CSCI 2041: Language Processing

Chris Kauffman

Last Updated:
Sun Nov 18 18:15:27 CST 2018

Logistics

Goals

- Basics of Language Processing
- Lexing
- Parsing
- Evaluation

Lab10: Lexing/Parsing

- Covers code from lecture
- Extend it to additional operations

Exam 3

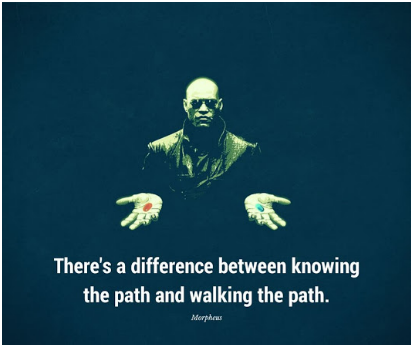
- Mon: Review
- Wed: Exam 3

Assignment 5

- Post later this week
- Arithmetic language interpreter
- Larger, worth 2X previous projects
- Due towards end of semester
- Will have **milestone** deadline

Knowing vs. Doing

- Several folks came to office hours with "My functors worked but I don't know why."
- Also, "I added this to the parser and it works but I don't understand it fully."
- Understanding **why** is very important and will serve you greatly, but... **it takes a lot of work**


There's a difference between knowing the path and walking the path.
Morpheus

Strive to understand what you do, but don't be afraid to do something you don't completely understand (especially if it's part of the assignment instructions)

An Opening Example

```
let x = 5 + 10*4 + 7*(3+2) in ...
```

- Above program fragment is something one would type in and expect compiled/run without trouble
- The trouble comes when writing the interpreter or compiler
- Will discuss issues associated with these for the next few sessions
- Ultimate goal: code to evaluate arithmetic expressions
- Will require
  - **Algebraic Types** for symbolic data
  - **Recursive functions** for lexing/parsing
  - **List processing** for parsing
  - **Recursive functions** on trees for evaluation

Good thing that you've mastered all these. . .

Roadmap for Processing

1. Get input

   ```
   let input = "5 + 10*4 + 7*(3+2)";;
   ```

2. Lex input to tokens

   ```
       [Int 5; Plus; Int 10; Times; Int 4; Plus;...
   ```

3. Parse tokens to expression tree

   ```
       Add(Const(5),
           Add(Mul(Const(10),
                   Const(4)),...
   ```

4. Evaluate tree

   ```
   80
   ```

Getting Input

- Generally the easy part
- Obtained via reading from a file or user typing in input
- Can usually assume it's stored in a string somewhere like

  ```
  let input = "5 + 10*4 + 7*(3+2)";;
  ```
- First step in compiler or interpreter is to get input like this

Lexing

lexical analysis ≡ *lexing* ≡ *tokenization*

- Raw input is just a bunch of characters
- Lexing is done to ease processing later on
- Group characters into **tokens** for operations, numbers, keywords, etc
- Assign some meaning to tokens via a symbolic name
- Identify characters that don't belong/not recognized
- Output of lexing is a stream of such tokens, will use a list of tokens in our work

```
let input = "5 + 10*4 + 7*(3+2)";;  (* Lexing: convert this string..  *
let lexed = [Int 5; Plus; Int 10;   (* Into this stream of tokens *)
             Times; Int 4; Plus;
             Int 7; Times;
             OParen; Int 3; Plus;
             Int 2; CParen];;
```

Lexing and Token Symbols

- Typical tokens are **symbolic** and may carry additional data
- A convenient way to represent them in OCaml is via algebraic types with variants for each token type

  ```
  (* algebraic types for tokens: lexing results *)
  type token =
    Plus | Times | OParen | CParen | Int of int;;
  ```
- Above tokens are very limited but sufficient for simple arithmetic
- More extensive arithmetic language processing would include subtraction, division, floating point numbers
- Fuller programming languages have variable identifiers, keywords like `let/in` and `for/do`

Exercise: Lexing Thought Questions

```
let input = "5 + 10*4 + 7*(3+2)";;  (* Lexing: convert this string..  *
let lexed = [Int 5; Plus; Int 10;   (* Into this stream of tokens *)
             Times; Int 4; Plus;
             Int 7; Times;
             OParen; Int 3; Plus;
             Int 2; CParen];;
```

1. Do all characters in the above string appear in the tokens? If not, which are not present and why?
2. Do all the tokens correspond to single characters or are groups of multiple characters associated with a single token?
3. Speculate on how the code for a lexing function will look

