Interpreters from scratch

What if Scheme is not available?

Then we need to implement Scheme.

We will let the to-do list based interpreter be our guide.

Tradeoff: Performance vs. Ease of implementation

Tradeoff: Compiler vs. Runtime system

Plan

Decide what language to implement.

Figure out the tradeoffs.

Implement it in C or assembly.

Scheme

```
expr = number
     | null
     | void
     | #f
     | (lambda (id ...) expr)
     | id
     | (let ((id expr) ...) expr)
     | (letrec ((id expr) ...) expr)
     | (expr expr ...)
     | (if expr expr expr)
     | (begin expr expr ...)
     | (set! id expr)
     | (and expr ...)
     | (or expr ...)
     | (not expr)
```

Tradeoffs

We have to have some compiler and some run-time system.

The runtime system supports at least garbage collection and primitive operations

The compiler will at least need to translate variables to lexical addresses.

An interpreter for Scheme will be easier to build than a compiler that translates Scheme to machine code.

Simplify

Scheme has quite a few different kinds of expressions to evaluate.

Let’s remove some.

How about let, letrec and begin?

Is this a very useful language?

Simplify - let

let is used to give names to values.

let’s evaluator looks something like this:

```
(let-exp (binding-ids bound-exprs body)
  (eval body (extend-env
               binding-ids
               (map (lambda (x) (eval x env))
                    bound-exprs)))))
```

Does this look similar to something else?

```
(eval body (extend-env ids args env))
```

Simplify - let

let is a bit like lambda.

Since we have lambda, perhaps we don’t need let.

Let’s try to write this program without let.

```
(let ((a (+ (c 1) 2))
      (b (+ (c 2) 2)))
  (+ a b))

(lambda (a b) (+ a b))
```

Simplify - let

let is a bit like lambda.

Since we have lambda, perhaps we don’t need let.

Let’s try to write this program without let.

```
(let ((a (+ (c 1) 2))
      (b (+ (c 2) 2)))
  (+ a b))

((lambda (a b) (+ a b))
 (+ (c 1) 2)
 (+ (c 2) 2))
```

Simplify - let -> lambda

We will implement a compiler that will translate let to lambda

(let ((id bound) ...) body) -> ((lambda (id ...) body) bound ...)

This is easy to do in the same compiler that computes variable addresses.

ASIDE: Scheme supports writing of this sort of compiler for Scheme itself with macros.

Simplify - begin

Begin evaluates a sequence of expressions.

Function application evaluates a sequence of expressions - its arguments.

(begin exp ... last-exp) -> ((lambda (dummy-id ...) last-exp) exp ...)

What if function application did not evaluate its arguments in order?

Simplify - letrec

Syntactically, letrec looks just like let.

Can we compile letrec to let?

Simplify - letrec -> let

An example.

```
(letrec ((a (lambda () (b)))
         (b (lambda () (a))))
  (a))
```

```
(let ((a (lambda () (b)))
      (b (lambda () (a))))
  (a))
```

Nope.

Simplify - letrec -> let

Scoping is more complicated for letrec than let.

We have to use let (lambda) to introduce the bindings.

```
(let ((a ...) (b ...)) ...)
```

What now?

After the bindings are introduced, we can put whatever values into them we want.

Simplify - letrec -> lambda

```
(let ((a void) (b void))
  (begin
    (set! a (lambda () (b)))
    (set! b (lambda () (a)))
    (a)))
```

(letrec ((id bound) ...) body) -> (let ((id void) ...) (set! id bound) ... body)

Simplify

We could translate cond to if.

We could get rid of and and or this way.

(and e1 e2 e3 e4) -> (if e1 (if e2 (if e3 e4 #f) #f) #f)

(or e1 e2 e3 e4) -> (if e1 e1 (if e2 e2 (if e3 e3 e4)))

Is this correct?

(or e1 e2) -> (let ((i1 e1)) (if i1 i1 e2)))

This is a more difficult compiler to implement.

Plan Revisited

Decide what language to implement.

Figure out the tradeoffs.

