Mini-LISP

The language that your project’s interpreter will process is a subset of [LISP](https://en.wikipedia.org/wiki/Lisp_%28programming_language%29), which we call it Mini-LISP for convenience. This handout first offers a general description, then goes into details such as lexical structure and grammar of the subset.

**Overview**

LISP is an ancient programming language based on [S-expressions](https://en.wikipedia.org/wiki/S-expression) and [lambda calculus](https://en.wikipedia.org/wiki/Lambda_calculus). All operations in Mini-LISP are written in parenthesized [prefix notation](https://en.wikipedia.org/wiki/Polish_notation). For example, a simple mathematical formula “(1 + 2) \* 3” written in Mini-LISP is:

(\* (+ 1 2) 3)

As a simplified language, Mini-LISP has only three types (**Boolean**, **number** and **function**) and a few operations.

**Type Definition**

* Boolean: Boolean type includes two values, #t for true and #f for false.
* Number: **Signed integer** **from *−(231)* to *231 – 1***, behavior out of this range is not defined.
* Function: See [Function](#Function).

Casting: Not allowed, but type checking is a bonus feature.

**Operation Overview**

|  |  |  |
| --- | --- | --- |
| Numerical Operators | | |
| Name | **Symbol** | **Example** |
| Plus | + | (+ 1 2) => 3 |
| Minus | - | (- 1 2) => -1 |
| Multiply | \* | (\* 2 3) => 6 |
| Divide | / | (/ 6 3) => 2 |
| Modulus | mod | (mod 8 3) => 2 |
| Greater | > | (> 1 2) => #f |
| Smaller | < | (< 1 2) => #t |
| Equal | = | (= 1 2) => #f |

|  |  |  |
| --- | --- | --- |
| Logical Operators | | |
| Name | **Symbol** | **Example** |
| And | and | (and #t #f) => #f |
| Or | or | (or #t #f) => #t |
| Not | not | (not #t) => #f |

**Other Operators:** define, fun, if

Note that all operators are **reserved words**, you cannot use any of these words as ID.

**Lexical Details**

Preliminary Definitions:

separator ::= ‘\t’(tab) | ‘\n’ | ‘\r’ | ‘ ’(space)

letter ::= [a-z]

digit ::= [0-9]

Token Definitions:

number ::= 0 | [1-9]digit\* | -[1-9]digit\*

*Examples: 0, 1, -23, 123456*

ID ::= letter (letter | digit | ‘-’)\*

*Examples: x, y, john, cat-food*

bool-val ::= #t | #f

**Grammar Overview**

PROGRAM ::= STMT+

STMT ::= EXP | DEF-STMT | PRINT-STMT

PRINT-STMT ::= (print-num EXP) | (print-bool EXP)

EXP ::= bool-val | number | VARIABLE | NUM-OP | LOGICAL-OP

| FUN-EXP | FUN-CALL | IF-EXP

NUM-OP ::= PLUS | MINUS | MULTIPLY | DIVIDE | MODULUS | GREATER

| SMALLER | EQUAL

PLUS ::= (+ EXP EXP+)

MINUS ::= (- EXP EXP)

MULTIPLY ::= (\* EXP EXP+)

DIVIDE ::= (/ EXP EXP)

MODULUS ::= (mod EXP EXP)

GREATER ::= (> EXP EXP)

SMALLER ::= (< EXP EXP)

EQUAL ::= (= EXP EXP+)

LOGICAL-OP ::= AND-OP | OR-OP | NOT-OP

AND-OP ::= (and EXP EXP+)

OR-OP ::= (or EXP EXP+)

NOT-OP ::= (not EXP)

DEF-STMT ::= (define VARIABLE EXP)

VARIABLE ::= id

FUN-EXP ::= (fun FUN\_IDs FUN-BODY)

FUN-IDs ::= (id\*)

FUN-BODY ::= EXP

FUN-CALL ::= (FUN-EXP PARAM\*) | (FUN-NAME PARAM\*)

PARAM ::= EXP

LAST-EXP ::= EXP

FUN-NAME ::= id

IF-EXP ::= (if TEST-EXP THAN-EXP ELSE-EXP)

TEST-EXP ::= EXP

THEN-EXP ::= EXP

ELSE-EXP ::= EXP

**Grammar and Behavior Definition**

1. Program

PROGRAM :: = STMT+

STMT ::= EXP | DEF-STMT | PRINT-STMT

1. Print

PRINT-STMT ::= (print-num EXP)

