Appendix B

Supporting code for μScheme

Contents

B.1 Uninteresting μScheme code

This code produces the largest integer that is not greater than the square root of `n`. This is a pathetic definition of square root, but it does work on perfect squares, and it's also useful for testing primality.

603a ⟨*definition of* `sqrt` 603a⟩≡ (82a 159c)

```
-> (define sqrt (n)
     (letrec ((find (lambda (r)
                      (if (> (* r r) n) (- r 1) (find (+ r 1))))))
       (find 0)))
```

The transcript actually includes the polymorphic-set code, but at the end, so as not to befoul non-polymorphic uses of the set operations.

603b ⟨*transcript* 603b⟩≡

⟨*polymorphic-set transcript* 103a⟩

B.2 Uninteresting parts of the μScheme interpreter

These implementations were deemed not interesting enough to include in Chapter 3.

For CS45600, Purdue University, Spring 2014 only -- do nc

B.2.1 Environments

Because environments can be copied, and the copies can be extended, we use a different representation of environments than we used in Impcore. First, and most important, environments are immutable, as we can see from the interface in Section 3.14.1. The operational semantics never mutates an environment, and there is really no need, because all the mutation is done on locations. Moreover, if we wanted to mutate environments, it wouldn't be safe to copy them just by copying pointers; this would make the evaluation of `lambda` expressions very expensive.

We choose a representation of environments that makes it easy to share and extend them: an environment contains a single binding and a pointer to the rest of the bindings in the environment.

604a ⟨*env.c* 604a⟩≡ 604b▷

```
#include "all.h"

struct Env {
    Name name;
    Value *loc;
    Env tl;
};
```

We look up a name by following `tl` pointers.

604b ⟨*env.c* 604a⟩+≡ ◁604a 604c▷

```
Value* find(Name name, Env env) {
    for (; env; env = env->tl)
        if (env->name == name)
            return env->loc;
    return NULL;
}
```

Function `bindalloc` *always* creates a new environment with a new binding. There is never any mutation.

604c ⟨*env.c* 604a⟩+≡ ◁604b 604d▷

```
Env bindalloc(Name name, Value val, Env env) {
    Env newenv = malloc(sizeof(*newenv));
    assert(newenv != NULL);

    newenv->name = name;
    newenv->loc  = allocate(val);
    newenv->tl   = env;
    return newenv;
}
```

allocate	126d
bindalloc	637c
env	637c
malloc	$\mathcal{B}$
name	637c
newenv	637c
val	637c

⟨*env.c* 604a⟩+≡ ◁604c 605a▷

```
Env bindalloclist(Namelist nl, Valuelist vl, Env env) {
    for (; nl && vl; nl = nl->tl, vl = vl->tl)
        env = bindalloc(nl->hd, vl->hd, env);
    return env;
}
```

For CS45600, Purdue University, Spring 2014 only — do nc

We include `printenv` in case it helps you debug your code.

605a ⟨*env.c* 604a⟩+≡ ◁604d

```
void printenv(FILE *output, va_list_box *box) {
    Env env = va_arg(box->ap, Env);
    char *prefix = " ";

    fprint(output, "{");
    for (; env; env = env->tl) {
        fprint(output, "%s%n -> %v", prefix, env->name, *env->loc);
        prefix = ", ";
    }
    fprint(output, " }");
}
```

B.2.2 Values

The implementation of the value interface is not especially interesting. The first part supports Booleans.

605b ⟨*value.c* 605b⟩≡ 605c▷

```
#include "all.h"

int istrue(Value v) {
    return v.alt != BOOL || v.u.bool;
}

Value truev, falsev;

void initvalue(void) {
    truev  = mkBool(1);
    falsev = mkBool(0);
}
```

The interface defines a function to return an unspecified value. "Unspecified" means we can pick any value we like. For example, we could just always use *nil*. Unfortunately, if we do that, careless persons will grow to rely on finding *nil*, and they shouldn't. To foil such carelessness, we choose an unhelpful value at random.