Implement it in C or assembly.
- Datatypes
- Code

Datatypes

```
typedef struct { int tag; val *v; } val_cexp;
typedef struct { int tag; int num_parms;
                 cexp *body; } lambda_cexp;
typedef struct { int tag; int dist;
                 int offset; } lexvar_cexp;
typedef struct { int tag; int num_args;
                cexp *fun; cexp **args; } app_cexp;
typedef struct { int tag; cexp *test;
                 cexp *t_br; cexp *f_br; } if_cexp;
typedef struct { int tag; int dist;
                 int offset; cexp *val; }
               lexsetbang_cexp;
typedef struct { int tag; int num_args;
                 cexp **args; } and_cexp;
```

Datatypes

```
cexp *make_app_cexp(cexp *f, int n, cexp **a)
{
  app_cexp *res = (app_cexp*)ALLOC(sizeof(app_cexp));
  res->tag = APP_CEXP;
  res->fun = f;
  res->num_args = n;
  res->args = a;
  return (cexp*)res;
}
```

Datatypes

```
env *make_extended_env(int i, char **c,
                       val **v, env *ev)
{
  extended_env *e =
    (extended_env*)ALLOC(sizeof(extended_env));
  e->tag = EXTENDED_ENV;
  e->num_bindings = i;
  e->syms = c;
  e->vals = v;
  e->next = ev;
  return (env*)e;
}
```

Code Example

Translate a simple Scheme program to C

even-odd.scm to even-odd.c

even? and odd? are like eval and apply_cont

Skeleton Code

```
cexp *exp_arg;
env *env_arg;
...
int main(int argc, char *argv
{
  exp_arg = compile(...);
  env_arg = make_extended_env(...);
  cont_arg = make_done_cont();
 cont_eval_exp:
  switch (exp_arg->tag)
  ...
 apply_cont:
  switch (cont_arg->tag)
  ...
 cons_prim:
  ...
}
```

Eval Code

```
case LAMBDA_CEXP:
  val_arg =
    make_closure(((lambda_cexp*)exp_arg)->num_parms,
                 ((lambda_cexp*)exp_arg)->body,
                 env_arg);
  goto apply_cont;

 case APP_CEXP:
   cont_arg =
     make_narg_cont(((app_cexp*)exp_arg)->num_args,
                    ((app_cexp*)exp_arg)->args,
                    env_arg,
                    cont_arg);
   exp_arg = ((app_cexp*)exp_arg)->fun;
   goto cont_eval_exp;
```

Apply_cont Code

```
case FARG_CONT:
{
  ...
  val *fun = ((farg_cont*)cont_arg)->fun;
  switch (fun->tag) {
  case CLOSURE:
    ...
    exp_arg = ((closure*)fun)->body;
    env_arg =
      make_extended_env(index, NULL,
             (val**)((farg_cont*)cont_arg)->args,
                        ((closure*)fun)->cenv);
    cont_arg = ((farg_cont*)cont_arg)->c;
    goto cont_eval_exp;
  ...
}
```

Apply_cont Code

```
case FARG_CONT:
{
  ...
  val *fun = ((farg_cont*)cont_arg)->fun;
  switch (fun->tag) {
  case PRIM_FUNC:
    ...
    prim_args = (val**)((farg_cont*)cont_arg)->args;
    cont_arg = ((farg_cont*)cont_arg)->c;
    goto goto *(((prim_func*)fun)->prim_addr);
  ...
}
```

Primitive code

```
    cons_prim:
     val_arg =
       make_pair_val(prim_args[0], prim_args[1]);
     goto apply_cont;

 numplus_prim:
  ...
  val_arg =
    make_number_val(((number_val*)prim_args[0])->n +
                    ((number_val*)prim_args[1])->n);
  goto apply_cont;
```

Success

Now we can program in a small subset of Scheme.

What if we want to handle more of Scheme?

What if we want to add a new feature to Scheme?

Adding a Function

Add a primitive by writing it in C and adding it to the initial environment.

Of course Scheme is already good at defining functions.

What if we want to add something that is not a function (like cond)?

What if we want static error detection?

Modify the compiler.

This is not a very flexible solution.

Macros

A program defines its own compiler extension.

```
(define-syntax let
  (syntax-rules ()
    ((_ ((id bound) ...) body)
     ((lambda (id ...) body) bound ...))))
```

let.scm

Hygiene

```
(define-syntax or
  (syntax-rules ()
    ((_) #f)
    ((_ e) e)
    ((_ e1 e2 e3 ...)
     (let ((t e1)) (if t t (or e2 e3 ...)))
```

Hygeine

```
(or e1 e2 e3 e4)
->
(let ((t e1)) (if t t (or e2 e3 e4)))
->
(let ((t e1)) (if t t (let ((t e2)) (if t t (or e3)))
->
(let ((t e1)) (if t t (let ((t e2)) (if t t e3))))
```

or.scm

EOPL

define-datatype and cases are defined by macros.

Syntax-case macros can perform arbitrary computation at compile time to build the compiled code

cases.scm

The Future

Implementing your own "Little Langage"

Powerful macro systems in languages other than Scheme