

1 Original Grammar

This grammar is highly ambiguous and definitely not LL(1).

```
Expr ::= Expr "+" Expr
      | Expr "-" Expr
      | Expr "*" Expr
      | Expr "/" Expr
      | Expr "^" Expr
      | "(" Expr ")" "!"
      | "(" Expr ")"
      | intlit | var
```

Here, `intlit` stands for “integer literal,” a string of digits 0 to 9, and `var` stands for “variable,” a string of upper- and lower-case letters.

2 LL(1) Grammar

The following grammar recognizes that same language, but is LL(1).

This means that, to parse a string `S` as given non-terminal `X`, where `X` has several rules of the form `X ::= ...`, one can determine which rule to use by looking only 1 character ahead in `S`.

An important consequence is that the grammar can be parsed by a recursive descent parser.

```
Expr      ::= Term Expr'
Expr'     ::= "+" Expr | "-" Expr | empty
Term      ::= Primary Term'
Term'     ::= "*" Term | "/" Term | empty
Primary   ::= Factor Primary'
Primary'  ::= "^" Primary | empty
Factor    ::= Base Factor'
Factor'   ::= "!" | empty
Base      ::= intlit | var | "(" Expr ")"
```

Note that, because the recursive “calls” are always on the right of an infix operator like “+” or “-”, such operators are all right associative in the parse tree representation.

This is a fundamental limit of recursive descent parsing, which dis-allows left recursion.

We will handle this problem by first parsing to an intermediate tree representation, applying functions to change the associativity of selected operators, and then converting the result into abstract syntax.

3 Recursive Descent Parsing Code

Following is code to parse the above grammar. It is split into four sections:

- **Types** The necessary types.
- **Tokenization** Simple code to convert a string into a list of tokens.
- **Parsing** The actual recursive descent parser, which takes in a list of tokens and outputs an intermediate tree representation.
- **Left-association and Conversion to Abstract Syntax Code** Code to left-associate operators by rearranging the intermediate tree, and code to convert the result into abstract syntax.

3.1 Types

Here are the types:

```
type token =
  LPAREN | RPAREN
  | INT of int | VAR of string
  | INF_PLUS | INF_MINUS | INF_TIMES | INF_DIV | INF_POWER
  | POST_FACTORIAL

type optree =
  TOKEN of token
  | UNARY of token * optree | BINARY of token * optree * optree

type exp =
  INTEGER of int | VARIABLE of string
  | PLUS of exp * exp | MINUS of exp * exp | TIMES of exp * exp
  | DIV of exp * exp | POWER of exp * exp
  | FACTORIAL of exp
```

`token` is the type of tokens, and `exp` is the type of the abstract syntax.
`optree` is an intermediate type that models general binary trees, with tokens at its nodes. By using `optree` as an intermediary representation,

we can write a generic function to re-structure parse trees to make certain operators left associative. This is necessary because, as you will notice, a non-left-recursive grammar produces parse trees where all infix operators are right associative.

3.2 Tokenization

The following code takes a string of characters and converts it into a list of tokens.

```
let get_token c =
  match c with
    '(' -> LPAREN | ')' -> RPAREN
  | '+' -> INF_PLUS | '-' -> INF_MINUS | '*' -> INF_TIMES | '/' -> INF_DIV
  | '^' -> INF_POWER
  | '!' -> POST_FACTORIAL
  | _ -> failwith ("unknown token: " ^ (String.make 1 c))

let is_int_char c =
  let code = Char.code c
  in
    (Char.code '0') <= code && code <= (Char.code '9')

let is_var_char c =
  let code = Char.code c
  in
    (Char.code 'a') <= code && code <= (Char.code 'z')
    || (Char.code 'A') <= code && code <= (Char.code 'Z')

let match_str str =
  let len = String.length str
  in
    if len <= 0 then None
    else
      Some (String.get str 0, String.sub str 1 (len - 1))

let rec get_seq f str =
  match match_str str with
    None -> ("",str)
  | Some(c,rest) ->
```

```

    if f c then
      let (res,str') = get_seq f rest
      in
        (String.make 1 c ^ res, str')
    else ("",str)

let rec tokenize str =
  match match_str str with
  None          -> []
  | Some (c,rest) ->
    match c with
    ' ' | '\t' | '\n' -> tokenize rest
    | _ when is_int_char c ->
      let (res,str') = get_seq is_int_char str
      in
        INT(int_of_string res)::tokenize str'
    | _ when is_var_char c ->
      let (res,str') = get_seq is_var_char str
      in
        VAR res::tokenize str'
    | _ -> get_token c::tokenize rest

```

3.3 Parsing

Here is the code that does the recursive descent parsing.