Answers: Lexing Thought Questions

```
let input = "5 + 10*4 + 7*(3+2)";;  (* Lexing: convert this string..  *)
let lexed = [Int 5; Plus; Int 10;   (* Into this stream of tokens *)
             Times; Int 4; Plus;
             Int 7; Times;
             OParen; Int 3; Plus;
             Int 2; CParen];;
```

1. Do all characters in the above string appear in the tokens? If not, which are not present and why?
   Spaces do not get tokenized; whitespace is commonly ignored.
2. Do all the tokens correspond to single characters or are groups of multiple characters associated with a single token?
   In the above example, most tokens are single characters but multi-character numbers like 10 are a single token.
3. Speculate on how the code for a lexing function will look
   Lexing functions must scan through the string matching characters and creating tokens. Our version will combine recursion and iteration.

A Simple Lexing Routine

```
let lex_string string =                      (* create a list of tokens  *)
  let len = String.length string in
  let rec lex pos =                          (* recursive helper on pos in string *)
    if pos >= len then                       (* off end of string ? *)
      []                                     (* end of input *)
    else                                     (* more to lex *)
      match string.[pos] with                (* match a single character *)
      |' ' | '\t' | '\n' -> lex (pos+1)      (* skip whitespace *)
      |'+' -> Plus :: (lex (pos+1))          (* single char ops become operators *)
      |'*' -> Times :: (lex (pos+1))         (* like add and multiply *)
      |'(' -> OParen :: (lex (pos+1))        (* and open/close parens *)
      |')' -> CParen :: (lex (pos+1))
      | d when is_digit d ->                 (* see a digit *)
         let stop = ref pos in               (* scan through until a non-digit is found *)
         while !stop < len && is_digit string.[!stop] do
           incr stop;
         done;
         let numstr = String.sub string pos (!stop - pos) in (* substring is the int *)
         let num = int_of_string numstr in   (* parse the integer *)
         Int(num) :: (lex !stop)             (* and tack onto the stream of tokens *)
      | _ ->                                 (* any other characters lead to failures *)
         let msg = sprintf "lex error at char %d, char '%c'" pos string.[pos] in
         failwith msg
  in                                         (* end helper *)
  lex 0                                      (* call helper *)
;;
```

Sample Calls to `lex_string`

```
# lex_string "123";;
- : token list = [Int 123]

# lex_string "*";;                 (* NOT VALID SYNTAX BUT THAT IS *)
- : token list = [Times]           (* THE BUSINESS OF THE PARSING PHASE *)

# lex_string "123 +";;
- : token list = [Int 123; Plus]

# lex_string "123 + 19";;
- : token list = [Int 123; Plus; Int 19]

# lex_string "123 + (19)";;
- : token list = [Int 123; Plus; OParen; Int 19; CParen]

# lex_string "123 * (1+19)";;
- : token list = [Int 123; Times; OParen; Int 1; Plus; Int 19; CParen]

# lex_string "/";;                 (* UNRECOGNIZED CHARACTERS *)
Exception: Failure "lex error at char 0, char '/'".

# lex_string "123 + 19 - 9";;
Exception: Failure "lex error at char 9, char '-'".
```

Syntax of a Language

- **Syntax** or **Grammar** of a language is the pattern of acceptable tokens
- A stream may be all valid tokens but the ordering of those tokens violates the syntax/grammar rules of the language.
  - "I like to program on weekends." *correct*
  - "On weekends, I like to program." *correct*
  - "On weekends, to program I like." *sure thing Yoda*
  - "On weekends. I like to program." *less correct*
  - "weekends like. to I On program," *you get the idea*
- **Parsing** is associated with determining if tokens adhere to the syntax of the language our not
- Accept input as valid or reject it as not valid
- Acceptable grammar usually converted into a parsed data structure: **Parse Tree** or **Abstract Syntax Tree (AST)**

Basic Parsing Approach

- A **Recursive Descent** parser divides parsing into a series of functions
- Each function handles a specific element of the grammar
- Functions call each other in an order that dictates **precedence** of grammar elements
  - Integer tokens and Variable names self-evaluate
  - Parenthesized expressions evaluate earlier than arithmetic
  - Multiplication and Division have higher precedence
  - Addition and subtraction have lower precedence
- Language documentation usually notes high/low precedence operations such as in OCaml's docs:

```
(* precedence level 6/11. *)
val (+) : int -> int -> int
(* precedence level 7/11. *)
val (*) : int -> int -> int
```

