        sprintf(s, FMT, ARG1, ARG2, ARG3); \
        m->ref      = ldbUtil_baseToExtRef("Im"); \
        m->def      = s; \
        m->externed = 0; \
        new->addr   = ADDR; \
        new->m      = m; \
        new->i      = ldbInfo_make(); \
        new->i->ext = &plcmt_extern; \
        new->i->ref = &plcmt_ref; \
        return new;

LdbPlcmt *ldbPlcmt_makeAbsolute(space, offset) char space; int offset; {
        plcmt_make(0, "%d '%c' Absolute%s", offset, space, "");
}

LdbPlcmt *ldbPlcmt_makeShifted(loc, offset) LdbPlcmt *loc; int offset; {
        plcmt_make(loc, "%s %d Shifted%s", plcmt_def(loc), offset, "");
}

LdbPlcmt *ldbPlcmt_makeIndirect(space, loc, offset) char space; LdbPlcmt *loc;
                                                    int offset; {
        plcmt_make(loc,"%s %d '%c' Indirect",plcmt_def(loc),offset,space);
}

⟨ldbPlcmt_makeAtLabel 83⟩

LdbInfo *ldbPlcmt_asInfo(a) LdbPlcmt *a; { ldbImmutable_asInfo(a); }
```

Defines:

ldbPlcmt_asInfo, used in chunks 61, 66, and 79.
ldbPlcmt_makeAbsolute, used in chunk 79.
ldbPlcmt_makeIndirect, used in chunk 79.
ldbPlcmt_makeShifted, used in chunk 79.
plcmt_def, never used.
plcmt_extern, never used.
plcmt_make, used in chunk 83.
plcmt_ref, never used.

Uses LDB_EXTREF_MAX 20c, LdbExtRef 20b 179, ldbImmutable_asInfo 28, ldbImmutable_extern 28, ldbInfo_make 25, ldbUtil_baseToExtRef 9, offset 83 241, and out 94a.

Lazy address management

- Abstraction : lazy address management using the anchor symbol
- Constructors:
  - `ldbPlcmt_initialize` : create an anchor symbol
- Observers :
  - `ldbPlcmt_finalize` : finalize emitting references to locations in assembly-language output
  - `ldbPlcmt_anchor` : return the name of the anchor symbol

Some addresses are bound by the linker and therefore are not available at compile time. We pass those addresses to the debugger by planting references in the assembly language output. The references are in known locations relative to the anchor symbol, and we refer to them by using the anchor symbol name and the offset of the reference. While we do not need to use this lazy address management any more because we eliminated the use of `.o.st` and `.lt` files, we still use the anchor symbol so that we can use the `ldb` debugger as it is.

82 ⟨*lazy address management* 82⟩≡ (9)

```
/** lazy address management using the anchor symbol *********************/
/** constructors ***********************************************************/
/* initialize lazy address management; create the anchor symbol
 *
 * precondition : true
 * postcondition: the anchor symbol is created
 */
extern void ldbPlcmt_initialize();

/** observers **************************************************************/
/* finalize emitting references to locations in assembly-language output
 *
 * precondition : compiler finished emitting assembly-language output
 * postcondition: all the bound labels are emitted
 */
extern void ldbPlcmt_finalize(LdbFile *out);

/* return the name of the anchor symbol
 *
 * precondition : the anchor symbol is created
 */
extern const char *ldbPlcmt_anchor();
```

Uses `ldbPlcmt_anchor` 83, `ldbPlcmt_finalize` 83, `ldbPlcmt_initialize` 83, and `out` 94a.

83 ⟨ldbPlcmt_makeAtLabel 83⟩≡ (80)

```
static const char *myanchor;
/* the offset from the anchor symbol of the next reference to be planted */
static int offset;
typedef struct ldblabeltoplcmt {
        int       index;
        LdbLabel *label;
        LdbPlcmt *plcmt;
} LdbLabelToPlcmt;
/* all the lazy addresses; (label, placement, offset) list
 * Most placements, like those of loci, are used only once, but procedure
 * placements are needed both in the lazy symbol table and in the full one.
 */
static LdbList labelToPlcmt=0;

void ldbPlcmt_initialize() {
        struct timeval time;
#if sparc
/* struct timezone tz; */
#else
        struct timezone tz;
#endif

#if sparc
  assert(!gettimeofday(&time, NULL));
#else
  assert(!gettimeofday(&time, &tz));
#endif
        myanchor = malloc(LDB_UTIL_MAXLABELNAME);
        sprintf((char *)myanchor, "_stanchor__V%lx_%lx", time.tv_sec,
                time.tv_usec);
}

/* plant a reference to each of the labels relative to the anchor symbol */
void ldbPlcmt_finalize(out) LdbFile *out; {
        int i;
        LdbList l = labelToPlcmt;
        LdbLabel *labelarray[offset];
        ldbUtil_exportName(myanchor, out); /* exports the anchor symbol */
        if (l)
                do {
                        LdbLabelToPlcmt *l2a   = (LdbLabelToPlcmt *) l->x;
                        labelarray[l2a->index] = l2a->label;
                } while ((l = l->link) != labelToPlcmt);
        for (i=0; i < offset; i++) ldbLabel_bindHere(labelarray[i], out);
        ldbUtil_emitDataSection(out);
```

```
}

const char *ldbPlcmt_anchor() { return myanchor; }

static LdbPlcmt *ldb_psCode(code) char *code; {
        plcmt_make(0, "{%s}%s%s", code, "", "");
}
/* return the placement of a specified label, create if not exist */
LdbPlcmt *ldbPlcmt_makeAtLabel(label) LdbLabel *label; {
        char    *psCode = malloc(LDB_EXTREF_MAX);
        LdbList  l      = labelToPlcmt;
        LdbPlcmt *placement;
        LdbLabelToPlcmt *l2a;
        if (l)
                do {
                        l2a = (LdbLabelToPlcmt *) l->x;
                        if (l2a->label == label) return l2a->plcmt;
                } while ((l = l->link) != labelToPlcmt);
        sprintf(psCode, "(%s) %d Lazy%s", myanchor, offset, ldbLabel_space(label));
        placement    = ldb_psCode(psCode);
        l2a          = (LdbLabelToPlcmt *)malloc(sizeof *l2a);
        l2a->label   = label;
        l2a->index   = offset++;
        l2a->plcmt   = placement;
        labelToPlcmt = ldbList_append(labelToPlcmt, l2a);
        return placement;
}
```

Defines:
labelToPlcmt, never used.
ldb_psCode, never used.
LdbLabelToPlcmt, never used.
ldbPlcmt_anchor, used in chunks 82 and 97.
ldbPlcmt_finalize, used in chunks 82 and 97.
ldbPlcmt_initialize, used in chunks 82 and 95.
ldbPlcmt_makeAtLabel, used in chunks 66 and 79.
myanchor, never used.
offset, used in chunks 79, 80, and 238.

Uses LDB_EXTREF_MAX 20c, LDB_UTIL_MAXLABELNAME 23a, ldbLabel_bindHere 86, ldbLabel_space 86, LdbList 8b, ldbList_append 11, out 94a, and plcmt_make 80.

2.4.7 Labels

- Abstraction : a reference to a location in an assembly output (`.s`)
- Constructors:
  - `ldbLabel_makeInCodeSpace` : return a fresh label with a given name in a code space
  - `ldbLabel_makeInDataSpace` : return a fresh label with a given name in a data space
- Observers :
  - `ldbLabel_name` : return this label's name
  - `ldbLabel_space` : return a space where this label is bound to
  - `ldbLabel_bindHere` : bind this label to a location in an assembly output

85 ⟨*label.h* 85⟩≡ (13b)

```
/** label ************************************************************/
/** constructors *****************************************************/
/* return a fresh label with a given name in a code space
 *
 * postcondition: label is unbound
 */
extern LdbLabel *ldbLabel_makeInCodeSpace(const char *name);

/* return a fresh label with a given name is a data space
 *
 * postcondition: label is unbound
 */
extern LdbLabel *ldbLabel_makeInDataSpace(const char *name);

/** observers ********************************************************/
/* return this label's name
 *
 * precondition : a given label's name is set when the label is created
 */
extern const char *ldbLabel_name(LdbLabel *label);

/* return the space name where a given label is bound to
 *
 * precondition : a given label's space name is set when the label is created
 */
extern const char *ldbLabel_space(LdbLabel *label);

/* bind a reference to a location in .s
```

```
 *
 * precondition : label is unbound
 * postcondition: label is bound and emitted to .s
 */
extern void ldbLabel_bindHere(LdbLabel *label, LdbFile *out);
```

Uses ldbLabel_bindHere 86, ldbLabel_makeInCodeSpace 86, ldbLabel_makeInDataSpace 86, ldbLabel_name 86, ldbLabel_space 86, and out 94a.

86 ⟨*label.c* 86⟩≡

⟨*included files* 10b⟩

```
#define label_make(NAME,SPACE) \
        LdbLabel *new = (LdbLabel *)malloc(sizeof *new); \
        new->name     = NAME; \
        new->space    = SPACE; \
        return new;

LdbLabel *ldbLabel_makeInCodeSpace(name) const char *name; {
        label_make(name, "Code");
}
LdbLabel *ldbLabel_makeInDataSpace(name) const char *name; {
        label_make(name, "Data");
}
const char *ldbLabel_name (label) LdbLabel *label; { return label->name; }
const char *ldbLabel_space(label) LdbLabel *label; { return label->space; }
void ldbLabel_bindHere(label, out) LdbLabel *label; LdbFile *out; {
        ldbUtil_emitName(label->name, out);
}
```

Defines:
ldbLabel_bindHere, used in chunks 83 and 85.
ldbLabel_makeInCodeSpace, used in chunk 85.
ldbLabel_makeInDataSpace, used in chunk 85.
ldbLabel_name, used in chunk 85.
ldbLabel_space, used in chunks 83 and 85.
Uses out 94a.

2.4.8 Stopping points

- Abstraction : a stopping point
- Constructors:
  - `ldbLocus_make` : create a stopping point
- Observers :
  - `ldbLocus_file` : return the file name of this stopping point
  - `ldbLocus_line` : return the line number of this stopping point
  - `ldbLocus_column` : return the column number of this stopping point
  - `ldbLocus_label` : return the label of this stopping point
  - `ldbLocus_env` : return the environment at this stopping point

Compilers usually have a set of stopping points where an execution of a given program might stop. For example, MiniJava stopping points occur before every expression, before the operands of `&&` and `||`, before every statement, at the beginning and end of block, and at the entry and exit to every method.

87a ⟨*type definitions in ldbinfo.h* 8b⟩+≡ (13a) ◁78a

```
typedef struct ldblocus      LdbLocus;
```

87b ⟨*abstractions in utils.h* 14b⟩+≡ (9) ◁78b 91▷

```
/** abstraction : locus ************************************************/
struct ldblocus {
        const char *file;
        int         line;
        int         column;
        LdbLabel   *label;
        LdbEnv     *env;    /* the environment at this stopping point */
};
```

88 ⟨*locus.h* 88⟩≡ (13b)

```
/** locus *************************************************************/
/** constructors ******************************************************/
/* create a fresh stopping point
 *
 * precondition : this stopping point does not exist
 * postcondition: this stopping point is created
 */
extern LdbLocus *ldbLocus_make(LdbSrcLoc *src, LdbLabel *label, LdbEnv *env);

/** observers *********************************************************/
/* return the file name of a given stopping point
 *
 * precondition  : true
 */
extern const char *ldbLocus_file(LdbLocus *locus);

/* return the line number of a given stopping point
 *
 * precondition  : true
 */
extern int ldbLocus_line(LdbLocus *locus);

/* return the column number of a given stopping point
 *
 * precondition  : true
 */
extern int ldbLocus_column(LdbLocus *locus);

/* return the label of a given stopping point
 *
 * precondition  : true
 */
extern LdbLabel *ldbLocus_label(LdbLocus *locus);

/* return the environment at a given stopping point
 *
 * precondition  : true
 */
extern LdbEnv *ldbLocus_env(LdbLocus *locus);
```

Uses ldbLocus_column 89, ldbLocus_env 89, ldbLocus_file 89, ldbLocus_label 89, ldbLocus_line 89, and ldbLocus_make 89.

89 ⟨*locus.c* 89⟩≡
⟨*included files* 10b⟩

```
/* set the name of this stopping point with a new unique name
 * This name has to be usable both as PostScript names and
 * as assembly-language labels.
 */
LdbLocus *ldbLocus_make(src, label, env) LdbSrcLoc *src; LdbLabel *label;
                                         LdbEnv *env; {
        LdbLocus *new = (LdbLocus *)malloc(sizeof *new);
        new->file     = ldbSrcLoc_file  (src);
        new->line     = ldbSrcLoc_line  (src);
        new->column   = ldbSrcLoc_column(src);
        new->label    = label;
        new->env      = env;
        return new;
}
const char *ldbLocus_file  (locus) LdbLocus *locus; { return locus->file;   }
int         ldbLocus_line  (locus) LdbLocus *locus; { return locus->line;   }
int         ldbLocus_column(locus) LdbLocus *locus; { return locus->column; }
LdbLabel   *ldbLocus_label (locus) LdbLocus *locus; { return locus->label;  }
LdbEnv     *ldbLocus_env   (locus) LdbLocus *locus; { return locus->env;    }
```

Defines:
`ldbLocus_column`, used in chunks 66 and 88.
`ldbLocus_env`, used in chunks 66 and 88.
`ldbLocus_file`, used in chunks 66 and 88.
`ldbLocus_label`, used in chunks 66 and 88.
`ldbLocus_line`, used in chunks 66 and 88.
`ldbLocus_make`, used in chunk 88.
Uses `ldbSrcLoc_column` 52a, `ldbSrcLoc_file` 52a, and `ldbSrcLoc_line` 52a.

2.4.9 Compilation units

- Abstraction : a compilation unit
- Constructors:
  - `ldbCompUnit_make` : create a compilation unit
  - `ldbCompUnit_addProc` : add a procedure to this compilation unit
  - `ldbCompUnit_addExported` : add an exported symbol to this compilation unit
  - `ldbCompUnit_addPrivate` : add a private symbol to this compilation unit
- Observers :
  - `ldbCompUnit_extern` : externalize this compilation unit

The `ldbCompUnit_extern` method emits the symbol table information of this compilation unit. There are entries in the `ldb_st` section which point to the top-level symbol table information and the actual data is placed in the `ldb_st_data` section. For example, the following information is emitted to the assembly-language output:

```
.section .ldb_st
.long .Lldb_st_src
.long .Lldb_st_arch
.long .Lldb_st_anchor
.long .Lldb_st_procs
.long .Lldb_st_externs
.long .Lldb_st_privates
.long .Lldb_st_locals
.long .Lldb_st_sourcemap
.section .ldb_st_data
.Lldb_st_src:
.byte 116
.byte 101
.byte 115
.byte 116
.byte 46
.byte 99
.byte 0
.Lldb_st_arch:
...
.Lldb_st_anchor:
...
.Lldb_st_procs:
.long 2
```

```
.long .LS23
.long .LS31
.Lldb_st_externs:
.long 2
.long .LIm161
.long .LS23
.long .LIm163
.long .LS31
.Lldb_st_privates:
.long 2
.long .LIm179
.long .LS16
.long .LIm181
.long .LS38
.Lldb_st_locals:
.long 1
.long .LS10
.Lldb_st_sourcemap:
.long 2
.long .LIm33
.long 1
.long .LS31
.long .LIm12
.long 1
.long .LS23
```

Since the compilation unit is the top-level information, `ldbCompUnit_ref` is not defined and `ldbCompUnit_make` creates a compilation unit, which is an implicit part of the inteface's state.

91 ⟨*abstractions in utils.h* 14b⟩+≡ (9) ◁87b

```
/** abstraction : compilation unit ***************************************/
typedef struct ldbcompunit {
        LdbTable   *t;
        const char *arch;   /* architecture name */
        const char *src;    /* source file  name */
        LdbFile  *asmout;   /* assembly-language output */
        LdbArray *procs;    /* procedure symbols in this  compilation unit  */
        LdbArray *exported; /* exported  symbols to other compilation units */
        LdbArray *privates; /* private   symbols in this  compilation unit  */
        LdbArray *locals;   /* local     symbols in this  compilation unit  */
        LdbTable *srcmap;   /* a mapping between file names and
                             * their procedures                             */
} LdbCompUnit;
```

Defines:
`LdbCompUnit`, used in chunks 94a and 95.

Uses `exported` 96a 254 and `privates` 96a 254.

93 ⟨*compilation-unit.h* 93⟩≡ (13b)

```
/** compilation unit ********************************************************/
/** constructors ************************************************************/
/* create an empty compilation unit
 *
 * precondition : architecture name, source file name, and
 *                a file handle for assembly-language output are given
 *                for example, "mips", "test.c", and a file handle for
 *                "test.s" are given
 * postcondition: the compilation unit has its architecture name,
 *                source file name, file handle for assembly-language
 *                output, and anchor symbol name
 */
extern void ldbCompUnit_make(const char *arch, const char *src, LdbFile *out);

/* add a procedure symbol to this compilation unit
 *
 * precondition : procedure symbol is not in the compilation unit
 * postcondition: procedure symbol is added to the compilation unit;
 *                this procedure symbol is defined in this compilation unit
 */
extern void ldbCompUnit_addProc(LdbProc *p);

/* add an exported symbol to this compilation unit
 *
 * precondition : exported symbol is not in the compilation unit
 * postcondition: exported symbol is added to the compilation unit;
 *                this symbol is exported to the other compilation units
 */
extern void ldbCompUnit_addExported(LdbSymbol *s);

/* add a privte symbol to this compilation unit
 *
 * precondition : private symbol is not in the compilation unit
 * postcondition: private symbol is added to the compilation unit;
 *                this symbol is private to this compilation unit
 */
extern void ldbCompUnit_addPrivate(LdbSymbol *s);

/** observers ***************************************************************/
/* externalize this compilation unit
 *
 * precondition : file handle must be at top level
 * postcondition: file handle is still at top level,
 *                and this compilation unit has been externalized
 */
```

```
extern void ldbCompUnit_extern(LdbFile *out);
```

Uses `exported` 96a 254, `ldbCompUnit_addExported` 96b, `ldbCompUnit_addPrivate` 96b, `ldbCompUnit_addProc` 95, `ldbCompUnit_extern` 97, `ldbCompUnit_make` 95, and `out` 94a.

94a ⟨*compilation unit : declarations* 94a⟩≡ (94c) 96a▷

```
static LdbCompUnit *cu;
static LdbFile     *out;     /* symbol-table output file handle */
```

Defines:
`cu`, used in chunks 95–97 and 100a.
`out`, used in chunks 14–17, 22–25, 31a, 33a, 34b, 36a, 37b, 39, 46, 49, 54a, 74, 80, 82, 83, 85, 86, 93, 94b, 96c, and 97.
Uses `LdbCompUnit` 91 251.

94b ⟨*compilation unit : definitions* 94b⟩≡ (94c) 96c▷

```
#define wr0(STRING)        ldbFile_write(out, STRING);
#define wr1(FMT,ARG)       ldbFile_write(out, FMT,ARG);
#define wr2(FMT,ARG1,ARG2) ldbFile_write(out, FMT,ARG1,ARG2);
```

Defines:
`wr0`, never used.
`wr1`, never used.
`wr2`, never used.
Uses `ldbFile_write` 17 and `out` 94a.

94c ⟨*compilation-unit.c* 94c⟩≡

⟨*included files* 10b⟩

⟨*compilation unit : declarations* 94a⟩
⟨*compilation unit : definitions* 94b⟩
⟨*compilation unit : constructors* 95⟩
⟨`ldbCompUnit_extern`*: emit the symbol table information* 97⟩

Constructors

95 ⟨*compilation unit : constructors* 95⟩≡ (94c) 96b▷

```
/* create an empty compilation unit */
void ldbCompUnit_make(arch, src, asmout) const char *arch, *src;
                                         LdbFile *asmout; {
        ldbPlcmt_initialize();
        cu = (LdbCompUnit *)malloc(sizeof *cu);
        cu->arch     = arch;
        cu->src      = src;
        cu->asmout   = asmout;
        cu->procs    = ldbArray_make("I");
        cu->exported = ldbArray_make("I");
        cu->privates = ldbArray_make("I");
        cu->locals   = ldbArray_make("I");
        cu->srcmap   = ldbTable_make("I");
}

#define addSym(ARR,SYM) \
        int i; \
        for (i=0; i<ARR->n; i++) \
                if (equallsym(SYM, ldbArray_get(ARR, i))) return; \
        ldbArray_addHigh(ARR, symbol2i(SYM));
/* add a procedure symbol to procs */
void ldbCompUnit_addProc(p) LdbProc *p; {
        ldbArray_addHigh(cu->procs, proc2i(p));
        savesourcemap(ldbProc_file(p), p);
        { LdbList locals = ldbProc_locals(p);
          LdbList l      = locals;
          if (l)
            do { addSym(cu->locals, (LdbSymbol *)l->x);
               } while ((l = l->link) != locals);
        }
}
```

Defines:
addSym, used in chunk 96b.
ldbCompUnit_addProc, used in chunk 93.
ldbCompUnit_make, used in chunk 93.
Uses cu 94a, equallsym 96b, exported 96a 254, ldbArray_addHigh 46, ldbArray_get 46, ldbArray_make 46, LdbCompUnit 91 251, LdbList 8b, ldbPlcmt_initialize 83, ldbProc_file 66, ldbProc_locals 66, ldbTable_make 49, n 21 179 186, privates 96a 254, proc2i 11, savesourcemap 100a, and symbol2i 11.

96a ⟨*compilation unit : declarations* 94a⟩+≡ (94c) ◁94a 99a▷

```
static int exported_id;
static int privates_id;
static const char *exported[LDB_ARRAY_MAX];
static const char *privates[LDB_ARRAY_MAX];
static int equallsym (void *, void *);
```

Defines:
`exported`, used in chunks 91, 93, 95–97, 249, and 251.
`exported_id`, used in chunk 96b.
`privates`, used in chunks 91, 95–97, and 251.
`privates_id`, used in chunk 96b.
Uses `equallsym` 96b and `LDB_ARRAY_MAX` 44b.

96b ⟨*compilation unit : constructors* 95⟩+≡ (94c) ◁95

```
static int equallsym(closure, data) void *closure; void *data; {
        return closure==(LdbSymbol *)data;
}
/* add an exported symbol to exported */
void ldbCompUnit_addExported(s) LdbSymbol *s; {
        addSym(cu->exported, s); exported[exported_id++] = s->name;
}
/* add a private symbol to privates */
void ldbCompUnit_addPrivate (s) LdbSymbol *s; {
        addSym(cu->privates, s); privates[privates_id++] = s->name;
}
```

Defines:
`equallsym`, used in chunks 95 and 96a.
`ldbCompUnit_addExported`, used in chunk 93.
`ldbCompUnit_addPrivate`, used in chunk 93.
Uses `addSym` 95, `cu` 94a, `exported` 96a 254, `exported_id` 96a, `privates` 96a 254, and `privates_id` 96a.

Structure of the output

Emitting the symbol table involves several iterations over symbols, all done by `dosymbols()`, which performs an action on each symbol in a given set of symbols:

96c ⟨*compilation unit : definitions* 94b⟩+≡ (94c) ◁94b 100a▷

```
/* dosymbols - do ACTION to each symbol in SYMBOLS */
#define dosymbols(SYMS,ACTION) { int i; for (i=0; i<SYMS->n; i++) { ACTION; } }
/* emitsymbols - emit each symbol in ARR */
#define emitsymbols(ARR) dosymbols(ARR, ldbInfo_extern((ARR->array[i]), out))
```

Defines:
`dosymbols`, used in chunk 97.
`emitsymbols`, used in chunk 97.
Uses `ldbInfo_extern` 25, `n` 21 179 186, and `out` 94a.

97 ⟨ldbCompUnit_extern: *emit the symbol table information* 97⟩≡ (94c)

```
#define emitsize(ARR) \
        s = malloc(sizeof(LDB_UTIL_MAXDIGIT)); \
        sprintf(s, "%d", ARR->n); \
        ldbUtil_emitName((const char *)s, out);
#define emitstr(STR) \
        LdbInfo *i = str2i(STR); \
        ldbInfo_extern(i, out); \
        ldbUtil_emitName((const char *)ldbInfo_ref(i), out);
#define emitref(REF) ldbUtil_emitName((const char*)ldbInfo_ref(REF), out)

static const char* mkLabelName(fmt) char *fmt; {
        const char *s = malloc(LDB_EXTREF_MAX);
        sprintf((char *)s, fmt, ldbPlcmt_anchor());
        s = ldbUtil_makeLabelNameFrom(s);
        return s;
}

/* emit the symbol table information */
void ldbCompUnit_extern(o) LdbFile *o; {
        char *s;
        const char *lab_src       = mkLabelName("%s_src");
        const char *lab_arch      = mkLabelName("%s_arch");
        const char *lab_anchor    = mkLabelName("%s_anchor");
        const char *lab_procs     = mkLabelName("%s_procs");
        const char *lab_externs   = mkLabelName("%s_externs");
        const char *lab_privates  = mkLabelName("%s_privates");
        const char *lab_locals    = mkLabelName("%s_locals");
        const char *lab_sourcemap = mkLabelName("%s_sourcemap");
        out = o;
        ldbPlcmt_finalize(out);
        /* emit all the procedures       */
        dosymbols  (cu->procs,    ldbInfo_extern(cu->procs->array[i] , out));
        emitsymbols(cu->exported); /* emit all the exported symbols */
        dosymbols  (cu->exported, ldbInfo_extern(name2i(exported[i]), out));
        emitsymbols(cu->privates); /* emit all the private symbols  */
        dosymbols  (cu->privates, ldbInfo_extern(name2i(privates[i]), out));
        emitsymbols(cu->locals);   /* emit all the local symbols    */
        ldbUtil_emitSection("ldb_st"         , out);
        ldbUtil_emitName   (lab_src          , out);
        ldbUtil_emitName   (lab_arch         , out);
        ldbUtil_emitName   (lab_anchor       , out);
        ldbUtil_emitName   (lab_procs        , out);
        ldbUtil_emitName   (lab_externs      , out);
        ldbUtil_emitName   (lab_privates     , out);
        ldbUtil_emitName   (lab_locals       , out);
```

```
        ldbUtil_emitName   (lab_sourcemap    , out);
        ldbUtil_emitSection("ldb_st_data"    , out);
        ldbUtil_emitLabel  (lab_src          , out);
        ldbUtil_emitStr    (cu->src          , out);
        ldbUtil_emitLabel  (lab_arch         , out);
        ldbUtil_emitStr    (cu->arch         , out);
        ldbUtil_emitLabel  (lab_anchor       , out);
        ldbUtil_emitStr    (ldbPlcmt_anchor(), out);
        ldbUtil_emitLabel  (lab_procs        , out);
        emitsize (cu->procs);    /* number of the procedures */
        dosymbols(cu->procs,     /* procedures in the compilation unit */
                  emitref(cu->procs->array[i]));
        ldbUtil_emitLabel(lab_externs, out);
        emitsize (cu->exported); /* number of the exported symbols */
        dosymbols(cu->exported,  /* exported symbols in the CU */
                  { emitref(name2i(exported[i]));
                    emitref(cu->exported->array[i]); });
        ldbUtil_emitLabel(lab_privates, out);
        emitsize (cu->privates); /* number of the private symbols */
        dosymbols(cu->privates,  /* private symbols in the CU */
                  { emitref(name2i(privates[i]));
                    emitref(cu->privates->array[i]); });
        ldbUtil_emitLabel(lab_locals, out);
        emitsize (cu->locals);   /* number of the local symbols */
        dosymbols(cu->locals,    /* local symbols in the CU */
                  emitref(cu->locals->array[i]));
        ldbUtil_emitLabel(lab_sourcemap, out);
        emitsize(cu->srcmap);    /* number of the source files */
        /* sourcemap in the compilation unit */
        { LdbList l = filenames;
          if (l)
            do {
                 char     *key = (char *)l->x;
                 LdbArray *arr = (LdbArray *)ldbTable_get(cu->srcmap, key);
                 emitstr (key);
                 emitsize(arr);
                 dosymbols(arr, emitref(arr->array[i]));
               } while ((l = l->link) != filenames);
        }
        ldbUtil_emitDataSection(out);
    }
```

Defines:
  `emitref`, never used.
  `emitsize`, never used.
  `emitstr`, never used.

`ldbCompUnit_extern`, used in chunk 93.
`mkLabelName`, never used.
Uses `cu` 94a, `dosymbols` 96c, `emitsymbols` 96c, `exported` 96a 254, `filenames` 99b, `LDB_EXTREF_MAX` 20c, `LDB_UTIL_MAXDIGIT` 10a, `ldbInfo_extern` 25, `ldbInfo_ref` 25, `LdbList` 8b, `ldbPlcmt_anchor` 83, `ldbPlcmt_finalize` 83, `ldbTable_get` 49, `n` 21 179 186, `name2i` 11, `out` 94a, `privates` 96a 254, and `str2i` 11.

Mapping source code to object code

To be able to set breakpoints by source coordinate, the debugger needs to be able to map any source coordinate to an execution point. It computes the map in two stages: first it checks the file name to find all the procedures that contain source code from that file; then it looks up the source coordinate in each of the procedures to find the nearest execution point, if any. The compiler supports this operation by emitting a dictionary mapping a source file name to an array of arrays of procedures that have source in that file. It builds up a list of filenames and the procedures associated with each by calling `savesourcemap` whenever the procedure is added to the compilation unit by calling `addProc`.

99a ⟨*compilation unit : declarations* 94a⟩+≡ (94c) ◁96a 99b▷

```
static int equal (void *, void *);
static void savesourcemap (const char *filename, LdbProc *proc);
```

Uses `equal` 100a and `savesourcemap` 100a.

99b ⟨*compilation unit : declarations* 94a⟩+≡ (94c) ◁99a

```
static LdbList filenames=0; /* list of all the file names */
```

Defines:
`filenames`, used in chunks 97 and 100a.
Uses `LdbList` 8b.

100a ⟨*compilation unit : definitions* 94b⟩+≡ (94c) ◁96c

```
static int equal(closure, data) void *closure; void *data; {
        return closure==data;
}
/* savesourcemap - incrementally build up cu->srcmap */
void savesourcemap(filename, proc) const char *filename; LdbProc *proc; {
        ⟨if filename and proc are the same as the previous call, return 100b⟩

        { LdbArray *arr = (LdbArray *)ldbTable_get(cu->srcmap, filename);
          /* if filename exists */
          if (arr) {
                /* and if new procedure */
                int j;
                for (j=0; j<arr->n; j++)
                        if (equal(proc, ldbArray_get(arr, j))) return;
                /* add it */
                ldbArray_addHigh(arr, proc2i(proc));
          } else {
          /* new filename; add new sourcemap with (filename, proc) */
                LdbArray *procs = ldbArray_make("I");
                ldbArray_addHigh(procs, proc2i(proc));
                ldbTable_put(cu->srcmap, filename, array2i(procs));
                filenames = ldbList_append(filenames, (char *)filename);
          }
        }
}
```

Defines:
  `equal`, used in chunks 16 and 99a.
  `savesourcemap`, used in chunks 95 and 99a.
Uses `array2i` 11, `cu` 94a, `filenames` 99b, `ldbArray_addHigh` 46, `ldbArray_get` 46, `ldbArray_make` 46, `ldbList_append` 11, `ldbTable_get` 49, `ldbTable_put` 49, `n` 21 179 186, and `proc2i` 11.

100b ⟨*if* `filename` *and* `proc` *are the same as the previous call,* `return` 100b⟩≡ (100a)

```
static const char *lastfile = 0;
static LdbProc    *lastproc = 0;

if (filename==lastfile && proc==lastproc)
      return;
lastfile = filename;
lastproc = proc;
```

Defines:
  `lastfile`, never used.
  `lastproc`, never used.

Chapter 3

The `ldb` interface in Java

3.1 Legend

We use the following naming conventions for the `ldb` interface in Java:

Token	Naming convention	Example
Class	Begin with `Ldb`. Initial upper case. Use embedded caps for multiword names.	`LdbFileHandle`
Instance, variables, methods	Initial lower case. Use embedded caps for multiword names.	`fileHandle`

Abbreviations are used only for the followings:

`arg`	argument
`comp`	compilation
`env`	environment
`ext`	external
`float`	floating-point numbers
`hex`	hexadecimal
`info`	information
`int`	integer
`loc`	location
`loci`	stopping points
`locus`	a stopping point
`plcmt`	placement
`proc`	procedure
`ps`	PostScript
`ref`	reference
`reg`	register
`src`	source
`str`	string
`var`	variable

3.2 Preliminaries

This section describes the preliminary `ldb` interface which should be the source-language independent, the compiler independent but implementation-language (ajva) dependent.

3.2.1 File handle

- Abstraction : a file handle to control emitting debugging information
- Constructors:
  - `LdbFileHandle`: open a specified file
- Observers :
  - `close`: close the open file
  - `write`: write a given string to the file
  - `println`: write a given string with a new line to the file
  - `push`: increase the current level
  - `pop`: decrease the current level
  - `assertTop`: make sure that the current level is 'top level'

`push`, `pop`, and `assertTop` are provided to maintain the current level of a file handle. So that the `ldb` interface can keep the following invariance:

- When a debugging information is to be emitted, the file handle is at 'top level'.

102 ⟨*LdbFileHandle.java* 102⟩≡

```
import java.io.FileOutputStream;