605c ⟨*value.c* 605b⟩+≡ ◁605b 606a▷

```
Value unspecified (void) {
    switch ((rand()>>4) & 0x3) {
        case 0:  return truev;
        case 1:  return mkNum(rand());
        case 2:  return mkPair(NULL, NULL);
        case 3:  return mkPrimitive(-12, NULL);
        default: return mkNil();
    }
}
```

fprint 33f
mkBool $\mathcal{A}$
mkNil $\mathcal{A}$
mkNum $\mathcal{A}$
mkPair $\mathcal{A}$
mkPrimitive $\mathcal{A}$
rand $\mathcal{B}$

With any luck, careless persons' code might make our interpreter dereference a `NULL` pointer, which is no worse than such persons deserve.

The rest of the code deals with printing—a complex and unpleasant task.

For CS45600, Purdue University, Spring 2014 only --- do nc

To print a closure nicely, we don't want to print entire environments, but only the parts of the environment that the closure actually depends on—the free variables of the `lambda` expression. Finding free variables is hard work. We start with a bunch of utility functions on names. Function `nameinlist` says whether a particular `Name` is on a `Namelist`.

606a ⟨*value.c* 605b⟩+≡ ◁605c 606b▷

```
static int nameinlist(Name n, Namelist nl) {
    for (; nl; nl=nl->tl)
        if (n == nl->hd)
            return 1;
    return 0;
}
```

Function `addname` adds a name to a list, unless it's already there.

606b ⟨*value.c* 605b⟩+≡ ◁606a 606c▷

```
static Namelist addname(Name n, Namelist nl) {
    if (nameinlist(n, nl))
        return nl;
    return mkNL(n, nl);
}
```

Function `freevars` is passed an expression, a list of variables known to be bound, and a list of variables known to be free. If the expression contains free variables not on either list, `freevars` adds them to the free list and returns the new free list. Function `freevars` uses function `addfree` to do the work.

606c ⟨*value.c* 605b⟩+≡ ◁606b 607a▷

```
static Namelist addfree(Name n, Namelist bound, Namelist free) {
    if (nameinlist(n, bound))
        return free;
    return addname(n, free);
}
```

mkNL 𝒜

For CS45600, Purdue University, Spring 2014 only --- do nc

Computing the free variables of an expression is as much work as evaluating the expression. We have to know all the rules for environments.

607a ⟨*value.c* 605b⟩+≡ ◁606c 608b▷

```
static Namelist freevars(Exp e, Namelist bound, Namelist free) {
    Namelist nl;
    Explist el;

    switch (e->alt) {
    case LITERAL:
        break;
    case VAR:
        free = addfree(e->u.var, bound, free);
        break;
    case IFX:
        free = freevars(e->u.ifx.cond, bound, free);
        free = freevars(e->u.ifx.true, bound, free);
        free = freevars(e->u.ifx.false, bound, free);
        break;
    case WHILEX:
        free = freevars(e->u.whilex.cond, bound, free);
        free = freevars(e->u.whilex.body, bound, free);
        break;
    case BEGIN:
        for (el = e->u.begin; el; el = el->tl)
            free = freevars(el->hd, bound, free);
        break;
    case SET:
        free = addfree(e->u.set.name, bound, free);
        free = freevars(e->u.set.exp, bound, free);
        break;
    case APPLY:
        free = freevars(e->u.apply.fn, bound, free);
        for (el = e->u.apply.actuals; el; el = el->tl)
            free = freevars(el->hd, bound, free);
        break;
    case LAMBDAX:
        ⟨let free be the free variables for e->u.lambdax 607b⟩
        break;
    case LETX:
        ⟨let free be the free variables for e->u.letx 608a⟩
        break;
    ⟨extra cases for finding free variables in μScheme expressions 620f⟩
    }
    return free;
}
```

The case for lambda expressions is the interesting one. Any variables that are bound by the `lambda` are added to the "known bound" list for the recursive examination of the `lambda`'s body.

