Appendix A

Supporting code for Impcore

A.1 Uninteresting interfaces

A.1.1 Lexical analysis

We first read the input as a list of parenthesized expressions called `Par`s, which are themselves either names or lists of `Par`s. This corresponds to the *concrete syntax* of the language itself. The simplicity of the representation is a direct result of the syntactic symplicity of Impcore (and our other languages).

585a ⟨*par.t* 585a⟩≡
```
Par* = ATOM (Name)
     | LIST (Parlist)
```

585b ⟨*shared type and structure definitions* 585b⟩≡
```
typedef struct Par *Par;
typedef enum { ATOM, LIST } Paralt;
struct Par { Paralt alt; union { Name atom; Parlist list; } u; };
```

585c ⟨*shared function prototypes* 585c⟩≡ (25) 585e▷
```
Par mkAtom(Name atom);
Par mkList(Parlist list);
struct Par mkAtomStruct(Name atom);
struct Par mkListStruct(Parlist list);
```

585d ⟨*shared type definitions* 585d⟩≡ (25)
```
typedef struct Parlist *Parlist; /* list of Par */
```

585e ⟨*shared function prototypes* 585c⟩+≡ (25) ◁585c 585f▷
```
Printer printpar;
```

The `readparlist` function reads and returns a list of parenthesized expressions from the given `Reader`, stopping when an end-of-expression and end-of-line coincide. The `doquote` flag specifies that an expression like `'(1 2 3)` should be turned into `(quote (1 2 3))`; it is used to implement μScheme. The `doprompt` flag specifies that prompts should be printed when reading input lines.

585f ⟨*shared function prototypes* 585c⟩+≡ (25) ◁585e 591c▷
```
Parlist readparlist(Reader r, int doquote, int doprompt);
```

A.2 Uninteresting implementations

Of the uninteresting implementations, the extensible printer is undoubtedly the most interesting, even if it is the least relevant to programming languages.

A.2.1 Printing

586a ⟨*print.c* 586a⟩≡ 586b▷

```
#include "all.h"

static Printer *printertab[256];
```

586b ⟨*print.c* 586a⟩+≡ ◁586a 586c▷

```
void installprinter(unsigned char c, Printer *printer) {
    printertab[c] = printer;
}
```

586c ⟨*print.c* 586a⟩+≡ ◁586b 586d▷

```
void fprint(FILE *output, const char *fmt, ...) {
    va_list_box box;

    assert(fmt);
    va_start(box.ap, fmt);
    vprint(output, fmt, &box);
    va_end(box.ap);
    fflush(output);
}
```

586d ⟨*print.c* 586a⟩+≡ ◁586c 586e▷

```
void print(const char *fmt, ...) {
    va_list_box box;

    assert(fmt);
    va_start(box.ap, fmt);
    vprint(stdout, fmt, &box);
    va_end(box.ap);
    fflush(stdout);
}
```

586e ⟨*print.c* 586a⟩+≡ ◁586d 587a▷

```
void vprint(FILE *output, const char *fmt, va_list_box *box) {
    unsigned char *p;
    for (p = (unsigned char*)fmt; *p; p++) {
        if (*p != '%') {
            putc(*p, output);
            continue;
        }
        if (printertab[*++p])
            printertab[*p](output, box);
        else {
            fprintf(output, "*%c*", *p);
            break;
        }
    }
}
```

_IO_putc B
fflush B
fprintf B
stdout B

587a ⟨*print.c* 586a⟩+≡ ◁586e 587b▷
```
void printpercent(FILE *output, va_list_box *box) {
    (void)box;
    putc('%', output);
}
```

587b ⟨*print.c* 586a⟩+≡ ◁587a 587c▷
```
void printstring(FILE *output, va_list_box *box) {
    const char *s = va_arg(box->ap, char*);
    fputs(s == NULL ? "<null>" : s, output);
}
```

587c ⟨*print.c* 586a⟩+≡ ◁587b
```
void printdecimal(FILE *output, va_list_box *box) {
    fprintf(output, "%d", va_arg(box->ap, int));
}
```

A.2.2 Error interface

Function `error` prints a message, then jumps to `errorjmp`.

