SOFTWARE FOUNDATIONS

VOLUME 1: LOGICAL FOUNDATIONS

TABLE OF CONTENTS INDEX ROADMAP

IMPPARSER

LEXING AND PARSING IN COQ

The development of the Imp language in `Imp.v` completely ignores issues of concrete syntax — how an ascii string that a programmer might write gets translated into abstract syntax trees defined by the datatypes `aexp`, `bexp`, and `com`. In this chapter, we illustrate how the rest of the story can be filled in by building a simple lexical analyzer and parser using Coq's functional programming facilities.

It is not important to understand all the details here (and accordingly, the explanations are fairly terse and there are no exercises). The main point is simply to demonstrate that it can be done. You are invited to look through the code — most of it is not very complicated, though the parser relies on some "monadic" programming idioms that may require a little work to make out — but most readers will probably want to just skim down to the Examples section at the very end to get the punchline.

```
Set Warnings "-notation-overridden,-parsing".
Require Import Coq.Strings.String.
Require Import Coq.Strings.Ascii.
Require Import Coq.Arith.Arith.
Require Import Coq.Arith.EqNat.
Require Import Coq.Lists.List.
Import ListNotations.
Require Import Maps Imp.
```

Internals

Lexical Analysis

```
Definition isWhite (c : ascii) : bool :=
  let n := nat_of_ascii c in
  orb (orb (beq_nat n 32) (* space *)
           (beq_nat n 9)) (* tab *)
```

```
      (orb (beq_nat n 10) (* linefeed *)
           (beq_nat n 13)). (* Carriage return. *)

Notation "x '<=?' y" := (leb x y)
  (at level 70, no associativity) : nat_scope.

Definition isLowerAlpha (c : ascii) : bool :=
  let n := nat_of_ascii c in
    andb (97 <=? n) (n <=? 122).

Definition isAlpha (c : ascii) : bool :=
  let n := nat_of_ascii c in
    orb (andb (65 <=? n) (n <=? 90))
        (andb (97 <=? n) (n <=? 122)).

Definition isDigit (c : ascii) : bool :=
  let n := nat_of_ascii c in
     andb (48 <=? n) (n <=? 57).

Inductive chartype := white | alpha | digit | other.

Definition classifyChar (c : ascii) : chartype :=
  if isWhite c then
    white
  else if isAlpha c then
    alpha
  else if isDigit c then
    digit
  else
    other.

Fixpoint list_of_string (s : string) : list ascii :=
  match s with
  | EmptyString ⇒ []
  | String c s ⇒ c :: (list_of_string s)
  end.

Fixpoint string_of_list (xs : list ascii) : string :=
  fold_right String EmptyString xs.

Definition token := string.

Fixpoint tokenize_helper (cls : chartype) (acc xs : list ascii)
                       : list (list ascii) :=
  let tk := match acc with [] ⇒ [] | _::_ ⇒ [rev acc] end in
  match xs with
  | [] ⇒ tk
  | (x::xs') ⇒
    match cls, classifyChar x, x with
    | _, _, "(" ⇒
      tk ++ ["("]::(tokenize_helper other [] xs')
    | _, _, ")" ⇒
      tk ++ [")"]::(tokenize_helper other [] xs')
    | _, white, _ ⇒
      tk ++ (tokenize_helper white [] xs')
    | alpha,alpha,x ⇒
      tokenize_helper alpha (x::acc) xs'
    | digit,digit,x ⇒
      tokenize_helper digit (x::acc) xs'
```

```
      | other,other,x ⇒
        tokenize_helper other (x::acc) xs'
      | _,tp,x ⇒
        tk ++ (tokenize_helper tp [x] xs')
      end
    end %char.

Definition tokenize (s : string) : list string :=
  map string_of_list (tokenize_helper white [] (list_of_string
s)).

Example tokenize_ex1 :
    tokenize "abc12=3 223*(3+(a+c))" %string
  = ["abc"; "12"; "="; "3"; "223";
       "*"; "("; "3"; "+"; "(";
       "a"; "+"; "c"; ")"; ")"]%string.
```

\+

Parsing

Options With Errors

An `option` type with error messages:

```
Inductive optionE (X:Type) : Type :=
  | SomeE : X → optionE X
  | NoneE : string → optionE X.

Arguments SomeE {X}.
Arguments NoneE {X}.
```