- Since parsing functions may call each other, they are usually **mutually recursive** necessitating the use of

```
let rec prec0 toks =
    ...
    and prec1 toks =
    ...
    and prec2 toks =
    ...
    and parse toks =
    ... ;;
```

Building an Abstract Syntax Tree (AST)

- `langproc.ml` builds an abstract syntax tree for arithmetic with integers, + and *
- Abstract syntax tree for these is comprised of the following

```
(* algebraic types for expression tree: parsing results *)
type expr =
  | Add of expr * expr
  | Mul of expr * expr
  | Const of int
;;

let input = "5 + 10*4 + 7*(3+2)";;
let parsed =
  Add(Const(5),
      Add(Mul(Const(10),
              Const(4)),
          Mul(Const(7),
              Add(Const(3),
                  Const(2)))))
;;
```

Exercise: Highest Precedence Grammar Elements

- `langproc.ml` provides a recursive descent parser with 3 precedence levels
- Below is highest precedence level

```
(* prec0: self-evaluating tokens like Int and parenthesized expressions *)
let rec prec0 toks =
  match toks with
  | [] ->                                          (* out of input *)
     raise (ParseError {msg="expected an expression"; toks=toks})
  | Int n ::  tail ->                              (* ints are self-evaluating *)
     (Const(n),tail)
  | OParen :: tail ->                              (* parenthesized expresion *)
     begin
       let (expr,rest) = parse tail in  (* start back at lowest precedence
       match rest with
       | CParen::tail -> (expr,tail)
       | _ -> raise (ParseError {msg="unclosed parentheses"; toks=rest})
     end
  | _ ->
     raise (ParseError {msg="syntax error"; toks=toks})
```

1. What kind of thing is parameter `toks`?
2. What types of tokens are handled by `prec0`
3. What kind of parsing errors can result at this level?

Answers: Highest Precedence Grammar Elements

1. What kind of thing is parameter `toks`?
   It is a list of token types. The head element is analyzed in the match/with.
2. What types of tokens are handled by `prec0`
   Int tokens which are converted to Const expressions and OParen/CParen tokens which continue parsing again.
3. What kind of parsing errors can result at this level?
   Running out of input, failure to close a parenthesis, and general syntax errors.

Exercise: Arithmetic Operations

```
(* prec1: multiplication *)
and prec1 toks =
  let (lexpr, rest) = prec0 toks in   (* try higher prec first *)
  match rest with
  | Times :: tail ->                  (* * is first *)
     let (rexpr,rest) = prec1 tail in (* recurse to get right-hand expr *)
     (Mul(lexpr,rexpr), rest)         (* multiplyt left/right expr *)
  | _ -> (lexpr, rest)                (* not a multiply *)

(* prec2: addition only *)
and prec2 toks =
  let (lexpr, rest) = prec1 toks in   (* try higher prec first *)
  match rest with
  | Plus :: tail ->                   (* + is first *)
     let (rexpr,rest) = prec2 tail in (* recurse to get right-hand expr *)
     (Add(lexpr,rexpr), rest)         (* add left/right expr *)
  | _ -> (lexpr, rest)                (* not an addition *)
```

1. `prec1` calls `prec0` and `prec2` calls `prec1`. In this scheme, does a large number indicate high or low precdence?
2. When `prec0` is called, what is the result? Is this the same for when `prec1` is called? how about `prec2`?
3. What happens if `prec1` and `prec2` cannot match a token at the beginning of the list of tokens?

Answers: Arithmetic Operations

1. `prec1` calls `prec0` and `prec2` calls `prec1`. In this scheme, does a large number indicate high or low precdence?
   Large numbers indicate lower precedence as prec0 are self-evaluating or parenthesized experssions. This is the opposite of the OCaml documentation where higher numbers indicate higher precedence.
2. When `prec0` is called, what is the result? Is this the same for when `prec1` is called? how about `prec2`?
   A pair of an expression (AST) and the remaining list of tokens is returned in both cases. This is also what prec0 returns.
3. What happens if `prec1` and `prec2` cannot match a token at the beginning of the list of tokens?
   They return the expression given by the prec0 or prec1 paired with the entire list of tokens with nothing removed. This backtracks to a previous function.