587d ⟨*error.c* 587d⟩≡ 587e▷
```
#include "all.h"

jmp_buf errorjmp;

void error(const char *fmt, ...) {
    va_list_box box;

    assert(fmt);
    fflush(stdout);
    fprint(stderr, "error: ");
    va_start(box.ap, fmt);
    vprint(stderr, fmt, &box);
    va_end(box.ap);
    fprint(stderr, "\n");
    fflush(stderr);
    longjmp(errorjmp, 1);
}
```

`checkargc` checks number of arguments.

587e ⟨*error.c* 587d⟩+≡ ◁587d 588a▷
```
void checkargc(Exp e, int expected, int actual) {
    if (expected != actual)
        error("expected %d but found %d argument%s in %e",
              expected, actual, actual == 1 ? "" : "s", e);
}
```

_IO_putc	*B*
fflush	*B*
fprint	33f
fprintf	*B*
fputs	*B*
longjmp	*B*
stderr	*B*
stdout	*B*
vprint	34d

If a list of names contains duplicates, `duplicatename` returns a duplicate. It is used to detect duplicate names in lists of formal parameters. Its cost is quadratic in the number of parameters, which should be very fast for any reasonable function.

588a ⟨*error.c* 587d⟩+≡ ◁587e

```
Name duplicatename(Namelist nl) {
    if (nl != NULL) {
        Name n = nl->hd;
        Namelist tail;
        for (tail = nl->tl; tail; tail = tail->tl)
            if (n == tail->hd)
                return n;
        return duplicatename(nl->tl);
    }
    return NULL;
}
```

The tail call could be turned into a loop, but it hardly seems worth it.

A.2.3 Input Readers

588b ⟨*read.c* 588b⟩≡ 588d▷

```
#include "all.h"
```

An input reader is just a line-reading function and some extra state.

588c ⟨*shared structure definitions* 588c⟩≡ (25)

```
struct Reader {
    char *buf;              /* holds the last line read */
    int nbuf;               /* size of buf */
    int line;               /* current line number */
    const char *readername; /* identifies this reader */

    FILE *fin;              /* filereader */
    const char *s;          /* stringreader */
};
```

The reader creators are simple.

588d ⟨*read.c* 588b⟩+≡ ◁588b 588e▷

```
Reader stringreader(const char *stringname, const char *s) {
    Reader r = calloc(1, sizeof(*r));
    assert(r);
    r->readername = stringname;
    r->s = s;
    return r;
}
```

calloc B

588e ⟨*read.c* 588b⟩+≡ ◁588d 589a▷

```
Reader filereader(const char *filename, FILE *fin) {
    Reader r = calloc(1, sizeof(*r));
    assert(r);
    r->readername = filename;
    r->fin = fin;
    return r;
}
```

Function `readline` returns a pointer to the next line from the input, which is held in a buffer that is reused on subsequent calls. Function `growbuf` makes sure the buffer is at least `n` bytes long.

589a ⟨*read.c* 588b⟩+≡ ◁588e 589b▷

```
static void growbuf(Reader r, int n) {
    if (r->nbuf < n) {
        r->buf = realloc(r->buf, n);
        assert(r->buf != NULL);
        r->nbuf = n;
    }
}
```

We cook `readline` to print out the line read if it begins with the string `;#`. This string is a special comment that helps us test the chunks marked ⟨*transcript* 10a⟩.

589b ⟨*read.c* 588b⟩+≡ ◁589a

```
char* readline(Reader r, char *prompt) {
    if (prompt)
        print("%s", prompt);

    r->line++;
    if (r->fin)
        ⟨set r->buf to next line from file, or return NULL if lines are exhausted 589c⟩
    else if (r->s)
        ⟨set r->buf to next line from string, or return NULL if lines are exhausted 590a⟩
    else
        assert(0);

    if (r->buf[0] == ';' && r->buf[1] == '#')
        print("%s\n", r->buf);

    return r->buf;
}
```

Returning the next line from a file requires us to continually call `fgets` until we get a newline character at the end of the returned string.

589c ⟨*set* `r->buf` *to next line from file, or return* `NULL` *if lines are exhausted* 589c⟩≡ (589b)

```
{
    int n;

    for (n = 0; n == 0 || r->buf[n-1] != '\n'; n = strlen(r->buf)) {
        growbuf(r, n+512);
        if (fgets(r->buf+n, 512, r->fin) == NULL)
            break;
    }
    if (n == 0)
        return NULL;
    if (r->buf[n-1] == '\n')
        r->buf[n-1] = '\0';
}
```

fgets B
print 33f
realloc B
strlen B

To return the next line in a string, we need to find it and update the pointer.