Some syntactic sugar to make writing nested match-expressions on optionE more convenient.

```
Notation "'DO' ( x , y ) <== e1 ; e2"
   := (match e1 with
         | SomeE (x,y) ⇒ e2
         | NoneE err ⇒ NoneE err
       end)
   (right associativity, at level 60).

Notation "'DO' ( x , y ) <-- e1 ; e2 'OR' e3"
   := (match e1 with
         | SomeE (x,y) ⇒ e2
         | NoneE err ⇒ e3
       end)
   (right associativity, at level 60, e2 at next level).
```

Generic Combinators for Building Parsers

```
Open Scope string_scope.

Definition parser (T : Type) :=
  list token → optionE (T * list token).
```

```
Fixpoint many_helper {T} (p : parser T) acc steps xs :=
  match steps, p xs with
  | 0, _ ⇒
      NoneE "Too many recursive calls"
  | _, NoneE _ ⇒
      SomeE ((rev acc), xs)
  | S steps', SomeE (t, xs') ⇒
      many_helper p (t::acc) steps' xs'
  end.
```

A (step-indexed) parser that expects zero or more ps:

```
Fixpoint many {T} (p : parser T) (steps : nat) : parser (list T)
:=
  many_helper p [] steps.
```

A parser that expects a given token, followed by p:

```
Definition firstExpect {T} (t : token) (p : parser T)
                     : parser T :=
  fun xs ⇒ match xs with
            | x::xs' ⇒
              if string_dec x t
              then p xs'
              else NoneE ("expected '" ++ t ++ "'.")
            | [] ⇒
              NoneE ("expected '" ++ t ++ "'.")
            end.
```

A parser that expects a particular token:

```
Definition expect (t : token) : parser unit :=
  firstExpect t (fun xs ⇒ SomeE(tt, xs)).
```

A Recursive-Descent Parser for Imp

Identifiers:

```
Definition parseIdentifier (xs : list token)
                         : optionE (string * list token) :=
match xs with
| [] ⇒ NoneE "Expected identifier"
| x::xs' ⇒
    if forallb isLowerAlpha (list_of_string x) then
      SomeE (x, xs')
    else
      NoneE ("Illegal identifier:'" ++ x ++ "'")
end.
```

Numbers:

```
Definition parseNumber (xs : list token)
                     : optionE (nat * list token) :=
match xs with
| [] ⇒ NoneE "Expected number"
| x::xs' ⇒
    if forallb isDigit (list_of_string x) then
```

```
        SomeE (fold_left
                 (fun n d ⇒
                    10 * n + (nat_of_ascii d -
                              nat_of_ascii "0"%char))
                 (list_of_string x)
                 0,
               xs')
      else
        NoneE "Expected number"
  end.
```

Parse arithmetic expressions

```
  Fixpoint parsePrimaryExp (steps:nat)
                           (xs : list token)
                         : optionE (aexp * list token) :=
    match steps with
    | 0 ⇒ NoneE "Too many recursive calls"
    | S steps' ⇒
        DO (i, rest) <-- parseIdentifier xs ;
            SomeE (AId i, rest)
        OR DO (n, rest) <-- parseNumber xs ;
            SomeE (ANum n, rest)
                  OR (DO (e, rest) <== firstExpect "("
                         (parseSumExp steps') xs;
            DO (u, rest') <== expect ")" rest ;
            SomeE(e,rest'))
    end

  with parseProductExp (steps:nat)
                       (xs : list token) :=
    match steps with
    | 0 ⇒ NoneE "Too many recursive calls"
    | S steps' ⇒
      DO (e, rest) <==
        parsePrimaryExp steps' xs ;
      DO (es, rest') <==
         many (firstExpect "*" (parsePrimaryExp steps'))
              steps' rest;
      SomeE (fold_left AMult es e, rest')
    end

  with parseSumExp (steps:nat) (xs : list token) :=
    match steps with
    | 0 ⇒ NoneE "Too many recursive calls"
    | S steps' ⇒
      DO (e, rest) <==
        parseProductExp steps' xs ;
      DO (es, rest') <==
        many (fun xs ⇒
          DO (e,rest') <--
             firstExpect "+"
               (parseProductExp steps') xs;
             SomeE ( (true, e), rest')
          OR DO (e,rest') <==
          firstExpect "-"
             (parseProductExp steps') xs;
```

```
            SomeE ( (false, e), rest'))
          steps' rest;
        SomeE (fold_left (fun e0 term ⇒
                            match term with
                              (true, e) ⇒ APlus e0 e
                            | (false, e) ⇒ AMinus e0 e
                            end)
                         es e,
               rest')
  end.

Definition parseAExp := parseSumExp.
```