The Full(-ish) Parser

3 precedence levels, 3 functions to handle them

```
(* parse tokens list of arithmetic
   language to an AST  *)
let parse_tokens tokens =

  (* prec0: self-evaluating tokens like
     Int and parenthesized expressions *)
  let rec prec0 toks =
    match toks with
    | [] ->
       raise (Expect Expression)
    | Int n ::  tail ->
       (Const(n),tail)
    | OParen :: tail ->
       begin
         let (expr,rest) = parse tail in
         match rest with
         | CParen::tail -> (expr,tail)
         | _ -> raise (Unclosed Paren)
       end
    | _ ->
       raise (Syntax Error)

  (* prec1: multiplication *)
  and prec1 toks =
    let (lexpr, rest) = prec0 toks in
    match rest with
    | Times :: tail ->
       let (rexpr,rest) = prec1 tail in
       (Mul(lexpr,rexpr), rest)
    | _ -> (lexpr, rest)

  (* prec2: addition only *)
  and prec2 toks =
    let (lexpr, rest) = prec1 toks in
    match rest with
    | Plus :: tail ->
       let (rexpr,rest) = prec2 tail in
       (Add(lexpr,rexpr), rest)
    | _ -> (lexpr, rest)

  (* top-level parsing entry *)
  and parse toks =
    prec2 toks
  in

  (* end helpers, main code for
     parse_tokens *)
  let (expr, rest) = parse tokens in
  match rest with
  | [] -> expr
  | _  -> raise (Tokens Remain)
;;
```

Sample Calls to parse_tokens

```
# parse_tokens (lex_string "11+5");;         (* Simple tree with Const leaves *)
- : expr = Add (Const 11,
                Const 5)

# parse_tokens (lex_string "11+5+2+9");;     (* Chain-esque tree of repeated Add *)
- : expr = Add (Const 11,
                Add (Const 5,
                     Add (Const 2,
                          Const 9)))

# parse_tokens (lex_string "11+5*2+9");;     (* Mult has higher precedence *)
- : expr = Add (Const 11,
                Add (Mul (Const 5,
                          Const 2),
                     Const 9))

# parse_tokens (lex_string "2*(11+5)*2");;   (* Parens change precedence *)
- : expr = Mul (Const 2,
                Mul (Add (Const 11,
                          Const 5),
                     Const 2))

# parse_tokens (lex_string "2 +");;          (* Parse Error Cases *)
Exception: ParseError {msg = "expected an expression"; toks = []}.

# parse_tokens (lex_string "2 * (3+5 " );;
Exception: ParseError {msg = "unclosed parentheses"; toks = []}.

# parse_tokens (lex_string "2 * 3 + * 5");;
Exception: ParseError {msg = "syntax error"; toks = [Times; Int 5]}.
```

Exercise: Show Parse Tree

- Show the parse tree or report errors for the following examples
- First few examples are already done for reference

```
# parse_tokens (lex_string "11+5+2+9");;     (* EXAMPLE A *)
- : expr = Add (Const 11,
                Add (Const 5,
                     Add (Const 2,
                          Const 9)))

# parse_tokens (lex_string "2*(11+5)*2");;   (* EXAMPLE B *)
- : expr = Mul (Const 2,
                Mul (Add (Const 11,
                          Const 5),
                     Const 2))

# parse_tokens (lex_string "(11 + 2)*5");;   (* PROBLEM 1 *)


# parse_tokens (lex_string "(11 + )*5");;    (* PROBLEM 2 *)


# parse_tokens (lex_string "11*5*2*9");;     (* PROBLEM 3 *)


# parse_tokens (lex_string "(11+2)+(5+9)");; (* PROBLEM 4 *)


# parse_tokens (lex_string "11*5*2+9");;     (* PROBLEM 5 *)
```

Answers: Show Parse Tree

```
# parse_tokens (lex_string "(11 + 2)*5");;    (* PROBLEM 1 *)
- : expr = Mul (Add (Const 11,                (* parens give addition higher  *)
                     Const 2),                (* precedence, later multiply *)
                Const 5)

# parse_tokens (lex_string "(11 + )*5");;     (* PROBLEM 2 *)
Exception: ParseError {msg = "syntax error"; toks = [CParen; Times; Int 5]}.

# parse_tokens (lex_string "11*5*2*9");;      (* PROBLEM 3 *)
- : expr = Mul (Const 11,                     (* chain-like tree of repeated *)
                Mul (Const 5,                 (* multiplications *)
                     Mul (Const 2,
                          Const 9)))

# parse_tokens (lex_string "(11+2)+(5+9)");; (* PROBLEM 4 *)
- : expr = Add (Add (Const 11,                (* Parens give first/last add *)
                     Const 2),                (* higher precedence than middle *)
                Add (Const 5,
                     Const 9))

# parse_tokens (lex_string "11*5*2+9");;      (* PROBLEM 5 *)
- : expr = Add (Mul (Const 11,                (* Multiply everything first *)
                     Mul (Const 5,            (* then add *)
                          Const 2)),
                Const 9)
```

These make for easy exam problems. . .