590a ⟨*set* `r->buf` *to next line from string, or return* `NULL` *if lines are exhausted* 590a⟩≡ (589b)

```
{
    const char *p;
    int len;

    if ((p = strchr(r->s, '\n')) == NULL)
        return NULL;

    p++;

    len = p - r->s;
    growbuf(r, len);
    strncpy(r->buf, r->s, len);
    r->buf[len-1] = '\0';   /* no newline */

    r->s = p;
}
```

A.2.4 Function environments

This code is continued from Chapter 2, which gives the implementation of value environments. Except for types, the code is identical to code in Chapter 2.

590b ⟨*env.c* 590b⟩≡ 590c▷

```
struct Funenv {
    Namelist nl;
    Funlist  fl;
};
```

590c ⟨*env.c* 590b⟩+≡ ◁590b 590d▷

```
Funenv mkFunenv(Namelist nl, Funlist fl) {
    Funenv e = malloc(sizeof *e);
    assert(e != NULL);
    assert(lengthNL(nl) == lengthFL(fl));
    e->nl = nl;
    e->fl = fl;
    return e;
}
```

lengthFL 33d
lengthNL 33e
malloc *B*
strchr *B*
strncpy *B*

⟨*env.c* 590b⟩+≡ ◁590c 590e▷

```
static Fun* findfun(Name name, Funenv env) {
    Namelist nl = env->nl;
    Funlist  fl = env->fl;

    for ( ; nl && fl; nl = nl->tl, fl = fl->tl)
        if (name == nl->hd)
            return &fl->hd;
    return NULL;
}
```

590e ⟨*env.c* 590b⟩+≡ ◁590d 591a▷

```
int isfunbound(Name name, Funenv env) {
    return findfun(name, env) != NULL;
}
```

591a ⟨*env.c* 590b⟩+≡ ◁590e 591b▷
```
Fun fetchfun(Name name, Funenv env) {
    Fun *fp = findfun(name, env);
    assert(fp != NULL);
    return *fp;
}
```

591b ⟨*env.c* 590b⟩+≡ ◁591a
```
void bindfun(Name name, Fun fun, Funenv env) {
    Fun *fp = findfun(name, env);
    if (fp != NULL)
        *fp = fun;              /* safe optimization */
    else {
        env->nl = mkNL(name, env->nl);
        env->fl = mkFL(fun,  env->fl);
    }
}
```

A.2.5 Stack-overflow detection

The first call to `checkoverflow` sets a "low-water mark" in the stack. Each later call checks the current stack pointer against that low-water mark. If the distance exceeds `limit`, `checkoverflow` calls `error`. Otherwise it returns the distance.

591c ⟨*shared function prototypes* 585c⟩+≡ (25) ◁585f
```
extern int checkoverflow(int limit);
```

We assume that the stack grows downward.

591d ⟨*overflow.c* 591d⟩≡
```
#include "all.h"

static volatile char *low_water_mark = NULL;

int checkoverflow(int limit) {
  volatile char c;
  if (low_water_mark == NULL) {
    low_water_mark = &c;
    return 0;
  } else if (low_water_mark - &c >= limit) {
    error("Recursion too deep");
    return -1; /* not reachable, but the compiler can't tell */
  } else {
    return (low_water_mark - &c);
  }
}
```

error 35b
mkFL 33d
mkNL 33e

A.2.6 Lexical Analysis

The implementation of lexical analysis is not central to this book; it can be skipped on first and subsequent readings.

591e ⟨*lex.c* 591e⟩≡ 592a▷
```
#include "all.h"
#include <ctype.h>
```

To parse input, we use helper routines that take a reference to a string pointer and advance the pointer, returning an object corresponding to the data read.

Function `readname` reads the next name from the input string. The `quote` flag controls whether a single quote (') is treated as a special token-breaking character like a parenthesis. Function `strntoname` returns the name that corresponds to the first `n` characters of a string.