Parsing boolean expressions:

```
Fixpoint parseAtomicExp (steps:nat)
                        (xs : list token) :=
match steps with
  | 0 ⇒ NoneE "Too many recursive calls"
  | S steps' ⇒
     DO (u,rest) <-- expect "true" xs;
         SomeE (BTrue,rest)
     OR DO (u,rest) <-- expect "false" xs;
         SomeE (BFalse,rest)
     OR DO (e,rest) <--
            firstExpect "!"
               (parseAtomicExp steps')
               xs;
         SomeE (BNot e, rest)
     OR DO (e,rest) <--
              firstExpect "("
                (parseConjunctionExp steps') xs;
          (DO (u,rest') <== expect ")" rest;
              SomeE (e, rest'))
     OR DO (e, rest) <== parseProductExp steps' xs;
            (DO (e', rest') <--
              firstExpect "="
                (parseAExp steps') rest;
              SomeE (BEq e e', rest')
             OR DO (e', rest') <--
               firstExpect "≤"
                 (parseAExp steps') rest;
               SomeE (BLe e e', rest')
             OR
               NoneE
      "Expected '=' or '≤' after arithmetic expression")
end

with parseConjunctionExp (steps:nat)
                         (xs : list token) :=
  match steps with
  | 0 ⇒ NoneE "Too many recursive calls"
  | S steps' ⇒
    DO (e, rest) <==
      parseAtomicExp steps' xs ;
    DO (es, rest') <==
       many (firstExpect "&&"
```

```
                (parseAtomicExp steps'))
             steps' rest;
      SomeE (fold_left BAnd es e, rest')
    end.

Definition parseBExp := parseConjunctionExp.

Check parseConjunctionExp.

Definition testParsing {X : Type}
           (p : nat →
                list token →
                optionE (X * list token))
           (s : string) :=
  let t := tokenize s in
  p 100 t.

(*
Eval compute in
  testParsing parseProductExp "x*y*(x*x)*x".

Eval compute in
  testParsing parseConjunctionExp "not((x=x||x*x<=
(x*x)*x)&&x=x".
*)
```

Parsing commands:

```
Fixpoint parseSimpleCommand (steps:nat)
                            (xs : list token) :=
  match steps with
  | 0 ⇒ NoneE "Too many recursive calls"
  | S steps' ⇒
    DO (u, rest) <-- expect "SKIP" xs;
      SomeE (SKIP, rest)
    OR DO (e,rest) <--
         firstExpect "IFB" (parseBExp steps') xs;
       DO (c,rest') <==
         firstExpect "THEN"
           (parseSequencedCommand steps') rest;
       DO (c',rest'') <==
         firstExpect "ELSE"
           (parseSequencedCommand steps') rest';
       DO (u,rest''') <==
         expect "END" rest'';
       SomeE(IFB e THEN c ELSE c' FI, rest''')
    OR DO (e,rest) <--
         firstExpect "WHILE"
           (parseBExp steps') xs;
       DO (c,rest') <==
         firstExpect "DO"
           (parseSequencedCommand steps') rest;
       DO (u,rest'') <==
         expect "END" rest';
       SomeE(WHILE e DO c END, rest'')
    OR DO (i, rest) <==
         parseIdentifier xs;
       DO (e, rest') <==
```

```
          firstExpect ":=" (parseAExp steps') rest;
        SomeE(i ::= e, rest')
    end

with parseSequencedCommand (steps:nat)
                           (xs : list token) :=
  match steps with
  | 0 ⇒ NoneE "Too many recursive calls"
  | S steps' ⇒
      DO (c, rest) <==
        parseSimpleCommand steps' xs;
      DO (c', rest') <--
        firstExpect ";;"
          (parseSequencedCommand steps') rest;
        SomeE(c ;; c', rest')
      OR
        SomeE(c, rest)
  end.

Definition bignumber := 1000.

Definition parse (str : string) : optionE (com * list token) :=
  let tokens := tokenize str in
  parseSequencedCommand bignumber tokens.
```

Examples

```
Example eg1 : parse "
  IFB x = y + 1 + 2 - y * 6 + 3 THEN
    x := x * 1;;
    y := 0
  ELSE
    SKIP
  END "
=
  SomeE (
     IFB "x" = "y" + 1 + 2 - "y" * 6 + 3 THEN
       "x" ::= "x" * 1;;
       "y" ::= 0
     ELSE
       SKIP
     FI,
     []).
Proof. reflexivity. Qed.
```