607b ⟨*let* free *be the free variables for* e->u.lambdax 607b⟩≡ (607a)

```
for (nl = e->u.lambdax.formals; nl; nl = nl->tl)
    bound = addname(nl->hd, bound);
free = freevars(e->u.lambdax.body, bound, free);
```

For CS45600, Purdue University, Spring 2014 only --- do nc

The let expressions are a bit tricky; we have to follow the rules exactly.

608a ⟨*let* free *be the free variables for* e->u.letx 608a⟩≡ (607a)

```
switch (e->u.letx.let) {
case LET:
    for (el = e->u.letx.el; el; el = el->tl)
        free = freevars(el->hd, bound, free);
    for (nl = e->u.letx.nl; nl; nl = nl->tl)
        bound = addname(nl->hd, bound);
    free = freevars(e->u.letx.body, bound, free);
    break;
case LETSTAR:
    for ( nl = e->u.letx.nl, el = e->u.letx.el
       ; nl && el
       ; nl = nl->tl, el = el->tl
       )
    {
        free  = freevars(el->hd, bound, free);
        bound = addname(nl->hd, bound);
    }
    free = freevars(e->u.letx.body, bound, free);
    break;
case LETREC:
    for (nl = e->u.letx.nl; nl; nl = nl->tl)
        bound = addname(nl->hd, bound);
    for (el = e->u.letx.el; el; el = el->tl)
        free = freevars(el->hd, bound, free);
    free = freevars(e->u.letx.body, bound, free);
    break;
}
```

We print a closure by printing the lambda expression, plus the values of the free variables that are not global variables. (If we included the global variables, we would be distracted by many bindings of `cons`, `car`, `+`, and so on.) Function `printnonglobals` does the hard work.

Recursive functions are represented by closures whose environments include pointers back to the recursive functions themselves. This means if we always print closures by printing the values of the free variables, the printer could loop forever. The `depth` parameter cuts off this loop, so eventually, when `depth` reaches 0, the printing functions print closures simply as `<procedure>`.

bound 607a
el 607a
fprint 33f
free 607a
freevars 607a
nl 607a
printnonglobals 610b

⟨*value.c* 605b⟩+≡ ◁607a 609a▷

```
static void printnonglobals(FILE *output, Namelist nl, Env env, int depth);

static void printclosureat(FILE *output, Lambda lambda, Env env, int depth) {
    if (depth > 0) {
        Namelist vars = freevars(lambda.body, lambda.formals, NULL);
        fprint(output, "<%\\, {", lambda);
        printnonglobals(output, vars, env, depth - 1);
        fprint(output, "}>");
    } else {
        fprint(output, "<procedure>");
    }
}
```

For CS45600, Purdue University, Spring 2014 only — do nc

609a ⟨*value.c* 605b⟩+≡ ◁608b 609b▷

```
void printclosure(FILE *output, va_list_box *box) {
    Lambda l = va_arg(box->ap, Lambda);
    Env e    = va_arg(box->ap, Env);
    printclosureat(output, l, e, 1);
}
```

The value-printing functions also need a depth parameter.

609b ⟨*value.c* 605b⟩+≡ ◁609a 609c▷

```
static void printvalueat(FILE *output, Value v, int depth);
⟨helper functions for printvalue 610a⟩
static void printvalueat(FILE *output, Value v, int depth) {
    switch (v.alt){
    case NIL:
        fprint(output, "()");
        return;
    case BOOL:
        fprint(output, v.u.bool ? "#t" : "#f");
        return;
    case NUM:
        fprint(output, "%d", v.u.num);
        return;
    case SYM:
        fprint(output, "%n", v.u.sym);
        return;
    case PRIMITIVE:
        fprint(output, "<procedure>");
        return;
    case PAIR:
        fprint(output, "(");
        printvalueat(output, *v.u.pair.car, depth);
        printtail(output, *v.u.pair.cdr, depth);
        return;
    case CLOSURE:
        printclosureat(output, v.u.closure.lambda, v.u.closure.env, depth);
        return;
    default:
        fprint(output, "<unknown v.alt=%d>", v.alt);
return;         assert(0);
    }
}
```

