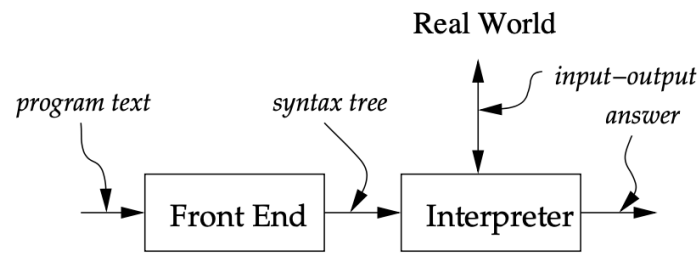


Project completed individually by Altun Hasanli.

Part A.



Execution via an interpreter - the 5 components:

- **Program Text:** Input Program in Source Language
- **Front End:** Scanner & Parser (implemented using a parser generator)
- **Syntax Tree:** AST of expressions that the parser generates
- **Interpreter:** Evaluates the AST into one of the expressed values (datatypes)
- **Answer:** Expressed Values given by the interpreter

The corresponding 5 components of MY-LET:

1: Lexical Specification - specifies how an input program written in the source language gets scanned and tokenized by the **Scanner (lexical analyzer)**. MY-LET allows for integers (negative and positive constants) and identifiers only, separated by comments and arbitrary whitespace. Lexical specification is defined in **lang.rkt**.

2: Grammar (Syntax) - specifies the source language (defined language) syntax in PL BNF rule format, and how it gets parsed into an **AST** by the **Parser (syntactic analyzer)**. Grammar specification is defined in **lang.rkt**. Programs in MY-LET are nested expressions that get evaluated to an expressed value. There are 10 different expression types.

3: Analyzer (Scanner & Parser: the two stages of the Front-End) - takes a program in source language and builds an AST. Lexical and grammar specifications are used in SLLGEN (Scheme LL(1) parser generator, defined in EOPL) procedures to generate another procedure that inputs an input program in the source language (PL syntax) and outputs an AST. (scan&parse in lang.rkt)

4: Expressed Values (datatypes) - specifies the final results (evaluated values) of expressions in MY-LET. They are: *num*, *rational*, *bool*, *list-of-nums*. These are structs with fields to hold the corresponding values in the **implementation language**. Defined and handled in **data-structures.rkt**. Datatypes, procedures to convert values from implementation language to defined language, and vice versa (extractors) are handled in the same file.

5: Interpreter (evaluator) - evaluates the different expressions of the language, extracts their semantic value (evaluation). Uses an environment to keep track of bound variables (defined and handled in environments.rkt and data-structures.rkt). Evaluation of a MY-LET program is handled in **value-of-program in interp.rkt**, and the recursive procedure **value-of** that it uses to handle the different expressions of MY-LET.

Part B.

The initial environment is defined in **init-env in environments.rkt**.
Using the syntax in EOPL, p.61,

```
;; init-env : () -> Env
;; usage: (init-env) = [z=304, y=302, x=301]
(define (init-env) (extend-env 'z (num-val 304)
                              (extend-env 'y (num-val 302)
                              (extend-env 'x (num-val 301)
                              (empty-env))))))
```

```
(init-env) => ρ
= [z=[304]] ([y=[302]] ([x=[301]] []))
= [z=[304]] ([y=[302]] [x=[301]])
= [z=[304]] [y=[302], x=[301]]
= [z=[304], y=[302], x=[301]]
```

Part C.

In MY-LET, expressed and denoted values are the same, and are as follows:

$\text{ExpVal} = \text{DenVal} = \text{Int} + \text{Bool} + \text{Pair}\langle \text{Int}, \text{Int} \rangle + \text{List}\langle \text{Int} \rangle$

Interface for them (as defined in data-structures.rkt):

```
num-val: Int -> ExpVal
bool-val: Bool -> ExpVal
rational-val: Pair<Int, Int> -> ExpVal
list-of-nums-val: List<Int> -> ExpVal

expval->num: Expval -> Int
expval->bool: Expval -> Bool
expval->rational: Expval -> Pair<Int, Int>
expval->list: Expval -> List<Int>
```

```
;; helper ;;
list-val: List<Int> -> ExpVal(List<ExpVal(Int)>)
```

Part D.

Check the corresponding files for implementations. Notes:

- list-exp is named **new-list-exp**
- Because of the way sloppy->expval is defined, I had to separate the logic for lists. “list-of-nums-val” creates a struct with a single field for the list. “list-val” calls the same procedure but turns the list of numbers into list of number structs first. This logic can further be expanded for car and cdr accessors for the list-of-nums, or better yet, the list can be constructed as a pair of a pair of a ..., but it wasn’t required.
- rational can be made into a struct with two fields: numerator and denominator, instead of a pair
- op-exp uses four helpers defined in interp.rkt
- simpl-exp uses euclid-gcd helper (in interp.rkt) which is $O(\log(n))$