Recursive Descent Parsing is a Search Process

- It is difficult but very worthwhile to reason about how the recursive descent parser works in total
- Parsing is really a **search process** in which each function tries to consume tokens with some failures allowing backtracking
- Gets even trickier to understand with parentheses involved which circle back to top of parsing functions

```
| TOKEN POSITION   | FUNCTION CALL STACK           | EXPRESSON TREE
|------------------+-------------------------------+-----------------------------------------
|  5  +  8  *  4   |                               |
| <5> +  8  *  4   | prec2                         |
| <5> +  8  *  4   | prec2,prec1                   |
| <5> +  8  *  4   | prec2,prec1,prec0             |
|  5 <+> 8  *  4   | prec2,prec1                   | Const(5)
|  5 <+> 8  *  4   | prec2                         | Const(5)
|  5  + <8> *  4   | prec2,prec2                   | Add(Const(5),..)
|  5  + <8> *  4   | prec2,prec2,prec1             | Add(Const(5),..)
|  5  + <8> *  4   | prec2,prec2,prec1,prec0       | Add(Const(5),..)
|  5  +  8 <*> 4   | prec2,prec2,prec1             | Add(Const(5),..) Const(8)
|  5  +  8  * <4>  | prec2,prec2,prec1,prec1       | Add(Const(5), Mul(Const(8), ...)
|  5  +  8  * <4>  | prec2,prec2,prec1,prec1,prec0 | Add(Const(5), Mul(Const(8), ...)
|  5  +  8  *  4<> | prec2,prec2,prec1,prec1,prec0 | Add(Const(5), Mul(Const(8), Const(4)))
```

Aside: Tracing Function Execution

- A **call trace** conveys a sequence of function calls with parameter and return values
- Useful to understand code flow within a group of functions, now the stack looks
- Like many programming environments with a REPL, OCaml can **automatically generate call traces**
- `#trace funcname;;` turns on tracing of function, shows calls with params and return values
- Look for trace features in every language either directly or via debugger tools

```
# let doub x = 2 * x;;
val doub : int -> int = <fun>

# let octosum (x,y) =
  (doub (doub x)) + (doub (doub y));;
val octosum : int * int -> int = <fun>

# #trace octosum;;
octosum is now traced.
# #trace doub;;
doub is now traced.

# octosum (2,7);;
octosum <-- (2, 7)
double <-- 7
double --> 14
double <-- 14
double --> 28
double <-- 2
double --> 4
double <-- 4
double --> 8
octosum --> 36
- : int = 36
```

Tracing Parsing Functions

```
#use "langproc_trace.ml";;      (* defines parsing funcs at top level *)
...                             (* rather than inside parse_tokens    *)
val prec2 : token list -> token list * expr = <fun>
val prec1 : token list -> token list * expr = <fun>
val prec0 : token list -> token list * expr = <fun>
...
# #trace prec2;;                (* trace each of the parsing funcs *)
prec2 is now traced.
# #trace prec1;;
prec1 is now traced.
# #trace prec0;;
prec0 is now traced.

# let parsed = parse_tokens (lex_string "7");;
prec2 <-- [Int 7]
prec1 <-- [Int 7]
prec0 <-- [Int 7]
prec0 --> ([], Const 7)
prec1 --> ([], Const 7)
prec2 --> ([], Const 7)
val parsed : expr = Const 7
```