/**
 * A handle on an open file.
 * Might one day include more, e.g., a string table, etc
 * this file represents a sequence of top-level declarations
 * (name-value bindings).
 * May be at 'top level' or not
 */
public class LdbFileHandle {
    ⟨LdbFileHandle.java: abstraction 103a⟩
    ⟨LdbFileHandle.java: constructors 103b⟩
    ⟨LdbFileHandle.java: observers 104⟩
}
```

103a ⟨*LdbFileHandle.java: abstraction* 103a⟩≡ (102)

```
/**
 * The file where debugging informaion would be written.
 */
private FileOutputStream file;
/**
 * Current level
 */
private int level = 0;
```

103b ⟨*LdbFileHandle.java: constructors* 103b⟩≡ (102)

```
/**
 * Class constructor
 *
 * precondition : file is not open
 * postcondition: file is open and its default current level is 0
 *
 * @param s output file name
 */
public LdbFileHandle (String s) throws java.io.IOException {
    file = new FileOutputStream(s);
}
```

104 ⟨*LdbFileHandle.java: observers* 104⟩≡ (102)

```
/**
 * Closes this file.
 *
 * precondition : file is open
 * postcondition: file is closed
 */
public void close() throws java.io.IOException { file.close(); }
/**
 * Writes the string to this file.
 *
 * precondition : file is open
 * postcondition: string is written to the file
 *
 * @param s the string to be written
 */
public void write(String s) throws java.io.IOException {
    file.write(s.getBytes());
}
/**
 * Writes the string with a new line to this file.
 *
 * precondition : file is open
 * postcondition: string and a new line are written to the file
 *
 * @param s the string to be written
 */
public void println(String s) throws java.io.IOException {
    file.write((s+"\n").getBytes());
}
/**
 * Increases current level.
 *
 * precondition : current level is equal or greater than 0
 * postcondition: current level is increased and greater than 0
 */
public void push() { level++; }
/**
 * Decreases current level.
 *
 * precondition : current level is greater than 0
 * postcondition: current level is decreased and equal or greater than 0
 */
public void pop()  { level--; }
/**
 * Makes sure that current level is ‘top level’.
```

```
 *
 * precondition : current level is equal or greater than 0
 * postcondition: if the current level is not 0, assertion error
 */
public void assertTop() { assert(level == 0); }
```

3.2.2 External references

- Abstraction : a reference to an item that has been externed
- Constructors:
  - `fromBase`: create an external reference from a base name and a generated number
  - `fromStr`: create an external reference from a string
- Observers :
  - `write`: write this external reference to a given file handle
  - `toStr`: return the string representation of this external reference

External references satisfy the following invariance:

- The external reference of every immutable information is its string representation.
- The external reference of every mutable information is a uniquely generated name.

105 ⟨*LdbExtRef.java* 105⟩≡

```
import java.util.Random;

/**
 * A reference to an item that has been externed.
 */
public class LdbExtRef {
    ⟨LdbExtRef.java: abstraction 106a⟩
    ⟨LdbExtRef.java: constructors 106b⟩
    ⟨LdbExtRef.java: observers 107a⟩
}
```

106a ⟨*LdbExtRef.java: abstraction* 106a⟩≡ (105)

```
/**
 * This reference's external name
 */
private String extref;
/*
 * Integer value to form an external name
 */
private static int n;
```

106b ⟨*LdbExtRef.java: constructors* 106b⟩≡ (105)

```
public LdbExtRef () {}

/**
 * External reference for LdbSymbol or LdbType
 *
 * @param base base name for this reference's external name
 */
public LdbExtRef fromBase(String base) {
    assert (base != "");
    extref = base + n++;
    return this;
}

/**
 * External reference for LdbStr
 *
 * @param s string content
 */
public LdbExtRef fromStr (String s) {
    extref = s;
    return this;
}
```

107a ⟨*LdbExtRef.java: observers* 107a⟩≡ (105)

```
/**
 * Writes this external reference to a given file handle
 *
 * precondition : a given file handle is open
 * postcondition: this external reference is written
 *
 * @param out a file handle to which thie external reference is written
 */
public void write(LdbFileHandle out) throws java.io.IOException {
    out.write(extref);
}

/**
 * Returns this external reference as a string
 *
 * @return this external reference as a string
 */
public String toStr() { return extref; }
```

3.2.3 Compiler supports

Utility functions from a compiler for the `ldb` interface.

107b ⟨*LdbUtil.java* 107b⟩≡

```
/**
 * Utility methods from the compiler for ldb I/F
 */
public class LdbUtil {
    ⟨LdbUtil.java: labels 108⟩
    ⟨LdbUtil.java: assembly output 109⟩
    ⟨LdbUtil.java: private data 110⟩
}
```

108 ⟨*LdbUtil.java: labels* 108⟩≡ (107b)

```
/**
 * Returns a fresh compiler/platform specific label name
 *
 * @return a new label name
 */
public static String makeLabelName() { return ""; }

/**
 * Returns a fresh compiler/platform specific label name from a given name
 *
 * @param n a base name for a new label
 * @return  a new label name
 */
public static String makeLabelNameFrom (String n) { return n; }

/**
 * Returns a base name from a given compiler/platform-specific label name
 *
 * @param label a label name
 * @return      a base name from the given label name
 */
public static String dropPrefixOfLabelName (String label) { return label; }

/**
 * Returns a no-op format to tag a given stopping point's name in the
 * assembly-language output
 *
 * @param locus the name of a stopping point to be planted
 * @return      a no-op format to tag a given stopping point
 */
public static String noOpFormat (String locus) { return locus; }
```

109 ⟨*LdbUtil.java: assembly output* 109⟩≡ (107b)

```
/**
 * Exports a given symbol name to a given file handle
 *
 * @param symbol a symbol name to be exported (anchor symbol)
 * @param h      a file handle (assembly-language output)
 */
public static void exportAsm(String symbol, LdbFileHandle h) {}

/**
 * Emits a given data name to a given file handle
 *
 * @param symbol a data name to be emitted (labels)
 * @param h      a file handle (assembly-language output)
 */
public static void valueAsm(String data, LdbFileHandle h) {}

/**
 * Emits a given label name to a given file handle
 *
 * @param data a label name to be emitted
 * @param h    a file handle (assembly-language output)
 */
public static void emitLabel(String data, LdbFileHandle h) {}

/**
 * Emits a given string to a given file handle
 *
 * @param data a string to be emitted
 * @param h    a file handle (assembly-language output)
 */
public static void emitStr(String data, LdbFileHandle h) {}

/**
 * Emits a given data name to a given file handle
 *
 * @param data a name to be emitted
 * @param h    a file handle (assembly-language output)
 */
public static void emitName(String data, LdbFileHandle h) {}

/**
 * Eexports a given anchor symbol name to a given file handle
 *
 * @param data a name to be exported
 * @param h    a file handle (assembly-language output)
```

```
 */
public static void exportName(String data, LdbFileHandle h) {}

/**
 * Emits a given section name to a given file handle
 *
 * @param data a section name to be emitted
 * @param h    a file handle (assembly-language output)
 */
public static void emitSectionOpen(String data, LdbFileHandle h) {}

/**
 * Emits a section closure    to a given file handle
 *
 * @param data a section closure to be emitted
 * @param h    a file handle (assembly-language output)
 */
public static void emitSectionClose(LdbFileHandle h) {}

/**
 * Emits a data section
 *
 * @param data a data section to be emitted
 * @param h    a file handle (assembly-language output)
 */
public static void emitDataSection(LdbFileHandle h) {}
```

110 ⟨*LdbUtil.java: private data* 110⟩≡ (107b)

```
/**
 * Emits compiler's private data possibly for its expression server
 *
 * @param h a file handle (symbol-table output)
 */
public static void emitPrivate(LdbFileHandle h) {}
```

3.3 Generic PostScript data

This section includes the `ldb` interface for generic PostScript data which should be the source-language independent, the compiler independent, but implementation-language (Java) dependent.

3.3.1 PostScript data (`ldb` information)

- Abstraction : the abstract base class for all PostScript data
- Constructors:
  - `setExtRef`: set this information's external reference
- Observers :
  - `extern`: externalize this information
  - `ref`: return this information's external reference

An `ldb` information satisfies the following invariance:

* `extern` is called after the information has been created.
* `ref` is called after the information has been externalized.

111a ⟨*LdbInfo.java* 111a⟩≡

```
/**
 * The abstract base class for all PostScript data.
 * This is specific to string-based or token-based representation of PostScript
 * (perhaps).
 */
public abstract class LdbInfo {
    ⟨LdbInfo.java: abstraction 111b⟩
    ⟨LdbInfo.java: constructors 112a⟩
    ⟨LdbInfo.java: observers 112b⟩
}
```

111b ⟨*LdbInfo.java: abstraction* 111b⟩≡ (111a)

```
/**
 * This information's external name.
 */
private LdbExtRef extRef;
```

112a ⟨*LdbInfo.java: constructors* 112a⟩≡ (111a)

```
/**
 * Sets this information's external reference.
 *
 * precondition : this information does not have its external reference
 * postcondition: this information's external reference is set
 *
 * @param e this information's external reference
 */
public void setExtRef(LdbExtRef e) { extRef = e; }
```

112b ⟨*LdbInfo.java: observers* 112b⟩≡ (111a)

```
/**
 * Externs this information.
 *
 * precondition : file handle must be at top level
 * postcondition: file handle is still at top level,
 *                and value has been externed
 *
 * @param f file where this information to be externed
 */
public void extern(LdbFileHandle f) {
    if (f != null) f.assertTop();
}

/**
 * Returns this information's external reference.
 *
 * precondition : this information's external reference has been set
 * postcondition: the reference may be written anywhere in a file handle
 *
 * @return this informaion's external reference
 */
public LdbExtRef ref() { return extRef; }
```

3.3.2 Immutable PostScript data

`LdbImmutable` extends the `LdbInfo` class.

- Abstraction : the abstract base class for all immutable PostScript data
- Constructors:
  - `setDef`: set this immutable information's definition
- Observers :
  - `getDef`: return this immutable information's definition

Each immutable information is externalized by `extern` as follows. If it's not yet externalized, its label is emitted followed by `0` to denote that this information is immutable and then the label for the string representation of the immutable information is emitted. The string representation and its label are emitted before the immutable information is emitted.

For example, `(Test.java)` is externalized as follows:

```
.LIg71:
.byte 40
.byte 84
.byte 101
.byte 115
.byte 116
.byte 46
.byte 106
.byte 97
.byte 118
.byte 97
.byte 41
.byte 0
.LIm12:
.long 0
.long .LIg71
```

Its label `.LIm12` is emitted followed by `0` and `.LIg71` which represents the string `(Test.java)`.

113 ⟨*LdbImmutable.java* 113⟩≡

```
/**
 * The abstract base class for all immutable information.
 */
public abstract class LdbImmutable extends LdbInfo {
    ⟨LdbImmutable.java: abstraction 114a⟩
    ⟨LdbImmutable.java: constructors 114b⟩
    ⟨LdbImmutable.java: observers 114c⟩
}
```

114a ⟨*LdbImmutable.java: abstraction* 114a⟩≡ (113)

```
/**
 * Whether this immutable information is externed
 */
private boolean isExterned = false;

/**
 * The definition of this immutable information
 */
private String myDef = null;
```

114b ⟨*LdbImmutable.java: constructors* 114b⟩≡ (113)

```
/**
 * Sets this immutable information's definition
 *
 * @param  this immutable information's definition
 */
public void setDef(String def) { myDef = def; }
```

114c ⟨*LdbImmutable.java: observers* 114c⟩≡ (113)

```
/**
 * Gets this immutable information's definition
 *
 * @return this immutable information's definition
 */
public String getDef() { return myDef; }

/**
 * Externs this immutable information.
 *
 * @param h file handle where this immutable information to be externalized
 */
public void extern(LdbFileHandle h) {
    super.extern(h);
    if (!isExterned) {
      String defLabel = new LdbExtRef().fromBase("Ig").toStr();
      setExtRef(new LdbExtRef().fromBase("Im"));
      LdbUtil.emitLabel(defLabel     , h);
      LdbUtil.emitStr  (myDef        , h);
      LdbUtil.emitLabel(ref().toStr(), h);
      LdbUtil.emitName ("0"          , h);
      LdbUtil.emitName (defLabel     , h);
      isExterned = true;
    }
}
```

3.3.3 PostScript null objects

Since `LdbNull` extends the `LdbImmutable` class, it inherits the costructors and observers of `LdbImmutable`. We don't explain the inherited methods.

- Abstraction : a PostScript null
- Constructors:
  - `LdbNull` : create a null object

115a ⟨*LdbNull.java* 115a⟩≡

```
/**
 * Null in ldb's view.
 */
public class LdbNull extends LdbImmutable {
    ⟨LdbNull.java: constructors 115b⟩
}
```

115b ⟨*LdbNull.java: constructors* 115b⟩≡ (115a)

```
/**
 * Class constructor.
 */
public LdbNull() { setDef("null"); }
```

3.3.4 PostScript integers

`LdbInt` extends the `LdbImmutable` class.

- Abstraction : a PostScript integer
- Constructors:
  - `LdbInt(int i)`: create a decimal integer
  - `LdbInt(int i, int j)`: create a binary, an octal, or a hexadecimal integer; `i` is the value and `j` is the base number

115c ⟨*LdbInt.java* 115c⟩≡

```
/**
 * Integer in ldb's view.
 */
public class LdbInt extends LdbImmutable {
    ⟨LdbInt.java: constructors 116a⟩
}
```

116a ⟨*LdbInt.java: constructors* 116a⟩≡ (115c)

```
/**
 * Class constructor for decimal integers.
 *
 * @param i this integer's value
 */
public LdbInt(int i) { setDef(new Integer(i).toString()); }

/**
 * Class constructor for binary, octal, and hexadecimal integers.
 *
 * @param i this integer's value
 * @param j this integer's radix
 */
public LdbInt(int i, int j) {
    if (j == 2)       setDef( "2#" + Integer.toBinaryString(i));
    else if (j == 8)  setDef( "8#" + Integer.toOctalString (i));
    else if (j == 16) setDef("16#" + Integer.toHexString   (i));
    else throw new Error("An unsigned integer in base " + j +
                         " is not supported");
}
```

3.3.5 PostScript strings

`LdbStr` extends the `LdbImmutable` class.

- Abstraction : a PostScript string
- Constructors:
  - `LdbStr`: create a PostScript string

116b ⟨*LdbStr.java* 116b⟩≡

```
/**
 * String in ldb's view.
 */
public class LdbStr extends LdbImmutable {
    ⟨LdbStr.java: constructors 116c⟩
}
```

116c ⟨*LdbStr.java: constructors* 116c⟩≡ (116b)

```
/**
 * Class constructor.
 *
 * @param s this string's content
 */
public LdbStr(String s) { setDef("(" + s + ")"); }
```

3.3.6 PostScript names

`LdbName` extends the `LdbImmutable` class.

- Abstraction : a PostScript name
- Constructors:
  - `LdbName`: create an PostScript name

117a ⟨*LdbName.java* 117a⟩≡

```
/**
 * Name in ldb's view.
 */
public class LdbName extends LdbImmutable {
    ⟨LdbName.java: constructors 117b⟩
}
```

117b ⟨*LdbName.java: constructors* 117b⟩≡ (117a)

```
/**
 * Class constructor.
 *
 * @param s this name's content
 */
public LdbName(String s) { setDef(s); }
```

3.3.7 PostScript procedures

`LdbPsProc` extends the `LdbImmutable` class.

- Abstraction : a PostScript procedure
- Constructors:
  - `LdbPsProc`: create a PostScript procedure

117c ⟨*LdbPsProc.java* 117c⟩≡

```
/**
 * PostScript procedure in ldb's view.
 */
public class LdbPsProc extends LdbImmutable {
    ⟨LdbPsProc.java: constructors 117d⟩
}
```

117d ⟨*LdbPsProc.java: constructors* 117d⟩≡ (117c)

```
/**
 * Class constructor.
 *
 * @param s this PostScript procedure's string representation
 */
public LdbPsProc(String s) { setDef("{" + s + "}"); }
```

3.3.8 PostScript codes

`LdbPsCode` extends the `LdbImmutable` class.

- Abstraction : a PostScript code
- Constructors:
  - `LdbPsCode`: create a PostScript code

118a ⟨*LdbPsCode.java* 118a⟩≡

```
/**
 * PostScript code in ldb's view.
 */
public class LdbPsCode extends LdbImmutable {
    ⟨LdbPsCode.java: constructors 118b⟩
}
```

118b ⟨*LdbPsCode.java: constructors* 118b⟩≡ (118a)

```
/**
 * Class constructor.
 *
 * @param s this code's string representation
 */
public LdbPsCode(String s) { setDef(s); }
```

3.3.9 Singleton immutable PostScript data

Because too many identical immutable objects can be created, the `LdbSingleton` class maintains a hash table to guarantee the singleton property for each immutable object.

119 ⟨*LdbSingleton.java* 119⟩≡

```
import java.util.HashMap;
import java.util.Iterator;

/**
 * Singleton immutable objects.
 */
public class LdbSingleton {
    private static HashMap<String, LdbInfo> map =
                new HashMap<String, LdbInfo>();
    public static final LdbNull newNull = new LdbNull();
    public static LdbInt newInt(int i) {
        String str = new Integer(i).toString();
        for (Iterator<String> ss = map.keySet().iterator(); ss.hasNext(); ) {
            String s = ss.next();
            if (s == str) return (LdbInt)map.get(s);
        }
        LdbInt ldbInt = new LdbInt(i);
        map.put(str, ldbInt);
        return ldbInt;
    }
    public static LdbInt newHex(int i) {
        String str = "16#" + Integer.toHexString(i);
        for (Iterator<String> ss = map.keySet().iterator(); ss.hasNext(); ) {
            String s = ss.next();
            if (s == str) return (LdbInt)map.get(s);
        }
        LdbInt ldbHex = new LdbInt(i, 16);
        map.put(str, ldbHex);
        return ldbHex;
    }
    public static LdbStr newStr(String str) {
        for (Iterator<String> ss = map.keySet().iterator(); ss.hasNext(); ) {
            String s = ss.next();
            if (s == str) return (LdbStr)map.get(s);
        }
        LdbStr ldbStr = new LdbStr(str);
        map.put(str, ldbStr);
        return ldbStr;
    }
    public static LdbName newName(String str) {
```

```
    for (Iterator<String> ss = map.keySet().iterator(); ss.hasNext(); ) {
        String s = ss.next();
        if (s == str) return (LdbName)map.get(s);
    }
    LdbName name = new LdbName(str);
    map.put(str, name);
    return name;
}
public static LdbPsProc newPsProc(String str) {
    for (Iterator<String> ss = map.keySet().iterator(); ss.hasNext(); ) {
        String s = ss.next();
        if (s == str) return (LdbPsProc)map.get(s);
    }
    LdbPsProc psProc = new LdbPsProc(str);
    map.put(str, psProc);
    return psProc;
}
public static LdbPsCode newPsCode(String str) {
    for (Iterator<String> ss = map.keySet().iterator(); ss.hasNext(); ) {
        String s = ss.next();
        if (s == str) return (LdbPsCode)map.get(s);
    }
    LdbPsCode psCode = new LdbPsCode(str);
    map.put(str, psCode);
    return psCode;
}
public static LdbValue newValue(int i) {
    String str = new Integer(i).toString();
    for (Iterator<String> ss = map.keySet().iterator(); ss.hasNext(); ) {
        String s = ss.next();
        if (s == str) return (LdbValue)map.get(s);
    }
    LdbValue value = new LdbValue(i);
    map.put(str, value);
    return value;
}
public static LdbValue newValue(float f) {
    String str = new Float(f).toString();
    for (Iterator<String> ss = map.keySet().iterator(); ss.hasNext(); ) {
        String s = ss.next();
        if (s == str) return (LdbValue)map.get(s);
    }
    LdbValue value = new LdbValue(f);
    map.put(str, value);
    return value;
}
```

```
public static LdbEnv emptyEnv() {
    String str = "EMPTY";
    for (Iterator<String> ss = map.keySet().iterator(); ss.hasNext(); ) {
        String s = ss.next();
        if (s == str) return (LdbEnv)map.get(s);
    }
    LdbEnv env = new LdbEnv();
    map.put(str, env);
    return env;
}
public static LdbEnv newEnv(LdbEnv e, LdbSymbol sym) {
    String str = e.getDef() + ":" + sym.name();
    for (Iterator<String> ss = map.keySet().iterator(); ss.hasNext(); ) {
        String s = ss.next();
        if (s == str) return (LdbEnv)map.get(s);
    }
    LdbEnv env = new LdbEnv(e, sym);
    map.put(str, env);
    return env;
}
public static LdbPlcmt newPlcmt(char space, int offset) {
    String str = offset + " '" + space + "' Absolute";
    for (Iterator<String> ss = map.keySet().iterator(); ss.hasNext(); ) {
        String s = ss.next();
        if (s == str) return (LdbPlcmt)map.get(s);
    }
    LdbPlcmt plcmt = new LdbPlcmt().absolute(space, offset);
    map.put(str, plcmt);
    return plcmt;
}
public static LdbPlcmt newPlcmt(LdbPlcmt p, int o) {
    String str = p.getDef() + " " + o + " Shifted";
    for (Iterator<String> ss = map.keySet().iterator(); ss.hasNext(); ) {
        String s = ss.next();
        if (s == str) return (LdbPlcmt)map.get(s);
    }
    LdbPlcmt plcmt = new LdbPlcmt().shifted(p, o);
    map.put(str, plcmt);
    return plcmt;
}
public static LdbPlcmt newPlcmt(char space, LdbPlcmt p, int offset) {
    String str = p.getDef() + " " + offset + " '" + space + "' Indirect";
    for (Iterator<String> ss = map.keySet().iterator(); ss.hasNext(); ) {
        String s = ss.next();
        if (s == str) return (LdbPlcmt)map.get(s);
    }
```

```
        LdbPlcmt plcmt = new LdbPlcmt().indirect(space, p, offset);
        map.put(str, plcmt);
        return plcmt;
    }
    public static LdbPlcmt newPlcmt(LdbLabel l) {
        String str = l.name;
        for (Iterator<String> ss = map.keySet().iterator(); ss.hasNext(); ) {
            String s = ss.next();
            if (s == str) return (LdbPlcmt)map.get(s);
        }
        LdbPlcmt plcmt = new LdbPlcmt().atLabel(l);
        map.put(str, plcmt);
        return plcmt;
    }
}
```

3.3.10 Mutable PostScript data

`LdbMutable` extends the `LdbInfo` class.

- Abstraction : the abstract base class for all mutable PostScript data
- Constructors:
  - `setExtBase` : set the base name for this mutable information's external reference
- Observers :
  - `getExtBase` : get the base name for this mutable information's external reference
  - `size` : get the size of this mutable information
  - `emitElements` : emit all the elements in this mutable information
  - `emitDefinition` : emit the definition of this mutable information
  - `mutate` : assert that this mutable information has not yet been externed

A mutable information satisfies the following invariance:

* `extern` is called in any states.

* `ref` is called only in the `EMITTED` state.

* `mutate` is called in the `NEW` state.

Each mutable information is externalized by `extern` as follows. If it's not yet externalized, its label is emitted followed by the number of its elements, its external reference, and every element in it.

For example, the following type dictionary

```
/T45 <<
        /decl    (type-decl)
        /printer {psproc}
        /symbol  S23
>> def
```

is externalized as follows:

```
.LT45:
.long 3
.long .LIm139   ;; T45
.long .LIm141   ;; decl
.long .LIm46    ;; (type-decl)
.long .LIm144   ;; printer
.long .LIm4     ;; {psproc}
.long .LIm146   ;; symbol
.long .LS23     ;; S23
```

Its label `.LT45` is emitted followed by the number of its elements `3`, its external reference `T45` denoted by `.LIm139`, and every key-value pair in the type dictionary.

124a ⟨*LdbMutable.java* 124a⟩≡

```
/**
 * The abstract base class for all mutable ldb information.
 */
public abstract class LdbMutable extends LdbInfo {
    ⟨LdbMutable.java: abstraction 124b⟩
    ⟨LdbMutable.java: constructors 124c⟩
    ⟨LdbMutable.java: observers 125⟩
}
```

124b ⟨*LdbMutable.java: abstraction* 124b⟩≡ (124a)

```
/**
 * State of this mutable information.
 */
public class State {}
public class NEW     extends State {}
public class EMITTED extends State {}
public State state = new NEW();

/**
 * Used to form an external name.
 * Every LdbSymbol's external name starts with "S",
 * every LdbType's   external name starts with "T", and
 * every LdbLoci's   external name starts with "L".
 */
private String extBase = "I";

/**
 * The external reference of this mutable information
 */
private LdbExtRef myExtRef = null;
```

124c ⟨*LdbMutable.java: constructors* 124c⟩≡ (124a)

```
/**
 * Sets the base name for this mutable information's external reference.
 *
 * @param  base base name for this mutable information's external reference
 */
public void setExtBase(String base) { extBase = base; }
```

125 ⟨*LdbMutable.java: observers* 125⟩≡ (124a)

```
/**
 * Gets the base name for this mutable information's external reference.
 *
 * @return base name for this mutable information's external reference
 */
public String getExtBase() { return extBase; }

/**
 * Gets the size of this mutable information.
 */
public abstract int size();

/**
 * Emits all the elements in this mutable information.
 */
public abstract void emitElements(LdbFileHandle h);

/**
 * Emits the definition of this mutable information.
 */
public abstract void emitDefinition(LdbFileHandle h) throws java.io.IOException;

/**
 * Externs this mutable information.
 *
 * @param h file handle where this mutable information to be externed
 */
public void extern(LdbFileHandle h) {
    if (state instanceof NEW) {
        super.extern(h);
        state = new EMITTED();
        // generates a unique name for this mutable information
        String name = new LdbExtRef().fromBase(getExtBase()).toStr();
        myExtRef = new LdbExtRef().fromStr(LdbUtil.makeLabelNameFrom(name));
        setExtRef(myExtRef);
        // emits used LdbInfo's in this mutable information
        emitElements(h);
        try {
            if (state instanceof EMITTED) {
                LdbInfo info = LdbSingleton.newName(LdbUtil.dropPrefixOfLabelName(myE
                info.extern(h);
                LdbUtil.emitLabel(myExtRef.toStr(),   h);
                LdbUtil.emitName (Integer.toString(size()), h);
                LdbUtil.emitName (info.ref().toStr(), h);
                h.push(); emitDefinition(h); h.pop();
```

```
            } else assert (false);
        } catch (java.io.IOException e) {
            System.out.println("java.io.IOException");
        }
    }
}

/**
 * Returns this mutable information's external reference.
 *
 * precondition : this mutable information has been externed
 * postcondition: the reference may be written anywhere in a file handle
 *
 * @return this informaion's external reference
 */
public LdbExtRef ref() {
    assert(state instanceof EMITTED);
    return super.ref();
}

/**
 * Asserts that this mutable information has not yet been externed
 *
 * precondition : this mutable information has not yet been externed
 * postcondition: this mutable information has not yet been externed
 *
 * @return this informaion's external reference
 */
public void mutate() {
    assert(state instanceof NEW);
}
```

3.3.11 PostScript arrays (lists)

`LdbArray` extends the `LdbMutable` class.

- Abstraction : a PostScript array whose elements are PostScript data
- Constructors:
  - `LdbArray` : create a PostScript array
  - `addHigh` : put a given value to the end of this array
- Observers :
  - `get` : get the value of the indexed component in this array
  - `size` : get the size of this array

127a ⟨*LdbArray.java* 127a⟩≡

```
import java.lang.reflect.Array;
import java.util.Iterator;

/**
 * Array in ldb's view.
 */
public class LdbArray extends LdbMutable {
    ⟨LdbArray.java: abstraction 127b⟩
    ⟨LdbArray.java: constructors 127c⟩
    ⟨LdbArray.java: observers 128⟩
}
```

127b ⟨*LdbArray.java: abstraction* 127b⟩≡ (127a)

```
private LdbInfo[] array = new LdbInfo[100];
public  int index = 0;
```

127c ⟨*LdbArray.java: constructors* 127c⟩≡ (127a)

```
/**
 * Puts the specified LdbInfo to the end of this array.
 *
 * precondition : addHigh [a,..,n] i
 * postcondition: [a,..,n,i]
 *
 * @param i the new LdbInfo to be put to this array
 */
public void addHigh(LdbInfo i) {
    super.mutate();
    Array.set(array, index++, i);
}
```

128 ⟨*LdbArray.java: observers* 128⟩≡ (127a)

```
/**
 * Emits all the elements in this array.
 *
 * @param h file handle where this array to be externed
 */
public void emitElements(LdbFileHandle h) {
    for (int i = 0; i < index; i++) {
        LdbInfo info = (LdbInfo)Array.get(array, i);
        if (info != null) info.extern(h);
    }
}

/**
 * Emits the definition of this array.
 *
 * @param h file handle where this array to be externed
 */
public void emitDefinition(LdbFileHandle h) throws java.io.IOException {
    for (int i = 0; i < index; i++) {
        LdbInfo info = (LdbInfo)Array.get(array, i);
        if (info != null) {
            LdbUtil.emitName(info.ref().toStr(), h);
            h.write(" "); info.ref().write(h);
        }
    }
}

/**
 * Returns the LdbInfo of the indexed component in the array.
 *
 * @param i array index
 * @return  the LdbInfo associated with the specified key in the array
 */
public LdbInfo get(int i) { return (LdbInfo)Array.get(array, i); }

/**
 * Returns the size of this array.
 *
 * @return  the size of this array
 */
public int size() { return index; }
```

3.3.12 PostScript dictionaries (tables)

`LdbTable` extends the `LdbMutable` class.

- Abstraction : a PostScript dictionary with string keys and PostScript values
- Constructors:
  - `LdbTable` : create a PostScript table
  - `put` : add a given key/value pair to this table; overwrite any existing entry with the same key
- Observers :
  - `get` : find the value bound to a given key in this table

129a ⟨*LdbTable.java* 129a⟩≡

```
import java.util.List;
import java.util.LinkedList;
import java.util.Iterator;

/**
 * Table or dictionary in ldb's view.
 * LdbSymbol and LdbType are subclasses of thid class.
 */
public class LdbTable extends LdbMutable {
    ⟨LdbTable.java: abstraction 129b⟩
    ⟨LdbTable.java: constructors 130⟩
    ⟨LdbTable.java: observers 131⟩
}
```

129b ⟨*LdbTable.java: abstraction* 129b⟩≡ (129a)

```
/**
 * LdbTable entry.
 */
private class Entry {
    public String key;
    public LdbInfo value;
    public Entry(String k, LdbInfo v) { key = k; value = v; }
}
private LinkedList<Entry> map = new LinkedList<Entry>();
public  int index = 0;
```

130 ⟨*LdbTable.java: constructors* 130⟩≡ (129a)

```
/**
 * Puts the specified value into the LdbTable, bound to the specified Key.
 *
 * precondition : table is not externed yet
 * postcondition: table is still not externed;
 *                a given key/value pair overwrites any
 *                existing entry with the same key
 *
 * @param k the specified key
 * @param v the new LdbInfo associated with the specified key
 */
public void put(String k, LdbInfo v) {
    Entry e = null;
    super.mutate();
    for (Iterator<Entry> entries = map.iterator(); entries.hasNext(); ) {
        e = entries.next();
        if (e.key == k) break;
    }
    if (e != null && e.key == k) { map.remove(e); index--; }
    map.add(new Entry(k, v));
    index++;
}
```

131 ⟨*LdbTable.java: observers* 131⟩≡ (129a)

```
/**
 * Emits all the elements in this LdbTable.
 *
 * @param h file handle where this LdbTable to be externed
 */
public void emitElements(LdbFileHandle h) {
    for (Iterator<Entry> entries = map.iterator();
         entries.hasNext(); ) {
        Entry   e = entries.next();
        String  k = e.key;
        LdbInfo i = e.value;
        if (i != null) { i.extern(h); LdbSingleton.newName(k).extern(h); }
    }
}

/**
 * Emits the definition of this LdbTable.
 *
 * @param h file handle where this LdbTable to be externed
 */
public void emitDefinition(LdbFileHandle h) throws java.io.IOException {
    for (Iterator<Entry> entries = map.iterator();
         entries.hasNext(); ) {
        Entry   e = entries.next();
        String  k = e.key;
        LdbInfo i = e.value;
        if (i != null) {
            LdbUtil.emitName(LdbSingleton.newName(k).ref().toStr(), h);
            LdbUtil.emitName(i.ref().toStr(), h);
        }
    }
}

/**
 * Returns the LdbInfo associated with the specified key in the LdbTable.
 *
 * precondition : table might be empty
 * postcondition: return an \ps~data associated with a given key, if any
 *
 * @param k a key in the LdbTable
 * @return  the LdbInfo associated with the specified key in the LdbTable
 */
public LdbInfo get(String k) {
    LdbInfo result = null;
    for (Iterator<Entry> entries = map.iterator(); entries.hasNext(); ) {
```

```
            Entry e = entries.next();
            if (e.key == k) result = e.value;
        }
        return result;
    }

    /**
     * Returns the size of this table.
     *
     * @return  the size of this table
     */
    public int size() { return index; }
```

3.4 `ldb` data

This section includes the `ldb` interface for `ldb` data which might be the source-language dependent but the compiler independent. All the compiler-dependent procedures used in this section are defined in Section 3.2.3.

3.4.1 Source-code locations

- Abstraction : a source-code location
- Constructors:
  - `LdbSrcLoc` : create a source location
- Observers :
  - `file` : return this source-code location's file name
  - `line` : return this source-code location's line number
  - `column` : return this source-code location's column number

132a ⟨*LdbSrcLoc.java* 132a⟩≡

```
/**
 * The source-code location.
 */
public class LdbSrcLoc {
    ⟨LdbSrcLoc.java: abstraction 132b⟩
    ⟨LdbSrcLoc.java: constructors 133a⟩
    ⟨LdbSrcLoc.java: observers 133b⟩
}
```

132b ⟨*LdbSrcLoc.java: abstraction* 132b⟩≡ (132a)