**Behavior: Print exp in decimal.**

| (print-bool EXP)

**Behavior: Print** #t **if EXP is true. Print** #f**, otherwise.**

1. Expression (EXP)

EXP ::= bool-val | number | VARIABLE

| NUM-OP | LOGICAL-OP | FUN-EXP | FUN-CALL | IF-EXP

1. Numerical Operations (NUM-OP)

NUM-OP ::= PLUS | MINUS | MULTIPLY | DIVIDE | MODULUS |

| GREATER | SMALLER | EQUAL

PLUS ::= (**+** EXP EXP+)

**Behavior: return sum of all EXP inside.**

Example: (+ 1 2 3 4) → 10

MINUS ::= (**-** EXP EXP)

**Behavior: return the result that the 1st** EXP **minus the 2nd** EXP**.**

Example: (- 2 1) → 1

MULTIPLY ::= (**\***  EXP EXP+)

**Behavior: return the product of all EXP inside.**

Example: (\* 1 2 3 4) → 24

DIVIDE ::= (**/** EXP EXP)

**Behavior: return the result that 1st** EXP **divided by 2nd** EXP**.**

Example: (/ 10 5) → 2

(/ 3 2) → 1 (just like C++)

MODULUS ::= (**mod**  EXP EXP)

**Behavior: return the modulus that 1st** EXP **divided by 2nd** EXP**.**

Example: (mod 8 5) → 3

GREATER ::= (**>** EXP EXP)

**Behavior: return** #t **if 1st** EXP **greater than 2nd** EXP**.** #f **otherwise.**

Example: (> 1 2) → #f

SMALLER ::= (**<** EXP EXP)

**Behavior: return** #t **if 1st** EXP **smaller than 2nd** EXP**.** #f **otherwise.**

Example: (< 1 2) → #t

EQUAL ::= (**=** EXP EXP+)

**Behavior: return** #t **if all** EXP**s** **are equal.** #f **otherwise.**

Example: (= (+ 1 1) 2 (/6 3)) → #t

1. Logical Operations (LOGICAL-OP)

LOGICAL-OP ::= AND-OP | OR-OP | NOT-OP

AND-OP ::= (**and** EXP EXP+)

**Behavior: return** #t **if all** EXP**s** **are true.** #f **otherwise.**

Example: (and #t (> 2 1)) → #t

OR-OP ::= (**or** EXP EXP+)

**Behavior: return** #t **if at least one** EXP **is true.** #f **otherwise.**

Example: (or (> 1 2) #f) → #f

NOT-OP ::= (**not** EXP)

**Behavior: return** #t **if** EXP **is false.** #f **otherwise.**

Example: (not (> 1 2)) → #t

1. define Statement (DEF-STMT)

DEF-STMT ::= (**define** id EXP)

VARIABLE ::= id

**Behavior: Define a variable named** id **whose value is** EXP**.**

Example:

(define x 5)

(+ x 1) → 6

Note: Redefining is not allowed.

1. Function

FUN-EXP ::= (**fun** FUN-IDs FUN-BODY)

FUN-IDs ::= (id\*)

FUN-BODY ::= EXP

FUN-CALL ::= (FUN-EXP PARAM\*)

| (FUN-NAME PARAM\*)

PARAM ::= EXP

LAST-EXP ::= EXP

FUN-NAME ::= id

**Behavior:**

FUN-EXP **defines a function. When a function is called, bind** FUN-IDs **to** PARAM**s, just like the** define **statement. If an** id **has been defined outside this function, prefer the definition inside the** FUN-EXP**. The variable definitions inside a function should not affect the outer scope. A** FUN-CALL **returns the evaluated result of** FUN-BODY **Note that variables used in** FUN-BODY **should be bound to** PARAM**s**

Examples:

((fun (x) (+ x 1)) 2) → 3

↑ fun-exp ↑ fun-call

(define foo (fun () 0))

(foo) → 0

(define x 1)

(define bar (fun (x y) (+ x y)))

(bar 2 3) → 5

x → 1

1. if Expression

IF-EXP ::= (**if** TEST-EXP THEN-EXP ELSE-EXP)

TEST-EXP ::= EXP

THEN-EXP ::= EXP

ELSE-EXP ::= EXP

**Behavior: When** TEST-EXP **is true, returns** THEN-EXP**. Otherwise, returns** ELSE-EXP**.**

Example:

(if (= 1 0) 1 2) → 2

(if #t 1 2) → 1