A More Intense Trace Example

- Indented (by hand) to show matching of call/return
- Token list param/return on the left and expr return on right

```
# parse_tokens (lex_string "7*(3+2)");;
prec2 <--                [Int 7; Times; OParen; Int 3; Plus; Int 2; CParen]
  prec1 <--              [Int 7; Times; OParen; Int 3; Plus; Int 2; CParen]
    prec0 <--            [Int 7; Times; OParen; Int 3; Plus; Int 2; CParen]
    prec0 -->           ([Times; OParen; Int 3; Plus; Int 2; CParen],  Const 7)
    prec1 <--            [OParen; Int 3; Plus; Int 2; CParen]
      prec0 <--          [OParen; Int 3; Plus; Int 2; CParen]
        prec2 <--        [Int 3; Plus; Int 2; CParen]
          prec1 <--      [Int 3; Plus; Int 2; CParen]
            prec0 <--    [Int 3; Plus; Int 2; CParen]
            prec0 -->   ([Plus; Int 2; CParen],                        Const 3)
          prec1 -->     ([Plus; Int 2; CParen],                        Const 3)
          prec2 <--      [Int 2; CParen]
            prec1 <--    [Int 2; CParen]
              prec0 <--  [Int 2; CParen]
              prec0 --> ([CParen],                                     Const 2)
            prec1 -->   ([CParen],                                     Const 2)
          prec2 -->     ([CParen],                                     Const 2)
        prec2 -->       ([CParen],                       Add (Const 3, Const 2))
      prec0 -->         ([],                             Add (Const 3, Const 2))
    prec1 -->           ([],                             Add (Const 3, Const 2))
  prec1 -->             ([],               Mul (Const 7, Add (Const 3, Const 2)))
prec2 -->               ([],               Mul (Const 7, Add (Const 3, Const 2)))
- : expr = Mul (Const 7, Add (Const 3, Const 2))
```

Adding more to the Language

- Lab 10 adds obvious stuff to the arithmetic expression language like subtraction, division, variable IDs
- More interesting stuff is programmatic
  - Name/value binding (including functions)
  - Control structures like if/then/else
  - Function application and Lambda Expressions
- We will delve into these soon but all have the same flavor
  - Add a `prec_ifthenelse toks` function to parse family
  - Looks for a pattern of tokens that fits

    ```
    if <expr> then <expr> else <expr>
    ```
  - Creates an associated parse tree associated with these like

    ```
    Cond(ifexpr, thenexpr, elseexpr)
    ```

But first, there's the small matter of **evaluating** a parsed expression in the simple case of arithmetic

Exercise: Evaluating an AST

Consider example parse tree below

```
# parse_tokens (lex_string "11*5*2+9");;
- : expr = Add (Mul (Const 11,
                     Mul (Const 5,
                          Const 2)),
                Const 9)
```

1. How does one evaluate such arithmetic expressions by hand?
2. What features do you expect from from code that `evaluate`'s the expression tree and produce an integer answer?
3. Write a version of `evaluate`

```
val evaluate : expr -> int = <fun>
```

Answers: Evaluating an AST

1. How does one evaluate such arithmetic expressions by hand?
   Usually ad-hoc but definitely multiply first then add
2. What features would code that would `evaluate` the expression tree and produce an integer answer?
   It will operate on the Parse Tree so will likely be recursive. The tree is comprised of algebraic types so pattern matching is expected.
3. Write a version of `evaluate`

```
(* Evaluate AST of expressions to produce an integer result *)
let rec evaluate expr =
  match expr with
  | Const i -> i
  | Add(lexpr,rexpr) ->
     let lans = evaluate lexpr in
     let rans = evaluate rexpr in
     lans + rans
  | Mul(lexpr,rexpr) ->
     let lans = evaluate lexpr in
     let rans = evaluate rexpr in
     lans * rans
;;
# evaluate (parse_tokens (lex_string "11*5*2+9"));;
- : int = 119
```

Aside: OCaml's "Threading" Operator |>

- At times have a series of functions to apply to data
  - Apply `f1` to data `x`
  - Apply `f2` to result of `f1`
  - Apply `f3` to result of `f2`, etc.
- Tedious and backwards looking to write

  ```
  let ans = f3 (f2 (f1 x)) in ...
  ```
- A reverse application operator is helpful, referred to in some contexts as a "threading" operator[1] as it threads data through functions in the order they appear
- OCaml has threading operator called `|>`

  ```
  let ans = x |> f1 |> f2 |> f3 ...
  ```

```
(* full cycle of lex, parse, evaluate *)
# let result = evaluate (parse_tokens (lex_string "2*3+5"));;
val result : int = 11

(* with |> threading operator (a.k.a reverse application) *)
# let result = "2*3+5" |> lex_string |> parse_tokens |> evaluate;;
val result : int = 11
```

[1]This kind of threading has nothing to do with multi-threaded programs; it is merely makes certain kinds of function applications easier on the eyes.