592a ⟨*lex.c* 591e⟩+≡ ◁591e 592b▷

```
static Name strntoname(char *s, int n) {
    char *t = malloc(n + 1);
    assert(t != NULL);
    strncpy(t, s, n);
    t[n] = '\0';
    return strtoname(t);
}
```

592b ⟨*lex.c* 591e⟩+≡ ◁592a 592c▷

```
static int isdelim(char c, int quote) {
    if (quote && c == '\'')
        return 1;

    return c == '\0' || c == '(' || c == ')' || c == ';' || isspace(c);
}
```

592c ⟨*lex.c* 591e⟩+≡ ◁592b 592d▷

```
static Name readname(char **ps, int quote) {
    char *p, *q;

    p = *ps;
    for (q = p; !isdelim(*q, quote); q++)
        ;
    *ps = q;
    return strntoname(p, q - p);
}
```

Function `readpar` uses `readname` to read the next (possibly parenthesized) expression from the input. Function `skipspace` advances the `ps` pointer over any initial whitespace. The argument to `isspace` is required to be unsigned.

592d ⟨*lex.c* 591e⟩+≡ ◁592c 593a▷

```
static void skipspace(char **ps) {
    while (isspace((unsigned char)**ps))
        (*ps)++;
}
```

malloc 𝓑
strncpy 𝓑
strtoname 28d

593a ⟨*lex.c* 591e⟩+≡ ◁592d 595a▷

```
enum {
    More       = 1 << 0,
    Quote      = 1 << 1,
    Rightparen = 1 << 2
};
static Par readpar(char **ps, Reader r, int flag, char *moreprompt) {
    if (*ps == NULL)
        return NULL;

    skipspace(ps);

    switch (**ps) {
    case '\0':
    case ';':
        ⟨readpar end of line and return 593b⟩
    case ')':
        ⟨readpar right parenthesis and return 593c⟩
    case '(':
        ⟨readpar left parenthesis and return 594a⟩
    default:
        if ((flag & Quote) && **ps == '\'') {
            ⟨readpar quoted expression and return 594b⟩
        } else {
            ⟨readpar name and return 594c⟩
        }
    }
}
```

If we encounter the end of a line before finding an expression, the result is determined by the More flag. If it is set, we read a new line and keep going. Otherwise we return NULL and set *ps to NULL to indicate the end of the line.

593b ⟨*readpar end of line and return* 593b⟩≡ (593a)

```
if ((flag & More) && (*ps = readline(r, moreprompt)) != NULL)
    return readpar(ps, r, flag, moreprompt);
*ps = NULL;
return NULL;
```

When we're not expecting one, a right parenthesis is a fatal error; if we are expecting one, we return NULL but leave *ps pointing at the parenthesis.

593c ⟨*readpar right parenthesis and return* 593c⟩≡ (593a)

```
if ((flag & Rightparen) == 0)
    error("unexpected right parenthesis in %s, line %d", r->readername, r->line);
return NULL;
```

error 35b
readline 31b

If we see a left parenthesis, we read tokens until we get a right parenthesis, adding them to the end of a list.

594a ⟨*readpar left parenthesis and return* 594a⟩≡ (593a)

```
{
    Par p = mkList(NULL);
    Parlist *ppl =&p->u.list;
    Par q;  /* next par read in, to be accumulated into p */

    (*ps)++;
    while ((q = readpar(ps, r, flag | More | Rightparen, moreprompt))) {
        *ppl = mkPL(q, NULL);
        ppl  = &(*ppl)->tl;
    }
    if (*ps == NULL)
        error("premature end of file reading list (missing right paren)");

    assert(**ps == ')');
    (*ps)++;    /* past right parenthesis */

    return p;
}
```

If we are lexing a language that uses a ' operator, when we see a ', we read the next `Par` (say `x`) and then return `(quote x)`.

594b ⟨*readpar quoted expression and return* 594b⟩≡ (593a)

```
{
    Par p;

    (*ps)++;
    p = readpar(ps, r, More|Quote, moreprompt);
    return mkList(mkPL(mkAtom(strtoname("quote")), mkPL(p, NULL)));
}
```

Names are easy.

⟨*readpar name and return* 594c⟩≡ (593a)

```
return mkAtom(readname(ps, flag & Quote));
```

Function `readparlist` reads `Par` expressions until an end of expression and an end of line coincide; it returns the list of expressions read. The strings used as primary and secondary prompts are also buried instead `readparlist`.