```
private String file;
private int line, column;
```

133a ⟨*LdbSrcLoc.java: constructors* 133a⟩≡ (132a)

```
/**
 * Class constructor specifying file name, line number and column number.
 *
 * @param s file   name   of this source-code location
 * @param l line   number of this source-code location
 * @param c column number of this source-code location
 */
public LdbSrcLoc(String f, int l, int c) {
    file   = f;
    line   = l;
    column = c;
}
```

133b ⟨*LdbSrcLoc.java: observers* 133b⟩≡ (132a)

```
/**
 * returns the file name of this source-code location
 */
public String file()   { return file; }

/**
 * returns the line number of this source-code location
 */
public int    line()   { return line; }

/**
 * returns the column number of this source-code location
 */
public int    column() { return column; }
```

3.4.2 Values of machine-level constants

`LdbValue` extends the `LdbImmutable` class.

- Abstraction : a machine-level value which is either an integer or a floating-point value
- Constructors:
  - `LdbValue(int i)` : create a PostScript code whose evaluation results in an integer value
  - `LdbValue(float f)` : create a PostScript code whose evaluation results in a floating-point value

134a ⟨*LdbValue.java* 134a⟩≡

```
/**
 * LdbValue in ldb's view.
 */
public class LdbValue extends LdbImmutable {
    ⟨LdbValue.java: constructors 134b⟩
}
```

134b ⟨*LdbValue.java: constructors* 134b⟩≡ (134a)

```
/**
 * Class constructor for integer values.
 *
 * @param i this integer's value
 */
public LdbValue(int   i) { setDef(i + " Immediate"); }

/**
 * Class constructor for floating-point values.
 *
 * @param f this floating-point's value
 */
public LdbValue(float f) { setDef(f + " Immediate"); }
```

3.4.3 Symbols

- Abstraction : a symbol; every symbol is one of a variable, a procedure, a constant, and a type
- Constructors:
  - `LdbSymbol` : create a symbol and initialize its name and source-code location
- Observers :
  - `name` : return this symbol's name

135a ⟨*LdbSymbol.java* 135a⟩≡

```
/**
 * The base class for all symbols in ldb's view.
 */
public class LdbSymbol extends LdbTable {
    ⟨LdbSymbol.java: abstraction 135b⟩
    ⟨LdbSymbol.java: constructors 135c⟩
    ⟨LdbSymbol.java: observers 136a⟩
}
```

135b ⟨*LdbSymbol.java: abstraction* 135b⟩≡ (135a)

```
private String name;
```

135c ⟨*LdbSymbol.java: constructors* 135c⟩≡ (135a)

```
/**
 * Class constructor specifying name and source location.
 * Every LdbSymbol's external name starts with "S".
 *
 * @param n name of this symbol
 * @param s source-code location of this symbol
 */
public LdbSymbol(String n, LdbSrcLoc s) {
    name = n;
    setExtBase("S");
    put("name",       LdbSingleton.newStr(name));
    put("sourcefile", LdbSingleton.newStr(s.file()));
    put("sourcey",    LdbSingleton.newInt(s.line()));
    put("sourcex",    LdbSingleton.newInt(s.column()));
}
```

136a ⟨*LdbSymbol.java: observers* 136a⟩≡ (135a)

```
/**
 * Returns this LdbSymbol's name
 *
 * @return this LdbSymbol's name
 */
public String name() { return name; }
```

Constants

- Abstraction : a constant symbol
- Constructors:
  - `LdbConstant` : create a constant symbol
  - `putType` : initialize this constant's type
  - `putValue` : initialize this constant's value

When a constant is created, its name and source location are given but its type is usually not yet provided. Thus, a constant's type is initialized when the type is avaiable by `putType`. A constant's value is either an integer or a floating-point value and this information is initialized by `putValue`.

136b ⟨*LdbConstant.java* 136b⟩≡

```
/**
 * The constant symbols in ldb's view.
 */
public class LdbConstant extends LdbSymbol {
    ⟨LdbConstant.java: constructors 137⟩
}
```

137 ⟨*LdbConstant.java: constructors* 137⟩≡ (136b)

```
/**
 * Class constructor specifying name, source location, and value.
 *
 * @param n name of this constant symbol
 * @param s source-code location of this constant symbol
 * @param t value of this constant symbol
 */
public LdbConstant(String n, LdbSrcLoc s) {
    super(n, s);
    put("kind", LdbSingleton.newStr("constant"));
}

/**
 * Initializes this constant's type.
 *
 * @param t type of this constant
 */
public void putType(LdbType t) { put("type", t); }

/**
 * Initializes this constant's value.
 *
 * @param v value of this constant
 */
public void putValue(LdbValue v) { put("value", v); }
```

Variables

- Abstraction : a variable symbol
- Constructors:
  - `LdbVar` : create a variable symbol
  - `putType` : initialize this variable's type
  - `putWhere` : initialize this variable's placement
  - `putLabel` : set this variable's assembly-language label
- Observers :
  - `label` : return this variable's assembly-language label

When a variable is created, its name and source location are given but its type is usually not yet provided. Thus, a variable's type is initialized when the type is avaiable by `putType`. A variable's placement is either in a stack frame or in a register and this information is initialized by `putWhere` after the register allocation phase. A variable's assembly-language label is used to refer the variable in the executable program. `putLabel` records the variable's label when it's available and `label` returns the label. The naming convention for labels is compiler dependent and labels are described in section 3.4.7.

138a ⟨*LdbVar.java* 138a⟩≡

```
/**
 * The variable symbols in ldb's view.
 */
public class LdbVar extends LdbSymbol {
    ⟨LdbVar.java: abstraction 138b⟩
    ⟨LdbVar.java: constructors 139⟩
    ⟨LdbVar.java: observers 140⟩
}
```

138b ⟨*LdbVar.java: abstraction* 138b⟩≡ (138a)

```
/**
 * This variable's assembly-language label.
 */
private LdbLabel label = null;
/**
 * Returns whether this variable has its location.
 */
private boolean hasWhere = false;
```

139 ⟨*LdbVar.java: constructors* 139⟩≡ (138a)

```
/**
 * Class constructor specifying name and source location.
 *
 * @param n name of this variable symbol
 * @param s source-code location of this variable symbol
 */
public LdbVar(String n, LdbSrcLoc s) {
    super(n, s);
    put("kind", LdbSingleton.newStr("variable"));
}

/**
 * Initializes this variable's type.
 *
 * @param t type of this variable
 */
public void putType(LdbType t) { put("type", t); }

/**
 * Initializes this variable's placement.
 * Since a variable appears many places,
 * the compiler might put its location several times
 * and raise java.util.ConcurrentModificationException.
 * Thus, we guarantee non-concurrent modification by hasWhere flag.
nn *
 * @param a this variable's location in a frame or in a register
 */
public void putWhere(LdbPlcmt p) {
    if (!hasWhere) {
        hasWhere = true;
        put("where", p);
    }
}

/**
 * Sets this variable's assembly-language label.
 *
 * @param l this variable's assembly-language label
 */
public void putLabel(LdbLabel l) { label = l; }
```

140 ⟨*LdbVar.java: observers* 140⟩≡ (138a)

```
/**
 * Returns this variable's assembly-language label.
 *
 * @return this variable's assembly-language label
 */
public LdbLabel label() { return label; }
```

Procedures

- Abstraction : a procedure symbol
- Constructors:
  - `LdbProc` : create a procedure symbol
  - `putType` : initialize this procedure's type
  - `putFormal` : set this procedure's last formal parameter, if any
  - `putArgEnv` : set this procedure's arguments environment
  - `putReturnType` : set this procedure's return type
  - `putFrameSize` : set this procedure's frame size
  - `putRegInfo` : set this procedure's callee-saved registers information
  - `putLabel` : set this procedure's assembly-language label
  - `addLocus` : add a stopping point in this procedure
  - `addLocal` : add a local symbol in this procedure
- Observers :
  - `file` : return this procedure's file name
  - `plcmt` : return this procedure's placement
  - `locals` : return this procedure's local symbols

When a procedure is created, its name and source location are given but its type is usually not yet provided. Thus, a procedure's type is initialized when the type is avaiable by `putType`. `putFormal` sets a procedure's the last formal parameter, if any and `putArgEnv` sets the procedure's arguments environment during the semantic analysis phase and `putReturnType` sets the procedure's return type after the type-checking phase. `putFrameSize` sets a procedure's stack frame size and `putRegInfo` sets a procedure's callee-saved registers information. Callee-saved registers information is given as a list of tuples `(mask, base)` where `mask` is the register-save mask in a decreasing order and `base` is the distance of least significant register from the heavy pointer.

`ldb` assumes that each stack frame can be identified by a single pointer, called the *heavy pointer*, the value of which remains unchanged for the lifetime of the frame. Since the frame and stack pointers are used in different ways on different architectures, `ldb` uses the term "heavy" pointer, not frame or stack pointer. The heavy pointer is realized as either the frame pointer or the stack pointer.

A procedure's assembly-language label is used to refer the procedure in the executable program. `putLabel` records the procedure's label when it's available. The naming convention for labels is compiler dependent and labels are described in section 3.4.7. A procedure records all the stopping points in it and each stopping point is added to the procedure by `addLocus` when the stopping

point is created. Compilers usually have several intermediate representations for a program and in which representation to put stopping points are compiler dependent. Stopping points are described in section 3.4.8. All the local symbols in a procedure is recorded by `addLocal`. `file` returns the procedure's file name to make a source map for the current compilation unit and `plcmt` and `locals` return the procedure's placement and local symbols, respectively.

142 ⟨*LdbProc.java* 142⟩≡

```
import java.lang.reflect.Array;
import java.util.List;
import java.util.LinkedList;

/**
 * The procedure symbols in ldb's view.
 */
public class LdbProc extends LdbSymbol {
    ⟨LdbProc.java: abstraction 143⟩
    ⟨LdbProc.java: constructors 144⟩
    ⟨LdbProc.java: observers 147a⟩
}
```

143 ⟨*LdbProc.java: abstraction* 143⟩≡ (142)

```
/**
 * This procedure's file name.
 */
private String file = "";
/**
 * Stopping points in this procedure.
 */
private LdbArray myloci = new LdbArray();
private LdbArray envs   = new LdbArray();
private LdbArray loci   = new LdbArray();
/**
 * This procedure's placement.
 */
private LdbPlcmt plcmt = null;
/**
 * This procedure's local symbols.
 */
private LinkedList<LdbSymbol> locals = new LinkedList<LdbSymbol>();

/**
 * Cached reginfo.
 */
private class RegI {
    public int[] regs;
    public LdbArray regInfo;
}
private static RegI cachedRegInfo;
```

144 ⟨*LdbProc.java: constructors* 144⟩≡ (142)

```
/**
 * Class constructor specifying name and source location.
 *
 * @param n name of this procedure symbol
 * @param s source-code location of this procedure symbol
 */
public LdbProc(String n, LdbSrcLoc s) {
    super(n, s);
    file = s.file();
    put("kind", LdbSingleton.newStr("procedure"));
    put("loci", myloci);
    myloci.setExtBase("L");
    myloci.addHigh(envs);
    myloci.addHigh(loci);
}

/**
 * Initializes this procedure's type.
 *
 * @param t type of this procedure
 */
public void putType(LdbType t) { put("type", t); }

/**
 * Initializes this procedure's the last formal parameter.
 *
 * @param s LdbSymbol of the last formal parameter of this procedure
 */
public void putFormal(LdbSymbol s) {
    if (s == null) put("formals", LdbSingleton.newNull);
    else put("formals", s);
}

/**
 * Initializes this procedure's arguments environment
 *
 * @param s environment including the arguments of this procedure
 */
public void putArgEnv(LdbEnv e) { put("argenv", e); }

/**
 * Initializes this procedure's return type.
 *
 * @param s string representation of the return type of this procedure
 */
```

```
public void putReturnType(String s) {
    put("returntype", LdbSingleton.newStr(s));
}

/**
 * Sets this procedure's frame size.
 *
 * @param i this procedure's frame size
 */
public void putFrameSize(int i) { put("framesize", LdbSingleton.newInt(i)); }

/**
 * Creates LdbArray from a given integer array.
 *
 * @param r integer array whose elements are the register usage masks and
 *          the distances of least significant register from hp in turn
 */
private LdbArray mkRegInfo(int[] r) {
    LdbArray regInfo = new LdbArray();
    for (int i = 0; i < Array.getLength(r); i++) {
        int mask = Array.getInt(r, i);
        int base = Array.getInt(r, i+1);
        regInfo.addHigh(LdbSingleton.newHex(mask));
        regInfo.addHigh(LdbSingleton.newInt(base));
        i++;
    }
    return regInfo;
}
/**
 * Sets this procedure's callee-saved registers information.
 *
 * @param r integer array whose elements are the register usage masks and
 *          the distances of least significant register from hp in turn
 */
public void putRegInfo(int[] r) {
    if (cachedRegInfo == null) {
        cachedRegInfo         = new RegI();
        cachedRegInfo.regs    = r;
        cachedRegInfo.regInfo = mkRegInfo(r);
    } else {
        int i;
        for (i = 0; i < Array.getLength(r); i++) {
            if (Array.getInt(r, i) != Array.getInt(cachedRegInfo.regs, i))
                break;
        }
        if (i != Array.getLength(r) ||
```

```
            Array.getLength(r) != cachedRegInfo.regInfo.size()) {
            cachedRegInfo         = new RegI();
            cachedRegInfo.regs    = r;
            cachedRegInfo.regInfo = mkRegInfo(r);
        }
    }
    put("reginfo", cachedRegInfo.regInfo);
}

/**
 * Sets this procedure's assembly-language label.
 *
 * @param l this procedure's assembly-language label
 */
public void putLabel(LdbLabel l) {
    plcmt = LdbSingleton.newPlcmt(l);
    put("where", plcmt);
}

/**
 * Adds a specified stopping point to this procedure's stopping points.
 *
 * @param locus a stopping point in this procedure
 */
public void addLocus(LdbLocus locus) {
    envs.addHigh(locus.env());
    LdbArray mylocus = new LdbArray();
    LdbPlcmt myplcmt = LdbSingleton.newPlcmt(locus.label());
    mylocus.addHigh(LdbSingleton.newStr(locus.file()));
    mylocus.addHigh(LdbSingleton.newInt(locus.line()));
    mylocus.addHigh(LdbSingleton.newInt(locus.column()));
    mylocus.addHigh(myplcmt);
    loci.addHigh(mylocus);
}

/**
 * Adds a specified local symbol to this procedure
 *
 * @param local a local symbol in this procedure
 */
public void addLocal(LdbSymbol local) { locals.add(local); }
```

147a ⟨*LdbProc.java: observers* 147a⟩≡ (142)

```
/**
 * Returns this procedure's file name.
 *
 * @return this procedure's file name
 */
public String file() { return file; }

/**
 * Returns this procedure's placement.
 *
 * @return this procedure's placement.
 */
public LdbPlcmt plcmt() { return plcmt; }

/**
 * Returns this procedure's local symbols.
 *
 * @return this procedure's local symbols.
 */
public LinkedList<LdbSymbol> locals() { return locals; }
```

Type symbols

- Abstraction : a type symbol
- Constructors:
  - `LdbTypeSymbol` : create a type symbol
  - `putType` : initialize this type symbol's type
  - `addField` : record every field in this type symbol
  - `addMethod` : record every method in this type symbol

When a type symbol is created, its name and source location are given but its type is usually not yet provided. Thus, a type symbol's type is initialized when the type is avaiable by `putType`.

147b ⟨*LdbTypeSymbol.java* 147b⟩≡

```
/**
 * The class symbols in ldb's view.
 */
public class LdbTypeSymbol extends LdbSymbol {
    ⟨LdbTypeSymbol.java: abstraction 148a⟩
    ⟨LdbTypeSymbol.java: constructors 148b⟩
}
```

148a ⟨*LdbTypeSymbol.java: abstraction* 148a⟩≡ (147b)

```
private LdbArray fields  = new LdbArray();
private LdbArray methods = new LdbArray();
```

148b ⟨*LdbTypeSymbol.java: constructors* 148b⟩≡ (147b)

```
/**
 * Class constructor specifying name and source location.
 *
 * @param n name of this class symbol
 * @param s source-code location of this class symbol
 */
public LdbTypeSymbol(String n, LdbSrcLoc s) {
    super(n, s);
    put("kind",    LdbSingleton.newStr("type"));
    put("fields",  fields);
    put("methods", methods);
}

/**
 * Initializes this type symbol's type.
 *
 * @param t type of this type symbol
 */
public void putType(LdbType t) { put("type", t); }

/**
 * Puts the LdbSymbol's of this symbol's fields one at a time.
 *
 * @param f the LdbSymbol of one of this symbol's fields
 */
public void addField (LdbSymbol f) { fields.addHigh(f); }

/**
 * Puts the LdbSymbol's of this symbol's methods one at a time.
 *
 * @param m the LdbSymbol of one of this symbol's methods
 */
public void addMethod(LdbSymbol m) { methods.addHigh(m); }
```

3.4.4 Environments

`LdbEnv` extends the `LdbInfo` class.

- Abstraction : an environment
- Constructors:
  - `LdbEnv`: create an environment

149a ⟨*LdbEnv.java* 149a⟩≡

```
/**
 * ldb's environment
 */
public class LdbEnv extends LdbInfo {
    ⟨LdbEnv.java: abstraction 149b⟩
    ⟨LdbEnv.java: constructors 150⟩
    ⟨LdbEnv.java: observers 151⟩
}
```

149b ⟨*LdbEnv.java: abstraction* 149b⟩≡ (149a)

```
/**
 * The external reference of this environment
 */
private LdbExtRef myExtRef = null;

/**
 * The definition of this immutable information
 */
private String myDef = null;

/**
 * Whether this environment is externed
 */
private boolean isExterned = false;

/**
 * The previous environment
 */
private LdbEnv env = null;

/**
 * the bottom-most symbol of this environment
 */
private LdbSymbol symbol = null;
```

150 ⟨*LdbEnv.java: constructors* 150⟩≡ (149a)

```
/**
 * Class constructor for an empty environment.
 */
public LdbEnv() {
    myDef    = "EMPTY";
    myExtRef = new LdbExtRef().fromBase("E");
    setExtRef(myExtRef);
}

/**
 * Class constructor for binding a symbol to an environment.
 *
 * @param genv the global environment
 * @param sym  newly bound symbol
 */
public LdbEnv(LdbEnv e, LdbSymbol sym) {
    env      = e;
    symbol   = sym;
    myDef    = e.getDef() + ":" + sym.name();
    myExtRef = new LdbExtRef().fromBase("E");
    setExtRef(myExtRef);
}
```

151 ⟨*LdbEnv.java: observers* 151⟩≡ (149a)

```
/**
 * Externs this environment
 *
 * @param h file handle where this environment to be externed
 */
public void extern(LdbFileHandle h) {
    if (!isExterned) {
        LdbName info = LdbSingleton.newName(ref().toStr());
        super.extern(h);
        if (symbol == null) {
            info.extern(h); LdbUtil.emitLabel(myExtRef.toStr(), h);
            LdbUtil.emitName("0"               , h);
            LdbUtil.emitName(info.ref().toStr(), h);
        } else {
            env.extern(h); symbol.extern(h); info.extern(h);
            LdbUtil.emitLabel(myExtRef.toStr()    , h);
            LdbUtil.emitName ("2"                 , h);
            LdbUtil.emitName (info.ref().toStr()  , h);
            h.push();
            LdbUtil.emitName (env.ref().toStr()   , h);
            LdbUtil.emitName (symbol.ref().toStr(), h);
            h.pop();
        }
        isExterned = true;
    }
}

/**
 * Gets this environment's definition
 *
 * @return this environment's definition
 */
public String getDef() { return myDef; }
```

3.4.5 Types

- Abstraction : a type
- Constructors:
  - `LdbType` : create a type

When a type is created, how to print a declaration of the type `decl` and how to print a value of the type `printer` are provided. This information is compiler dependent. For example, let's consider the following structure `Fib` and its type `FIB`:

```
/S4 <<>> def
/T5 << /decl       (Fib %s)
       /printer    {CLASS}
       /symbol     S4          >> def
%% Backpatches
S4 /name       (Fib)     put
S4 /sourcefile (Fib.java)  put
S4 /sourcey    1         put
S4 /sourcex    6         put
S4 /kind       (type)    put
S4 /type       T5        put
S4 /vars       I6        put
S4 /procs      I7        put
```

In order to print the class `Fib` denoted by `S4` here, the printing method `CLASS` in `FIB` which is `T5` needs to know which variables and functions are in `S4`. Thus, the `symbol` key in `T5` is bound to `S4` and now `CLASS` can access the variables and functions in `S4` through the `symbol` key.

152 ⟨*LdbType.java* 152⟩≡

```
/**
 * The base class for all types in ldb's view.
 */
public class LdbType extends LdbTable {
    ⟨LdbType.java: constructors 153a⟩
}
```

153a ⟨*LdbType.java: constructors* 153a⟩≡ (152)

```
/**
 * Class constructor.
 * Every LdbType's external name starts with "T".
 *
 * @param decl    how to print a declartion of this type
 * @param printer how to print a value      of this type
 */
public LdbType(String decl, LdbPsProc printer) {
    setExtBase("T");
    put("decl", LdbSingleton.newStr(decl));
    put("printer", printer);
}
```

3.4.6 Placements

- Abstraction : an object location
- Constructors:
  - `absolute` : create a fundamental kind of location
    `absolute(s, o) = s[o]`
  - `shifted` : create a location relative to some other location
    `shifted(s[o'], o) = s[o'+o]`
  - `indirect` : create one level of indirection, with displacement
    `indirect(s, s'[o'], o) = s[s'[o']+o]`
  - `atLabel` : create a placement of a given label

153b ⟨*LdbPlcmt.java* 153b⟩≡

```
import java.lang.reflect.Array;
import java.util.HashMap;
import java.util.Iterator;

/**
 * Object location.
 */
public class LdbPlcmt extends LdbImmutable {
    ⟨LdbPlcmt.java: abstraction 154⟩
    ⟨LdbPlcmt.java: constructors 155⟩
    ⟨LdbPlcmt.java: observers 156a⟩
}
```

154 ⟨*LdbPlcmt.java: abstraction* 154⟩≡ (153b) 156b ▷

```
/**
 * a placement embedded in this placement
 */
private LdbPlcmt subPlcmt = null;
```

155 ⟨*LdbPlcmt.java: constructors* 155⟩≡ (153b) 157a▷

```
/**
 * Creates a fundamental kind of location.
 *
 * postcondition: absolute(s, o) = s[o]
 *
 * @param s space
 * @param o offset
 */
public LdbPlcmt absolute(char s, int o) {
    setDef(o + " '" + s + "' Absolute");
    return this;
}

/**
 * Creates a location relative to some other location.
 *
 * postcondition: shifted(s[o'], o) = s[o'+o]
 *
 * @param a address
 * @param o offset
 */
public LdbPlcmt shifted(LdbPlcmt a, int o) {
    subPlcmt = a;
    setDef(a.getDef() + " " + o + " Shifted");
    return this;
}

/**
 * Creates one level of indirection, with displacement.
 *
 * postcondition: indirect(s, s'[o'], o) = s[s'[o'] + o]
 *
 * @param s space
 * @param a base address
 * @param o offset
 */
public LdbPlcmt indirect(char s, LdbPlcmt a, int o) {
    subPlcmt = a;
    setDef(a.getDef() + " " + o + " '" + s + "' Indirect");
    return this;
}

⟨atLabel 158a⟩
```

156a ⟨*LdbPlcmt.java: observers* 156a⟩≡ (153b) 157b ▷

```
/**
 * Externs this placement
 *
 * @param h file handle where this placement to be externed
 */
public void extern(LdbFileHandle h) {
    if (subPlcmt != null) subPlcmt.extern(h);
    super.extern(h);
}
```

Lazy address management

- Abstraction : lazy address management using the anchor symbol
- Constructors:
  - `initialize` : create an anchor symbol
- Observers :
  - `finalize` : finalize emitting references to locations in assembly-language output
  - `anchor` : return the name of the anchor symbol

Some addresses are bound by the linker and therefore are not available at compile time. We pass those addresses to the debugger by planting references in the assembly language output. The references are in known locations relative to the anchor symbol, and we refer to them by using the anchor symbol name and the offset of the reference. While we do not need to use this lazy address management any more because we eliminated the use of `.o.st` and `.lt` files, we still use the anchor symbol so that we can use the `ldb` debugger as it is.

156b ⟨*LdbPlcmt.java: abstraction* 154⟩+≡ (153b) ◁154

```
/**
 * All the lazy placements.
 */
private static HashMap<LdbLabel, Integer> map =
    new HashMap<LdbLabel, Integer>();
/**
 * The offset from the anchor symbol of the next reference to be planted.
 */
private static int offset = 0;
```

157a ⟨*LdbPlcmt.java: constructors* 155⟩+≡ (153b) ◁155

```
/**
 * Initializes lazy address management; create anchor symbol
 *
 * @return the anchor symbol
 */
public static void initialize() {
    // time/version stamp for this compilation
    anchor = "_stanchor__V" + java.lang.System.currentTimeMillis();
}
```

157b ⟨*LdbPlcmt.java: observers* 156a⟩+≡ (153b) ◁156a

```
/**
 * Finalize emitting references to locations in asssembly-language output
 * Plants a reference to each of the labels relative to the anchor symbol
 *
 * precondition : compiler finished emitting assembly-language output
 * postcondition: all the bound labels are emitted
 */
public static void finalize(LdbFileHandle h) throws java.io.IOException {
    // exports the anchor symbol
    LdbUtil.exportName(anchor, h);
    LdbLabel[] array = new LdbLabel[offset];
    for (Iterator<LdbLabel> ls = map.keySet().iterator(); ls.hasNext(); ) {
        LdbLabel l = ls.next();
        Array.set(array, map.get(l).intValue(), l);
    }
    for (int i = 0; i < offset; i++) {
        ((LdbLabel)Array.get(array, i)).bindHere(h);
    }
    LdbUtil.emitSectionClose(h);
}

/**
 * Name of the anchor symbol.
 */
public static String anchor;
```

158a ⟨atLabel 158a⟩≡ (155)

```
/**
 * Creates a placement of a given label.
 *
 * @param  label label
 * @return a placement of a given label
 */
public LdbPlcmt atLabel(LdbLabel label) {
    setDef("{(" + anchor + ") " + offset + " Lazy" + label.space + "}");
    map.put(label, new Integer(offset++));
    return this;
}
```

3.4.7 Labels

- Abstraction : a reference to a location in an assembly output (`.s`)
- Constructors:
  - `inCodeSpace` : return a fresh label with a given name in a code space
  - `inDataSpace` : return a fresh label with a given name in a data space
- Observers :
  - `name` : return this label's name
  - `space` : return a space where this label is bound to
  - `bindHere` : bind this label to a location in an assembly output

158b ⟨*LdbLabel.java* 158b⟩≡

```
/**
 * A reference to a location in an assembly output (.s).
 */
public class LdbLabel {
    ⟨LdbLabel.java: constructors 159⟩
    ⟨LdbLabel.java: observers 160⟩
}
```

159 ⟨*LdbLabel.java: constructors* 159⟩≡ (158b)

```
public LdbLabel () {}

/**
 * Label with a given name in a code space.
 *
 * precondition : label does not exist
 * postcondition: label is unbound
 *
 * @param n label name
 */
public LdbLabel inCodeSpace(String n) {
    name  = n;
    space = "Code";
    return this;
}

/**
 * Label with a given name in a data space.
 *
 * precondition : label does not exist
 * postcondition: label is unbound
 *
 * @param n label name
 */
public LdbLabel inDataSpace(String n) {
    name  = n;
    space = "Data";
    return this;
}
```

160 ⟨*LdbLabel.java: observers* 160⟩≡ (158b)

```
/**
 * This label's name
 */
public String name  = "";

/**
 * The space where this label is bound to
 */
public String space = "";

/**
 * Bind this label to a location in .s
 *
 * precondition : label is unbound
 * postcondition: label is bound and emitted to .s
 *
 * @param h assembly-language output
 */
public void bindHere(LdbFileHandle h) throws java.io.IOException {
    LdbUtil.emitName(name, h);
}
```

3.4.8 Stopping points

- Abstraction : a stopping point
- Constructors:
  - `LdbLocus` : create a stopping point
- Observers :
  - `file` : return the file name of this stopping point
  - `line` : return the line number of this stopping point
  - `column` : return the column number of this stopping point
  - `label` : return the label of this stopping point
  - `env` : return the environment at this stopping point

Compilers usually have a set of stopping points where an execution of a given program might stop. For example, MiniJava places a stopping point before each expression, including nested expressions, as well as before each statement and at the end of each block.

161a ⟨*LdbLocus.java* 161a⟩≡

```
/**
 * Stopping point.
 */
public class LdbLocus {
    ⟨LdbLocus.java: abstraction 161b⟩
    ⟨LdbLocus.java: constructors 162⟩
    ⟨LdbLocus.java: observers 163⟩
}
```

161b ⟨*LdbLocus.java: abstraction* 161b⟩≡ (161a)

```
private LdbSrcLoc src;
private LdbLabel  label;
private LdbEnv    env;
```

162 ⟨*LdbLocus.java: constructors* 162⟩≡ (161a)

```
/**
 * Class constructor
 * Sets the name of this stopping point with a new unique name.
 * This name has to be usable both as PostScript names and
 * as assembly-language labels.
 *
 * @param s the source location of this stopping point
 * @param e the environment at this stopping point
 */
public LdbLocus(LdbSrcLoc s, LdbLabel l, LdbEnv e) {
    src   = s;
    label = l;
    env   = e;
}
```

163 ⟨*LdbLocus.java: observers* 163⟩≡ (161a)

```
/**
 * Returns the file name of this stopping point.
 *
 * @return the file name of this stopping point
 */
public String file() { return src.file(); }

/**
 * Returns the line number of this stopping point.
 *
 * @return the line number of this stopping point
 */
public int line() { return src.line(); }

/**
 * Returns the column number of this stopping point.
 *
 * @return the column number of this stopping point
 */
public int column() { return src.column(); }

/**
 * Returns the label of this stopping point.
 *
 * @return the label of this stopping point
 */
public LdbLabel label() { return label; }

/**
 * Returns the environment at this stopping point.
 *
 * @return the environment at this stopping point
 */
public LdbEnv env() { return env; }
```

3.4.9 Compilation units

- Abstraction : a compilation unit
- Constructors:
  - `LdbCompUnit` : create a compilation unit
  - `addProc` : add a procedure to this compilation unit
  - `addExported` : add a public class to this compilation unit
  - `addPrivate` : add a private class to this compilation unit
- Observers :
  - `extern` : externalize this compilation unit

The `extern` method emits the symbol table information of this compilation unit. There are entries in the `ldb_st` section which point to the top-level symbol table information and the actual data is placed in the `ldb_st_data` section. For example, the following information is emitted to the assembly-language output:

```
.section .ldb_st
.long .Lldb_st_src
.long .Lldb_st_arch
.long .Lldb_st_anchor
.long .Lldb_st_procs
.long .Lldb_st_externs
.long .Lldb_st_privates
.long .Lldb_st_locals
.long .Lldb_st_sourcemap
.section .ldb_st_data
.Lldb_st_src:
.byte 116
.byte 101
.byte 115
.byte 116
.byte 46
.byte 99
.byte 0
.Lldb_st_arch:
...
.Lldb_st_anchor:
...
.Lldb_st_procs:
.long 2
.long .LS23
.long .LS31
.Lldb_st_externs:
```

```
.long 2
.long .LIm161
.long .LS23
.long .LIm163
.long .LS31
.Lldb_st_privates:
.long 2
.long .LIm179
.long .LS16
.long .LIm181
.long .LS38
.Lldb_st_locals:
.long 1
.long .LS10
.Lldb_st_sourcemap:
.long 2
.long .LIm33
.long 1
.long .LS31
.long .LIm12
.long 1
.long .LS23
```

Since the compilation unit is the top-level information, this class does not have the `ref` method.

165 ⟨*LdbCompUnit.java* 165⟩≡

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.LinkedList;

/**
 * Emits the symbol table information of this compilation unit.
 */
public class LdbCompUnit extends LdbInfo {
    ⟨LdbCompUnit.java: abstraction 166⟩
    ⟨LdbCompUnit.java: constructors 167⟩
    ⟨LdbCompUnit.java: observers 169⟩
}
```

166 ⟨*LdbCompUnit.java: abstraction* 166⟩≡ (165)

```
/**
 * Procedures in this compilation unit.
 */
private LdbArray           procs     = new LdbArray();
/**
 * Exported symbols to other compilation units: public classes.
 */
private LdbArray           exported  = new LdbArray();
private LinkedList<String> exportedL = new LinkedList<String>();
/**
 * Private symbols  in this  compilation unit : non-public classes.
 */
private LdbArray           privates  = new LdbArray();
private LinkedList<String> privatesL = new LinkedList<String>();
/**
 * Local symbols    in this  compilation unit.
 */
private LdbArray           locals    = new LdbArray();
/**
 * Mapping between source file names and their procedures.
 */
private LdbTable           srcmap    = new LdbTable();
private LinkedList<String> filenames = new LinkedList<String>();
/**
 * ldb's name for the cpu architecture.
 */
private String arch;
/**
 * The name of Java source program.
 */
private String src;
/**
 * Assembly-languague output
 */
private LdbFileHandle asm;
```