The default depth is 0; that is, by default the interpreter doesn't print closures at all. By changing this default depth, you can get more information.

fprint 33f

609c ⟨*value.c* 605b⟩+≡ ◁609b 610b▷

```
void printvalue(FILE *output, va_list_box *box) {
    printvalueat(output, va_arg(box->ap, Value), 0);
}
```

For CS45600, Purdue University, Spring 2014 only — do nc

Function printtail handles the correct printing of lists. If a cons cell doesn't have another cons cell or NIL in its cdr field, the car and cdr are separated by a dot.

610a ⟨*helper functions for* printvalue 610a⟩≡ (609b)

```
static void printtail(FILE *output, Value v, int depth) {
    switch (v.alt) {
    case NIL:
        fprint(output, ")");
        break;
    case PAIR:
        fprint(output, " ");
        printvalueat(output, *v.u.pair.car, depth);
        printtail(output, *v.u.pair.cdr, depth);
        break;
    default:
        fprint(output, " . ");
        printvalueat(output, v, depth);
        fprint(output, ")");
        break;
    }
}
```

Finally, the implementation of printnonglobals.

610b ⟨*value.c* 605b⟩+≡ ◁609c

```
Env *globalenv;
static void printnonglobals(FILE *output, Namelist nl, Env env, int depth) {
    char *prefix = "";
    for (; nl; nl = nl->tl) {
        Value *loc = find(nl->hd, env);
        if (loc && (globalenv == NULL || find(nl->hd, *globalenv) != loc)) {
            fprint(output, "%s%n -> ", prefix, nl->hd);
            prefix = ", ";
            printvalueat(output, *loc, depth);
        }
    }
}
```

B.2.3 Error checking

duplicatename 35d
error 35b
find 126b
fprint 33f
printvalueat 609b

Here are a few bits of error checking that were omitted from Chapter 3.

⟨*if* e->u.lambdax.formals *contains a duplicate, call* error 610c⟩≡ (130a)

```
if (duplicatename(e->u.lambdax.formals) != NULL)
    error("formal parameter %n appears twice in lambda",
          duplicatename(e->u.lambdax.formals));
```

610d ⟨*if* e->u.letx.nl *contains a duplicate, complain of error in* let 610d⟩≡ (131b)

```
if (duplicatename(e->u.letx.nl) != NULL)
    error("bound name %n appears twice in let", duplicatename(e->u.letx.nl));
```

610e ⟨*if* e->u.letx.nl *contains a duplicate, complain of error in* letrec 610e⟩≡ (131b)

```
if (duplicatename(e->u.letx.nl) != NULL)
    error("bound name %n appears twice in letrec", duplicatename(e->u.letx.nl));
```

For CS45600, Purdue University, Spring 2014 only -- do nc

611a ⟨*if* d->u.define.lambda.formals *contains a duplicate, call* error 611a⟩≡ (134c)

```
if (duplicatename(d->u.define.lambda.formals) != NULL)
    error("formal parameter %n appears twice in definition of function %n",
          duplicatename(d->u.define.lambda.formals), d->u.define.name);
```

B.2.4 Expressions

Here is the (boring) code that prints abstract-syntax trees.

