P4. “Spla in Spla”

Kelley van Evert
s4046854

Spla

- P1..P3 → Splb: “Simple C-like stuff...”
  - i.e.
    - first-order functions at the toplevel
    - lexical scope, by block (function body, if, while, { ... })
    - follow the standard C-like stack discipline
    - lists and tuples by reference, on the heap

- P4 → Spla: **whatever needed to do the language thing**
  - Original goal: parse itself
  - Revised goal: ?

Language overview

- “whatever needed to do the language thing”
  - Parser combinators → **(1)** higher order functions
  - Somewhat misc. expressivity
    - **(2)** Nested function definitions, as expressions
    - Strings, chars:
      - **(3)** Type aliases, e.g. `type @char = int; type @string = [@char];`
        (which really don't amount to anything interesting)
      - Slightly extend SSM to include “trapchr” instruction, then just `(map print str)`
  - Unneccessary semantic enhancements
    - Partial function application
    - Lazy evaluation, though this is a subtle point
    - Lists/tuples/functions by value

Variable binding

- Binding order etc?
  - “Just do something”
    - shadow new variables at block level, only allow access to initialized variables
  - Mutual recursion?
    - Nope; just let the programmer write:

```
(int → bool, int → bool) is_even_odd = (
  fun (n) { … },
  fun (n) { … }
);
(int → bool) is_even = is_even_odd.fst();
(int → bool) is_odd  = is_even_odd.snd();
```

(Note that function definition and application can't occur in a single expression.)

    - Hack:
      - Treat sequential declarations on functions and simple things (i.e. no applications) as such.

Blocks / lexical scoping

- Blocks / lexical scopes
  - Per `let`, `{ … }`, function body, and global
- Introduction of lexical contexts:
  - Global @ program execution
  - `let`, `{ … }` @ entry
  - Function body @ function application
- We have to keep all these contexts
  - Because variables can outlive execution of scopes
  - Instead of on the stack, we keep them on the heap

Blocks / lexical scoping

- Context layout:

1. “Function context?” flag
2. Parent context
3. Return context (to be restored after exit)
4... locals

Blocks / lexical scoping

- Obviously wildly inefficient (in time and space)
- Is it a problem?
  - For “seriously” parsing Spla itself, somewhat
  - I had to hack SSM to have 10x as much heap to just parse simple things like `int x = 10;`
- What can we do?
  - Lambda-lift instead *– but I didn't*
  - Remove contexts without variable declarations (at any depth) after execution
    - Put them on the stack *– I tried this, but such a hassle*
  - Compile to C++ (with lambda lifting), x86 or LLVM instead of SSM

    – am doing this (x86) at the moment

Some more pragmatics

- Intermediate “intstructions”, i.e. macros
  - `enter_ctxt n :: n → 0`
    - *Create and enter new lexical context with top n stack cells as contents – cf.* `link n`
  - `exit_ctxt :: 0 → 0`
    - *Exit/restore lexical context – cf.* `unlink`
  - `lex_load u i :: 0 → 1`
    - Get variable #1, u contexts up
  - `lex_store u i :: 1 → 0`
    - Store variable #1, u contexts up

Some more pragmatics

- Intermediate “intstructions”, i.e. macros
  - `ret' :: 0 → 0`
    - *Jumps to runtime code that traverses up the context tree until jumping out of function context*

Returning to the original goal

- (Parsing Spla in Spla)
- Maybe I'll just:
  - parse expressions (same thing, but less)
  - or type-check (less computationally intensive, but the same HO functions and monads etc.)

```
type @parser s a = s -> [(a, s)];
type @char       = int;
type @string     = [@char];

// List operations
// ===============

([t] -> int) length = fun (list) {
    if (isEmpty(list)) {
      return 0;
    } else {
      return 1 + length(list.tl);
    }
  };

([t] -> [t] -> [t]) concat = fun (a, b) {
    if (isEmpty(a)) {
      return b;
    } else {
      return a.hd : concat(a.tl, b);
    }
  };

([[t]] -> [t]) concat_many = fun (lists) {
    if (isEmpty(lists)) {
      return [];
    } else {
      return concat(lists.hd,
        concat_many(lists.tl));
    }
  };

((a -> b) -> [a] -> [b]) map =
  fun (f, list) {
    if (isEmpty(list)) {
      return [];
    } else {
      return f(list.hd) : map(f, list.tl);
    }
  };

((a -> b -> a) -> a -> [b] -> a) foldl =
  fun (f, e, list) {
    if (isEmpty(list)) {
      return e;
    } else {
      return foldl(f, f(e, list.hd), list.tl);
    }
  };

([a] -> [a]) reverse = fun (list) {
    [a] rev = [];
    while (!isEmpty(list)) {
      rev = list.hd : rev;
      list = list.tl;
    }
    return rev;
  };

// The parser monad
// ================

(a -> @parser s a) mreturn = fun (e) {
    return fun (input) {
      return (e, input) : [];
    };
  };

(@parser s a -> (a -> @parser s b) ->
 @parser s b) mbind = fun (p, f) {
    return fun (input) {
      return concat_many(map(fun (r) {
        (@parser s b) g = f(r.fst);
        return g(r.snd);
      }, p(input)));
    };
  };

(@parser s a) mzero = fun (input) {
    return [];
  };

(@parser s a -> @parser s a ->
 @parser s a) mplus = fun (p, q) {
    return fun(input) {
      return concat(p(input), q(input));
    };
  };

([@parser s a] -> @parser s a) mplus_list =
  fun (parsers) {
    return foldl(mplus, mzero, parsers);
  };

// List input parsers
// ==================

(@parser [t] t) next = fun (input) {
    return (input.hd, input.tl) : [];
  };

((t -> bool) -> @parser [t] t) sat =
  fun (P) {
    return fun (input) {
      if (P(input.hd)) {
        return next(input);
      } else {
        return [];
      }
    };
  };

(t -> @parser [t] t) element = fun (e1) {
    return sat(fun (e2) {
      return (e1 == e2);
    });
  };

// String input parsers
// ====================

(@parser (@string) (@char)) digit = sat(fun (c) {
    return c >= 48 && c <= 57;
  });

(@parser (@string) (@char)) lower = sat(fun (c) {
    return c >= 97 && c <= 122;
  });

(@parser (@string) (@char)) upper = sat(fun (c) {
    return c >= 65 && c <= 90;
  });

(@parser (@string) (@char)) alpha =
  mplus(lower, upper);

(@parser (@string) (@char)) alphanum =
  mplus(alpha, digit);

(@string -> @parser (@string) (@string)) pstring =
  fun (match) {
    if (isEmpty(match)) {
      return mreturn([]);
    } else {
      return mbind(element(match.hd), fun (x) {
        return mbind(pstring(match.tl), fun (x) {
          return mreturn(match);
        });
      });
    }
  };

([@string] -> @parser (@string) (@string)) pstring_any =
  fun (match_any) {
    return mplus_list(map(pstring, match_any));
  };
```

(Most of) the parser combinator library

(Des questions)