167 ⟨*LdbCompUnit.java: constructors* 167⟩≡ (165)

```
/**
 * Class constructor.
 * Creates the anchor symbol.
 */
public LdbCompUnit(String archName, String srcName, LdbFileHandle asmOut) {
    arch = archName;
    src  = srcName;
    asm  = asmOut;
    LdbPlcmt.initialize();
}

private void addSym(LdbArray arr, LdbSymbol s) {
    for (int j = 0; j < arr.size(); j++)
        if (s == arr.get(j)) return;
    arr.addHigh(s);
}

private void addLst(LinkedList lst, String s) {
    if (!lst.contains(s)) lst.addLast(s);
}

/**
 * Adds a procedure to procs and srcmap.
 *
 * @param p added procedure
 */
public void addProc(LdbProc p) {
    String file = p.file();
    procs.addHigh(p);
    LdbArray procedures = (LdbArray)srcmap.get(file);
    if (procedures == null) {
        LdbArray ps = new LdbArray();
        ps.addHigh(p);
        srcmap.put(file, ps);
        addLst(filenames, file);
    } else addSym(procedures, p);
    /* add a local symbol to locals */
    for (Iterator<LdbSymbol> ls = p.locals().iterator(); ls.hasNext(); ) {
        addSym(locals, ls.next());
    }
}

/**
 * Adds a public class to exported - not used.
 *
```

```
 * @param l added exported symbol
 */
public void addExported(LdbSymbol l) {
    addSym(exported, l);
    addLst(exportedL, l.name());
}

/**
 * Adds a private class to privates.
 *
 * @param l added private symbol
 */
public void addPrivate(LdbSymbol l) {
    addSym(privates, l);
    addLst(privatesL, l.name());
}
```

169 ⟨*LdbCompUnit.java: observers* 169⟩≡ (165)

```
/**
 * Emits the symbol table information of this compilation unit.
 *
 * @param o file where this information to be emitted
 */
public void extern(LdbFileHandle h) {
    try {
        LdbPlcmt.finalize(h);
        LdbUtil.emitDataSection(h);
        int i;
        // emit all the procedures
        for (i = 0; i < procs.size()   ; i++) procs.get(i).extern(h);
        // emit all the exported symbols
        for (i = 0; i < exported.size(); i++) exported.get(i).extern(h);
        for (Iterator<String> ls = exportedL.iterator(); ls.hasNext();) {
            LdbSingleton.newName(ls.next()).extern(h);
        }
        // emit all the private symbols
        for (i = 0; i < privates.size(); i++) privates.get(i).extern(h);
        for (Iterator<String> ls = privatesL.iterator(); ls.hasNext();) {
            LdbSingleton.newName(ls.next()).extern(h);
        }
        // emit all the local symbols
        for (i = 0; i < locals.size()  ; i++) locals.get(i).extern(h);
        String anchor = LdbPlcmt.anchor;
        String lab_src       = anchor + "_src";
        String lab_arch      = anchor + "_arch";
        String lab_anchor    = anchor + "_anchor";
        String lab_procs     = anchor + "_procs";
        String lab_externs   = anchor + "_externs";
        String lab_privates  = anchor + "_privates";
        String lab_locals    = anchor + "_locals";
        String lab_sourcemap = anchor + "_sourcemap";
        LdbUtil.emitSectionClose(h);
        LdbUtil.emitSectionOpen("ldb_st", h);
        LdbUtil.emitName (lab_src      , h);
        LdbUtil.emitName (lab_arch     , h);
        LdbUtil.emitName (lab_anchor   , h);
        LdbUtil.emitName (lab_procs    , h);
        LdbUtil.emitName (lab_externs  , h);
        LdbUtil.emitName (lab_privates , h);
        LdbUtil.emitName (lab_locals   , h);
        LdbUtil.emitName (lab_sourcemap, h);
        LdbUtil.emitSectionClose(h);
        LdbUtil.emitSectionOpen("ldb_st_data", h);
```

```
        LdbUtil.emitLabel(lab_src      , h);
        LdbUtil.emitStr  (    src      , h);
        LdbUtil.emitLabel(lab_arch     , h);
        LdbUtil.emitStr  (    arch     , h);
        LdbUtil.emitLabel(lab_anchor   , h);
        LdbUtil.emitStr  (    anchor   , h);
        emitRefs1(           procs   , lab_procs   , h);
        emitRefs2(exportedL, exported, lab_externs , h);
        emitRefs2(privatesL, privates, lab_privates, h);
        emitRefs1(           locals  , lab_locals  , h);
        LdbUtil.emitLabel(lab_sourcemap, h);
        emitSize(filenames.size(), h);
        for (Iterator<String> ns = filenames.iterator(); ns.hasNext(); ) {
            String file = ns.next();
            LdbArray ps = (LdbArray)srcmap.get(file);
            emitStr(file, h); emitSize(ps.size(), h);
            for (i = 0; i < ps.size(); i++) emitRef(ps.get(i).ref(), h);
        }
        LdbUtil.emitSectionClose(h);
    } catch (java.io.IOException e) {
        System.out.println("java.io.IOException");
    }
}

private void emitRef(LdbExtRef ref, LdbFileHandle h) {
    LdbUtil.emitName(ref.toStr(), h);
}
private void emitStr(String str, LdbFileHandle h) {
    LdbStr ldbstr = LdbSingleton.newStr(str);
    ldbstr.extern(h); LdbUtil.emitName(ldbstr.ref().toStr(), h);
}
private void emitSize(int size, LdbFileHandle h) {
    LdbUtil.emitName(new Integer(size).toString(), h);
}
private void emitRefs1(LdbArray symsarr, String lab, LdbFileHandle h) {
    LdbUtil.emitLabel(lab, h);
    emitSize(symsarr.size(), h);
    for(int i = 0; i < symsarr.size(); i++) {
        emitRef(symsarr.get(i).ref(), h);
    }
}
private void emitRefs2(LinkedList<String> symslist, LdbArray symsarr,
                       String lab, LdbFileHandle h) {
    LdbUtil.emitLabel(lab, h);
    emitSize(symsarr.size(), h);
    for(int i = 0; i < symsarr.size(); i++) {
```

```
        emitRef(LdbSingleton.newName(symslist.get(i)).ref(), h);
        emitRef(symsarr.get(i).ref(), h);
    }
}

/**
 * Since the compilation unit is the top-level information,
 * the ref method should not be called.
 */
public LdbExtRef ref() { assert(false); return super.ref(); }
```

Chapter 4

The `ldb` interface in Standard ML

4.1 Legend

We use the following naming conventions for the `ldb` interface in Standard ML:

Token	Naming convention	Example
Variables	Initial lower case. Use embedded caps for multiword names.	`getItem`
Constructors	Initial upper case. Use embedded caps for multiword names.	`EmptyQueue`
Types	All lower case. Use underscores for multiword names.	`priority_queue`
Signatures	Begin with `LDB_`. All upper case. Use underscores for multiword names.	`LDB_PRIORITY_QUEUE`
Structures	Begin with `Ldb`. Initial upper case. Use embedded caps for multiword names.	`LdbPriorityQueue`
Functors	Same as structure convention, except `Fun` completes the name.	`LdbPriorityQueueFun`

The functions in the `ldb` interface are implemented as methods of an object; the first argument of each observer function has the type of the object. For example, `LdbArray.addHigh` is a method of an object `LdbArray` and the first argument of `LdbArray.addHigh` has the type `LdbArray.t`. The only exception is `LdbCompUnit`. Because a compilation unit is the top-level information, the `ldb` interface makes it an implicit part of the interface's state and the observers of the compilation unit do not have the compilation unit as their first arguments. In addition, a conversion function is defined in a subtype. For example, a conversion function between `LdbArray` and its super type `LdbInfo` is defined as `LdbArray.asInfo` instead of `LdbInfo.ofArray`.

Abbreviations are used only for the followings:

`arg`	argument
`comp`	compilation
`env`	environment
`ext`	external
`float`	floating-point numbers
`hex`	hexadecimal
`info`	information
`int`	integer
`loc`	location
`loci`	stopping points
`locus`	a stopping point
`plcmt`	placement
`proc`	procedure
`ps`	PostScript
`ref`	reference
`reg`	register
`src`	source
`str`	string
`var`	variable

4.2 Preliminaries

This section describes the preliminary `ldb` interface which should be the source-language independent, the compiler independent but implementation-language (Standard ML) dependent.

4.2.1 Utility functions

173 ⟨*ldbutil.sig* 173⟩≡

```
signature LDB_UTIL =
sig
  val inc     : int ref -> unit
  val dec     : int ref -> unit
  val assert  : bool -> unit
  val addList : 'a -> 'a list ref -> unit
end
```

Defines:
`addList`, used in chunks 222 and 241.
`assert`, used in chunks 177 and 198.
`dec`, used in chunk 177.
`inc`, used in chunks 177, 179, 186, and 241.
`LDB_UTIL`, used in chunk 174.

174 ⟨*ldbutil.sml* 174⟩≡

```
structure LdbUtil :> LDB_UTIL =
struct
  fun inc i = i := !i + 1
  fun dec i = i := !i - 1
  fun assert true  = ()
    | assert false = raise Fail "assertion failure"
  fun addList a l = l := a::(!l)
end
```

Defines:
`addList`, used in chunks 222 and 241.
`assert`, used in chunks 177 and 198.
`dec`, used in chunk 177.
`inc`, used in chunks 177, 179, 186, and 241.
`LdbUtil`, used in chunks 177, 179, 186, 198, 222, and 241.
Uses `LDB_UTIL` 173.

4.2.2 File handle

- Abstraction : a file handle to control emitting debugging information
- Constructors:
  - `openFile`: open a specified file
- Observers :
  - `closeFile`: close the open file
  - `write`: write a given string to the file
  - `push`: increase the current level
  - `pop`: decrease the current level
  - `assertTop`: make sure that the current level is 'top level'

`push`, `pop`, and `assertTop` are provided to maintain the current level of a file handle. So that the `ldb` interface can keep the following invariance:

- When a debugging information is to be emitted, the file handle is at 'top level'.

175 ⟨*ldbfilehandle.sig* 175⟩≡

```
signature LDB_FILE_HANDLE =
sig
  (** abstraction ***********************************************************)
  (* a file handle to control emitting debugging information;
   * include an open file stream and its current level
   *)
  type t

  (** constructors **********************************************************)
  (* open a specified file
   *
   * precondition : file is not open
   * postcondition: file is open and its default current level is 0
   *)
  val openFile : string -> t

  (** observers **************************************************************)
  (* close the open file
   *
   * precondition : file is open
   * postcondition: file is closed
   *)
  val closeFile : t -> unit
```

```
    (* write a string to the file
     *
     * precondition : file is open
     * postcondition: string is written to the file
     *)
    val write : t -> string -> unit

    (* increase current level
     *
     * precondition : current level is equal or greater than 0
     * postcondition: current level is increased and greater than 0
     *)
    val push : t -> unit

    (* decrease current level
     *
     * precondition : current level is greater than 0
     * postcondition: current level is decreased and equal or greater than 0
     *)
    val pop : t -> unit

    (* make sure that the current level is top level
     *
     * precondition : current level is equal or greater than 0
     * postcondition: if the current level is not 0, assertion error
     *)
    val assertTop : t -> unit
  end
```

Defines:
  `assertTop`, used in chunk 186.
  `closeFile`, never used.
  `LDB_FILE_HANDLE`, used in chunk 177.
  `openFile`, never used.
  `pop`, used in chunks 198 and 230.
  `push`, used in chunks 198 and 230.
  `t`, used in chunks 180, 185, 190–93, 196, 200, 202, 206b, 239, 241, and 254.
  `write`, used in chunk 251.
Uses `file` 204 205 218 222 245 247 and `make` 185 186 187 189 194 198 201 203 204 205 207 208 209 211 212 215 218 222 226 228 229 230 232 234

177 ⟨*ldbfilehandle.sml* 177⟩≡

```
structure LdbFileHandle :> LDB_FILE_HANDLE =
struct
  type t = TextIO.outstream * int ref
  (* open  a specified file *)
  fun openFile f =
    let val level = ref 0 (* current level *)
    in  (TextIO.openOut f, level)
    end
  (* close a specified file *)
  fun closeFile (out, level) = TextIO.closeOut out
  (* write a specified string to this file *)
  fun write (out, _) s = TextIO.output(out, s)
  (* increase current level *)
  fun push (_, level) = LdbUtil.inc level
  (* decrease current level *)
  fun pop  (_, level) = LdbUtil.dec level
  (* make sure that the current level is top level *)
  fun assertTop (_, level) = LdbUtil.assert (!level = 0)
end
structure FH = LdbFileHandle
```

Defines:
`assertTop`, used in chunk 186.
`closeFile`, never used.
`FH`, used in chunks 179, 186, 198, 230, and 251.
`LdbFileHandle`, used in chunks 178, 180, 184, 185, 239, 243, and 249.
`openFile`, never used.
`pop`, used in chunks 198 and 230.
`push`, used in chunks 198 and 230.
`t`, used in chunks 180, 185, 190–93, 196, 200, 202, 206b, 239, 241, and 254.
`write`, used in chunk 251.
Uses `assert` 173 174, `dec` 173 174, `file` 204 205 218 222 245 247, `inc` 173 174, `LDB_FILE_HANDLE` 175, `LdbUtil` 174, and `make` 185 186 187 189 194 198 201 203 204 205 207 208 209 211 212 215 218 222 226 2

4.2.3 External references

- Abstraction : a reference to an item that has been externed
- Constructors:
  - `fromBase`: create an external reference from a base name and a generated number
  - `fromStr`: create an external reference from a string
- Observers :
  - `write`: write this external reference to a given file handle
  - `toStr`: return the string representation of this external reference

External references satisfy the following invariance:

- The external reference of every immutable information is its string representation.
- The external reference of every mutable information is a uniquely generated name.

178 ⟨*ldbextref.sig* 178⟩≡

```
signature LDB_EXT_REF =
sig
  (** abstraction ***********************************************************)
  (* a reference to an item that has been externed *)
  type t

  (** constructors **********************************************************)
  (* create an external reference from a base name
   *
   * postcondition: this external reference's string representation is
   *                its generated name which is the given base name
   *                appended by its unique index
   *)
  val fromBase : string -> t

  (* create an external reference from a string
   *
   * postcondition: this external reference's string representation is
   *                the string surrounded by parentheses
   *)
  val fromStr : string -> t

  (** observers *************************************************************)
  (* write an external reference
```

```
     *
     * precondition : a given file handle is open
     * postcondition: external reference is written
     *)
    val write : LdbFileHandle.t -> t -> unit

    (* return an external reference as a string
     *
     * postcondition: true
     *)
    val toStr : t -> string
  end
```

Defines:
  `fromBase`, used in chunks 189, 198, 230, and 237.
  `fromStr`, used in chunk 198.
  `LDB_EXT_REF`, used in chunk 179.
  `t`, used in chunks 180, 185, 190–93, 196, 200, 202, 206b, 239, 241, and 254.
  `toStr`, used in chunks 189, 198, 201, 203, 230, 237, and 251.
  `write`, used in chunk 251.
Uses `file` 204 205 218 222 245 247, `LdbFileHandle` 177, and `name` 207 208.

179 ⟨*ldbextref.sml* 179⟩≡

```
  structure LdbExtRef :> LDB_EXT_REF =
  struct
    type t = string
    val n = ref 0 (* integer value to form an external name *)
    fun fromBase b   = (b ^ (LdbUtil.inc n; (Int.toString (!n))))
    fun fromStr s = s
    fun toStr   s = s
    (* write a specified external reference as a string *)
    fun write out extref = FH.write out extref
  end
  structure ER = LdbExtRef
```

Defines:
  `ER`, used in chunks 186, 189, 198, 201, 203, 230, 237, and 251.
  `fromBase`, used in chunks 189, 198, 230, and 237.
  `fromStr`, used in chunk 198.
  `LdbExtRef`, used in chunks 9, 20a, 21, 23–25, 27, 28, 31a, 33a, 34b, 36a, 37b, 39, 41–43, 46, 49, 54a, 72b, 74, 80, 184, and 185.
  `n`, used in chunks 11, 17, 30c, 31a, 44–47, 49, 95–97, 100a, and 200.
  `t`, used in chunks 180, 185, 190–93, 196, 200, 202, 206b, 239, 241, and 254.
  `toStr`, used in chunks 189, 198, 201, 203, 230, 237, and 251.
  `write`, used in chunk 251.
Uses `FH` 177, `inc` 173 174, `LDB_EXT_REF` 178, `LdbUtil` 174, and `name` 207 208.

4.2.4 Compiler supports

Utility functions from a compiler for the `ldb` interface.

180 ⟨*ldbcompilersupport.sig* 180⟩≡

```
signature LDB_COMPILER_SUPPORT =
sig
  (* labels ************************************************************)
  (* return a fresh compiler/platform specific label name *)
  val makeLabelName         : unit   -> string
  (* return a fresh compiler/platform specific label name from a given name *)
  val makeLabelNameFrom     : string -> string
  (* return a base name from a given compiler/platform-specific label name *)
  val dropPrefixOfLabelName : string -> string
  (* return a no-op format to tag a given stopping point's name in the
   * assembly-language output *)
  val noOpFormat            : string -> string

  (* assembly output ***************************************************)
  (* emit a given label name to a given file handle *)
  val emitLabel   : string -> LdbFileHandle.t -> unit
  (* emit a given string to a given file handle *)
  val emitStr     : string -> LdbFileHandle.t -> unit
  (* emit a given data name to a given file handle *)
  val emitName    : string -> LdbFileHandle.t -> unit
  (* export a given anchor symbol name to a given file handle *)
  val exportName  : string -> LdbFileHandle.t -> unit
  (* emit a given section name to a given file handle *)
  val emitSectionOpen  : string -> LdbFileHandle.t -> unit
  (* emit a section closure    to a given file handle *)
  val emitSectionClose :           LdbFileHandle.t -> unit
  (* emit a data section *)
  val emitDataSection  :           LdbFileHandle.t -> unit

  (* private data ******************************************************)
  (* emit compiler's private data possibly for its expression server *)
  val emitPrivate      :           LdbFileHandle.t -> unit
end
```

Defines:
`dropPrefixOfLabelName`, used in chunk 198.
`emitDataSection`, used in chunk 251.
`emitLabel`, used in chunks 189, 198, 230, 237, and 251.
`emitName`, used in chunks 189, 198, 201, 203, 230, 237, 244, and 251.
`emitPrivate`, never used.
`emitSectionClose`, used in chunks 241 and 251.
`emitSectionOpen`, used in chunk 251.
`emitStr`, used in chunks 189, 237, and 251.
`exportName`, used in chunk 241.
`LDB_COMPILER_SUPPORT`, used in chunk 182.

`makeLabelName`, never used.
`makeLabelNameFrom`, used in chunk 198.
`noOpFormat`, never used.
Uses `file` 204 205 218 222 245 247, `label` 212 215 245 247, `LdbFileHandle` 177, `name` 207 208,
and `t` 175 177 178 179 184 186 187 189 194 198 201 203 204 205 206a 207 208 209 211 212 215 218 222 226 228 229 230 232 23

182 ⟨*ldbcompilersupport.sml* 182⟩≡

```
structure LdbCompilerSupport :> LDB_COMPILER_SUPPORT =
struct
  (* labels ************************************************************)
  (* return a fresh compiler/platform specific label name *)
  fun makeLabelName () = ""
  (* return a fresh compiler/platform specific label name from a given name *)
  fun makeLabelNameFrom      name = name
  (* return a base name from a given compiler/platform-specific label name *)
  fun dropPrefixOfLabelName name = name
  (* return a no-op format to tag a given stopping point's name in the
   * assembly-language output *)
  fun noOpFormat label = label

  (* assembly output ***************************************************)
  (* emit a given label name to a given file handle *)
  fun emitLabel label out = ()
  (* emit a given string to a given file handle *)
  fun emitStr     str out = ()
  (* emit a given data name to a given file handle *)
  fun emitName    name out = ()
  (* export a given anchor symbol name to a given file handle *)
  fun exportName  name out = ()
  (* emit a given section name to a given file handle *)
  fun emitSectionOpen  section out = ()
  (* emit a section closure    to a given file handle *)
  fun emitSectionClose         out = ()
  (* emit a data section *)
  fun emitDataSection          out = ()

  (* private data ******************************************************)
  (* emit compiler's private data possibly for its expression server *)
  fun emitPrivate              out = ()
end
structure CS = LdbCompilerSupport
```

Defines:
`CS`, used in chunks 189, 198, 201, 203, 230, 237, 241, 244, and 251.
`dropPrefixOfLabelName`, used in chunk 198.
`emitDataSection`, used in chunk 251.
`emitLabel`, used in chunks 189, 198, 230, 237, and 251.
`emitName`, used in chunks 189, 198, 201, 203, 230, 237, 244, and 251.
`emitPrivate`, never used.
`emitSectionClose`, used in chunks 241 and 251.
`emitSectionOpen`, used in chunk 251.
`emitStr`, used in chunks 189, 237, and 251.
`exportName`, used in chunk 241.
`LdbCompilerSupport`, never used.
`makeLabelName`, never used.

`makeLabelNameFrom`, used in chunk 198.
`noOpFormat`, never used.
Uses `file` 204 205 218 222 245 247, `label` 212 215 245 247, `LDB_COMPILER_SUPPORT` 180, and `name` 207 208.

4.3 Generic PostScript data

This section includes the `ldb` interface for generic PostScript data which should be the source-language independent, the compiler independent, but implementation-language (Standard ML) dependent.

4.3.1 PostScript data (`ldb` information)

- Abstraction : the base for all the PostScript data
- Observers :
  - `extern`: externalize this information
  - `ref'`: return this information's external reference

An `ldb` information satisfies the following invariance:

* `extern` is called after the information has been created.

* `ref'` is called after the information has been externalized.

184 ⟨*ldbinfo.sig* 184⟩≡ 185 ▷

```
signature LDB_INFO =
sig
  (** abstraction ***********************************************************)
  (* the base type for all the ldb information *)
  type t

  (** observers **************************************************************)
  (* extern a specified information
   *
   * precondition : file handle must be at top level
   * postcondition: file handle is still at top level,
   *                and information has been externed
   *)
  val extern : t -> LdbFileHandle.t -> unit

  (* return a specified information's external reference
   *
   * precondition : a given information has already been externed
   * postcondition: this external reference can be written anywhere
   *                in a file handle
   *)
  val ref' : t -> LdbExtRef.t
end
```

Defines:
`extern`, used in chunks 185, 189, 198, 201, 203, 230, and 237.
`LDB_INFO`, used in chunks 185, 187, 194, 218, 229, 232, and 235.

`ref'`, used in chunks 185, 189, 198, 201, 203, 230, 237, and 251.
`t`, used in chunks 180, 185, 190–93, 196, 200, 202, 206b, 239, 241, and 254.
Uses `file` 204 205 218 222 245 247, `LdbExtRef` 20b 179, and `LdbFileHandle` 177.

- Constructors:
  - `make`: create a fresh information
- Observers :
  - `eq` : return whether given two information are the same

The following signature `LDB_INFO'` reveals the constructor of `ldb` information. It also includes the observers `extern` and `ref'` from `LDB_INFO` but `ref'` here has a different invariance as follows:

* `ref'` is called after `extern` called; it's okay whether `extern` has not been returned.

185 ⟨*ldbinfo.sig* 184⟩+≡ ◁184

```
signature LDB_INFO' =
sig
  include LDB_INFO
  (** constructors *******************************************************)
  (* create a fresh information
   *
   * precondition : this information does not exist
   * postcondition: this information is created
   *
  *)
  val make : { extern : LdbFileHandle.t -> unit
             , ref'   : unit -> LdbExtRef.t
             } -> t

  (** observers ***********************************************************)
  (* return whether given two symbols are the same
   *
   * precondition : true
   *)
  val eq : t -> t -> bool
end
```

Defines:
`eq`, used in chunk 254.
`LDB_INFO'`, used in chunks 186 and 198.
`make`, used in chunks 175, 177, 190–93, 206b, 235, 238, and 241.
Uses `extern` 184 186 249 251, `LDB_INFO` 184, `LdbExtRef` 20b 179, `LdbFileHandle` 177,
`ref'` 184 186, and `t` 175 177 178 179 184 186 187 189 194 198 201 203 204 205 206a 207 208 209 211 212 215 218 222 226 228

186 ⟨*ldbinfo.sml* 186⟩≡

```
structure LdbInfo :> LDB_INFO' =
struct
  type t = { extern : FH.t -> unit
           , ref'   : unit -> ER.t
           , id     : int
           }
  val n = ref 0
  fun make {extern, ref'=ref'} =
    {extern=extern, ref'=ref', id=(LdbUtil.inc n; !n)}
  fun extern (i: t) out = (FH.assertTop out; #extern i out)
  fun ref'   (i: t)     = #ref' i ()
  fun eq (i1 : t) (i2 : t) = (#id i1) = (#id i2)
end
```

Defines:
`eq`, used in chunk 254.
`extern`, used in chunks 185, 189, 198, 201, 203, 230, and 237.
`LdbInfo`, used in chunks 187, 189, 194, 198, 201, 203, 218, 222, 229, 230, 232, 234, 235, 237, 251, and 254.
`make`, used in chunks 175, 177, 190–93, 206b, 235, 238, and 241.
`n`, used in chunks 11, 17, 30c, 31a, 44–47, 49, 95–97, 100a, and 200.
`ref'`, used in chunks 185, 189, 198, 201, 203, 230, 237, and 251.
`t`, used in chunks 180, 185, 190–93, 196, 200, 202, 206b, 239, 241, and 254.
Uses `assertTop` 175 177, `ER` 179, `FH` 177, `inc` 173 174, `LDB_INFO'` 185, and `LdbUtil` 174.

4.3.2 Immutable PostScript data

- Abstraction : an immutable PostScript data
- Constructors:
  - `make`: create a fresh immutable PostScript data
- Observers :
  - `asInfo` : return a given immutable information as an `ldb` information (subsumption)

Each immutable information is externalized by `extern` as follows. If it's not yet externalized, its label is emitted followed by `0` to denote that this information is immutable and then the label for the string representation of the immutable information is emitted. The string representation and its label are emitted before the immutable information is emitted.

For example, `(test.sml)` is externalized as follows:

```
.LIg71:
.byte 40
.byte 116
.byte 101
.byte 115
.byte 116
.byte 46
.byte 115
.byte 109
.byte 108
.byte 41
.byte 0
.LIm12:
.long 0
.long .LIg71
```

Its label `.LIm12` is emitted followed by `0` and `.LIg71` which represents the string `(test.sml)`.

187 ⟨*ldbimmutable.sig* 187⟩≡

```
signature LDB_HASH_TABLE =
sig
  type table
  val  add   : table -> string -> LdbInfo.t -> table
  val  find  : table -> string -> LdbInfo.t option
  val  empty : table
end
signature LDB_IMMUTABLE =
sig
```

```
    (** abstraction ***********************************************************)
    structure I : LDB_INFO
    (* the base type for all the immutable information *)
    type rep
    type t

    (** constructors ************************************************************)
    (* create a fresh immutable information
     *
     * postcondition: this immutable information is created
     *)
    val make : rep -> t

    (** observers ***************************************************************)
    (* return a given immutable information as an ldb information (subsumption)
     *
     * precondition : true
     *)
    val asInfo : t -> I.t
  end where type I.t = LdbInfo.t
```

Defines:
  add, used in chunks 202, 218, 226, 230, 237, 249, and 254.
  asInfo, used in chunks 190–93, 211, 215, 222, 228, 234, and 251.
  empty, used in chunks 202, 249, and 254.
  find, used in chunks 201, 203, 230, 237, and 254.
  I, used in chunks 196, 200, 202, 228, and 254.
  LDB_HASH_TABLE, used in chunk 189.
  LDB_IMMUTABLE, used in chunks 189–93 and 206a.
  make, used in chunks 175, 177, 190–93, 206b, 235, 238, and 241.
  rep, used in chunks 190–93 and 206a.
  t, used in chunks 180, 185, 190–93, 196, 200, 202, 206b, 239, 241, and 254.
  table, used in chunks 202 and 251.
Uses LDB_INFO 184 and LdbInfo 186.

189 ⟨*ldbimmutable.sml* 189⟩≡

```
structure LdbHashTable : LDB_HASH_TABLE =
struct
  structure HashTable = Table(type key = string val gt = String.>)
  type table = LdbInfo.t HashTable.table
  val  empty = HashTable.empty
  fun add  table def info = HashTable.insert((def,info), table)
  fun find table def = HashTable.find(def, table)
end
val ldbHashTable = ref LdbHashTable.empty
functor LdbImmutableFun ( type t
                          val asPS : t -> string
                        ) :> LDB_IMMUTABLE where type rep = t =
struct
  structure I = LdbInfo
  type rep    = t
  type t      = I.t
  fun make i  =
    let val isExterned = ref false
        val myref = ER.fromBase "Im"
        val def   = asPS i
        fun extern out =
          if (!isExterned) then ()
          else let val defLabel = ER.toStr(ER.fromBase "Ig")
               in  ( CS.emitLabel defLabel         out
                   ; CS.emitStr   def              out
                   ; CS.emitLabel (ER.toStr myref) out
                   ; CS.emitName  "0"              out
                   ; CS.emitName  defLabel         out
                   ; isExterned := true
                   )
               end
    in  case LdbHashTable.find (!ldbHashTable) def
          of SOME info => info
           | NONE      =>
            let val info = I.make { extern = extern
                                  , ref'   = (fn _ => myref)
                                  }
            in  ( ldbHashTable := LdbHashTable.add (!ldbHashTable) def info
                ; info )
            end
    end
  fun asInfo i = i
end
```

Defines:
add, used in chunks 202, 218, 226, 230, 237, 249, and 254.

`asInfo`, used in chunks 190–93, 211, 215, 222, 228, 234, and 251.
`empty`, used in chunks 202, 249, and 254.
`find`, used in chunks 201, 203, 230, 237, and 254.
`HashTable`, never used.
`I`, used in chunks 196, 200, 202, 228, and 254.
`LdbHashTable`, used in chunks 230 and 237.
`ldbHashTable`, used in chunks 230 and 237.
`LdbImmutableFun`, used in chunks 190–93 and 206b.
`make`, used in chunks 175, 177, 190–93, 206b, 235, 238, and 241.
`rep`, used in chunks 190–93 and 206a.
`t`, used in chunks 180, 185, 190–93, 196, 200, 202, 206b, 239, 241, and 254.
`table`, used in chunks 202 and 251.
Uses `CS` 182, `emitLabel` 180 182, `emitName` 180 182, `emitStr` 180 182, `ER` 179, `extern` 184 186 249 251, `fromBase` 178 179, `LDB_HASH_TABLE` 187, `LDB_IMMUTABLE` 187, `LdbInfo` 186, `ref'` 184 186, and `toStr` 178 179 238.

4.3.3 PostScript null objects

- Abstraction : a PostScript null
- Constructors:
  - `ldbNull` : create a null object

190a ⟨*ldbnull.sig* 190a⟩≡

```
signature LDB_NULL = LDB_IMMUTABLE where type rep = unit
```

Defines:
`LDB_NULL`, used in chunk 190b.
Uses `LDB_IMMUTABLE` 187 and `rep` 187 189 196 198 198 201 203.

190b ⟨*ldbnull.sml* 190b⟩≡

```
structure LdbNull :> LDB_NULL = LdbImmutableFun ( type t   = unit
                                                  val asPS = fn _ => "null" )
val ldbNull = LdbNull.asInfo(LdbNull.make())
```

Defines:
`LdbNull`, never used.
`ldbNull`, used in chunk 222.
Uses `asInfo` 187 189 194 198 201 203 229 230 235 237, `LDB_NULL` 190a, `LdbImmutableFun` 189, `make` 185 186 187 189 194 198 201 203 204 205 207 208 209 211 212 215 218 222 226 228 229 230 232 234 237 245 247 249 254, and `t` 175 177 178 179 184 186 187 189 194 198 201 203 204 205 206a 207 208 209 211 212 215 218 222 226 228 229 230 232 234 235 237 243 2

4.3.4 PostScript integers

- Abstraction : a PostScript integer
- Constructors:
  - `ldbInt` : create a PostScript decimal integer
  - `ldbHex` : create a PostScript hexadecimal integer

191a ⟨*ldbint.sig* 191a⟩≡
```
signature LDB_INT = LDB_IMMUTABLE where type rep = int
```
Defines:
`LDB_INT`, used in chunk 191b.
Uses `LDB_IMMUTABLE` 187 and `rep` 187 189 196 198 198 201 203.

191b ⟨*ldbint.sml* 191b⟩≡
```
fun d2h d =
  let fun d2s 10 = "a"
        | d2s 11 = "b"
        | d2s 12 = "c"
        | d2s 13 = "d"
        | d2s 14 = "e"
        | d2s 15 = "f"
        | d2s  d = Int.toString d
  in  (if d >= 16 then (d2h (d div 16)) else "") ^ d2s (d mod 16)
  end
structure LdbInt :> LDB_INT =
  LdbImmutableFun ( type t   = int
                    val asPS = fn i => Int.toString i)
structure LdbHex :> LDB_INT =
  LdbImmutableFun ( type t   = int
                    val asPS = fn i => "16#" ^ (d2h i))
val ldbInt = LdbInt.asInfo o LdbInt.make
val ldbHex = LdbHex.asInfo o LdbHex.make
```
Defines:
`d2h`, never used.
`LdbHex`, never used.
`ldbHex`, used in chunk 222.
`LdbInt`, never used.
`ldbInt`, used in chunks 208 and 222.
Uses `asInfo` 187 189 194 198 201 203 229 230 235 237, `LDB_INT` 191a, `LdbImmutableFun` 189,
`make` 185 186 187 189 194 198 201 203 204 205 207 208 209 211 212 215 218 222 226 228 229 230 232 234 237 245 247 249 254,
and `t` 175 177 178 179 184 186 187 189 194 198 201 203 204 205 206a 207 208 209 211 212 215 218 222 226 228 229 230 232 23

4.3.5 PostScript strings

- Abstraction : a PostScript string
- Constructors:
  - `ldbStr` : create a PostScript string

192a ⟨*ldbstr.sig* 192a⟩≡
```
signature LDB_STR = LDB_IMMUTABLE where type rep = string
```
Defines:
`LDB_STR`, used in chunk 192b.
Uses `LDB_IMMUTABLE` 187 and `rep` 187 189 196 198 198 201 203.