595a ⟨*lex.c* 591e⟩+≡ ◁593a 595b▷

```
Parlist readparlist(Reader r, int doquote, int doprompt) {
    char *s;
    Par p;
    Parlist pl = NULL;
    Parlist *ppl = &pl;

    while (pl == NULL) {
        if ((s = readline(r, doprompt ? "-> " : "")) == NULL)
            return NULL;

        while ((p = readpar(&s, r, doquote ? Quote : 0, doprompt ? "   " : ""))) {
            *ppl = mkPL(p, NULL);
            ppl  = &(*ppl)->tl;
        }
    }
    return pl;
}
```

595b ⟨*lex.c* 591e⟩+≡ ◁595a

```
void printpar(FILE *output, va_list_box *box) {
    Par p = va_arg(box->ap, Par);
    if (p == NULL) {
        fprint(output, "<null>");
        return;
    }

    switch (p->alt){
    case ATOM:
        fprint(output, "%n", p->u.atom);
        break;
    case LIST:
        fprint(output, "(%P)", p->u.list);
        break;
    }
}
```

fprint 33f
mkPL,
in Impcore 33b
in μScheme
A
readline 31b

A.2.7 parse.c: Parser

595c ⟨*parse.c* 595c⟩≡ 596a▷

```
#include "all.h"
```

⟨*parse.c declarations* 596d⟩

The parser needs to take concrete syntax, check to see that it is well formed, and produce abstract syntax. It provides the `readtop` function.

At the top level, parsing amounts to looking for top level constructs and passing the rest of the work to `parseexp`, which parses the input into `Exp`s.

596a ⟨*parse.c* 595c⟩+≡ ◁595c 597a▷

```
static Def parse(Par p) {
    switch (p->alt) {
    case ATOM:
        ⟨parse atom and return the result 596b⟩
    case LIST:
        ⟨parse list and return the result 596c⟩
    }
    assert(0);
    return NULL;
}
```

If we have a name, we treat it as an expression.

596b ⟨*parse* `atom` *and return the result* 596b⟩≡ (596a)

```
return mkExp(parseexp(p));
```

If we have a list, we need to look for `define`, `val`, and `use`.

596c ⟨*parse* `list` *and return the result* 596c⟩≡ (596a)

```
{
    Name first;
    Parlist pl = p->u.list;
    if (pl == NULL)
        error("%p: empty list", p);
    if (nthPL(pl, 0)->alt != ATOM)
        error("%p: first item of list not name", p);

    first = nthPL(pl, 0)->u.atom;
    if (first == strtoname("define")) {
        ⟨parse define and return the result 597b⟩
    }
    if (first == strtoname("val")) {
        ⟨parse val and return the result 597c⟩
    }
    if (first == strtoname("use")) {
        ⟨parse use and return the result 597d⟩
    }
    return mkExp(parseexp(p));
}
```

error 35b
mkExp 27b
nthPL 33b
parseexp 598b
strtoname 28d

The `getnamelist` function turns a `Parlist` that is a list of names into a `Namelist`, calling `error` if the `Parlist` contains any sublists. The `Par*` parameter is used only to print a good error message.

596d ⟨*parse.c declarations* 596d⟩≡ (595c) 598a▷

```
static Namelist getnamelist(Name f, Par p, Parlist pl);
```

597a ⟨*parse.c* 595c⟩+≡ ◁596a 597e▷

```
static Namelist getnamelist(Name f, Par p, Parlist pl) {
    if (pl == NULL)
        return NULL;
    if (pl->hd->alt != ATOM)
        error("%p: formal-parameter list of function %n contains something that is not a name", p, f);
    return mkNL(pl->hd->u.atom, getnamelist(f, p, pl->tl));
}
```

Parsing the top-level expressions requires checking the argument counts and then parsing the subpieces. For function definitions, we could check that formal parameters have distinct names, but that check is part of the operational semantics for function definition.