611b ⟨*ast.c* 611b⟩≡ 611c▷

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

611c ⟨*ast.c* 611b⟩+≡ ◁611b 612▷

```
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 %\\)", d->u.define.name, d->u.define.lambda);
        break;
    case USE:
        fprint(output, "(use %n)", d->u.use);
        break;
    default:
        assert(0);
    }
}
```


For CS45600, Purdue University, Spring 2014 only --- do nc

612 ⟨*ast.c* 611b⟩+≡ ◁611c 613a▷

```
static void printlet(FILE *output, Exp let) {
    Namelist nl;
    Explist el;

    switch (let->u.letx.let) {
    case LET:
        fprint(output, "(let (");
        break;
    case LETSTAR:
        fprint(output, "(let* (");
        break;
    case LETREC:
        fprint(output, "(letrec (");
        break;
    default:
        assert(0);
    }
    for (nl = let->u.letx.nl, el = let->u.letx.el;
         nl && el;
         nl = nl->tl, el = el->tl)
        fprint(output, "(%n %e)%s", nl->hd, el->hd, nl->tl?" ":"");
    fprint(output, ") %e)", let->u.letx.body);
}
```

fprint 33f

For CS45600, Purdue University, Spring 2014 only --- do nc

613a ⟨*ast.c* 611b⟩+≡ ◁612 613b▷

```
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:
        if (e->u.literal.alt == NUM || e->u.literal.alt == BOOL)
            fprint(output, "%v", e->u.literal);
        else
            fprint(output, "'%v", e->u.literal);
        break;
    case VAR:
        fprint(output, "%n", e->u.var);
        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.body);
        break;
    case BEGIN:
        fprint(output, "(begin%s%E)", e->u.begin ? " " : "", e->u.begin);
        break;
    case SET:
        fprint(output, "(set %n %e)", e->u.set.name, e->u.set.exp);
        break;
    case LETX:
        printlet(output, e);
        break;
    case LAMBDAX:
        fprint(output, "%\\", e->u.lambdax);
        break;
    case APPLY:
        fprint(output, "(%e%s%E)", e->u.apply.fn,
               e->u.apply.actuals ? " " : "", e->u.apply.actuals);
        break;
    ⟨extra cases for printing μScheme ASTs 620e⟩
    }
}
```

fprint 33f

613b ⟨*ast.c* 611b⟩+≡ ◁613a

```
void printlambda(FILE *output, va_list_box *box) {
    Lambda l = va_arg(box->ap, Lambda);
    fprint(output, "(lambda (%N) %e)", l.formals, l.body);
}
```

For CS45600, Purdue University, Spring 2014 only — do nc

B.2.5 Parsing

This is really *not* interesting.

614a ⟨*parse.c* 614a⟩≡ 614b▷
```
#include "all.h"

⟨parse.c declarations 615d⟩
```

614b ⟨*parse.c* 614a⟩+≡ ◁614a 615e▷
```
Def parse(Par p) {
    switch (p->alt) {
    case ATOM:
        ⟨parse ATOM and return 614c⟩
    case LIST:
        ⟨parse LIST and return 614d⟩
    }
    assert(0);
    return NULL;
}
```

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

614c ⟨*parse ATOM and return* 614c⟩≡ (614b)
```
return mkExp(parseexp(p));
```

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

614d ⟨*parse LIST and return* 614d⟩≡ (614b)
```
{
    Parlist pl = p->u.list;

    if (pl == NULL)
        error("%p: empty list", p);
    if (nthPL(pl, 0)->alt == ATOM) {
        Name first = nthPL(pl, 0)->u.atom;
        if (first == strtoname("define")) {
            ⟨parse define and return 615a⟩
        }
        if (first == strtoname("val")) {
            ⟨parse val and return 615b⟩
        }
        if (first == strtoname("use")) {
            ⟨parse use and return 615c⟩
        }
    }

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

error	35b
mkExp	$\mathcal{A}$
nthPL	$\mathcal{A}$
parseexp	616c
strtoname	28d

For CS45600, Purdue University, Spring 2014 only --- do nc

Parsing the toplevel expressions requires checking the argument counts and then parsing the subpieces.

615a ⟨*parse define and return* 615a⟩≡ (614d)