192b ⟨*ldbstr.sml* 192b⟩≡
```
structure LdbStr :> LDB_STR =
  LdbImmutableFun ( type t   = string
                    val asPS = fn s => ("("^s^")") )
val ldbStr = LdbStr.asInfo o LdbStr.make
```
Defines:
`LdbStr`, never used.
`ldbStr`, used in chunks 208, 222, 234, and 251.
Uses `asInfo` 187 189 194 198 201 203 229 230 235 237, `LDB_STR` 192a, `LdbImmutableFun` 189,
`make` 185 186 187 189 194 198 201 203 204 205 207 208 209 211 212 215 218 222 226 228 229 230 232 234 237 245 247 249 254,
and `t` 175 177 178 179 184 186 187 189 194 198 201 203 204 205 206a 207 208 209 211 212 215 218 222 226 228 229 230 232 234 235 237 243 2

4.3.6 PostScript names

- Abstraction : a PostScript name
- Constructors:
  - `ldbName` : create a PostScript name

192c ⟨*ldbname.sig* 192c⟩≡
```
signature LDB_NAME = LDB_IMMUTABLE where type rep = string
```
Defines:
`LDB_NAME`, used in chunk 192d.
Uses `LDB_IMMUTABLE` 187 and `rep` 187 189 196 198 198 201 203.

192d ⟨*ldbname.sml* 192d⟩≡
```
structure LdbName :> LDB_NAME = LdbImmutableFun ( type t   = string
                                                  val asPS = fn s => s )
val ldbName = LdbName.asInfo o LdbName.make
```
Defines:
`LdbName`, never used.
`ldbName`, used in chunks 198, 203, 230, and 251.
Uses `asInfo` 187 189 194 198 201 203 229 230 235 237, `LDB_NAME` 192c, `LdbImmutableFun` 189,
`make` 185 186 187 189 194 198 201 203 204 205 207 208 209 211 212 215 218 222 226 228 229 230 232 234 237 245 247 249 254,
and `t` 175 177 178 179 184 186 187 189 194 198 201 203 204 205 206a 207 208 209 211 212 215 218 222 226 228 229 230 232 234 235 237 243 2

4.3.7 PostScript procedures

- Abstraction : a PostScript procedure
- Constructors:
  - `ldbPsProc` : create a PostScript procedure

193a ⟨*ldbpsproc.sig* 193a⟩≡
```
signature LDB_PS_PROC = LDB_IMMUTABLE where type rep = string
```
Defines:
`LDB_PS_PROC`, used in chunks 193b and 232.
Uses `LDB_IMMUTABLE` 187 and `rep` 187 189 196 198 198 201 203.

193b ⟨*ldbpsproc.sml* 193b⟩≡
```
structure LdbPsProc :> LDB_PS_PROC =
  LdbImmutableFun ( type t   = string
                    val asPS = fn s => ("{"^s^"}") )
val ldbPsProc = LdbPsProc.asInfo o LdbPsProc.make
```
Defines:
`LdbPsProc`, used in chunks 232 and 234.
`ldbPsProc`, never used.
Uses `asInfo` 187 189 194 198 201 203 229 230 235 237, `LDB_PS_PROC` 193a, `LdbImmutableFun` 189,
`make` 185 186 187 189 194 198 201 203 204 205 207 208 209 211 212 215 218 222 226 228 229 230 232 234 237 245 247 249 254,
and `t` 175 177 178 179 184 186 187 189 194 198 201 203 204 205 206a 207 208 209 211 212 215 218 222 226 228 229 230 232 23

4.3.8 PostScript codes

- Abstraction : a PostScript code
- Constructors:
  - `ldbPsCode` : create a PostScript code

193c ⟨*ldbpscode.sig* 193c⟩≡
```
signature LDB_PS_CODE = LDB_IMMUTABLE where type rep = string
```
Defines:
`LDB_PS_CODE`, used in chunk 193d.
Uses `LDB_IMMUTABLE` 187 and `rep` 187 189 196 198 198 201 203.

193d ⟨*ldbpscode.sml* 193d⟩≡
```
structure LdbPsCode :> LDB_PS_CODE = LdbImmutableFun ( type t   = string
                                                       val asPS = fn s => (s) )
val ldbPsCode = LdbPsCode.asInfo o LdbPsCode.make
```
Defines:
`LdbPsCode`, never used.
`ldbPsCode`, never used.
Uses `asInfo` 187 189 194 198 201 203 229 230 235 237, `LDB_PS_CODE` 193c, `LdbImmutableFun` 189,
`make` 185 186 187 189 194 198 201 203 204 205 207 208 209 211 212 215 218 222 226 228 229 230 232 234 237 245 247 249 254,
and `t` 175 177 178 179 184 186 187 189 194 198 201 203 204 205 206a 207 208 209 211 212 215 218 222 226 228 229 230 232 23

4.3.9 Mutable PostScript data

- Abstraction : mutable PostScript data
- Constructors:
  - `make`: create a fresh mutable PostScript data
- Observers :
  - `asInfo` : return a given mutable information as an `ldb` information (subsumption)

A mutable information satisfies the following invariance:

* `extern` is called in any states.

* `ref'` is called only in the `EMITTED` state.

Each mutable information is externalized by `extern` as follows. If it's not yet externalized, its label is emitted followed by the number of its elements, its external reference, and every element in it.

For example, the following type dictionary

```
/T45 <<
        /decl    (type-decl)
        /printer {psproc}
        /symbol  S23
>> def
```

is externalized as follows:

```
.LT45:
.long 3
.long .LIm139   ;; T45
.long .LIm141   ;; decl
.long .LIm46    ;; (type-decl)
.long .LIm144   ;; printer
.long .LIm4     ;; {psproc}
.long .LIm146   ;; symbol
.long .LS23     ;; S23
```

Its label `.LT45` is emitted followed by the number of its elements `3`, its external reference `T45` denoted by `.LIm139`, and every key-value pair in the type dictionary.

194 ⟨*ldbmutable.sig* 194⟩≡ 196 ▷

```
signature LDB_MUTABLE =
sig
  (** abstraction *********************************************************)
  structure I : LDB_INFO
  (* the base type for all the mutable information *)
  type t

  (** constructors *********************************************************)
  (* create a fresh mutable information
   *
   * precondition : this mutable information does not exist
   * postcondition: this mutable information is created
   *)
  val make : {basename:string} -> t

  (** observers ************************************************************)
  (* return a given mutable information as an ldb information (subsumption)
   *
   * precondition : true
   *)
  val asInfo : t -> I.t
end where type I.t = LdbInfo.t
```

Defines:
asInfo, used in chunks 190–93, 211, 215, 222, 228, 234, and 251.
I, used in chunks 196, 200, 202, 228, and 254.
LDB_MUTABLE, used in chunks 196, 200, and 202.
make, used in chunks 175, 177, 190–93, 206b, 235, 238, and 241.
t, used in chunks 180, 185, 190–93, 196, 200, 202, 206b, 239, 241, and 254.
Uses LDB_INFO 184 and LdbInfo 186.

- Observers :
  - `mutate` : perform a given function to a given mutable information
  - `get` : get an element from a given mutable information
  - `size` : get the size of a given mutable information

The following signature `LDB_MUTABLE'` reveals the representation of mutable informtion `rep` and the observers `mutate` and `get`. It also includes the `make` constructor and the `asInfo` observer from `LDB_MUTABLE`. `mutate` and `get` have the following invariance:

* `mutate` is called in the `NEW` state.
* `get` is called in any states.
* `size` is called in any states.

196 ⟨*ldbmutable.sig* 194⟩+≡ ◁194

```
signature LDB_MUTABLE' =
sig
  include LDB_MUTABLE
  (** abstraction ***********************************************************)
  (* the representation of this mutable information *)
  type rep

  (** observers *************************************************************)
  (* perform a given function to a given mutable information
   *
   * precondition : this mutable information has not yet been externed
   * postcondition: this mutable information has not yet been externed
   *)
  val mutate : t -> (rep -> unit) -> unit

  (* return the value of an element in this mutable information
   *
   * precondition : true
   *)
  val get : t -> (rep -> I.t) -> I.t

  (* return the size of this mutable information
   *
   * precondition : true
   *)
  val size : t -> int
end
```

Defines:
`get`, used in chunks 251 and 254.

`LDB_MUTABLE'`, used in chunk 198.
`mutate`, used in chunks 201 and 203.
`rep`, used in chunks 190–93 and 206a.
`size`, used in chunks 203, 218, 222, and 251.
Uses `I` 187 189 194 198 198 201 203 218 222 229 230 232 234 235 237 251, `LDB_MUTABLE` 194,
and `t` 175 177 178 179 184 186 187 189 194 198 201 203 204 205 206a 207 208 209 211 212 215 218 222 226 228 229 230 232 23

198 ⟨*ldbmutable.sml* 198⟩≡

```
signature LDB_MUTABLE_ARG =
sig
  structure I : LDB_INFO'
  type rep
  val makeRep : unit -> rep
  val emitElements   : FH.t -> rep -> unit
  val emitDefinition : FH.t -> rep -> unit
  val size           : rep -> int
end where type I.t = LdbInfo.t

functor LdbMutableFun ( MA : LDB_MUTABLE_ARG
                      ) :> LDB_MUTABLE' where type rep = MA.rep =
struct
  structure I    = MA.I
  datatype state = NEW | EMITTED
  type t         = I.t * { rep    : MA.rep
                         , state  : state ref
                         , mutate : (MA.rep -> unit) -> unit
                         , get    : (MA.rep -> I.t ) -> I.t
                         , size   : unit -> int
                         }
  type rep       = MA.rep

  fun make {basename=b} =
    let val state = ref NEW
        val rep   = MA.makeRep()
        val myRef = ER.fromStr(CS.makeLabelNameFrom(ER.toStr(ER.fromBase b)))
        fun extern out =
          let val toS = ER.toStr
          in  case !state
                of NEW =>
                   ( state := EMITTED
                   ; MA.emitElements out rep
                   ; (case !state
                        of EMITTED =>
                           let val info =
                               ldbName(CS.dropPrefixOfLabelName(toS myRef))
                           in  ( I.extern     info                          out
                               ; CS.emitLabel (toS myRef)                   out
                               ; CS.emitName  (Int.toString (MA.size rep)) out
                               ; CS.emitName  (toS (I.ref' info))          out
                               ; FH.push out
                               ; MA.emitDefinition out rep
                               ; FH.pop  out
                               )
```

```
                              end
                            | _ => LdbUtil.assert false)
                      )
                    | EMITTED => ()         (* if already emitted then return *)
            end

          fun myref () = (LdbUtil.assert(!state = EMITTED); myRef)
          fun mutate f = (LdbUtil.assert(!state = NEW);     f rep)
          fun get    f = f rep
          fun size  () = MA.size rep

      in  (I.make {extern=extern,ref'=myref},
           {rep=rep,state=state,mutate=mutate,get=get,size=size})
      end

    fun asInfo ((i, _) : t) = i
    fun mutate ((_, i) : t) = #mutate i
    fun get    ((_, i) : t) = #get    i
    fun size   ((_, i) : t) = #size   i ()
  end
```

Defines:
asInfo, used in chunks 190–93, 211, 215, 222, 228, 234, and 251.
emitDefinition, used in chunks 201 and 203.
emitElements, used in chunks 201 and 203.
get, used in chunks 251 and 254.
I, used in chunks 196, 200, 202, 228, and 254.
LDB_MUTABLE_ARG, never used.
LdbMutableFun, used in chunks 201 and 203.
make, used in chunks 175, 177, 190–93, 206b, 235, 238, and 241.
makeRep, used in chunks 201 and 203.
mutate, used in chunks 201 and 203.
rep, used in chunks 190–93 and 206a.
size, used in chunks 203, 218, 222, and 251.
state, never used.
t, used in chunks 180, 185, 190–93, 196, 200, 202, 206b, 239, 241, and 254.
Uses assert 173 174, CS 182, dropPrefixOfLabelName 180 182, emitLabel 180 182, emitName 180 182, ER 179, extern 184 186 249 251, FH 177, fromBase 178 179, fromStr 178 179, LDB_INFO' 185, LDB_MUTABLE' 196, LdbInfo 186, ldbName 192d, LdbUtil 174, makeLabelNameFrom 180 182, pop 175 177, push 175 177, ref' 184 186, and toStr 178 179 238.

4.3.10 PostScript arrays (lists)

- Abstraction : a PostScript array whose elements are PostScript data
- Constructors:
  - `addHigh` : put a given value to the end of this array
- Observers :
  - `get` : get the value of the indexed component in this array
  - `size` : get the size of a given mutable information

200 ⟨*ldbarray.sig* 200⟩≡

```
signature LDB_ARRAY =
sig
  include LDB_MUTABLE
  (** constructors ************************************************************)
  (* put a given value to the end of this array
   *
   * precondition : addHigh [a,..,n] i
   * postcondition: [a,..,n,i]
   *)
  val addHigh : t -> I.t -> unit

  (** observers ***************************************************************)
  (* return the value of the indexed component in this array
   *
   * precondition : true
   *)
  val get : t -> int -> I.t

  (* return the size of this mutable information
   *
   * precondition : true
   *)
  val size : t -> int
end
```

Defines:
`addHigh`, used in chunks 222, 228, and 254.
`get`, used in chunks 251 and 254.
`LDB_ARRAY`, used in chunk 201.
`size`, used in chunks 203, 218, 222, and 251.
Uses `I` 187 189 194 198 198 201 203 218 222 229 230 232 234 235 237 251, `LDB_MUTABLE` 194, `n` 21 179 186, `put` 202 203, and `t` 175 177 178 179 184 186 187 189 194 198 201 203 204 205 206a 207 208 209 211 212 215 218 222 226 228 229

201 ⟨*ldbarray.sml* 201⟩≡

```
structure LdbArray :> LDB_ARRAY =
struct
  structure R = LdbMutableFun (
        structure I = LdbInfo
        type rep    = (int * I.t) list ref
        fun makeRep() = ref [] : rep
        fun emitElements out rep = app (fn (_,v) => I.extern v out) (rev(!rep))
        fun emitDefinition out rep =
          app (fn (_,v) => CS.emitName (ER.toStr (I.ref' v)) out) (rev(!rep))
        fun size rep = List.length (!rep)
  )
  structure I = R.I
  type rep    = R.rep
  type t      = R.t
  val make    = R.make
  val asInfo  = R.asInfo
  val size    = R.size
  fun addHigh a v =
    R.mutate a (fn rep => rep := (case !rep
                                    of [] => (0, v)
                                     | (i,_)::_ => (i+1, v))::(!rep))
  fun get a i =
    R.get a (fn rep => case List.find (fn (idx,_) => idx=i) (!rep)
                         of SOME (_,v) => v | _ => raise Fail "LdbArray.get")
end
```

Defines:
  `addHigh`, used in chunks 222, 228, and 254.
  `asInfo`, used in chunks 190–93, 211, 215, 222, 228, 234, and 251.
  `get`, used in chunks 251 and 254.
  `I`, used in chunks 196, 200, 202, 228, and 254.
  `LdbArray`, used in chunks 222, 228, and 251.
  `make`, used in chunks 175, 177, 190–93, 206b, 235, 238, and 241.
  `R`, never used.
  `rep`, used in chunks 190–93 and 206a.
  `size`, used in chunks 203, 218, 222, and 251.
  `t`, used in chunks 180, 185, 190–93, 196, 200, 202, 206b, 239, 241, and 254.
Uses `CS` 182, `emitDefinition` 198, `emitElements` 198, `emitName` 180 182, `ER` 179, `extern` 184 186 249 251, `find` 187 189, `LDB_ARRAY` 200, `LdbInfo` 186, `LdbMutableFun` 198, `makeRep` 198, `mutate` 196 198, `ref'` 184 186, and `toStr` 178 179 238.

4.3.11 PostScript dictionaries (tables)

- Abstraction : a PostScript dictionary with string keys and PostScript values
- Constructors:
  - `put`: add a given key/value pair to this table; overwrite any existing entry with the same key
- Observers :
  - `get`: find the value bound to a given key in this table

202 ⟨*ldbtable.sig* 202⟩≡

```
signature LDB_TABLE =
sig
  include LDB_MUTABLE
  (** constructors ************************************************************)
  (* add a given key/value pair to this table
   *
   * precondition : put <<...(k,v)...>> k i
   * postcondition: <<...(k,i)...>>
   *)
  val put : t -> string -> I.t -> unit

  (** observers ***************************************************************)
  (* return the value associated with the specified key in this table
   *
   * precondition : table might be empty
   * postcondition: return a value associated with a given key, if any
   *)
  val get : t -> string -> I.t
end
```

Defines:
`get`, used in chunks 251 and 254.
`LDB_TABLE`, used in chunks 203, 207, and 232.
`put`, used in chunks 200, 208, 211, 215, 222, 228, and 234.
Uses `add` 187 189, `empty` 187 189, `I` 187 189 194 198 198 201 203 218 222 229 230 232 234 235 237 251, `LDB_MUTABLE` 194, `t` 175 177 178 179 184 186 187 189 194 198 201 203 204 205 206a 207 208 209 211 212 215 218 222 226 228 229 230 232 234 and `table` 187 189.

203 ⟨*ldbtable.sml* 203⟩≡

```
structure LdbTable :> LDB_TABLE =
struct
  structure R = LdbMutableFun (
        structure I = LdbInfo
        type rep    = (string * I.t) list ref
        fun makeRep() = ref [] : rep
        fun emitElements   out rep =
          app (fn (k,v) => ( I.extern v           out
                           ; I.extern (ldbName k) out
                           )) (rev(!rep))
        fun emitDefinition out rep =
          app (fn (k,v) => ( CS.emitName (ER.toStr (I.ref' (ldbName k))) out
                           ; CS.emitName (ER.toStr (I.ref' v))           out
                           )) (rev(!rep))
        fun size rep = List.length (!rep)
  )
  structure I = R.I
  type rep    = R.rep
  type t      = R.t
  val make    = R.make
  val asInfo  = R.asInfo
  fun put t key v =
    R.mutate t (fn rep => rep := (key,v) ::
                                 (List.filter (fn (k,_) => k<>key) (!rep)))
  fun get t key =
    R.get t (fn rep => case List.find (fn (k,_) => key=k) (!rep)
                         of SOME (_,v) => v | _ => raise Fail "LdbTable.get")
end
```

Defines:
asInfo, used in chunks 190–93, 211, 215, 222, 228, 234, and 251.
get, used in chunks 251 and 254.
I, used in chunks 196, 200, 202, 228, and 254.
LdbTable, used in chunks 207, 208, 211, 215, 222, 228, 230, 232, 234, and 251.
make, used in chunks 175, 177, 190–93, 206b, 235, 238, and 241.
put, used in chunks 200, 208, 211, 215, 222, 228, and 234.
R, never used.
rep, used in chunks 190–93 and 206a.
t, used in chunks 180, 185, 190–93, 196, 200, 202, 206b, 239, 241, and 254.
Uses CS 182, emitDefinition 198, emitElements 198, emitName 180 182, ER 179, extern 184 186 249 251, find 187 189, LDB_TABLE 202, LdbInfo 186, LdbMutableFun 198, ldbName 192d, makeRep 198, mutate 196 198, ref' 184 186, size 196 198 198 200 201, and toStr 178 179 238.

4.4 `ldb` data

This section includes the `ldb` interface for `ldb` data which might be the source-language dependent but the compiler independent. All the compiler-dependent procedures used in this section are defined in Section 4.2.4.

4.4.1 Source-code locations

- Abstraction : a source-code location
- Constructors:
  - `make` : create a source-code location
- Observers :
  - `file` : return this source-code location's file name
  - `line` : return this source-code location's line number
  - `column` : return this source-code location's column number

204 ⟨*ldbsrcloc.sig* 204⟩≡

```
signature LDB_SRC_LOC =
sig
  (** abstraction *********************************************************)
  type t

  (** constructors *********************************************************)
  (* create a fresh source-code location
   *
   * precondition : this source-code location does not exist
   * postcondition: this source-code location is created
   *)
  val make : { file : string, line : int, column : int } -> t

  (** observers ************************************************************)
  (* return a given source-code location's file name
   *
   * precondition : true
   *)
  val file   : t -> string

  (* return a given source-code location's line number
   *
   * precondition : true
   *)
  val line   : t -> int
```

```
    (* return a given source-code location's column number
     *
     * precondition : true
     *)
    val column : t -> int
  end
```

Defines:
`column`, used in chunks 208 and 222.
`file`, used in chunks 175, 177, 178, 180, 182, 184, 208, 249, 251, and 254.
`LDB_SRC_LOC`, used in chunks 205, 207, 209, 212, 218, 226, and 245.
`line`, used in chunks 208 and 222.
`make`, used in chunks 175, 177, 190–93, 206b, 235, 238, and 241.
`t`, used in chunks 180, 185, 190–93, 196, 200, 202, 206b, 239, 241, and 254.
Uses `name` 207 208.

205 ⟨*ldbsrcloc.sml* 205⟩≡

```
  structure LdbSrcLoc :> LDB_SRC_LOC =
  struct
    type t = { file : string, line : int, column : int }
    fun make src = src
    fun file   ({file  ,...} : t) = file
    fun line   ({line  ,...} : t) = line
    fun column ({column,...} : t) = column
  end
```

Defines:
`column`, used in chunks 208 and 222.
`file`, used in chunks 175, 177, 178, 180, 182, 184, 208, 249, 251, and 254.
`LdbSrcLoc`, used in chunks 207–9, 211, 212, 215, 218, 222, 226, 228, 245, and 247.
`line`, used in chunks 208 and 222.
`make`, used in chunks 175, 177, 190–93, 206b, 235, 238, and 241.
`t`, used in chunks 180, 185, 190–93, 196, 200, 202, 206b, 239, 241, and 254.
Uses `LDB_SRC_LOC` 204 and `src` 254.

4.4.2 Values of machine-level constants

- Abstraction : a machine-level value which is either an integer or a floating-point value
- Constructors:
  - `ldbValueInt` : create a PostScript code whose evaluation results in an integer value
  - `ldbValueFloat` : create a PostScript code whose evaluation results in a floating-point value
- Observers :
  - `asInfo` : return a given value as an `ldb` information (subsumption)

206a ⟨*ldbvalue.sig* 206a⟩≡

```
datatype t = Int of int | Float of real
signature LDB_VALUE = LDB_IMMUTABLE where type rep = t
```

Defines:
`LDB_VALUE`, used in chunks 206b and 209.
`t`, used in chunks 180, 185, 190–93, 196, 200, 202, 206b, 239, 241, and 254.
Uses `LDB_IMMUTABLE` 187 and `rep` 187 189 196 198 198 201 203.

206b ⟨*ldbvalue.sml* 206b⟩≡

```
structure LdbValue :> LDB_VALUE = LdbImmutableFun ( type t = t
          val asPS = fn v => case v
                               of Int   i => (Int.toString  i) ^ " Immediate"
                                | Float f => (Real.toString f) ^ " Immediate")
fun ldbValueInt   i = LdbValue.make (Int   i)
fun ldbValueFloat f = LdbValue.make (Float f)
```

Defines:
`LdbValue`, used in chunks 209 and 211.
`ldbValueFloat`, never used.
`ldbValueInt`, never used.
Uses `LDB_VALUE` 206a, `LdbImmutableFun` 189, `make` 185 186 187 189 194 198 201 203 204 205 207 208 209 211 212 215 218 222 226 228 229 230 2
and `t` 175 177 178 179 184 186 187 189 194 198 201 203 204 205 206a 207 208 209 211 212 215 218 222 226 228 229 230 232 234 235 237 243 2

4.4.3 Symbols

- Abstraction : a symbol; every symbol is one of a variable, a procedure, a constant, and a type.
- Constructors:
  - `make` : create an `ldb` symbol and initialize its name and source location
- Observers :
  - `asTable` : return a given symbol as a PostScript dictionary (subsumption)
  - `name` : return this symbol's name

207 ⟨*ldbsymbol.sig* 207⟩≡

```
signature LDB_SYMBOL =
sig
  (** abstraction ************************************************************)
  structure T : LDB_TABLE
  structure S : LDB_SRC_LOC
  (* the base type for all the symbols *)
  type t

  (** constructors ***********************************************************)
  (* create a fresh symbol
   *
   * precondition : this symbol does not exist
   * postcondition: this symbol is created
   *)
  val make : { name : string, kind : string, src : S.t } -> t

  (** observers **************************************************************)
  (* return a given symbol as a PostScript dictionary (subsumption)
   *
   * precondition : true
   *)
  val asTable : t -> T.t

  (* return a given symbol's name
   *
   * precondition : true
   *)
  val name  : t -> string
end where type T.t = LdbTable.t
    where type S.t = LdbSrcLoc.t
```

Defines:
asTable, used in chunks 211, 215, 222, 228, 230, and 251.
LDB_SYMBOL, used in chunks 208, 209, 212, 218, 226, 229, and 249.
make, used in chunks 175, 177, 190–93, 206b, 235, 238, and 241.
name, used in chunks 178–80, 182, 204, 209, 211, 212, 215, 218, 222, 226, 228, 230, 234, 239, 243–45, 247, 249, 251, and 254.
S, used in chunk 254.
T, never used.
t, used in chunks 180, 185, 190–93, 196, 200, 202, 206b, 239, 241, and 254.
Uses LDB_SRC_LOC 204, LDB_TABLE 202, LdbSrcLoc 205, LdbTable 203, and src 254.

208 ⟨*ldbsymbol.sml* 208⟩≡

```
structure LdbSymbol :> LDB_SYMBOL =
struct
  structure T = LdbTable
  structure S = LdbSrcLoc
  type t      = T.t * { name  : unit -> string }

  fun make { name, kind, src } =
    (* every symbol's external name starts with "S" *)
    let val sym      = T.make {basename="S"}
        fun myname() = name
    in  ( T.put sym "name"       (ldbStr name)
        ; T.put sym "sourcefile" (ldbStr (S.file   src))
        ; T.put sym "sourcey"    (ldbInt (S.line   src))
        ; T.put sym "sourcex"    (ldbInt (S.column src))
        ; T.put sym "kind"       (ldbStr kind)
        ; (sym, { name  = myname })
        )
    end

  fun asTable ((mi, _) : t) = mi
  fun name    ((_,  s) : t) = #name      s ()
end
```

Defines:
asTable, used in chunks 211, 215, 222, 228, 230, and 251.
LdbSymbol, used in chunks 209, 211, 212, 215, 218, 222, 226, 228–30, and 251.
make, used in chunks 175, 177, 190–93, 206b, 235, 238, and 241.
name, used in chunks 178–80, 182, 204, 209, 211, 212, 215, 218, 222, 226, 228, 230, 234, 239, 243–45, 247, 249, 251, and 254.
S, used in chunk 254.
T, never used.
t, used in chunks 180, 185, 190–93, 196, 200, 202, 206b, 239, 241, and 254.
Uses column 204 205 245 247, file 204 205 218 222 245 247, LDB_SYMBOL 207, ldbInt 191b, LdbSrcLoc 205, ldbStr 192b, LdbTable 203, line 204 205 245 247, put 202 203, and src 254.

Constants

- Abstraction : a constant symbol
- Constructors:
  - `make` : create a constant symbol
  - `putType` : initialize this constant's type
  - `putValue` : initialize this constant's value
- Observers :
  - `asSymbol` : return a given constant as a symbol (subsumption)

When a constant is created, its name and source location are given but its type is usually not yet provided. Thus, a constant's type is initialized when the type is avaiable by `putType`. A constant's value is either an integer or a floating-point value and this information is initialized by `putValue`.

209 ⟨*ldbconstant.sig* 209⟩≡

```
signature LDB_CONSTANT =
sig
  (** abstraction ***********************************************************)
  structure V  : LDB_VALUE
  structure S  : LDB_SYMBOL
  structure SL : LDB_SRC_LOC
  structure TY : LDB_TYPE
  (* constant symbol *)
  type t

  (** constructors **********************************************************)
  (* create a fresh constant
   *
   * precondition : this constant does not exist
   * postcondition: this constant is created
   *)
  val make : { name : string, src : SL.t } -> t

  (* initialize a given constant's type
   *
   * precondition : after type checking
   *)
  val putType : t -> TY.t -> unit

  (* initialize a given constant's value
   *
   * precondition : after lexing
   *)
```

```
    val putValue  : t -> V.t -> unit

    (** observers ************************************************************)
    (* return a given constant as a symbol (subsumption)
     *
     * precondition : true
     *)
    val asSymbol  : t -> S.t
  end where type V.t  = LdbValue.t
      where type S.t  = LdbSymbol.t
      where type SL.t = LdbSrcLoc.t
      where type TY.t = LdbType.t
```

Defines:
asSymbol, used in chunk 251.
LDB_CONSTANT, used in chunk 211.
make, used in chunks 175, 177, 190–93, 206b, 235, 238, and 241.
putType, never used.
putValue, never used.
S, used in chunk 254.
SL, never used.
t, used in chunks 180, 185, 190–93, 196, 200, 202, 206b, 239, 241, and 254.
TY, never used.
V, never used.
Uses LDB_SRC_LOC 204, LDB_SYMBOL 207, LDB_TYPE 232, LDB_VALUE 206a, LdbSrcLoc 205, LdbSymbol 208, LdbType 234, LdbValue 206b, name 207 208, and src 254.

211 ⟨*ldbconstant.sml* 211⟩≡

```
structure LdbConstant :> LDB_CONSTANT =
struct
  structure V  = LdbValue
  structure S  = LdbSymbol
  structure SL = LdbSrcLoc
  structure T  = LdbTable
  structure TY = LdbType
  type t = S.t * { putType  : TY.t -> unit
                 , putValue : V.t  -> unit
                 }

  fun make { name, src } =
    let val c = S.make { name = name, kind = "constant", src  = src }
        fun putType  t = T.put (S.asTable c) "type"  (T.asInfo(TY.asTable t))
        fun putValue v = T.put (S.asTable c) "value" (V.asInfo v)
    in  (c, {putType = putType, putValue = putValue})
    end

  fun putType  ((_, c) : t) = #putType  c
  fun putValue ((_, c) : t) = #putValue c
  fun asSymbol ((s, _) : t) = s
end
```

Defines:
`asSymbol`, used in chunk 251.
`LdbConstant`, never used.
`make`, used in chunks 175, 177, 190–93, 206b, 235, 238, and 241.
`putType`, never used.
`putValue`, never used.
`S`, used in chunk 254.
`SL`, never used.
`T`, never used.
`t`, used in chunks 180, 185, 190–93, 196, 200, 202, 206b, 239, 241, and 254.
`TY`, never used.
`V`, never used.
Uses `asInfo` 187 189 194 198 201 203 229 230 235 237, `asTable` 207 208 232 234, `LDB_CONSTANT` 209, `LdbSrcLoc` 205, `LdbSymbol` 208, `LdbTable` 203, `LdbType` 234, `LdbValue` 206b, `name` 207 208, `put` 202 203, and `src` 254.

Variables

- Abstraction : a variable symbol
- Constructors:
  - `make` : create a variable symbol
  - `putType` : initialize this variable's type
  - `putWhere` : initialize this variable's placement
  - `putLabel` : set this variable's assembly-language label
- Observers :
  - `asSymbol` : return a given variable as a symbol (subsumption)
  - `label` : return this variable's assembly-language label

When a variable is created, its name and source location are given but its type is usually not yet provided. Thus, a variable's type is initialized when the type is avaiable by `putType`. A variable's placement is either in a stack frame or in a register and this information is initialized by `putWhere` after the register allocation phase. A variable's assembly-language label is used to refer the variable in the executable program. `putLabel` records the variable's label when it's available and `label` returns the label. The naming convention for labels is compiler dependent and labels are described in section 4.4.7.

212 ⟨*ldbvar.sig* 212⟩≡

```
signature LDB_VAR =
sig
  (** abstraction *********************************************************)
  structure S  : LDB_SYMBOL
  structure P  : LDB_PLCMT
  structure L  : LDB_LABEL
  structure SL : LDB_SRC_LOC
  structure TY : LDB_TYPE
  (* variable symbol *)
  type t

  (** constructors ********************************************************)
  (* create a fresh variable
   *
   * precondition : this variable does not exist
   * postcondition: this variable is created
   *)
  val make : { name : string, src : SL.t } -> t

  (* initialize a given variable's type
   *
```

```
     * precondition : after type checking
     *)
    val putType : t -> TY.t -> unit

    (* initialize a given variable's placement
     *
     * precondition : after register allocation
     *)
    val putWhere  : t -> P.t -> unit

    (* set a given variable's assembly-language label
     *
     * precondition : after the variable's label is known
     *)
    val putLabel   : t -> L.t -> unit

    (** observers ***********************************************************)
    (* return a given variable as a symbol (subsumption)
     *
     * precondition : true
     *)
    val asSymbol  : t -> S.t

    (* returns a given variable's assembly-language label
     *
     * precondition : true
     *)
    val label     : t -> L.t
  end where type S.t  = LdbSymbol.t
      where type P.t  = LdbPlcmt.t
      where type L.t  = LdbLabel.t
      where type SL.t = LdbSrcLoc.t
      where type TY.t = LdbType.t
```

Defines:
asSymbol, used in chunk 251.
L, never used.
label, used in chunks 180, 182, 218, 222, 241, 243, and 244.
LDB_VAR, used in chunk 215.
make, used in chunks 175, 177, 190–93, 206b, 235, 238, and 241.
P, used in chunk 254.
putLabel, never used.
putType, never used.
putWhere, never used.
S, used in chunk 254.
SL, never used.
t, used in chunks 180, 185, 190–93, 196, 200, 202, 206b, 239, 241, and 254.
TY, never used.
Uses LDB_LABEL 243, LDB_PLCMT 235, LDB_SRC_LOC 204, LDB_SYMBOL 207, LDB_TYPE 232, LdbLabel 244, LdbPlcmt 237, LdbSrcLoc 205, LdbSymbol 208, LdbType 234, name 207 208,

and `src` 254.