597b ⟨*parse* `define` *and return the result* 597b⟩≡ (596c)

```
if (lengthPL(pl) != 4 || nthPL(pl, 1)->alt != ATOM || nthPL(pl, 2)->alt != LIST)
    error("%p: usage: (define fun (formals) body)", p);

{
  Name     name    = nthPL(pl, 1)->u.atom;
  Namelist formals = getnamelist(name, p, nthPL(pl, 2)->u.list);
  Exp      body    = parseexp(nthPL(pl, 3));
  return mkDefine(name, mkUserfun(formals, body));
}
```

597c ⟨*parse* `val` *and return the result* 597c⟩≡ (596c)

```
Exp var, exp;

if (lengthPL(pl) != 3)
    error("%p: usage: (val var exp)", p);

var = parseexp(nthPL(pl, 1));
if (var->alt != VAR)
    error("%p: usage: (val var exp) (bad variable)", p);

exp = parseexp(nthPL(pl, 2));
return mkVal(var->u.var, exp);
```

597d ⟨*parse* `use` *and return the result* 597d⟩≡ (596c)

```
if (lengthPL(pl) != 2 || nthPL(pl, 1)->alt != ATOM)
    error("%p: usage: (use filename)", p);

return mkUse(nthPL(pl, 1)->u.atom);
```

Now we can move on to parsing Exps. The `parselist` helper function repeatedly calls `parseexp` to parse a list of Par expressions, eventually returning a list of Exps.

597e ⟨*parse.c* 595c⟩+≡ ◁597a 598b▷

```
static Explist parselist(Parlist pl) {
    Exp e;

    if (pl == NULL)
        return NULL;

    e = parseexp(pl->hd); /* force pl->hd to be parsed first */
    return mkEL(e, parselist(pl->tl));
}
```

Parsing expressions first depends on whether we have a name or a list.

598a ⟨*parse.c declarations* 596d⟩+≡ (595c) ◁596d
```
static Exp parseexp(Par);
```

598b ⟨*parse.c* 595c⟩+≡ ◁597e 600b▷
```
static Exp parseexp(Par p) {
    switch (p->alt) {
    case ATOM:
        ⟨parseexp atom and return the result 598c⟩
    case LIST:
        ⟨parseexp list and return the result 599a⟩
    default:
        assert(0);
        return NULL;
    }
}
```

If we have a name, it must be either a literal value or a variable.

598c ⟨*parseexp* atom *and return the result* 598c⟩≡ (598b)
```
{
    const char *s = nametostr(p->u.atom);
    char *t;
    long l = strtol(s, &t, 10);
    if (*t == '\0') /* the number is the whole string */
        return mkLiteral(l);
    else
        return mkVar(p->u.atom);
}
```

mkLiteral	$\mathcal{A}$
mkVar	$\mathcal{A}$
nametostr	28d
strtol	$\mathcal{B}$

If we have a list, we need to look at the first element, which must be a name.

599a ⟨*parseexp* `list` *and return the result* 599a⟩≡ (598b)

```
{
    Parlist pl;
    Name first;
    Explist argl;

    pl = p->u.list;
    if (pl == NULL)
        error("%p: empty list in input", p);
    if (pl->hd->alt != ATOM)
        error("%p: first item of list not name", p);

    first = pl->hd->u.atom;
    argl  = parselist(pl->tl);
    if (first == strtoname("begin")) {
        ⟨parseexp begin and return the result 599b⟩
    } else if (first == strtoname("if")) {
        ⟨parseexp if and return the result 599c⟩
    } else if (first == strtoname("set")) {
        ⟨parseexp set and return the result 599e⟩
    } else if (first == strtoname("while")) {
        ⟨parseexp while and return the result 599d⟩
    } else {
        ⟨parseexp function application and return the result 600a⟩
    }
}
```

A `begin` expression can have any number of parameters.

599b ⟨*parseexp* `begin` *and return the result* 599b⟩≡ (599a)

```
return mkBegin(argl);
```

An `if` expression needs three parameters.

599c ⟨*parseexp* `if` *and return the result* 599c⟩≡ (599a)

```
if (lengthEL(argl) != 3)
    error("%p: usage: (if cond true false)", p);
return mkIfx(nthEL(argl, 0), nthEL(argl, 1), nthEL(argl, 2));
```

A `while` loop needs two.

599d ⟨*parseexp* `while` *and return the result* 599d⟩≡ (599a)

```
if (lengthEL(argl) != 2)
    error("%p: usage: (while cond body)", p);
return mkWhilex(nthEL(argl, 0), nthEL(argl, 1));
```

A `set` expression requires a variable and a value.