```
Name name;
Lambda l;

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

name      = nthPL(pl, 1)->u.atom;
l.formals = getnamelist(p, nthPL(pl, 2)->u.list);
l.body    = parseexp(nthPL(pl, 3));
return      mkDefine(name, l);
```

615b ⟨*parse val and return* 615b⟩≡ (614d)

```
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);
```

615c ⟨*parse use and return* 615c⟩≡ (614d)

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

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

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

615d ⟨*parse.c declarations* 615d⟩≡ (614a 595c) 615f▷

```
Namelist getnamelist(Par p, Parlist pl);
```

615e ⟨*parse.c* 614a⟩+≡ ◁614b 616a▷

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

error	35b
lengthPL	$\mathcal{A}$
mkDefine	$\mathcal{A}$
mkNL	$\mathcal{A}$
mkUse	$\mathcal{A}$
mkVal	$\mathcal{A}$
nthPL	$\mathcal{A}$
parseexp	616c
pl	614d

Now we can move on to parsing `Exp`s. The `parselist` helper function parses a list of `Par` expressions and calls `parseexp` repeatedly to return a list of `Exp`s.

615f ⟨*parse.c declarations* 615d⟩+≡ (614a 595c) ◁615d 616b▷

```
Explist parselist(Parlist);
```

For CS45600, Purdue University, Spring 2014 only --- do nc

616a ⟨*parse.c* 614a⟩+≡ ◁615e 616c▷

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

    if (pl == NULL)
        return NULL;

    e = parseexp(pl->hd);
    return mkEL(e, parselist(pl->tl));
}
```

616b ⟨*parse.c declarations* 615d⟩+≡ (614a 595c) ◁615f 618b▷

```
Exp parseexp(Par);
```

616c ⟨*parse.c* 614a⟩+≡ ◁616a 618c▷

```
Exp parseexp(Par p) {
    switch (p->alt) {
    case ATOM:
        ⟨parseexp ATOM and return 616d⟩
    case LIST:
        ⟨parseexp LIST and return 617a⟩
    default:
        assert(0);
        return NULL;
    }
}
```

To parse an atom, we need to check whether it is a boolean or a number. Otherwise it is a variable.

616d ⟨*parseexp* ATOM *and return* 616d⟩≡ (616c)

```
{
    Name n = p->u.atom;
    const char *s; /* string form of n */
    char *t;       /* nondigits in s, if any */
    long l;        /* number represented by s, if any */

    if (n == strtoname("#t"))
        return mkLiteral(truev);
    else if (n == strtoname("#f"))
        return mkLiteral(falsev);

    s = nametostr(n);
    l = strtol(s, &t, 10);
    if (*t == '\0' && *s != '\0') /* all the characters in s are digits base 10 */
        return mkLiteral(mkNum(l));
    else
        return mkVar(n);
}
```

falsev	127b
mkEL	𝒜
mkLiteral	𝒜
mkNum	𝒜
mkVar	𝒜
nametostr	28d
strtol	ℬ
strtoname	28d
truev	127b

For CS45600, Purdue University, Spring 2014 only — do nc

If we have a list, we need to see if the first element is a special name. In anticipation of code we need for Chapter 4, we put the parsed expression into `rv` (for "return value") instead of returning it directly. We treat `lambda`, the let family, and `quote` specially, because their arguments are not always expressions. For other phrases, the arguments are almost always expressions, so we call `parselist`. This means we cheat a bit on `set`, but so be it.