215 ⟨*ldbvar.sml* 215⟩≡

```
structure LdbVar :> LDB_VAR =
struct
  structure S  = LdbSymbol
  structure P  = LdbPlcmt
  structure L  = LdbLabel
  structure SL = LdbSrcLoc
  structure T  = LdbTable
  structure TY = LdbType
  type t       = S.t * { putType  : TY.t -> unit
                       , putWhere : P.t  -> unit
                       , putLabel : L.t  -> unit
                       , label    : unit -> L.t
                       }

  fun make { name, src } =
    let val var = S.make { name = name, kind = "variable", src = src }
        val label = ref (L.inDataSpace "")
        fun putType  t = T.put (S.asTable var) "type"  (T.asInfo(TY.asTable t))
        fun putWhere p = T.put (S.asTable var) "where" (P.asInfo p)
        fun putLabel l = label := l
        fun mylabel () = !label
    in  (var, { putType = putType, putWhere = putWhere, putLabel = putLabel
              , label = mylabel})
    end

  fun putType   ((_, v) : t) = #putType   v
  fun putWhere  ((_, v) : t) = #putWhere  v
  fun putLabel  ((_, v) : t) = #putLabel  v
  fun asSymbol  ((s, _) : t) = s
  fun label     ((_, v) : t) = #label     v ()
end
```

Defines:
`asSymbol`, used in chunk 251.
`L`, never used.
`label`, used in chunks 180, 182, 218, 222, 241, 243, and 244.
`LdbVar`, never used.
`make`, used in chunks 175, 177, 190–93, 206b, 235, 238, and 241.
`P`, used in chunk 254.
`putLabel`, never used.
`putType`, never used.
`putWhere`, never used.
`S`, used in chunk 254.
`SL`, never used.
`T`, never used.
`t`, used in chunks 180, 185, 190–93, 196, 200, 202, 206b, 239, 241, and 254.
`TY`, never used.
Uses `asInfo` 187 189 194 198 201 203 229 230 235 237, `asTable` 207 208 232 234, `inDataSpace` 243 244, `LDB_VAR` 212, `LdbLabel` 244, `LdbPlcmt` 237, `LdbSrcLoc` 205,

Procedures

- Abstraction : a procedure symbol
- Constructors:
  - `make` : create a procedure symbol
  - `putType` : initialize this procedure's type
  - `putFormal` : set this procedure's last formal parameter, if any
  - `putArgEnv` : set this procedure's arguments environment
  - `putReturnType` : set this procedure's return type
  - `putFrameSize` : set this procedure's frame size
  - `putRegInfo` : set this procedure's callee-saved registers information
  - `putLabel` : set this procedure's assembly-language label
  - `addLocus` : add a stopping point in this procedure
  - `addLocal` : add a local symbol in this procedure
- Observers :
  - `file` : return this procedure's file name
  - `plcmt` : return this procedure's placement
  - `locals` : return this procedure's local symbols
  - `asSymbol` : return a given procedure as a symbol (subsumption)

When a procedure is created, its name and source location are given but its type is usually not yet provided. Thus, a procedure's type is initialized when the type is avaiable by `putType`. `putFormal` sets a procedure's the last formal parameter, if any and `putArgEnv` sets the procedure's arguments environment during the semantic analysis phase and `putReturnType` sets the procedure's return type after the type-checking phase. `putFrameSize` sets a procedure's stack frame size and `putRegInfo` sets a procedure's callee-saved registers information. Callee-saved registers information is given as a list of tuples `(mask, base)` where `mask` is the register-save mask in a decreasing order and `base` is the distance of least significant register from the heavy pointer.

`ldb` assumes that each stack frame can be identified by a single pointer, called the *heavy pointer*, the value of which remains unchanged for the lifetime of the frame. Since the frame and stack pointers are used in different ways on different architectures, `ldb` uses the term "heavy" pointer, not frame or stack pointer. The heavy pointer is realized as either the frame pointer or the stack pointer.

A procedure's assembly-language label is used to refer the procedure in the executable program. `putLabel` records the procedure's label when it's available. The naming convention for labels is compiler dependent and labels are described in section 4.4.7. A procedure records all the stopping points in it and

each stopping point is added to the procedure by `addLocus` when the stopping point is created. Compilers usually have several intermediate representations for a program and in which representation to put stopping points are compiler dependent. Stopping points are described in section 4.4.8. All the local symbols in a procedure is recorded by `addLocal`. `file` returns the procedure's file name to make a source map for the current compilation unit and `plcmt` and `locals` return the procedure's placement and local symbols, respectively.

218 ⟨*ldbproc.sig* 218⟩≡

```
signature LDB_PROC =
sig
  (** abstraction ************************************************************)
  structure I  : LDB_INFO
  structure S  : LDB_SYMBOL
  structure L  : LDB_LABEL
  structure E  : LDB_ENV
  structure P  : LDB_PLCMT
  structure SP : LDB_LOCUS
  structure SL : LDB_SRC_LOC
  structure TY : LDB_TYPE
  (* procedure symbol *)
  type t

  (** constructors ***********************************************************)
  (* create a fresh procedure
   *
   * precondition : this procedure does not exist
   * postcondition: this procedure is created
   *)
  val make : { name : string, src : SL.t } -> t

  (* initialize a given procedure's type
   *
   * precondition : after type checking
   *)
  val putType     : t -> TY.t -> unit

  (* initialize a given procedure's the last formal parameter
   *
   * precondition : during semantic analysis
   *)
  val putFormal : t -> S.t option -> unit

  (* initialize a given procedure's arguments environment
   *
   * precondition : during semantic analysis
```

```
 *)
val putArgEnv     : t -> E.t -> unit

(* initialize a given procedure's return type
 *
 * precondition : after type checking
 *)
val putReturnType : t -> string -> unit

(* set a given procedure's frame size
 *
 * precondition : after register allocation
 *)
val putFrameSize  : t -> int -> unit

(* set a given procedure's callee-saved registers information
 *
 * precondition : after register allocation
 *)
val putRegInfo    : t -> (int * int) list -> unit

(* set a given procedure's assembly-language label
 *
 * precondition : after the procedure's label is known
 *)
val putLabel      : t -> LdbLabel.t -> unit

(* add a specified stopping point to a given procedure's stopping points
 *
 * precondition : after a given stopping point is created
 *)
val addLocus      : t -> SP.t -> unit

(* add a local symbol in this procedure
 *
 * precondition : symbol is not   in this procedure
 * postcondition: symbol is added to this procedure
 *)
val addLocal      : t -> S.t -> unit

(** observers *******************************************************)
(* returns a given procedure's file name
 *
 * precondition : true
 *)
val file     : t -> string
```

```
    (* returns a given procedure's placement
     *
     * precondition : true
     *)
    val plcmt    : t -> P.t

    (* returns a given procedure's local symbols
     *
     * precondition : true
     *)
    val locals   : t -> S.t list

    (* return a given procedure as a symbol (subsumption)
     *
     * precondition : true
     *)
    val asSymbol : t -> S.t
  end where type I.t  = LdbInfo.t
      where type S.t  = LdbSymbol.t
      where type L.t  = LdbLabel.t
      where type E.t  = LdbEnv.t
      where type P.t  = LdbPlcmt.t
      where type SP.t = LdbLocus.t
      where type SL.t = LdbSrcLoc.t
      where type TY.t = LdbType.t
```

Defines:
addLocal, never used.
addLocus, never used.
asSymbol, used in chunk 251.
E, used in chunk 230.
file, used in chunks 175, 177, 178, 180, 182, 184, 208, 249, 251, and 254.
I, used in chunks 196, 200, 202, 228, and 254.
L, never used.
LDB_PROC, used in chunks 222 and 249.
locals, used in chunk 251.
make, used in chunks 175, 177, 190–93, 206b, 235, 238, and 241.
P, used in chunk 254.
plcmt, used in chunk 241.
putArgEnv, never used.
putFormal, never used.
putFrameSize, never used.
putLabel, never used.
putRegInfo, never used.
putReturnType, never used.
putType, never used.
S, used in chunk 254.
SL, never used.
SP, never used.
t, used in chunks 180, 185, 190–93, 196, 200, 202, 206b, 239, 241, and 254.
TY, never used.

Uses add 187 189, label 212 215 245 247, LDB_ENV 229, LDB_INFO 184, LDB_LABEL 243, LDB_LOCUS 245, LDB_PLCMT 235, LDB_SRC_LOC 204, LDB_SYMBOL 207, LDB_TYPE 232, LdbEnv 230, LdbInfo 186, LdbLabel 244, LdbLocus 247, LdbPlcmt 237, LdbSrcLoc 205, LdbSymbol 208, LdbType 234, name 207 208, size 196 198 198 200 201, and src 254.

222 ⟨*ldbproc.sml* 222⟩≡

```
structure LdbProc :> LDB_PROC =
struct
  structure I  = LdbInfo
  structure S  = LdbSymbol
  structure L  = LdbLabel
  structure E  = LdbEnv
  structure P  = LdbPlcmt
  structure SP = LdbLocus
  structure SL = LdbSrcLoc
  structure TY = LdbType
  structure T  = LdbTable
  structure A  = LdbArray
  type t       = S.t * { putType       : TY.t             -> unit
                       , putFormal     : S.t option       -> unit
                       , putArgEnv     : E.t              -> unit
                       , putReturnType : string           -> unit
                       , putFrameSize  : int              -> unit
                       , putRegInfo    : (int * int) list -> unit
                       , putLabel      : L.t              -> unit
                       , addLocus      : SP.t             -> unit
                       , addLocal      : S.t              -> unit
                       , file          : unit             -> string
                       , plcmt         : unit             -> P.t
                       , locals        : unit             -> S.t list
                       }
  val cached_regInfo = ref NONE
  fun make { name, src } =
    let val proc     = S.make { name = name, kind = "procedure", src = src }
        val myloci   = A.make {basename="L"}
        val lociEnvs = A.make {basename="I"}
        val lociLoci = A.make {basename="I"}
        val myplcmt  = ref (NONE : P.t option)
        val mylocals = ref ([]   : S.t list)
        val putP     = T.put (S.asTable proc)
        fun putFormal formal =
          putP "formals" (case formal of NONE   => ldbNull
                                       | SOME s => T.asInfo(S.asTable s))
        fun putType       ty  = putP "type"       (T.asInfo(TY.asTable ty))
        fun putArgEnv     env = putP "argenv"     (E.asInfo env)
        fun putReturnType ty  = putP "returntype" (ldbStr ty)
        fun putFrameSize size = putP "framesize"  (ldbInt size)
        (* the argument is a list of (mask, base)
         * where mask is the register-save mask
         *   and base is the distance of least significant register
         *               from heavy pointer
```

```
 *)
fun makeRegInfo regs =
  let val regInfo = A.make {basename="I"}
  in  ( app (fn (mask, base) => ( A.addHigh regInfo (ldbHex mask)
                                ; A.addHigh regInfo (ldbInt base)
                                )) regs
      ; regInfo
      )
  end
fun sameRegInfo [] [] = true
  | sameRegInfo (x::xs) (y::ys) =
    if x = y then sameRegInfo xs ys else false
  | sameRegInfo _  _  = false
fun putRegInfo regs =
  let val regInfo =
          case !cached_regInfo
            of SOME (lst, arr) =>
               if sameRegInfo regs lst then arr else makeRegInfo regs
             | NONE => makeRegInfo regs
  in  ( cached_regInfo := SOME (regs, regInfo)
      ; putP "reginfo" (A.asInfo regInfo)
      )
  end
fun putLabel label =
  let val plcmt = ldbPlcmtAtLabel label
  in  ( myplcmt := SOME plcmt; putP "where" (P.asInfo plcmt) )
  end

fun addLocus locus =
  let val mylocus    = A.make {basename="I"}
      val locusplcmt = ldbPlcmtAtLabel (SP.label locus)
  in  ( A.addHigh lociEnvs (E.asInfo (SP.env    locus))
      ; A.addHigh mylocus  (ldbStr   (SP.file   locus))
      ; A.addHigh mylocus  (ldbInt   (SP.line   locus))
      ; A.addHigh mylocus  (ldbInt   (SP.column locus))
      ; A.addHigh mylocus  (P.asInfo locusplcmt)
      ; A.addHigh lociLoci (A.asInfo mylocus)
      )
  end
fun addLocal l = LdbUtil.addList l mylocals
in  ( putP      "loci" (A.asInfo myloci)
    ; A.addHigh myloci (A.asInfo lociEnvs)
    ; A.addHigh myloci (A.asInfo lociLoci)
    ; (proc, { putType      = putType     , putFormal     = putFormal
             , putArgEnv    = putArgEnv   , putReturnType = putReturnType
             , putFrameSize = putFrameSize, putRegInfo    = putRegInfo
```

```
                , putLabel     = putLabel    , addLocus      = addLocus
                , addLocal     = addLocal
                , file   = fn _ => SL.file src
                , plcmt  = fn _ => (case !myplcmt of SOME p => p
                                                   | _ => raise Fail "Proc")
                , locals = fn _ => !mylocals
                }))
    end

  fun putType       ((_, p) : t) = #putType       p
  fun putFormal     ((_, p) : t) = #putFormal     p
  fun putArgEnv     ((_, p) : t) = #putArgEnv     p
  fun putReturnType ((_, p) : t) = #putReturnType p
  fun putFrameSize  ((_, p) : t) = #putFrameSize  p
  fun putRegInfo    ((_, p) : t) = #putRegInfo    p
  fun putLabel      ((_, p) : t) = #putLabel      p
  fun addLocus      ((_, p) : t) = #addLocus      p
  fun addLocal      ((_, p) : t) = #addLocal      p
  fun file          ((_, p) : t) = #file          p ()
  fun plcmt         ((_, p) : t) = #plcmt         p ()
  fun locals        ((_, p) : t) = #locals        p ()
  fun asSymbol      ((s, _) : t) = s
end
```

Defines:
A, used in chunk 254.
addLocal, never used.
addLocus, never used.
asSymbol, used in chunk 251.
cached_regInfo, never used.
E, used in chunk 230.
file, used in chunks 175, 177, 178, 180, 182, 184, 208, 249, 251, and 254.
I, used in chunks 196, 200, 202, 228, and 254.
L, never used.
LdbProc, used in chunk 251.
locals, used in chunk 251.
make, used in chunks 175, 177, 190–93, 206b, 235, 238, and 241.
P, used in chunk 254.
plcmt, used in chunk 241.
putArgEnv, never used.
putFormal, never used.
putFrameSize, never used.
putLabel, never used.
putRegInfo, never used.
putReturnType, never used.
putType, never used.
S, used in chunk 254.
SL, never used.
SP, never used.
T, never used.
t, used in chunks 180, 185, 190–93, 196, 200, 202, 206b, 239, 241, and 254.
TY, never used.

Uses addHigh 200 201, addList 173 174, asInfo 187 189 194 198 201 203 229 230 235 237, asTable 207 208 232 234, column 204 205 245 247, env 245 247, label 212 215 245 247, LDB_PROC 218, LdbArray 201, LdbEnv 230, ldbHex 191b, LdbInfo 186, ldbInt 191b, LdbLabel 244, LdbLocus 247, ldbNull 190b, LdbPlcmt 237, ldbPlcmtAtLabel 239 241, LdbSrcLoc 205, ldbStr 192b, LdbSymbol 208, LdbTable 203, LdbType 234, LdbUtil 174, line 204 205 245 247, name 207 208, put 202 203, size 196 198 198 200 201, and src 254.

Type symbols

- Abstraction : a type symbol
- Constructors:
  - `make` : create a type symbol
  - `putType` : initialize this type symbol's type
  - `addVar` : record every variable in this type symbol
  - `addProc` : record every procedure in this type symbol
- Observers :
  - `asSymbol` : return a given type symbol as a symbol (subsumption)

When a type symbol is created, its name and source location are given but its type is usually not yet provided. Thus, a type symbol's type is initialized when the type is avaiable by `putType`.

226 ⟨*ldbtypesymbol.sig* 226⟩≡

```
signature LDB_TYPE_SYMBOL =
sig
  (** abstraction ***********************************************************)
  structure S  : LDB_SYMBOL
  structure SL : LDB_SRC_LOC
  structure TY : LDB_TYPE
  (* type symbol *)
  type t

  (** constructors **********************************************************)
  (* create a fresh type symbol
   *
   * precondition : this type symbol does not exist
   * postcondition: this type symbol is created
   *)
  val make : { name : string, src : SL.t } -> t

  (* initialize a given type symbol's type
   *
   * precondition : after type checking
   *)
  val putType : t -> TY.t -> unit

  (* add a given variable to the set of variables in a given type symbol
   *
   * precondition : true
   *)
  val addVar  : t -> S.t -> unit
```

```
    (* add a given procedure to the set of procedures in a given type symbol
     *
     * precondition : true
     *)
    val addProc : t -> S.t -> unit

    (** observers ***********************************************************)
    (* return a given type symbol as a symbol (subsumption)
     *
     * precondition : true
     *)
    val asSymbol : t -> S.t
  end where type S.t  = LdbSymbol.t
        where type SL.t = LdbSrcLoc.t
        where type TY.t = LdbType.t
```

Defines:
addProc, used in chunk 251.
addVar, never used.
asSymbol, used in chunk 251.
LDB_TYPE_SYMBOL, used in chunk 228.
make, used in chunks 175, 177, 190–93, 206b, 235, 238, and 241.
putType, never used.
S, used in chunk 254.
SL, never used.
t, used in chunks 180, 185, 190–93, 196, 200, 202, 206b, 239, 241, and 254.
TY, never used.
Uses add 187 189, LDB_SRC_LOC 204, LDB_SYMBOL 207, LDB_TYPE 232, LdbSrcLoc 205, LdbSymbol 208, LdbType 234, name 207 208, and src 254.

228 ⟨*ldbtypesymbol.sml* 228⟩≡

```
structure LdbTypeSymbol :> LDB_TYPE_SYMBOL =
struct
  structure S  = LdbSymbol
  structure A  = LdbArray
  structure T  = LdbTable
  structure SL = LdbSrcLoc
  structure TY = LdbType
  type t       = S.t * { putType : TY.t -> unit
                       , addVar  : S.t  -> unit
                       , addProc : S.t  -> unit
                       }

  fun make { name, src } =
    let val ty    = S.make { name = name, kind = "type", src = src }
        val vars  = A.make {basename="I"}
        val procs = A.make {basename="I"}
        fun putType t    = T.put (S.asTable ty) "type" (T.asInfo(TY.asTable t))
        fun addVar  var  = A.addHigh vars  (T.asInfo (S.asTable var))
        fun addProc proc = A.addHigh procs (T.asInfo (S.asTable proc))
    in  ( T.put (S.asTable ty) "vars"  (A.asInfo vars)
        ; T.put (S.asTable ty) "procs" (A.asInfo procs)
        ; (ty, {putType=putType, addVar=addVar, addProc=addProc})
        )
    end

  fun putType  ((_, s) : t) = #putType s
  fun addVar   ((_, s) : t) = #addVar  s
  fun addProc  ((_, s) : t) = #addProc s
  fun asSymbol ((s, _) : t) = s
end
```

Defines:
A, used in chunk 254.
addProc, used in chunk 251.
addVar, never used.
asSymbol, used in chunk 251.
LdbTypeSymbol, never used.
make, used in chunks 175, 177, 190–93, 206b, 235, 238, and 241.
putType, never used.
S, used in chunk 254.
SL, never used.
T, never used.
t, used in chunks 180, 185, 190–93, 196, 200, 202, 206b, 239, 241, and 254.
TY, never used.
Uses addHigh 200 201, asInfo 187 189 194 198 201 203 229 230 235 237,
asTable 207 208 232 234, I 187 189 194 198 198 201 203 218 222 229 230 232 234 235 237 251,
LDB_TYPE_SYMBOL 226, LdbArray 201, LdbSrcLoc 205, LdbSymbol 208, LdbTable 203,
LdbType 234, name 207 208, procs 254, put 202 203, and src 254.

4.4.4 Environments

- Abstraction : an environment
- Constructors:
  - `emptyEnv` : create an empty environment
  - `ldbEnv` : create a new environment

229 ⟨*ldbenv.sig* 229⟩≡

```
signature LDB_ENV =
sig
  (** abstraction *******************************************************)
  structure I : LDB_INFO
  structure S : LDB_SYMBOL
  (* environment *)
  type t

  (** constructors ******************************************************)
  (* create a fresh environment
   *
   * postcondition: this environment is created
   *)
  val make : (t * S.t) option -> t

  (** observers *********************************************************)
  (* return this placement information's definition
   *
   * precondition : this placement's definition is set
   *)
  val getDef : t -> string

  (* return a given environment as an ldb information (subsumption)
   *
   * precondition : true
   *)
  val asInfo : t -> I.t
end where type I.t = LdbInfo.t
    where type S.t = LdbSymbol.t
```

Defines:
`asInfo`, used in chunks 190–93, 211, 215, 222, 228, 234, and 251.
`getDef`, used in chunk 238.
`I`, used in chunks 196, 200, 202, 228, and 254.
`LDB_ENV`, used in chunks 218, 230, and 245.
`make`, used in chunks 175, 177, 190–93, 206b, 235, 238, and 241.
`S`, used in chunk 254.
`t`, used in chunks 180, 185, 190–93, 196, 200, 202, 206b, 239, 241, and 254.
Uses `LDB_INFO` 184, `LDB_SYMBOL` 207, `LdbInfo` 186, and `LdbSymbol` 208.

230 ⟨*ldbenv.sml* 230⟩≡

```
structure LdbEnv :> LDB_ENV =
struct
  structure  I = LdbInfo
  structure  S = LdbSymbol
  type       t = I.t * string
  fun getDef (_, d) = d
  fun asInfo (i, _) = i
  fun make arg =
    let val isExterned = ref false
        val myref = ER.fromBase "E"
        val def   = case arg
                     of NONE            => "NONE"
                      | SOME (env, sym) => (getDef env) ^ ":" ^ (S.name sym)
        fun extern out =
          if (!isExterned) then ()
          else let val toS  = ER.toStr
                   val info = ldbName(toS myref)
               in  case arg
                     of NONE            =>
                        ( I.extern     info                out
                        ; CS.emitLabel (toS myref)         out
                        ; CS.emitName  "0"                 out
                        ; CS.emitName  (toS (I.ref' info)) out
                        ; isExterned := true
                        )
                      | SOME (env, sym) =>
                        let val eAsI = asInfo env
                            val sAsI = LdbTable.asInfo(S.asTable sym)
                        in  ( I.extern     eAsI                out
                            ; I.extern     sAsI                out
                            ; I.extern     info                out
                            ; CS.emitLabel (toS myref)         out
                            ; CS.emitName  "2"                 out
                            ; CS.emitName  (toS(I.ref' info))  out
                            ; FH.push                          out
                            ; CS.emitName  (toS(I.ref' eAsI))  out
                            ; CS.emitName  (toS(I.ref' sAsI))  out
                            ; FH.pop                           out
                            ; isExterned := true
                            )
                        end
               end
    in  case LdbHashTable.find (!ldbHashTable) def
          of SOME info => (info, def)
           | NONE      =>
```

```
                let val info = I.make { extern = extern
                                      , ref'   = (fn _ => myref)
                                      }
                in  ( ldbHashTable := LdbHashTable.add (!ldbHashTable) def info
                    ; (info, def) )
                end
        end
    end
    fun emptyEnv    () = LdbEnv.make NONE
    fun ldbEnv env sym = LdbEnv.make (SOME (env, sym))
```

Defines:
asInfo, used in chunks 190–93, 211, 215, 222, 228, 234, and 251.
emptyEnv, never used.
getDef, used in chunk 238.
I, used in chunks 196, 200, 202, 228, and 254.
LdbEnv, used in chunks 218, 222, 245, and 247.
ldbEnv, never used.
make, used in chunks 175, 177, 190–93, 206b, 235, 238, and 241.
S, used in chunk 254.
t, used in chunks 180, 185, 190–93, 196, 200, 202, 206b, 239, 241, and 254.

Uses add 187 189, asTable 207 208 232 234, CS 182, E 218 222 245 247, emitLabel 180 182, emitName 180 182, env 245 247, ER 179, extern 184 186 249 251, FH 177, find 187 189, fromBase 178 179, LDB_ENV 229, LdbHashTable 189, ldbHashTable 189, LdbInfo 186, ldbName 192d, LdbSymbol 208, LdbTable 203, name 207 208, pop 175 177, push 175 177, ref' 184 186, and toStr 178 179 238.

4.4.5 Types

- Abstraction : a type
- Constructors:
  - `make` : create an `ldb` type
- Observers :
  - `asTable` : return a given type as a PostScript dictionary (subsumption)

When a type is created, how to print a declaration of the type `decl` and how to print a value of the type `printer` are provided. This information is compiler dependent. For example, let's consider the following structure `Fib` and its type `FIB`:

```
/S4 <<>> def
/T5 << /decl       (%s : FIB)
       /printer    {STRUCT}
       /symbol     S4          >> def
%% Backpatches
S4 /name       (Fib)     put
S4 /sourcefile (fib.sml) put
S4 /sourcey    1         put
S4 /sourcex    6         put
S4 /kind       (type)    put
S4 /type       T5        put
S4 /vars       I6        put
S4 /procs      I7        put
```

In order to print the structure `Fib` denoted by `S4` here, the printing method `STRUCT` in `FIB` which is `T5` needs to know which variables and functions are in `S4`. Thus, the `symbol` key in `T5` is bound to `S4` and now `STRUCT` can access the variables and functions in `S4` through the `symbol` key.

232 ⟨*ldbtype.sig* 232⟩≡

```
signature LDB_TYPE =
sig
  (** abstraction ***********************************************************)
  structure I : LDB_INFO
  structure P : LDB_PS_PROC
  structure T : LDB_TABLE
  (* ldb type *)
  type t

  (** constructors **********************************************************)
  (* create a fresh type
```

```
     *
     * precondition : this type does not exist
     * postcondition: this type is created
     *)
    val make : { decl    : string
               , printer : P.t    } -> t

    (** observers ***********************************************************)
    (* return a given type as a PostScript dictionary (subsumption)
     *
     * precondition : true
     *)
    val asTable : t -> T.t
  end where type I.t = LdbInfo.t
      where type P.t = LdbPsProc.t
      where type T.t = LdbTable.t
```

Defines:
asTable, used in chunks 211, 215, 222, 228, 230, and 251.
I, used in chunks 196, 200, 202, 228, and 254.
LDB_TYPE, used in chunks 209, 212, 218, 226, and 234.
make, used in chunks 175, 177, 190–93, 206b, 235, 238, and 241.
P, used in chunk 254.
T, never used.
t, used in chunks 180, 185, 190–93, 196, 200, 202, 206b, 239, 241, and 254.
Uses LDB_INFO 184, LDB_PS_PROC 193a, LDB_TABLE 202, LdbInfo 186, LdbPsProc 193b, and LdbTable 203.

234 ⟨*ldbtype.sml* 234⟩≡

```
structure LdbType :> LDB_TYPE =
struct
  structure I = LdbInfo
  structure P = LdbPsProc
  structure T = LdbTable
  type t      = T.t

  fun make { decl, printer } =
    (* every type's external name starts with "T" *)
    let val ty = T.make {basename="T"}
        (* how to print a declartion of this type *)
    in  ( T.put ty "decl"    (ldbStr           decl)
        (* how to print a value      of this type *)
        ; T.put ty "printer" (LdbPsProc.asInfo printer)
        ; ty
        )
    end

  fun asTable t = t
end
```

Defines:
asTable, used in chunks 211, 215, 222, 228, 230, and 251.
I, used in chunks 196, 200, 202, 228, and 254.
LdbType, used in chunks 209, 211, 212, 215, 218, 222, 226, and 228.
make, used in chunks 175, 177, 190–93, 206b, 235, 238, and 241.
P, used in chunk 254.
T, never used.
t, used in chunks 180, 185, 190–93, 196, 200, 202, 206b, 239, 241, and 254.
Uses asInfo 187 189 194 198 201 203 229 230 235 237, LDB_TYPE 232, LdbInfo 186, LdbPsProc 193b, ldbStr 192b, LdbTable 203, name 207 208, and put 202 203.

4.4.6 Placements

- Abstraction : an object location
- Constructors:
  - `ldbPlcmtAbsolute` : create a fundamental kind of location `ldbPlcmtAbsolute(s, o) = s[o]`
  - `ldbPlcmtShifted` : create a location relative to some other location `ldbPlcmtShifted(s[o'], o) = s[o'+o]`
  - `ldbPlcmtIndirect` : create one level of indirection, with displacement `ldbPlcmtIndirect(s, s'[o'], o) = s[s'[o']+o]`
  - `ldbPlcmtAtLabel` : create a placement of a given label
- Observers :
  - `asInfo` : return a given address as an ldb information (subsumption)

235 ⟨*ldbplcmt.sig* 235⟩≡

```
signature LDB_PLCMT =
sig
  (** abstraction *********************************************************)
  structure I : LDB_INFO
  (* ldb placement *)
  type t

  (** constructors ********************************************************)
  (* create a fresh placement
   *
   * postcondition: this placement is created
   *)
   val make : string -> t option -> t * bool

  (** observers ***********************************************************)
  (* return this placement information's definition
   *
   * precondition : this placement's definition is set
   *)
  val getDef : t -> string

  (* return a given immutable information as an ldb information (subsumption)
   *
   * precondition : true
   *)
  val asInfo : t -> I.t
end where type I.t = LdbInfo.t
```

Defines:

asInfo, used in chunks 190–93, 211, 215, 222, 228, 234, and 251.
getDef, used in chunk 238.
I, used in chunks 196, 200, 202, 228, and 254.
LDB_PLCMT, used in chunks 212, 218, and 237.
t, used in chunks 180, 185, 190–93, 196, 200, 202, 206b, 239, 241, and 254.
Uses LDB_INFO 184, LdbInfo 186, and make 185 186 187 189 194 198 201 203 204 205 207 208 209 211 212 215 218 222 226 228 229 230 232 234 2

237 ⟨*ldbplcmt.sml* 237⟩≡

```
structure LdbPlcmt :> LDB_PLCMT =
struct
  structure I = LdbInfo
  type t      = I.t * string
  fun getDef (_, d) = d
  fun asInfo (i, _) = i
  fun make def loc =
    let val isExterned = ref false
        val myref = ER.fromBase "Im"
        fun extern out =
          if (!isExterned) then ()
          else let val defLabel = ER.toStr(ER.fromBase "Ig")
               in  ( case loc of SOME l => I.extern (asInfo(l)) out
                               | NONE => ()
                   ; CS.emitLabel defLabel         out
                   ; CS.emitStr   def              out
                   ; CS.emitLabel (ER.toStr myref) out
                   ; CS.emitName  "0"              out
                   ; CS.emitName  defLabel         out
                   ; isExterned := true
                   )
               end
    in  case LdbHashTable.find (!ldbHashTable) def
          of SOME info => ((info, def), false)
           | NONE      =>
             let val info = I.make { extern = extern
                                   , ref'   = (fn _ => myref)
                                   }
             in  ( ldbHashTable := LdbHashTable.add (!ldbHashTable) def info
                 ; ((info, def), true))
             end
    end
end
```

Defines:
  `asInfo`, used in chunks 190–93, 211, 215, 222, 228, 234, and 251.
  `getDef`, used in chunk 238.
  `I`, used in chunks 196, 200, 202, 228, and 254.
  `LdbPlcmt`, used in chunks 212, 215, 218, 222, 238, 239, and 241.
  `make`, used in chunks 175, 177, 190–93, 206b, 235, 238, and 241.
  `t`, used in chunks 180, 185, 190–93, 196, 200, 202, 206b, 239, 241, and 254.
Uses `add` 187 189, `CS` 182, `emitLabel` 180 182, `emitName` 180 182, `emitStr` 180 182, `ER` 179, `extern` 184 186 249 251, `find` 187 189, `fromBase` 178 179, `LDB_PLCMT` 235, `LdbHashTable` 189, `ldbHashTable` 189, `LdbInfo` 186, `ref'` 184 186, and `toStr` 178 179 238.

238 ⟨*ldbaddress.sml* 238⟩≡

⟨*lazy address management* 241⟩

```
val i2s   = Int.toString
val c2s   = Char.toString
val toStr = LdbPlcmt.getDef
fun ldbPlcmtAbsolute space     offset =
  #1(LdbPlcmt.make (i2s offset ^ " '" ^ c2s space ^ "' Absolute") NONE)
fun ldbPlcmtShifted        loc offset =
  #1(LdbPlcmt.make
     (LdbPlcmt.getDef loc ^ " " ^ i2s offset ^ " Shifted") (SOME loc))
fun ldbPlcmtIndirect space loc offset =
  #1(LdbPlcmt.make
     (LdbPlcmt.getDef loc ^ " " ^ i2s offset ^ " '" ^ c2s space ^ "' Indirect")
     (SOME loc))
open LdbLazy
```

Defines:

`c2s`, never used.
`i2s`, never used.
`ldbPlcmtAbsolute`, never used.
`ldbPlcmtIndirect`, never used.
`ldbPlcmtShifted`, never used.
`toStr`, used in chunks 189, 198, 201, 203, 230, 237, and 251.