The main function, `parse`, comes last and has type `token list -> optree`.

The functions to parse the individual non-terminals come earlier, and each have type `token list -> optree * token list`. They take token lists and returns pairs consisting of `optrees` and remaining tokens to be parsed.

```

(** Expr and Expr' **)
let rec parse_top tokens =
  match parse_notplusminus tokens with
  (res,INF_PLUS::toks) -> (
    match parse_top toks with
    (res',toks') -> (BINARY(INF_PLUS,res,res'),toks')
  )
  | (res,INF_MINUS::toks) -> (

```

```

        match parse_top toks with
          (res',toks') -> (BINARY(INF_MINUS,res,res'),toks')
        )
    | x                -> x

(***) Term and Term' (***)
and parse_notplusminus tokens =
  match parse_nottimesdiv tokens with
    (res,INF_TIMES::toks) -> (
      match parse_notplusminus toks with
        (res',toks') -> (BINARY(INF_TIMES,res,res'),toks')
      )
    | (res,INF_DIV::toks) -> (
      match parse_notplusminus toks with
        (res',toks') -> (BINARY(INF_DIV,res,res'),toks')
      )
    | x                -> x

(***) Primary and Primary' (***)
and parse_nottimesdiv tokens =
  match parse_notpower tokens with
    (res,INF_POWER::toks) -> (
      match parse_nottimesdiv toks with
        (res',toks') -> (BINARY(INF_POWER,res,res'),toks')
      )
    | x                -> x

(***) Factor (***)
and parse_notpower tokens =
  match parse_notfactorial tokens with
    (res,POST_FACTORIAL::toks) -> (UNARY(POST_FACTORIAL, res),toks)
    | x                -> x

(***) Base (***)
and parse_notfactorial tokens =
  match tokens with
    INT i::toks -> (TOKEN (INT i),toks)
    | VAR s::toks -> (TOKEN (VAR s),toks)
    | LPAREN::toks ->
      (

```

```

        match parse_top toks with
          (res,RPAREN::toks') -> (res,toks')
        | _                    -> failwith "parse_notfactorial: ')' expected"
      )
    | _ -> failwith "parse_notfactorial"

```

```

(*
  parse: token list -> optree

```

This top-level function takes in a list of tokens and passes it to the recursive descent parser, extracting the final tree from the result.

```

*)
let parse tokens =
  match parse_top tokens with
    (res, []) -> res
  | _         -> failwith "parse: tokens remain"

```

3.4 Left-association and Conversion to Abstract Syntax

The following function `p` takes an optree and makes '-', '+', '*', and '/' left associative (by repeatedly calling `left_assoc`. It then converts the optree into an `exp` (our abstract syntax type) using `conv`.

```

(* left_append : token -> optree -> optree

```

appends `t'` to the left of `t` (helper for `left_assoc`)

```

*)
let rec left_append op t' t =
  match t with
    BINARY(op',t1,t2) when op' = op -> BINARY(op',left_append op t' t1,t2)
  | _ -> BINARY(op,t',t)

```

```

(* left_assoc : optree -> token -> optree

```

re-arranges the tree to associate given operator to the left

```

*)
let rec left_assoc t op =
  match t with
    TOKEN _ -> t

```

```

    | UNARY (op',t') -> UNARY (op',left_assoc t' op)
    | BINARY(op',t1,t2) when op' = op -> left_append op t1 (left_assoc t2 op)
    | BINARY(op',t1,t2) -> BINARY(op',left_assoc t1 op,left_assoc t2 op)

(*
conv : optree -> exp

convert to exp
*)
let rec conv t =
  match t with
  | TOKEN (INT i) -> INTEGER i | TOKEN (VAR s) -> VARIABLE s
  | UNARY (POST_FACTORIAL, t') -> FACTORIAL (conv t')
  | BINARY (INF_PLUS, t1, t2) -> PLUS (conv t1, conv t2)
  | BINARY (INF_MINUS, t1, t2) -> MINUS (conv t1, conv t2)
  | BINARY (INF_TIMES, t1, t2) -> TIMES (conv t1, conv t2)
  | BINARY (INF_DIV, t1, t2) -> DIV (conv t1, conv t2)
  | BINARY (INF_POWER, t1, t2) -> POWER (conv t1, conv t2)
  | _ -> failwith "conv: illegal token"

(* p : string -> exp

Takes in a string, tokenizes it, parses the result into
an optree, left associates '+' '-' '*' and '/', and then
converts the result into the type of our abstract syntax.
*)
let p str =
  conv (
    List.fold_left left_assoc (parse (tokenize str))
    [INF_PLUS; INF_MINUS; INF_TIMES; INF_DIV]
  )

```