617a ⟨*parseexp* LIST *and return* 617a⟩≡ (616c)

```
{
    Parlist pl;                 /* parenthesized list we are parsing */
    Name first;                 /* first element, as a name (or NULL if not name) */
    Explist el;                 /* remaining elements, as expressions */
    Exp rv;                     /* result of parsing */

    pl = p->u.list;
    if (pl == NULL)
        error("%p: empty list in input", p);

    first = pl->hd->alt == ATOM ? pl->hd->u.atom : NULL;
    if (first == strtoname("lambda")) {
        ⟨parseexp lambda and put the result in rv 617b⟩
    } else if (first == strtoname("let")
           ||  first == strtoname("let*")
           ||  first == strtoname("letrec")) {
        ⟨parseexp let and put the result in rv 618a⟩
    } else if (first == strtoname("quote")) {
        ⟨parseexp quote and put the result in rv 618d⟩
    } else {
        el = parselist(pl->tl);
        if (first == strtoname("begin")) {
            ⟨parseexp begin and put the result in rv 619d⟩
        } else if (first == strtoname("if")) {
            ⟨parseexp if and put the result in rv 619e⟩
        } else if (first == strtoname("set")) {
            ⟨parseexp set and put the result in rv 620a⟩
        } else if (first == strtoname("while")) {
            ⟨parseexp while and put the result in rv 619f⟩
        } else {
            ⟨parseexp application and put the result in rv 619c⟩
        }
    }
    return rv;
}
```

error	35b	
lengthPL	𝒜	
mkLambda	𝒜	
mkLambdax	𝒜	
nthPL	𝒜	
parseexp	616c	
pl	639	
rv	639	
strtoname	28d	

617b ⟨*parseexp* lambda *and put the result in* rv 617b⟩≡ (617a)

```
Par q;

if (lengthPL(pl->tl) != 2)
    error("%p: usage: (lambda (formals) exp)", p);
q = nthPL(pl->tl, 0);
if (q->alt != LIST)
    error("%p: usage: (lambda (formals) exp)", p);
rv = mkLambdax(mkLambda(getnamelist(p, q->u.list), parseexp(nthPL(pl->tl, 1))));
```

For CS45600, Purdue University, Spring 2014 only --- do nc

618a ⟨*parseexp let and put the result in* rv 618a⟩≡ (617a)

```
Letkeyword letword;
Par letbindings;

if (first == strtoname("let"))
    letword = LET;
else if (first == strtoname("let*"))
    letword = LETSTAR;
else if (first == strtoname("letrec"))
    letword = LETREC;
else
    assert(0);

if (lengthPL(pl->tl) != 2)
    error("%p: usage: (%n (letlist) exp)", p, first);

letbindings = nthPL(pl->tl, 0);
if (letbindings->alt != LIST)
    error("%p: usage: (%n (letlist) exp)", p, first);

rv = mkLetx(letword, NULL, NULL, parseexp(nthPL(pl->tl, 1)));

parseletbindings(p, letbindings->u.list, rv);
```

Function parseletbindings adds bindings to a let expression.

⟨*parse.c declarations* 615d⟩+≡ (614a 595c) ◁616b 618e▷

```
static void parseletbindings(Par p, Parlist bindings, Exp letexp);
```

⟨*parse.c* 614a⟩+≡ ◁616c 619a▷

```
static void parseletbindings(Par p, Parlist bindings, Exp letexp) {
    if (bindings) {
        Par t = bindings->hd;
        Name n;  /* name bound in t (if t is well formed) */
        Exp e;   /* expression on RHS of t (if t is well formed) */
        parseletbindings(p, bindings->tl, letexp);
        if (t->alt != LIST || lengthPL(t->u.list) != 2
        ||  nthPL(t->u.list, 0)->alt != ATOM)
            error("%p: usage: (letX (letlist) exp)", p);
        n = nthPL(t->u.list, 0)->u.atom;
        e = parseexp(nthPL(t->u.list, 1));
        letexp->u.letx.nl = mkNL(n, letexp->u.letx.nl);
        letexp->u.letx.el = mkEL(e, letexp->u.letx.el);
    }
}
```

618d ⟨*parseexp* quote *and put the result in* rv 618d⟩≡ (617a)

```
{
    if (lengthPL(pl) != 2)
        error("%p: quote needs exactly one argument", p);
    rv = mkLiteral(parsesx(nthPL(pl, 1)));
}
```