Uses `getDef` 229 230 235 237, `LdbLazy` 241, `LdbPlcmt` 237, `make` 185 186 187 189 194 198 201 203 204 205 207 208 209 211 212 215 218 222 226 2
`offset` 83 241, and `space` 243 244.

Lazy address management

- Abstraction : lazy address management using the anchor symbol
- Constructors:
  - `ldbInitialize` : create an anchor symbol
- Observers :
  - `ldbFinalize` : finalize emitting references to locations in assembly-language output
  - `ldbAnchor` : return the name of the anchor symbol

Some addresses are bound by the linker and therefore are not available at compile time. We pass those addresses to the debugger by planting references in the assembly language output. The references are in known locations relative to the anchor symbol, and we refer to them by using the anchor symbol name and the offset of the reference. While we do not need to use this lazy address management any more because we eliminated the use of `.o.st` and `.lt` files, we still use the anchor symbol so that we can use the `ldb` debugger as it is.

239 ⟨*ldbaddress.sig* 239⟩≡

```
signature LDB_LAZY =
sig
  (** constructors ********************************************************)
  (* initialize lazy address management; create anchor symbol
   *
   * precondition : true
   * postcondition: the anchor symbol is created
   *)
  val ldbInitialize      : unit -> unit

  (** observers ************************************************************)
  (* finalize emitting references to locations in assembly-language output
   *
   * precondition : compiler finished emitting assembly-language output
   * postcondition: all the bound labels are emitted
   *)
  val ldbFinalize        : LdbFileHandle.t -> unit

  (* return the name of the anchor symbol
   *
   * precondition : the anchor symbol is created
   *)
  val ldbAnchor : unit -> string

  val ldbPlcmtAtLabel : LdbLabel.t -> LdbPlcmt.t
```

```
end
```

Defines:
  LDB_LAZY, used in chunk 241.
  ldbAnchor, used in chunk 251.
  ldbFinalize, used in chunk 251.
  ldbInitialize, used in chunk 254.
  ldbPlcmtAtLabel, used in chunk 222.

Uses LdbFileHandle 177, LdbLabel 244, LdbPlcmt 237, name 207 208,
  and t 175 177 178 179 184 186 187 189 194 198 201 203 204 205 206a 207 208 209 211 212 215 218 222 226 228 229 230 232 234 235 237 243 2

241 ⟨*lazy address management* 241⟩≡ (238)

```
structure LdbLazy :> LDB_LAZY =
struct
  val i2s = Int.toString
  (* Declare secofday/usecofday to be implemented
   * by calling the C functions. *)
  val  secofday = _import "secofday": unit -> int;
  val usecofday = _import "usecofday": unit -> int;
  (* for SML/NJ
  val gettimeofday: unit -> (int * int) =
      Unsafe.CInterface.c_function "SMLNJ-Time" "timeofday"
  *)
  val myanchor = ref ""
  (* the offset from the anchor symbol of the next reference to be planted *)
  val offset        = ref 0
  (* all the lazy placements; (label, offset) list
   *)
  val label2offset = ref ([]: (LdbLabel.t * int) list)

  fun ldbAnchor() = !myanchor
  fun ldbInitialize() =
    (* time/version stamp for this compilation *)
    myanchor := ("_stanchor__V" ^ i2s(secofday()) ^ "_" ^ i2s(usecofday()))

  (* plant a reference to each of the labels relative to the anchor symbol *)
  fun ldbFinalize out =
    let val labelArray = Array.array(!offset, LdbLabel.inDataSpace "")
    in  ( CS.exportName (!myanchor) out (* exports the anchor symbol *)
        ; app (fn (l,i) => Array.update(labelArray, i, l)) (!label2offset)
        ; Array.app (fn l => LdbLabel.bindHere out l) labelArray
        ; CS.emitSectionClose out
        )
    end
  (* return the placement of a specified label, create if not exist *)
  fun ldbPlcmtAtLabel label =
    let val (plcmt, new) =
            LdbPlcmt.make ( "{(" ^ !myanchor ^ ") " ^ i2s(!offset)
                          ^ " Lazy" ^ LdbLabel.space label ) NONE
    in  if new then ( LdbUtil.addList (label, !offset) label2offset
                    ; LdbUtil.inc offset; plcmt )
                else plcmt
    end
end
```

Defines:
`gettimeofday`, never used.
`i2s`, never used.

label2offset, never used.
ldbAnchor, used in chunk 251.
ldbFinalize, used in chunk 251.
ldbInitialize, used in chunk 254.
LdbLazy, used in chunk 238.
ldbPlcmtAtLabel, used in chunk 222.
myanchor, never used.
offset, used in chunks 79, 80, and 238.
secofday, never used.
usecofday, never used.
Uses addList 173 174, bindHere 243 244, CS 182, emitSectionClose 180 182,
exportName 180 182, inc 173 174, inDataSpace 243 244, label 212 215 245 247,
LDB_LAZY 239, LdbLabel 244, LdbPlcmt 237, LdbUtil 174, make 185 186 187 189 194 198 201 203 204 205 207 208 209 211 212 215 218 222 22
plcmt 218 222, space 243 244, and t 175 177 178 179 184 186 187 189 194 198 201 203 204 205 206a 207 208 209 211 212 215 218 222 226 228

4.4.7 Labels

- Abstraction : a reference to a location in an assembly output (`.s`)
- Constructors:
  - `inCodeSpace` : return a fresh label with a given name in a code space
  - `inDataSpace` : return a fresh label with a given name in a data space
- Observers :
  - `name` : return this label's name
  - `space` : return a space where this label is bound to
  - `bindHere` : bind this label to a location in an assembly output
  - `eq` : return whether given two labels are the same

243 ⟨*ldblabel.sig* 243⟩≡

```
signature LDB_LABEL =
sig
  (** abstraction ***********************************************************)
  (* a reference to a location in .s *)
  type t

  (** constructors **********************************************************)
  (* return a fresh label with a given name in a code space
   *
   * postcondition: label is unbound
   *)
  val inCodeSpace : string -> t

  (* return a fresh label with a given name is a data space
   *
   * postcondition: label is unbound
   *)
  val inDataSpace : string -> t

  (** observers *************************************************************)
  (* return the space name where a given label is bound to
   *
   * precondition : a given label's space name is set when the label is created
   *)
  val space : t -> string

  (* bind a reference to a location in .s
   *
   * precondition : label is unbound
   * postcondition: label is bound and emitted to .s
```

```
     *)
    val bindHere : LdbFileHandle.t -> t -> unit

    (* return whether given two labels are the same
     *
     * precondition : true
     *)
    val eq : t -> t -> bool
  end
```

Defines:
bindHere, used in chunk 241.
eq, used in chunk 254.
inCodeSpace, never used.
inDataSpace, used in chunks 215 and 241.
LDB_LABEL, used in chunks 212, 218, 244, and 245.
space, used in chunks 238 and 241.
t, used in chunks 180, 185, 190–93, 196, 200, 202, 206b, 239, 241, and 254.
Uses label 212 215 245 247, LdbFileHandle 177, and name 207 208.

244 ⟨*ldblabel.sml* 244⟩≡

```
  structure LdbLabel :> LDB_LABEL =
  struct
    type t = { name : string, space : string }
    fun inCodeSpace name = { name = name, space = "Code" }
    fun inDataSpace name = { name = name, space = "Data" }
    fun space        (label : t) = #space label
    fun bindHere out (label : t) = CS.emitName (#name label) out
    fun eq {name=name1,space=space1} {name=name2,space=space2} =
      name1=name2 andalso space1=space2
  end
```

Defines:
bindHere, used in chunk 241.
eq, used in chunk 254.
inCodeSpace, never used.
inDataSpace, used in chunks 215 and 241.
LdbLabel, used in chunks 212, 215, 218, 222, 239, 241, 245, and 247.
space, used in chunks 238 and 241.
t, used in chunks 180, 185, 190–93, 196, 200, 202, 206b, 239, 241, and 254.
Uses CS 182, emitName 180 182, label 212 215 245 247, LDB_LABEL 243, and name 207 208.

4.4.8 Stopping points

- Abstraction : a stopping point
- Constructors:
  - `make` : create a stopping point
- Observers :
  - `file` : return the file name of this stopping point
  - `line` : return the line number of this stopping point
  - `column` : return the column number of this stopping point
  - `label` : return the label of this stopping point
  - `env` : return the environment at this stopping point

Compilers usually have a set of stopping points where an execution of a given program might be able to stop. For example, MiniJava places a stopping point before each expression, including nested expressions, as well as before each xstatement and at the end of each block.

245 ⟨*ldblocus.sig* 245⟩≡

```
signature LDB_LOCUS =
sig
  (** abstraction ***********************************************************)
  structure E  : LDB_ENV
  structure L  : LDB_LABEL
  structure SL : LDB_SRC_LOC
  (* a stopping point *)
  type t

  (** constructors **********************************************************)
  (* create a fresh stopping point
   *
   * precondition : this stopping point does not exist
   * postcondition: this stopping point is created
   *)
  val make : { src : SL.t, label : L.t, env : E.t } -> t

  (** observers **************************************************************)
  (* return the file name of a given stopping point
   *
   * precondition  : true
   *)
  val file   : t -> string

  (* return the line number of a given stopping point
```

```
     *
     * precondition  : true
     *)
    val line   : t -> int

    (* return the column number of a given stopping point
     *
     * precondition  : true
     *)
    val column : t -> int

    (* return the label of a given stopping point
     *
     * precondition  : true
     *)
    val label  : t -> L.t

    (* return the environment at a given stopping point
     *
     * precondition  : true
     *)
    val env     : t -> E.t
  end where type E.t  = LdbEnv.t
      where type L.t  = LdbLabel.t
      where type SL.t = LdbSrcLoc.t
```

Defines:
column, used in chunks 208 and 222.
E, used in chunk 230.
env, used in chunks 222 and 230.
file, used in chunks 175, 177, 178, 180, 182, 184, 208, 249, 251, and 254.
L, never used.
label, used in chunks 180, 182, 218, 222, 241, 243, and 244.
LDB_LOCUS, used in chunks 218 and 247.
line, used in chunks 208 and 222.
make, used in chunks 175, 177, 190–93, 206b, 235, 238, and 241.
SL, never used.
t, used in chunks 180, 185, 190–93, 196, 200, 202, 206b, 239, 241, and 254.
Uses LDB_ENV 229, LDB_LABEL 243, LDB_SRC_LOC 204, LdbEnv 230, LdbLabel 244, LdbSrcLoc 205, name 207 208, and src 254.

247 ⟨*ldblocus.sml* 247⟩≡

```
structure LdbLocus :> LDB_LOCUS =
struct
  structure E  = LdbEnv
  structure L  = LdbLabel
  structure SL = LdbSrcLoc
  type t = { src : SL.t, label : L.t, env : E.t }
  (* set the name of this stopping point with a new unique name
   * This name has to be usable both as PostScript names and
   * as assembly-language labels.
   *)
  fun make { src, label, env } = { src = src, label = label, env = env }
  fun file   (sp: t) = SL.file   (#src sp)
  fun line   (sp: t) = SL.line   (#src sp)
  fun column (sp: t) = SL.column (#src sp)
  fun label  (sp: t) = #label sp
  fun env    (sp: t) = #env   sp
end
```

Defines:
`column`, used in chunks 208 and 222.
`E`, used in chunk 230.
`env`, used in chunks 222 and 230.
`file`, used in chunks 175, 177, 178, 180, 182, 184, 208, 249, 251, and 254.
`L`, never used.
`label`, used in chunks 180, 182, 218, 222, 241, 243, and 244.
`LdbLocus`, used in chunks 218 and 222.
`line`, used in chunks 208 and 222.
`make`, used in chunks 175, 177, 190–93, 206b, 235, 238, and 241.
`SL`, never used.
`t`, used in chunks 180, 185, 190–93, 196, 200, 202, 206b, 239, 241, and 254.
Uses `LDB_LOCUS` 245, `LdbEnv` 230, `LdbLabel` 244, `LdbSrcLoc` 205, `name` 207 208, and `src` 254.

4.4.9 Compilation units

- Abstraction : a compilation unit
- Constructors:
  - `make` : create a compilation unit
  - `addProc` : add a procedure to this compilation unit
  - `addExported` : add an exported symbol to this compilation unit
  - `addPrivate` : add a private symbol to this compilation unit
- Observers :
  - `extern` : externalize this compilation unit

The `extern` method emits the symbol table information of this compilation unit. There are entries in the `ldb_st` section which point to the top-level symbol table information and the actual data is placed in the `ldb_st_data` section. For example, the following information is emitted to the assembly-language output:

```
.section .ldb_st
.long .Lldb_st_src
.long .Lldb_st_arch
.long .Lldb_st_anchor
.long .Lldb_st_procs
.long .Lldb_st_externs
.long .Lldb_st_privates
.long .Lldb_st_locals
.long .Lldb_st_sourcemap
.section .ldb_st_data
.Lldb_st_src:
.byte 116
.byte 101
.byte 115
.byte 116
.byte 46
.byte 99
.byte 0
.Lldb_st_arch:
...
.Lldb_st_anchor:
...
.Lldb_st_procs:
.long 2
.long .LS23
.long .LS31
.Lldb_st_externs:
```

```
.long 2
.long .LIm161
.long .LS23
.long .LIm163
.long .LS31
.Lldb_st_privates:
.long 2
.long .LIm179
.long .LS16
.long .LIm181
.long .LS38
.Lldb_st_locals:
.long 1
.long .LS10
.Lldb_st_sourcemap:
.long 2
.long .LIm33
.long 1
.long .LS31
.long .LIm12
.long 1
.long .LS23
```

Since the compilation unit is the top-level information, `ref'` is not defined and `make` creates a compilation unit, which is an implicit part of the inteface's state.

249 ⟨*ldbcompunit.sig* 249⟩≡

```
signature LDB_COMP_UNIT =
sig
  (** abstraction ********************************************************)
  structure P : LDB_PROC
  structure S : LDB_SYMBOL
  (* compilation unit *)
  type t

  (* constructors ***************************************************)
  (* create an empty compilation unit
   *
   * precondition : architecture name, source file name, and
   *                a file handle for assembly-language output are given
   *                for example, "mips", "test.sml", and a file handle for
   *                "test.s" are given
   * postcondition: compilation unit has its architecture name,
   *                source file name, file handle for assembly-language
   *                output, and anchor symbol name
   *)
```

```
    val make : { arch : string, src : string, asm : LdbFileHandle.t } -> t

    (* add a procedure symbol to this compilation unit
     *
     * precondition : procedure symbol is not in the compilation unit
     * postcondition: procedure symbol is added to the compilation unit;
     *                this procedure symbol is defined in this compilation unit
     *)
    val addProc : t -> P.t -> unit

    (* add an exported symbol to this compilation unit
     *
     * precondition : exported symbol is not in the compilation unit
     * postcondition: exported symbol is added to the compilation unit;
     *                this symbol is exported to the other compilation units
     *)
    val addExported : t -> S.t -> unit

    (* add a privte symbol to this compilation unit
     *
     * precondition : private symbol is not in the compilation unit
     * postcondition: private symbol is added to the compilation unit;
     *                this symbol is private to this compilation unit
     *)
    val addPrivate : t -> S.t -> unit

    (** observers ***********************************************************)
    (* extern this compilation unit
     *
     * precondition : file handle must be at top level
     * postcondition: file handle is still at top level,
     *                and this compilation unit has been externed
     *)
    val extern : t -> LdbFileHandle.t -> unit
  end
```

Defines:
addExported, used in chunk 251.
addPrivate, used in chunk 251.
addProc, used in chunk 251.
extern, used in chunks 185, 189, 198, 201, 203, 230, and 237.
LDB_COMP_UNIT, used in chunk 251.
make, used in chunks 175, 177, 190–93, 206b, 235, 238, and 241.
P, used in chunk 254.
S, used in chunk 254.
t, used in chunks 180, 185, 190–93, 196, 200, 202, 206b, 239, 241, and 254.
Uses add 187 189, arch 254, asm 254, empty 187 189, exported 96a 254,
file 204 205 218 222 245 247, LDB_PROC 218, LDB_SYMBOL 207, LdbFileHandle 177,
name 207 208, and src 254.

251 ⟨*ldbcompunit.sml* 251⟩≡

```
structure LdbCompUnit : LDB_COMP_UNIT =
struct
  structure P = LdbProc
  structure S = LdbSymbol
  structure A = LdbArray
  structure T = LdbTable
  structure I = LdbInfo
  type t = { addProc     : P.t -> unit
           , addExported : S.t -> unit
           , addPrivate  : S.t -> unit
           , arch        : unit -> string  (* architecture  name *)
           , src         : unit -> string  (* source file   name *)
           , asm         : unit -> FH.t    (* assembly-language output *)
           , procs       : unit -> A.t     (* procedures       in this  CU  *)
           , exported    : unit -> A.t     (* exported symbols to other CUs *)
           , privates    : unit -> A.t     (* private  symbols in this  CU  *)
           , locals      : unit -> A.t     (* local    symbols in this  CU  *)
           (* a mapping between file names and their procedures   *)
           , srcmap      : unit -> (string * A.t) list
           }
  val s2i = T.asInfo o S.asTable
  val p2i = T.asInfo o S.asTable o P.asSymbol
  val a2i = LdbArray.asInfo
  val procsL    = ref ([] : P.t    list)
  val exportedL = ref ([] : string list)
  val privatesL = ref ([] : string list)
  val localsL   = ref ([] : string list)
  ⟨constructors 254⟩
  (* emit the symbol table information *)
  fun extern (cu : t) out =
    let val fhWrite = FH.write out
        val erWrite = ER.write out
        fun dosymbols action syms size =
          let fun dosym i =
                if i < size then (action (A.get syms i); dosym (i+1)) else ()
          in  (dosym 0) handle Fail "Array.get" => ()
          end
        fun dosymbols2 action syms size =
          let fun dosym i =
                if i < size then (action (A.get syms i) i; dosym (i+1)) else ()
          in  (dosym 0) handle Fail "Array.get" => ()
          end
        val anchor        = ldbAnchor()
        val lab_src       = anchor ^ "_src"
        val lab_arch      = anchor ^ "_arch"
```

```
    val lab_anchor    = anchor ^ "_anchor"
    val lab_procs     = anchor ^ "_procs"
    val lab_externs   = anchor ^ "_externs"
    val lab_privates  = anchor ^ "_privates"
    val lab_locals    = anchor ^ "_locals"
    val lab_sourcemap = anchor ^ "_sourcemap"
    fun emitRef  extref = CS.emitName (ER.toStr extref)              out
    fun emitSize ldbarr = CS.emitName (Int.toString(A.size ldbarr))  out
    fun emitStr  str    =
      let val i = ldbStr str
      in  (I.extern i out; CS.emitName (ER.toStr(I.ref' i)) out )
      end
    fun emitRefs ftn symslist symsarr lab_syms =
      ( CS.emitLabel lab_syms out; emitSize symsarr
      ; ftn symsarr (length symslist))
    fun emitRefs1 symsL =
      emitRefs (dosymbols (fn s => emitRef (I.ref' s))) symsL
    fun emitRefs2 symsL =
      emitRefs (dosymbols2(fn s => fn i =>
                           ( emitRef(I.ref'(ldbName(List.nth(symsL,i))))
                           ; emitRef(I.ref' s)))) symsL
in  ( ldbFinalize (asm cu)
    ; CS.emitDataSection out
      (* emit all procedures       *)
    ; dosymbols (fn s => I.extern s out) (procs    cu) (length(!procsL))
      (* emit all exported symbols *)
    ; dosymbols (fn s => I.extern s out) (exported cu) (length(!exportedL))
    ; app       (fn s => I.extern (ldbName s) out) (!exportedL)
      (* emit all private  symbols *)
    ; dosymbols (fn s => I.extern s out) (privates cu) (length(!privatesL))
    ; app       (fn s => I.extern (ldbName s) out) (!privatesL)
      (* emit all local    symbols *)
    ; dosymbols (fn s => I.extern s out) (locals   cu) (length(!localsL))
    ; CS.emitSectionClose              out
    ; CS.emitSectionOpen "ldb_st"      out
    ; CS.emitName        lab_src       out
    ; CS.emitName        lab_arch      out
    ; CS.emitName        lab_anchor    out
    ; CS.emitName        lab_procs     out
    ; CS.emitName        lab_externs   out
    ; CS.emitName        lab_privates  out
    ; CS.emitName        lab_locals    out
    ; CS.emitName        lab_sourcemap out
    ; CS.emitSectionClose              out
    ; CS.emitSectionOpen "ldb_st_data" out
    ; CS.emitLabel       lab_src       out
```

```
          ; CS.emitStr         (src cu)      out
          ; CS.emitLabel       lab_arch      out
          ; CS.emitStr         (arch cu)     out
          ; CS.emitLabel       lab_anchor    out
          ; CS.emitStr             anchor    out
          ; emitRefs1 (!procsL)    (procs    cu) lab_procs
          ; emitRefs2 (!exportedL) (exported cu) lab_externs
          ; emitRefs2 (!privatesL) (privates cu) lab_privates
          ; emitRefs1 (!localsL)   (locals   cu) lab_locals
          ; CS.emitLabel lab_sourcemap                    out
          ; CS.emitName (Int.toString(length (srcmap cu))) out
          ; app (fn (file, ps) => ( emitStr file; emitSize ps
                                  ; dosymbols (fn s => emitRef(I.ref' s))
                                              ps (A.size ps) )) (srcmap cu)
          ; CS.emitSectionClose out
          )
      end
  end
```

Defines:
  A, used in chunk 254.
  a2i, never used.
  exportedL, used in chunk 254.
  extern, used in chunks 185, 189, 198, 201, 203, 230, and 237.
  I, used in chunks 196, 200, 202, 228, and 254.
  LdbCompUnit, used in chunks 94a and 95.
  localsL, used in chunk 254.
  P, used in chunk 254.
  p2i, used in chunk 254.
  privatesL, used in chunk 254.
  procsL, used in chunk 254.
  S, used in chunk 254.
  s2i, used in chunk 254.
  T, never used.
  t, used in chunks 180, 185, 190–93, 196, 200, 202, 206b, 239, 241, and 254.
Uses addExported 249 254, addPrivate 249 254, addProc 226 228 249 254, arch 254, asInfo 187 189 194 198 201 203 229 230 235 237, asm 254, asSymbol 209 211 212 215 218 222 226 228, asTable 207 208 232 234, CS 182, emitDataSection 180 182, emitLabel 180 182, emitName 180 182, emitSectionClose 180 182, emitSectionOpen 180 182, emitStr 180 182, ER 179, exported 96a 254, FH 177, file 204 205 218 222 245 247, get 196 198 200 201 202 203, LDB_COMP_UNIT 249, ldbAnchor 239 241, LdbArray 201, ldbFinalize 239 241, LdbInfo 186, ldbName 192d, LdbProc 222, ldbStr 192b, LdbSymbol 208, LdbTable 203, locals 218 222 254, name 207 208, privates 96a 254, procs 254, ref' 184 186, size 196 198 198 200 201, src 254, srcmap 254, table 187 189, toStr 178 179 238, and write 175 177 178 179.

254 ⟨*constructors* 254⟩≡ (251)

```
(* create an empty compilation unit *)
fun make {arch, src, asm} =
  let val arch     = ref arch
      val src      = ref src
      val asm      = ref asm
      val procs    = ref (A.make {basename="I"})
      val exported = ref (A.make {basename="I"})
      val privates = ref (A.make {basename="I"})
      val locals   = ref (A.make {basename="I"})
      val srcmap   = ref ([] : (string * A.t) list)
      fun addSym arr sym =
        let fun exist i =
              (if (LdbInfo.eq (A.get arr i) sym) then true else exist (i+1))
              handle Fail "Array.get" => false
        in  if (exist 0) then () else A.addHigh arr sym
        end
      fun addLst elmt lst = lst := (!lst)@[elmt]
      fun addProc p =
        let val file = P.file p
            val pi   = p2i p
        in  ( A.addHigh (!procs) pi
            ; addLst p procsL
            ; case List.find (fn (f,_) => f=file) (!srcmap)
                of SOME (_, arr) => addSym arr pi
                 | _ => let val ps = A.make {basename="I"}
                        in  ( A.addHigh ps pi
                            ; addLst (file, ps) srcmap
                            )
                        end
            ; app (fn s => ( addSym (!locals) (s2i s)
                           ; addLst (S.name s) localsL)) (P.locals p)
            )
        end
      fun addExported s =
        (addSym (!exported) (s2i s); addLst (S.name s) exportedL)
      fun addPrivate  s =
        (addSym (!privates) (s2i s); addLst (S.name s) privatesL)
      fun myarch     () = !arch
      fun mysrc      () = !src
      fun myasm      () = !asm
      fun myprocs    () = !procs
      fun myexported () = !exported
      fun myprivates () = !privates
      fun mylocals   () = !locals
      fun mysrcmap   () = !srcmap
```

```
  in  ( ldbInitialize()
      ; { addProc    = addProc   , addExported = addExported
        , addPrivate = addPrivate
        , arch       = myarch    , src         = mysrc
        , asm        = myasm     , procs       = myprocs
        , exported   = myexported, privates    = myprivates
        , locals     = mylocals  , srcmap      = mysrcmap
        })
  end

  (* add a procedure symbol to procs *)
  fun addProc     (cu : t) = #addProc     cu
  (* add an exported symbol to exported *)
  fun addExported (cu : t) = #addExported cu
  (* add a private symbol to privates *)
  fun addPrivate  (cu : t) = #addPrivate  cu

  fun arch     (cu : t) = #arch     cu ()
  fun src      (cu : t) = #src      cu ()
  fun asm      (cu : t) = #asm      cu ()
  fun procs    (cu : t) = #procs    cu ()
  fun exported (cu : t) = #exported cu ()
  fun privates (cu : t) = #privates cu ()
  fun locals   (cu : t) = #locals   cu ()
  fun srcmap   (cu : t) = #srcmap   cu ()
```

Defines:
`addExported`, used in chunk 251.
`addPrivate`, used in chunk 251.
`addProc`, used in chunk 251.
`arch`, used in chunks 249 and 251.
`asm`, used in chunks 249 and 251.
`exported`, used in chunks 91, 93, 95–97, 249, and 251.
`locals`, used in chunk 251.
`make`, used in chunks 175, 177, 190–93, 206b, 235, 238, and 241.
`privates`, used in chunks 91, 95–97, and 251.
`procs`, used in chunks 228 and 251.
`src`, used in chunks 205, 207–9, 211, 212, 215, 218, 222, 226, 228, 245, 247, 249, and 251.
`srcmap`, used in chunk 251.

Uses `A` 222 228 251, `add` 187 189, `addHigh` 200 201, `empty` 187 189, `eq` 185 186 243 244,
`exportedL` 251, `file` 204 205 218 222 245 247, `find` 187 189, `get` 196 198 200 201 202 203,
`I` 187 189 194 198 198 201 203 218 222 229 230 232 234 235 237 251,
`LdbInfo` 186, `ldbInitialize` 239 241, `localsL` 251, `name` 207 208,
`P` 212 215 218 222 232 234 249 251, `p2i` 251, `privatesL` 251, `procsL` 251,
`S` 207 208 209 211 212 215 218 222 226 228 229 230 249 251, `s2i` 251,
and `t` 175 177 178 179 184 186 187 189 194 198 201 203 204 205 206a 207 208 209 211 212 215 218 222 226 228 229 230 232 23

Chapter 5

The `ldb` interface in Objective Caml

5.1 Legend

We use the following naming conventions for the `ldb` interface in Objective Caml:

Token	Naming convention	Example
Types, variables, and functions	Initial lower case. Use embedded caps for multiword names.	`priorityQueue`
Constructors	Initial upper case. Use embedded caps for multiword names.	`PriorityQueue`
Classes	Begin with `ldb`. Initial upper case. Use embedded caps for multiword names.	`ldbPsProc`

Abbreviations are used only for the followings:

`arg`	argument
`comp`	compilation
`env`	environment
`ext`	external
`float`	floating-point numbers
`hex`	hexadecimal
`info`	information
`int`	integer
`loc`	location
`loci`	stopping points
`locus`	a stopping point
`plcmt`	placement
`proc`	procedure
`ps`	PostScript
`ref`	reference
`reg`	register
`src`	source
`str`	string
`var`	variable

5.2 Preliminaries

This section describes the preliminary `ldb` interface which should be the source-language independent, the compiler independent but implementation-language (Objective Caml) dependent.

5.2.1 File handle

- Abstraction : a file handle to control emitting debugging information
- Constructors:
  - `openFile`: open a specified file
  - `stdout`: create a file handle for `stdout`
- Observers :
  - `closeFile`: close the open file
  - `write`: write a given string to the file
  - `push`: increase the current level
  - `pop` : decrease the current level
  - `assertTop`: make sure that the current level is 'top level'

`push`, `pop`, and `assertTop` are provided to maintain the current level of a file handle. So that the `ldb` interface can keep the following invariance:

- When a debugging information is to be emitted, the file handle is at 'top level'.

258 ⟨*ldbfilehandle.mli* 258⟩≡

```
(** abstraction ***********************************************************)
(* a file handle to control emitting debugging information;
 * include an open file stream and its current level
 *)
type t

(** constructors **********************************************************)
(* open a specified file
 *
 * precondition : file is not open
 * postcondition: file is open and its default current level is 0
 *)
val openFile : string -> t

(* create a file handle for stdout
 *
 * precondition : true
 *)
val stdout : unit -> t

(** observers *************************************************************)
(* close the open file
 *
 * precondition : file is open
 * postcondition: file is closed
 *)
val closeFile : t -> unit

(* write a string to the file
 *
 * precondition : file is open
 * postcondition: string is written to the file
 *)
val write : t -> string -> unit

(* increase current level
 *
 * precondition : current level is equal or greater than 0
 * postcondition: current level is increased and greater than 0
 *)
val push : t -> unit
```

```
(* decrease current level
 *
 * precondition : current level is greater than 0
 * postcondition: current level is decreased and equal or greater than 0
 *)
val pop : t -> unit

(* make sure that the current level is top level
 *
 * precondition : current level is equal or greater than 0
 * postcondition: if the current level is not 0, assertion error
 *)
val assertTop : t -> unit
```

5.2.2 External references

- Abstraction : a reference to an item that has been externed
- Constructors:
  - `fromBase` : create an external reference from a base name and a generated number
  - `fromStr`: create an external reference from a string
- Observers :
  - `write`: write this external reference to a given file handle
  - `toStr`: return the string representation of this external reference

External references satisfy the following invariance:

- The external reference of every immutable information is its string representation.
- The external reference of every mutable information is a uniquely generated name.

260 ⟨*ldbextref.mli* 260⟩≡

```
(** abstraction ***********************************************************)
(* a reference to an item that has been externed *)
type t

(** constructors **********************************************************)
(* create an external reference from a base name
 *
 * postcondition: this external reference's string representation is
 *                its generated name which is the given base name
 *                appended by its unique index
 *)
val fromBase : string -> t

(* create an external reference from a string
 *
 * postcondition: this external reference's string representation is
 *                the string surrounded by parentheses
 *)
val fromStr : string -> t

(** observers *************************************************************)
(* write an external reference
 *
 * precondition : a given file handle is open
```

```
 * postcondition: external reference is written
 *)
val write : Ldbfilehandle.t -> t -> unit

(* return an external reference as a string
 *
 * postcondition: true
 *)
val toStr : t -> string
```

5.2.3 Compiler supports

Utility functions from a compiler for the `ldb` interface.

261 ⟨*ldbutil.mli* 261⟩≡

```
(* labels ************************************************************)
(* return a fresh compiler/platform specific label name *)
val makeLabelName         : unit     -> string
(* return a fresh compiler/platform specific label name from a given name *)
val makeLabelNameFrom     : string   -> string
(* return a base name from a given compiler/platform-specific label name *)
val dropPrefixOfLabelName : string -> string
(* return a no-op format to tag a given stopping point's name in the
 * assembly-language output *)
val noOpFormat            : string -> string

(* assembly output ***************************************************)
(* emit a given label name to a given file handle *)
val emitLabel   : string -> Ldbfilehandle.t -> unit
(* emit a given string to a given file handle *)
val emitStr     : string -> Ldbfilehandle.t -> unit
(* emit a given data name to a given file handle *)
val emitName    : string -> Ldbfilehandle.t -> unit
(* export a given anchor symbol name to a given file handle *)
val exportName  : string -> Ldbfilehandle.t -> unit
(* emit a given section name to a given file handle *)
val emitSectionOpen  : string -> Ldbfilehandle.t -> unit
(* emit a section closure    to a given file handle *)
val emitSectionClose :           Ldbfilehandle.t -> unit
(* emit a data section *)
val emitDataSection  :           Ldbfilehandle.t -> unit

(* private data ******************************************************)
(* emit compiler's private data possibly for its expression server *)
val emitPrivate      :           Ldbfilehandle.t -> unit
```

5.3 Generic PostScript data

This section includes the `ldb` interface for generic PostScript data which should be the source-language independent, the compiler independent, but implementation-language (Objective Caml) dependent.

5.3.1 PostScript data (`ldb` information)

- Abstraction : the base class for all the PostScript data
- Constructors:
  - `setExtRef`: set this information's external reference
- Observers :
  - `extern`: externalize this information
  - `ref`: return this information's external reference

An `ldb` information satisfies the following invariance:

* `extern` is called after the information has been created.

* `ref` is called after the information has been externalized.