599e ⟨*parseexp* `set` *and return the result* 599e⟩≡ (599a)

```
if (lengthEL(argl) != 2)
    error("%p: usage: (set var exp)", p);
if (nthEL(argl, 0)->alt != VAR)
    error("%p: set needs variable as first param", p);
return mkSet(nthEL(argl, 0)->u.var, nthEL(argl, 1));
```

Anything else must be a function application. We can't check the number of parameters here, because the function definition might change before evaluation, or might not be present yet (as occurs, for example, when defining recursive functions).

600a ⟨*parseexp function application and return the result* 600a⟩≡ (599a)

```
return mkApply(first, argl);
```

Now we can assemble `readtop`. We keep a list of read but not yet parsed `Pars` in `tr->pl`.

600b ⟨*parse.c* 595c⟩+≡ ◁598b 600c▷

```
struct Defreader {
    int doprompt;
    Reader r;
    Parlist pl;
};
```

600c ⟨*parse.c* 595c⟩+≡ ◁600b 600d▷

```
Def readdef(Defreader dr) {
    Par p;

    if (dr->pl == NULL) {
        dr->pl = readparlist(dr->r, 0, dr->doprompt);
        if (dr->pl == NULL)
            return NULL;
    }

    p      = dr->pl->hd;
    dr->pl = dr->pl->tl;
    return parse(p);
}
```

600d ⟨*parse.c* 595c⟩+≡ ◁600c

```
Defreader defreader(Reader r, int doprompt) {
    Defreader dr = malloc(sizeof(*dr));
    assert(dr != NULL);
    dr->r        = r;
    dr->doprompt = doprompt;
    dr->pl       = NULL;
    return dr;
}
```

A.2.8 Printers

601 ⟨*ast.c* 601⟩≡ 602a▷

```
#include "all.h"

void printexp(FILE *output, va_list_box *box) {
    Exp e = va_arg(box->ap, Exp);
    if (e == NULL) {
        fprint(output, "<null>");
        return;
    }

    switch (e->alt){
    case LITERAL:
        fprint(output, "%v", e->u.literal);
        break;
    case VAR:
        fprint(output, "%n", e->u.var);
        break;
    case SET:
        fprint(output, "(set %n %e)", e->u.set.name, e->u.set.exp);
        break;
    case IFX:
        fprint(output, "(if %e %e %e)", e->u.ifx.cond, e->u.ifx.true, e->u.ifx.false);
        break;
    case WHILEX:
        fprint(output, "(while %e %e)", e->u.whilex.cond, e->u.whilex.exp);
        break;
    case BEGIN:
        fprint(output, "(begin%s%E)", e->u.begin?" ":"", e->u.begin);
        break;
    case APPLY:
        fprint(output, "(%n%s%E)", e->u.apply.name,
                    e->u.apply.actuals?" ":"", e->u.apply.actuals);
        break;
    }
}
```

fprint 33f

602a ⟨*ast.c* 601⟩+≡ ◁601

```
void printdef(FILE *output, va_list_box *box) {
    Def d = va_arg(box->ap, Def);
    if (d == NULL) {
        fprint(output, "<null>");
        return;
    }

    switch (d->alt) {
    case VAL:
        fprint(output, "(val %n %e)", d->u.val.name, d->u.val.exp);
        break;
    case EXP:
        fprint(output, "%e", d->u.exp);
        break;
    case DEFINE:
        fprint(output, "(define %n (%N) %e)", d->u.define.name,
                       d->u.define.userfun.formals,
                 d->u.define.userfun.body);
        break;
    case USE:
        fprint(output, "(use %n)", d->u.use);
        break;
    default:
        assert(0);
    }
}
```

602b ⟨*fun.c* 602b⟩≡

```
#include "all.h"

void printfun(FILE *output, va_list_box *box) {
    Fun f = va_arg(box->ap, Fun);
    switch (f.alt) {
    case PRIMITIVE:
        fprint(output, "<%n>", f.u.primitive);
        break;
    case USERDEF:
        fprint(output, "<userfun (%N) %e>", f.u.userdef.formals, f.u.userdef.body);
        break;
    default:
        assert(0);
    }
}
```

fprint 33f

602c ⟨*value.c* 602c⟩≡

```
#include "all.h"

void printvalue(FILE *output, va_list_box *box) {
    Value v = va_arg(box->ap, Value);
    fprint(output, "%d", v);
}
```