618e ⟨*parse.c declarations* 615d⟩+≡ (614a 595c) ◁618b

```
Value parsesx(Par p);
```


error 35b
first 617a
lengthPL A
mkEL A
mkLetx A
mkLiteral A
mkNL A
nthPL A
parseexp 616c
parsesx 619a
pl, in μScheme 617a
in μScheme (in GC?!) 639
rv, in μScheme 617a
in μScheme (in GC?!) 639
strtoname 28d

For CS45600, Purdue University, Spring 2014 only --- do nc

619a ⟨*parse.c* 614a⟩+≡ ◁618c 620b▷

```
Value parsesx(Par p) {
    switch (p->alt) {
    case ATOM:
        {
            Name n        = p->u.atom;
            const char *s = nametostr(n);
            long l;           /* value of digits in s, if any */
            char *t;          /* first nondigit in s */

            l = strtol(s, &t, 10);
            if (*t == '\0' && *s != '\0')  /* s is all digits */
                return mkNum(l);
            else if (strcmp(s, "#t") == 0)
                return truev;
            else if (strcmp(s, "#f") == 0)
                return falsev;
            else if (strcmp(s, ".") == 0)
                error("this interpreter cannot handle . in quoted S-expressions");
            else
                return mkSym(n);
        }
    case LIST:
        ⟨parsesx LIST and return 619b⟩
    }
    assert(0);
    return falsev;
}
```

619b ⟨*parsesx* LIST *and return* 619b⟩≡ (619a)

```
if (p->u.list == NULL)
    return mkNil();
else
    return mkPair(allocate(parsesx(p->u.list->hd)),
                  allocate(parsesx(mkList(p->u.list->tl))));
```

Parsing function application.

619c ⟨*parseexp application and put the result in* rv 619c⟩≡ (617a)

```
rv = mkApply(parseexp(pl->hd), el);
```

A begin statement can have any number of parameters.

619d ⟨*parseexp* begin *and put the result in* rv 619d⟩≡ (617a)

```
rv = mkBegin(el);
```

An if statement needs three parameters.

619e ⟨*parseexp* if *and put the result in* rv 619e⟩≡ (617a)

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

A while loop needs two.

619f ⟨*parseexp* while *and put the result in* rv 619f⟩≡ (617a)

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


For CS45600, Purdue University, Spring 2014 only — do nc

A set statement requires a variable and a value.

620a ⟨*parseexp* set *and put the result in* rv 620a⟩≡ (617a)

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

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

620b ⟨*parse.c* 614a⟩+≡ ◁619a 620c▷

```
struct Defreader {
    int doprompt;  /* whether to prompt at each definition */
    Reader r;      /* underlying reader of Pars */
    Parlist pl;    /* Pars read but not yet parsed */
};
```

620c ⟨*parse.c* 614a⟩+≡ ◁620b 620d▷

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

    if (dr->pl == NULL) {
        dr->pl = readparlist(dr->r, 1, dr->doprompt);
        if (dr->pl == NULL)
            return NULL;
    }

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

⟨*parse.c* 614a⟩+≡ ◁620c

```
Defreader defreader(Reader r, int doprompt) {
    Defreader dr;

    dr = malloc(sizeof(*dr));
    assert(dr != NULL);

    dr->r        = r;
    dr->doprompt = doprompt;
    dr->pl       = NULL;
    return dr;
}
```


B.2.6 Future work

These empty definitions are placeholders for future work in which Chapter 4 will be split into two chapters: one on a new interpreter for μScheme that uses a stack-based abstract machine, and one on a garbage collector that works with that new interpreter.

620e ⟨*extra cases for printing μScheme ASTs* 620e⟩≡ (613a)

620f ⟨*extra cases for finding free variables in μScheme expressions* 620f⟩≡ (607a)

For CS45600, Purdue University, Spring 2014 only --- do nc