262 ⟨*ldbinfo.mli* 262⟩≡
```
⟨ldbInfoType 263⟩
class ldbInfo : ldbInfoType
```

263 ⟨*ldbInfoType* 263⟩≡ (262)

```
class type ldbInfoType =
object
  (** constructors ************************************************************)
  (* set this information's external reference
   *
   * precondition : this information does not have its external reference
   * postcondition: this information's external reference is set
   *)
  method setExtRef : Ldbextref.t -> unit

  (** observers ****************************************************************)
  (* extern a specified information
   *
   * precondition : file handle must be at top level
   * postcondition: file handle is still at top level,
   *                and information has been externed
   *)
  method extern : Ldbfilehandle.t -> unit

  (* return a specified information's external reference
   *
   * precondition : a given information has already been externed
   * postcondition: this external reference can be written anywhere
   *                in a file handle
   *)
  method ref : Ldbextref.t
end
```

5.3.2 Immutable PostScript data

- Abstraction : an immutable PostScript data
- Constructors:
  - `setDef`: set this immutable information's definition
- Observers :
  - `getDef`: return this immutable information's definition

Each immutable information is externalized by `extern` as follows. If it's not yet externalized, its label is emitted followed by `0` to denote that this information is immutable and then the label for the string representation of the immutable information is emitted. The string representation and its label are emitted before the immutable information is emitted.

For example, `(test.ml)` is externalized as follows:

```
.LIg71:
.byte 40
.byte 116
.byte 101
.byte 115
.byte 116
.byte 46
.byte 109
.byte 108
.byte 41
.byte 0
.LIm12:
.long 0
.long .LIg71
```

Its label `.LIm12` is emitted followed by `0` and `.LIg71` which represents the string `(test.ml)`.

264 ⟨*ldbimmutable.mli* 264⟩≡
⟨*ldbImmutableType* 265a⟩
```
class ldbImmutable : ldbImmutableType
val hashT : (string, Ldbinfo.ldbInfo) Hashtbl.t
```

265a ⟨*ldbImmutableType* 265a⟩≡ (264)

```
class type ldbImmutableType =
object
  inherit Ldbinfo.ldbInfo
  (** constructors ********************************************************)
  (* set this immutable information's definition
   *
   * precondition : this immutable information does not have its definition
   * postcondition: this immutable information's definition is set
   *)
  method setDef : string -> unit

  (** observers ************************************************************)
  (* return this immutable information's definition
   *
   * precondition : this immutable information's definition is set
   *)
  method getDef : string
end
```

5.3.3 PostScript null objects

Since `ldbNull` extends the `ldbImmutable` class, it inherits the costructors and observers of `ldbImmutable`. We don't explain the inherited methods.

- Abstraction : a PostScript null
- Constructors:
  - `newNull` : create a null object

265b ⟨*ldbnull.mli* 265b⟩≡

```
val newNull : Ldbinfo.ldbInfo
```

5.3.4 PostScript integers

`ldbInt` extends the `ldbImmutable` class.

- Abstraction : a PostScript integer
- Constructors:
  - `newInt` : create a PostScript decimal integer
  - `newHex` : create a PostScript hexadecimal integer

265c ⟨*ldbint.mli* 265c⟩≡

```
val newInt : int -> Ldbinfo.ldbInfo
val newHex : int -> Ldbinfo.ldbInfo
```

5.3.5 PostScript strings

`ldbStr` extends the `ldbImmutable` class.

- Abstraction : a PostScript string
- Constructors:
  - `newStr` : create a PostScript string

266a ⟨*ldbstr.mli* 266a⟩≡
```
val newStr : string -> Ldbinfo.ldbInfo
```

5.3.6 PostScript names

`ldbName` extends the `ldbImmutable` class.

- Abstraction : a PostScript name
- Constructors:
  - `newName` : create a PostScript name

266b ⟨*ldbname.mli* 266b⟩≡
```
val newName : string -> Ldbinfo.ldbInfo
```

5.3.7 PostScript procedures

`ldbPsProc` extends the `ldbImmutable` class.

- Abstraction : a PostScript procedure
- Constructors:
  - `newPsProc` : create a PostScript procedure

266c ⟨*ldbpsproc.mli* 266c⟩≡
```
class ldbPsProc : string -> Ldbimmutable.ldbImmutable
val   newPsProc : string -> ldbPsProc
```

5.3.8 PostScript codes

`ldbPsCode` extends the `ldbImmutable` class.

- Abstraction : a PostScript code
- Constructors:
  - `newPsCode` : create a PostScript code

266d ⟨*ldbpscode.mli* 266d⟩≡
```
val newPsCode : string -> Ldbinfo.ldbInfo
```

5.3.9 Mutable PostScript data

- Abstraction : the virtual base class for all mutable PostScript data
- Constructors:
  - `setExtBase` : set the base name for this mutable information's external reference
- Observers :
  - `getExtBase` : get the base name for this mutable information's external reference
  - `size` : get the size of this mutable information
  - `emitElements` : emit all the elements in this mutable information
  - `emitDefinition` : emit the definition of this mutable information
  - `mutate` : assert that this mutable information has not yet been externed

A mutable information satisfies the following invariance:

* `extern` is called in any states.

* `ref` is called only in the `EMITTED` state.

* `mutate` is called in the `NEW` state.

Each mutable information is externalized by `extern` as follows. If it's not yet externalized, its label is emitted followed by the number of its elements, its external reference, and every element in it.

For example, the following type dictionary

```
/T45 <<
        /decl    (type-decl)
        /printer {psproc}
        /symbol  S23
>> def
```

is externalized as follows:

```
.LT45:
.long 3
.long .LIm139   ;; T45
.long .LIm141   ;; decl
.long .LIm46    ;; (type-decl)
.long .LIm144   ;; printer
.long .LIm4     ;; {psproc}
.long .LIm146   ;; symbol
.long .LS23     ;; S23
```

Its label `.LT45` is emitted followed by the number of its elements `3`, its external reference `T45` denoted by `.LIm139`, and every key-value pair in the type dictionary.

268 ⟨*ldbmutable.mli* 268⟩≡
⟨*ldbMutableType* 269⟩
```
class virtual ldbMutable : ldbMutableType
```

269 ⟨*ldbMutableType* 269⟩≡ (268)

```
class type virtual ldbMutableType =
object
  inherit Ldbinfo.ldbInfo
  (** constructors ************************************************)
  (* set the base name for this mutable information's external reference
   *
   * precondition : the external reference of this mutable information has
   *                not been set yet
   * postcondition: the external reference of this mutable information is set
   *)
  method setExtBase : string -> unit

  (** observers ***************************************************)
  (* get the base name for this mutable information's external reference
   *
   * precondition : the external reference of this mutable information has
   *                already been set
   *)
  method getExtBase : string

  (* get the size of this mutable information
   *
   * precondition : true
   *)
  method virtual size : int

  (* emit all the elements in this mutable information
   *
   * precondition : this mutable information has not been emitted
   * postcondition: this mutable information has not been emitted;
   *                all the elements of this mutable information are emitted
   *)
  method virtual emitElements      : Ldbfilehandle.t -> unit

  (* emit the definition of this mutable information
   *
   * precondition : this mutable information has not been emitted
   * postcondition: this mutable information is emitted
   *)
  method virtual emitDefinition    : Ldbfilehandle.t -> unit

  (*
   * assert that this mutable information has not yet been externed
   *
   * precondition : this mutable information has not yet been externed
```

```
    * postcondition: this mutable information has not yet been externed
    *)
   method mutate : unit
 end
```

5.3.10 PostScript arrays (lists)

`ldbArray` extends the `ldbMutable` class.

- Abstraction : a PostScript array whose elements are PostScript data
- Constructors:
  - `addHigh` : put a given value to the end of this array
- Observers :
  - `get` : get the value of the indexed component in this array
  - `size` : get the size of this array

270 ⟨*ldbarray.mli* 270⟩≡
⟨*ldbArrayType* 271a⟩
```
class ldbArray : ldbArrayType
```

271a ⟨*ldbArrayType* 271a⟩≡ (270)

```
class type ldbArrayType =
object
  inherit Ldbmutable.ldbMutable
  (** constructors ************************************************************)
  (* put a given value to the end of this array
   *
   * precondition : addHigh [a,..,n] i
   * postcondition: [a,..,n,i]
   *)
  method addHigh : Ldbinfo.ldbInfo -> unit

  (** observers ***************************************************************)
  (* return the value of the indexed component in this array
   *
   * precondition : true
   *)
  method get : int -> Ldbinfo.ldbInfo

  (* return the size of this array
   *
   * precondition : true
   *)
  method size           : int

  method emitElements   : Ldbfilehandle.t -> unit
  method emitDefinition : Ldbfilehandle.t -> unit
end
```

5.3.11 PostScript dictionaries (tables)

`ldbTable` extends the `ldbMutable` class.

- Abstraction : a PostScript dictionary with string keys and PostScript values
- Constructors:
  - `put` : add a given key/value pair to this table; overwrite any existing entry with the same key
- Observers :
  - `get`: find the value bound to a given key in this table

271b ⟨*ldbtable.mli* 271b⟩≡

```
⟨ldbTableType 272a⟩
class ldbTable : ldbTableType
```

272a ⟨*ldbTableType* 272a⟩≡ (271b)

```
class type ldbTableType =
object
  inherit Ldbmutable.ldbMutable
  (** constructors ************************************************************)
  (* add a given key/value pair to this table
   *
   * precondition : put <<...(k,v)...>> k i
   * postcondition: <<...(k,i)...>>
   *)
  method put : string -> Ldbinfo.ldbInfo -> unit

  (** observers ***************************************************************)
  (* return the value associated with the specified key in this table
   *
   * precondition : table might be empty
   * postcondition: return a value associated with a given key, if any
   *)
  method get : string -> Ldbinfo.ldbInfo

  method size           : int
  method emitElements   : Ldbfilehandle.t -> unit
  method emitDefinition : Ldbfilehandle.t -> unit
  method set_cache :  (int * int) list * Ldbarray.ldbArray -> unit
  method get_cache : ((int * int) list * Ldbarray.ldbArray) option
end
```

5.4 `ldb` data

This section includes the `ldb` interface for `ldb` data which might be the source-language dependent but the compiler independent. All the compiler-dependent procedures used in this section are defined in Section 5.2.3.

5.4.1 Source-code locations

- Abstraction : a source-code location
- Observers :
  - `file` : return this source-code location's file name
  - `line` : return this source-code location's line number
  - `column` : return this source-code location's column number

272b ⟨*ldbsrcloc.mli* 272b⟩≡

```
⟨srclocType 273a⟩
class ldbSrcLoc : string -> int -> int -> srcLocType
```

273a ⟨*srclocType* 273a⟩≡ (272b)

```
class type srcLocType =
object
  (** observers ************************************************************)
  (* return a given source-code location's file name
   *
   * precondition : true
   *)
  method file : string

  (* return a given source-code location's line number
   *
   * precondition : true
   *)
  method line : int

  (* return a given source-code location's column number
   *
   * precondition : true
   *)
  method column : int
end
```

5.4.2 Values of machine-level constants

- Abstraction : a machine-level value which is either an integer or a floating-point value
- Constructors:
  - `newValueInt` : create a PostScript code whose evaluation results in an integer value
  - `newValueFloat` : create a PostScript code whose evaluation results in a floating-point value

273b ⟨*ldbvalue.mli* 273b⟩≡

```
class ldbValue      : string -> Ldbimmutable.ldbImmutable
val   newValueInt   : int    -> ldbValue
val   newValueFloat : float  -> ldbValue
```

5.4.3 Symbols

- Abstraction : a symbol; every symbol is one of a variable, a procedure, a constant, and a type.
- Observers :
  - `name` : return this symbol's name

274a ⟨*ldbsymbol.mli* 274a⟩≡
```
⟨ldbSymbolType 274b⟩
class ldbSymbol : string -> Ldbsrcloc.ldbSrcLoc -> ldbSymbolType
```

274b ⟨*ldbSymbolType* 274b⟩≡ (274a)
```
class type ldbSymbolType =
object
  inherit Ldbtable.ldbTable

  (** observers *************************************************************)
  (* return a given symbol's name
   *
   * precondition : true
   *)
  method name : string
end
```

Constants

- Abstraction : a constant symbol
- Constructors:
  - `putType` : initialize this constant's type
  - `putValue` : initialize this constant's value

When a constant is created, its name and source location are given but its type is usually not yet provided. Thus, a constant's type is initialized when the type is avaiable by `putType`. A constant's value is either an integer or a floating-point value and this information is initialized by `putValue`.

274c ⟨*ldbconstant.mli* 274c⟩≡
```
⟨ldbConstantType 275a⟩
class ldbConstant : string -> Ldbsrcloc.ldbSrcLoc -> ldbConstantType
```

275a ⟨*ldbConstantType* 275a⟩≡ (274c)

```
class type ldbConstantType = object
  inherit Ldbsymbol.ldbSymbolType

  (** constructors ******************************************************)
  (* initialize a given constant's type
   *
   * precondition : after type checking
   *)
  method putType : Ldbtype.ldbType -> unit

  (* initialize a given constant's value
   *
   * precondition : after lexing
   *)
  method putValue : Ldbvalue.ldbValue -> unit
end
```

Variables

- Abstraction : a variable symbol
- Constructors:
  - `putType` : initialize this variable's type
  - `putWhere` : initialize this variable's placement
  - `putLabel` : set this variable's assembly-language label
- Observers :
  - `label` : return this variable's assembly-language label

When a variable is created, its name and source location are given but its type is usually not yet provided. Thus, a variable's type is initialized when the type is avaiable by `putType`. A variable's placement is either in a stack frame or in a register and this information is initialized by `putWhere` after the register allocation phase. A variable's assembly-language label is used to refer the variable in the executable program. `putLabel` records the variable's label when it's available and `label` returns the label. The naming convention for labels is compiler dependent and labels are described in section 5.4.7.

275b ⟨*ldbvar.mli* 275b⟩≡

```
⟨ldb VarType 276⟩
class ldbVar : string -> Ldbsrcloc.ldbSrcLoc -> ldbVarType
```

276 ⟨*ldbVarType* 276⟩≡ (275b)

```
class type ldbVarType = object
  inherit Ldbsymbol.ldbSymbolType

  (** constructors ************************************************************)
  (* initialize a given variable's type
   *
   * precondition : after type checking
   *)
  method putType : Ldbtype.ldbType -> unit

  (* initialize a given variable's placement
   *
   * precondition : after register allocation
   *)
  method putWhere  : Ldbplcmt.ldbPlcmt -> unit

  (* set a given variable's assembly-language label
   *
   * precondition : after the variable's label is known
   *)
  method putLabel  : Ldblabel.t -> unit

  (** observers ***************************************************************)
  (* returns a given variable's assembly-language label
   *
   * precondition : true
   *)
  method label     : Ldblabel.t
end
```

Procedures

- Abstraction : a procedure symbol
- Constructors:
  - `putType` : initialize this procedure's type
  - `putFormal` : set this procedure's last formal parameter, if any
  - `putArgEnv` : set this procedure's arguments environment
  - `putReturnType` : set this procedure's return type
  - `putFrameSize` : set this procedure's frame size
  - `putRegInfo` : set this procedure's callee-saved registers information
  - `putLabel` : set this procedure's assembly-language label
  - `addLocus` : add a stopping point in this procedure
  - `addLocal` : add a local symbol in this procedure
- Observers :
  - `file` : return this procedure's file name
  - `plcmt` : return this procedure's placement
  - `locals` : return this procedure's local symbols

When a procedure is created, its name and source location are given but its type is usually not yet provided. Thus, a procedure's type is initialized when the type is avaiable by `putType`. `putFormal` sets a procedure's the last formal parameter, if any and `putArgEnv` sets the procedure's arguments environment during the semantic analysis phase and `putReturnType` sets the procedure's return type after the type-checking phase. `putFrameSize` sets a procedure's stack frame size and `putRegInfo` sets a procedure's callee-saved registers information. Callee-saved registers information is given as a list of tuples `(mask, base)` where `mask` is the register-save mask in a decreasing order and `base` is the distance of least significant register from the heavy pointer.

`ldb` assumes that each stack frame can be identified by a single pointer, called the *heavy pointer*, the value of which remains unchanged for the lifetime of the frame. Since the frame and stack pointers are used in different ways on different architectures, `ldb` uses the term "heavy" pointer, not frame or stack pointer. The heavy pointer is realized as either the frame pointer or the stack pointer.

A procedure's assembly-language label is used to refer the procedure in the executable program. `putLabel` records the procedure's label when it's available. The naming convention for labels is compiler dependent and labels are described in section 5.4.7. A procedure records all the stopping points in it and each stopping point is added to the procedure by `addLocus` when the stopping point is created. Compilers usually have several intermediate representations for a program and in which representation to put stopping points are compiler

dependent. Stopping points are described in section 5.4.8. All the local symbols in a procedure is recorded by `addLocal`. `file` returns the procedure's file name to make a source map for the current compilation unit and `plcmt` and `locals` return the procedure's placement and local symbols, respectively.

278 ⟨*ldbproc.mli* 278⟩≡
⟨*ldbProcType* 279⟩
```
class ldbProc : string -> Ldbsrcloc.ldbSrcLoc -> ldbProcType
```

279 ⟨*ldbProcType* 279⟩≡ (278)

```
class type ldbProcType = object
  inherit Ldbsymbol.ldbSymbolType

  (** constructors ******************************************************)
  (* initialize a given procedure's type
   *
   * precondition : after type checking
   *)
  method putType : Ldbtype.ldbType -> unit

  (* initialize a given procedure's the last formal parameter
   *
   * precondition : during semantic analysis
   *)
  method putFormal     : Ldbsymbol.ldbSymbol option -> unit

  (* initialize a given procedure's arguments environment
   *
   * precondition : during semantic analysis
   *)
  method putArgEnv     : Ldbenv.ldbEnv -> unit

  (* initialize a given procedure's return type
   *
   * precondition : after type checking
   *)
  method putReturnType : string -> unit

  (* set a given procedure's frame size
   *
   * precondition : after register allocation
   *)
  method putFrameSize  : int -> unit

  (* set a given procedure's callee-saved registers information
   *
   * precondition : after register allocation
   *)
  method putRegInfo    : (int * int) list -> unit

  (* set a given procedure's assembly-language label
   *
   * precondition : after the procedure's label is known
   *)
  method putLabel      : Ldblabel.t -> unit
```

```
  (* add a specified stopping point to a given procedure's stopping points
   *
   * precondition : after a given stopping point is created
   *)
  method addLocus      : Ldblocus.t -> unit

  (* add a local symbol in this procedure
   *
   * precondition : symbol is not   in this procedure
   * postcondition: symbol is added to this procedure
   *)
  method addLocal      : Ldbsymbol.ldbSymbol -> unit

  (** observers ***********************************************************)
  (* returns a given procedure's file name
   *
   * precondition : true
   *)
  method file      : string

  (* returns a given procedure's placement
   *
   * precondition : true
   *)
  method plcmt : Ldbplcmt.ldbPlcmt

  (* returns a given procedure's local symbols
   *
   * precondition : true
   *)
  method locals   : Ldbsymbol.ldbSymbol list
end
```

Type symbols

- Abstraction : a type symbol
- Constructors:
  - `putType` : initialize this type symbol's type
  - `addVar` : record every variable in this type symbol
  - `addProc` : record every procedure in this type symbol

When a type symbol is created, its name and source location are given but its type is usually not yet provided. Thus, a type symbol's type is initialized when the type is avaiable by `putType`.

281a ⟨*ldbtypesymbol.mli* 281a⟩≡
```
⟨ldbTypeSymbolType 281b⟩
class ldbTypeSymbol : string -> Ldbsrcloc.ldbSrcLoc -> ldbTypeSymbolType
```

281b ⟨*ldbTypeSymbolType* 281b⟩≡ (281a)
```
class type ldbTypeSymbolType = object
  inherit Ldbsymbol.ldbSymbolType

  (** constructors ************************************************)
  (* initialize a given type symbol's type
   *
   * precondition : after type checking
   *)
  method putType : Ldbtype.ldbType -> unit

  (* add a given variable to the set of variables in a given type symbol
   *
   * precondition : true
   *)
  method addVar  : Ldbvar.ldbVar   -> unit

  (* add a given procedure to the set of procedures in a given type symbol
   *
   * precondition : true
   *)
  method addProc : Ldbproc.ldbProc -> unit
end
```

5.4.4 Environments

`ldbEnv` extends the `ldbImmutable` class.

- Abstraction : an environment
- Constructors:
  - `emptyEnv` : create an empty environment
  - `newEnv` : create a new environment

282a ⟨*ldbenv.mli* 282a⟩≡
⟨*ldbEnvType* 282b⟩
```
class ldbEnv : (ldbEnvType * Ldbsymbol.ldbSymbol) option -> ldbEnvType
val emptyEnv : ldbEnv
val newEnv   : ldbEnv -> Ldbsymbol.ldbSymbol -> ldbEnv
```

282b ⟨*ldbEnvType* 282b⟩≡ (282a)
```
class type ldbEnvType = Ldbinfo.ldbInfo
```

5.4.5 Types

- Abstraction : a type

When a type is created, how to print a declaration of the type `decl` and how to print a value of the type `printer` are provided. This information is compiler dependent. For example, let's consider the following structure `Fib` and its type `FIB`:

```
/S4 <<>> def
/T5 << /decl       (%s : FIB)
       /printer    {MODULE}
       /symbol     S4          >> def
%% Backpatches
S4 /name       (Fib)     put
S4 /sourcefile (fib.ml)  put
S4 /sourcey    1         put
S4 /sourcex    6         put
S4 /kind       (type)    put
S4 /type       T5        put
S4 /vars       I6        put
S4 /procs      I7        put
```

In order to print the class `Fib` denoted by `S4` here, the printing method `CLASS` in `FIB` which is `T5` needs to know which variables and functions are in `S4`. Thus, the `symbol` key in `T5` is bound to `S4` and now `CLASS` can access the variables and functions in `S4` through the `symbol` key.

282c ⟨*ldbtype.mli* 282c⟩≡
⟨*ldbTypeType* 283a⟩
```
class ldbType : string -> Ldbpsproc.ldbPsProc -> ldbTypeType
```

283a ⟨*ldbTypeType* 283a⟩≡ (282c)

```
class type ldbTypeType = Ldbtable.ldbTable
```

5.4.6 Placements

- Abstraction : an object location
- Constructors:
  - `newPlcmtAbsolute` : create a fundamental kind of location
    `newPlcmtAbsolute(s, o) = s[o]`
  - `newPlcmtShifted` : create a location relative to some other location
    `newPlcmtShifted(s[o'], o) = s[o'+o]`
  - `newPlcmtIndirect` : create one level of indirection, with displacement
    `newPlcmtIndirect(s, s'[o'], o) = s[s'[o']+o]`
  - `newPlcmtAtLabel` : create a placement of a given label

283b ⟨*ldbplcmt.mli* 283b⟩≡

```
⟨lazy address management 284⟩
class ldbPlcmt : string -> ldbPlcmt option -> Ldbimmutable.ldbImmutable
type space  = char
type offset = int

(* create a fundamental kind of location
 *
 * postcondition: absolute s o = s[o]
 *)
val newPlcmtAbsolute : space -> offset -> ldbPlcmt

(* create a location relative to some other location
 *
 * postcondition: shifted s[o] o' = s[o+o']
 *)
val newPlcmtShifted : ldbPlcmt -> offset -> ldbPlcmt

(* create one level of indirection, with displacement
 *
 * postcondition: indirect s s[o] o' = s[s[o]+o']
 *)
val newPlcmtIndirect : space -> ldbPlcmt -> offset -> ldbPlcmt

(* create a placement of a given label
 *
 * postcondition: label gets its placement
 *)
val newPlcmtAtLabel : Ldblabel.t -> ldbPlcmt
```

Lazy address management

- Abstraction : lazy address management using the anchor symbol
- Constructors:
  - `initialize` : create an anchor symbol
- Observers :
  - `finalize` : finalize emitting references to locations in assembly-language output
  - `anchor` : return the name of the anchor symbol

Some addresses are bound by the linker and therefore are not available at compile time. We pass those addresses to the debugger by planting references in the assembly language output. The references are in known locations relative to the anchor symbol, and we refer to them by using the anchor symbol name and the offset of the reference. While we do not need to use this lazy address management any more because we eliminated the use of `.o.st` and `.lt` files, we still use the anchor symbol so that we can use the `ldb` debugger as it is.

284 ⟨*lazy address management* 284⟩≡ (283b)

```
(** constructors *************************************************)
(* initialize lazy address management; create anchor symbol
 *
 * precondition : true
 * postcondition: the anchor symbol is created
 *)
val initialize: unit -> unit

(** observers *****************************************************)
(* finalize emitting references to locations in assembly-language output
 *
 * precondition : compiler finished emitting assembly-language output
 * postcondition: all the bound labels are emitted
 *)
val finalize: Ldbfilehandle.t -> unit

(* return the name of the anchor symbol
 *
 * precondition : the anchor symbol is created
 *)
val anchor : unit -> string
```

285a ⟨*lazy address management for* atLabel 285a⟩≡

```
let gettimeofday() =
  let time = Unix.gettimeofday() in
  let tm   = Unix.gmtime time    in
  (tm.Unix.tm_min, tm.Unix.tm_sec)
let myanchor = ref ""
(* the offset from the anchor symbol of the next reference to be planted *)
let offset        = ref 0
(* all the lazy placements; (label, offset) list
 *)
let label2offset = ref ([]: (Ldblabel.t * int) list)

let anchor() = !myanchor
let initialize() =
  let (min, sec) = gettimeofday()
  in  (* time/version stamp for this compilation *)
      myanchor := Printf.sprintf "_stanchor__V%d_%d" min sec

(* plant a reference to each of the labels relative to the anchor symbol *)
let finalize out =
  let labelArray = Array.make (!offset) (Ldblabel.inDataSpace "")
        (* exports the anchor symbol *)
  in  ( Ldbutil.exportName(!myanchor) out
      ; List.iter  (fun (l,i) -> Array.set labelArray i l) (!label2offset)
      ; Array.iter (fun l -> Ldblabel.bindHere out l) labelArray
      ; Ldbutil.emitSectionClose out
      )
```

285b ⟨atLabel 285b⟩≡

```
(* return the placement of a specified label, create if not exist *)
let newPlcmtAtLabel label =
  let s = Printf.sprintf "{(%s) %d Lazy%s}" (!myanchor) (!offset)
                         (Ldblabel.space label)
  in  try Hashtbl.find hashT s
      with Not_found -> let newplcmt = new ldbPlcmt s None in
                        ( Hashtbl.add hashT s newplcmt
                        ; label2offset := (label, !offset) :: (!label2offset)
                        ; incr offset
                        ; newplcmt)
```

5.4.7 Labels

- Abstraction : a reference to a location in an assembly output (`.s`)
- Constructors:
  - `inCodeSpace` : return a fresh label with a given name in a code space
  - `inDataSpace` : return a fresh label with a given name in a data space
- Observers :
  - `name` : return this label's name
  - `space` : return a space where this label is bound to
  - `bindHere` : bind this label to a location in an assembly output

286 ⟨*ldblabel.mli* 286⟩≡

```
(** abstraction ***********************************************************)
(* a reference to a location in .s *)
type t

(** constructors **********************************************************)
(* return a fresh label with a given name in a code space
 *
 * postcondition: label is unbound
 *)
val inCodeSpace : string -> t

(* return a fresh label with a given name is a data space
 *
 * postcondition: label is unbound
 *)
val inDataSpace : string -> t

(** observers *************************************************************)
(* return the space name where a given label is bound to
 *
 * precondition : a given label's space name is set when the label is created
 *)
val space : t -> string

(* bind a reference to a location in .s
 *
 * precondition : label is unbound
 * postcondition: label is bound and emitted to .s
 *)
val bindHere : Ldbfilehandle.t -> t -> unit
```

5.4.8 Stopping points

- Abstraction : a stopping point
- Constructors:
  - `make` : create a stopping point
- Observers :
  - `file` : return the file name of this stopping point
  - `line` : return the line number of this stopping point
  - `column` : return the column number of this stopping point
  - `label` : return the label of this stopping point
  - `env` : return the environment at this stopping point

Compilers usually have a set of stopping points where an execution of a given program might stop. For example, MiniJava places a stopping point before each expression, including nested expressions, as well as before each statement and at the end of each block.

287 ⟨*ldblocus.mli* 287⟩≡

```
(** abstraction ************************************************************)
(* a stopping point *)
type t

(** constructors ***********************************************************)
(* create a fresh stopping point
 *
 * precondition : this stopping point does not exist
 * postcondition: this stopping point is created
 *)
val make : Ldbsrcloc.ldbSrcLoc -> Ldblabel.t -> Ldbenv.ldbEnv -> t

(** observers **************************************************************)
(* return the file name of a given stopping point
 *
 * precondition  : true
 *)
val file : t -> string

(* return the line number of a given stopping point
 *
 * precondition  : true
 *)
val line     : t -> int
```

```
(* return the column number of a given stopping point
 *
 * precondition  : true
 *)
val column   : t -> int

(* return the label of a given stopping point
 *
 * precondition  : true
 *)
val label     : t -> Ldblabel.t

(* return the environment at a given stopping point
 *
 * precondition  : true
 *)
val env      : t -> Ldbenv.ldbEnv
```

5.4.9 Compilation units

- Abstraction : a compilation unit
- Constructors:
  - `addProc` : add a procedure to this compilation unit
  - `addExported` : add an exported to this compilation unit
  - `addPrivate` : add a private symbol to this compilation unit
- Observers :
  - `extern` : externalize this compilation unit

The `extern` method emits the symbol table information of this compilation unit. There are entries in the `ldb_st` section which point to the top-level symbol table information and the actual data is placed in the `ldb_st_data` section. For example, the following information is emitted to the assembly-language output:

```
.section .ldb_st
.long .Lldb_st_src
.long .Lldb_st_arch
.long .Lldb_st_anchor
.long .Lldb_st_procs
.long .Lldb_st_externs
.long .Lldb_st_privates
.long .Lldb_st_locals
.long .Lldb_st_sourcemap
.section .ldb_st_data
.Lldb_st_src:
.byte 116
.byte 101
.byte 115
.byte 116
.byte 46
.byte 99
.byte 0
.Lldb_st_arch:
...
.Lldb_st_anchor:
...
.Lldb_st_procs:
.long 2
.long .LS23
.long .LS31
.Lldb_st_externs:
.long 2
.long .LIm161
```

```
.long .LS23
.long .LIm163
.long .LS31
.Lldb_st_privates:
.long 2
.long .LIm179
.long .LS16
.long .LIm181
.long .LS38
.Lldb_st_locals:
.long 1
.long .LS10
.Lldb_st_sourcemap:
.long 2
.long .LIm33
.long 1
.long .LS31
.long .LIm12
.long 1
.long .LS23
```

Since the compilation unit is the top-level information, this class does not have the `ref` method.

290 ⟨*ldbcompunit.mli* 290⟩≡
⟨*ldbCompUnitType* 291⟩
```
class ldbCompUnit : string -> string -> Ldbfilehandle.t -> ldbCompUnitType
```

291 ⟨*ldbCompUnitType* 291⟩≡ (290)

```
class type ldbCompUnitType = object
  inherit Ldbinfo.ldbInfo
  (** constructors ********************************************************)
  (* add a procedure symbol to this compilation unit
   *
   * precondition : procedure symbol is not in the compilation unit
   * postcondition: procedure symbol is added to the compilation unit;
   *                this procedure symbol is defined in this compilation unit
   *)
  method addProc : Ldbproc.ldbProc -> unit

  (* add an exported symbol to this compilation unit
   *
   * precondition : exported symbol is not in the compilation unit
   * postcondition: exported symbol is added to the compilation unit;
   *                this symbol is exported to the other compilation units
   *)
  method addExported : Ldbsymbol.ldbSymbol -> unit

  (* add a privte symbol to this compilation unit
   *
   * precondition : private symbol is not in the compilation unit
   * postcondition: private symbol is added to the compilation unit;
   *                this symbol is private to this compilation unit
   *)
  method addPrivate : Ldbsymbol.ldbSymbol -> unit
end
```

292 ⟨*constructors* 292⟩≡

```
method private addSym arr sym =
  let rec exist i =
          try if ((arr#get i) = sym) then true else exist (i+1) with _ -> false
  in  if (exist 0) then () else arr#addHigh sym

(* add a procedure symbol to procs and incrementally build up srcmap *)
method addProc (p : Ldbproc.ldbProc) =
  let file = p#file in
  ( procs#addHigh (p :> Ldbinfo.ldbInfo)
  ; procsL <- procsL@[p]
  ; try match List.find (fun (f,_) -> f=file) srcmap with
        | (_,arr) -> self#addSym arr (p :> Ldbinfo.ldbInfo)
    with _ -> let ps = new Ldbarray.ldbArray
              in  ( ps#addHigh (p :> Ldbinfo.ldbInfo)
                  ; srcmap <- srcmap@[(file, ps)]
                  )
    (* add a local symbol to locals *)
  ; List.iter (fun s -> ( self#addSym locals (s :> Ldbinfo.ldbInfo)
                        ; localsL <- localsL@[s#name])) p#locals
  )

(* add an exported symbol to exported *)
method addExported (s : Ldbsymbol.ldbSymbol) =
  ( self#addSym exported (s :> Ldbinfo.ldbInfo)
  ; exportedL <- exportedL@[s#name])

(* add a private symbol to privates *)
method addPrivate  (s : Ldbsymbol.ldbSymbol) =
  ( self#addSym privates (s :> Ldbinfo.ldbInfo)
  ; privatesL <- privatesL@[s#name])
```

Bibliography

Sukyoung Ryu and Norman Ramsey. 2005 (April). Source-level debugging for multiple languages with modest programming effort. In *14th International Conference on Compiler Construction (CC 2005)*, Lecture Notes in Computer Science.