**L4850: A Dynamically Typed Language for Instruction**

L4850 is a programming language designed for practice interpretation. L4850 is a simple dynamically typed language that contains elements of functional, object-oriented and imperative paradigms. L4850 is intended to be simple enough to interpret in a single semester by any student willing to put in some effort. Each feature included in the language was added specifically to help students learn about the meaning of programs written in different paradigms. The goal of this project is to help student understand what it means to express something in a language by writing an operational semantics (fancy words for interpreter). In the process, I hope that you will understand object-oriented design better and also learn about functional programming.

# 1 L4850 Syntax

## 1.1 Lexical Properties of L4850

1. In L4850, blanks are significant.

2. In L4850, all keywords are reserved; that is, the programmer cannot use an L4850 keyword as the name of a variable. The valid keywords are:

   $\langle$**ASSIGN**$\rangle$ $\rightarrow$ `assign`
   $\langle$**COND**$\rangle$ $\rightarrow$ `cond`
   $\langle$**DEFCLASS**$\rangle$ $\rightarrow$ `defclass`
   $\langle$**DEFUNC**$\rangle$ $\rightarrow$ `defunc`
   $\langle$**ELSE**$\rangle$ $\rightarrow$ `else`
   $\langle$**FALSE**$\rangle$ $\rightarrow$ `false`
   $\langle$**FI**$\rangle$ $\rightarrow$ `fi`
   $\langle$**FUNC**$\rangle$ $\rightarrow$ `func`
   $\langle$**IF**$\rangle$ $\rightarrow$ `if`
   $\langle$**LOAD**$\rangle$ $\rightarrow$ `load`
   $\langle$**METHOD**$\rangle$ $\rightarrow$ `method`
   $\langle$**NEW**$\rangle$ $\rightarrow$ `new`
   $\langle$**THEN**$\rangle$ $\rightarrow$ `then`
   $\langle$**TO**$\rangle$ $\rightarrow$ `to`
   $\langle$**TRUE**$\rangle$ $\rightarrow$ `true`
   $\langle$**VARS**$\rangle$ $\rightarrow$ `vars`
   $\langle$**WITH**$\rangle$ $\rightarrow$ `with`

   (Note that L4850 is *case sensitive,* that is, the variable `X` differs from `x`. Thus, `cond` is a keyword, but `COND` can be a variable name.)

3. The following special characters have meanings in an L4850 program.

   ⟨**AND**⟩ → `&&`
   ⟨**COMMA**⟩ → `,`
   ⟨**DIVIDE**⟩ → `/`
   ⟨**DOT**⟩ → `.`
   ⟨**EQUAL**⟩ → `==`
   ⟨**GREATER**⟩ → `>`
   ⟨**GREATEREQUAL**⟩ → `>=`
   ⟨**INVOKE**⟩ → `->`
   ⟨**LB**⟩ → `{`
   ⟨**LBK**⟩ → `[`
   ⟨**LESS**⟩ → `<`
   ⟨**LESSEQUAL**⟩ → `<=`
   ⟨**LP**⟩ → `(`
   ⟨**MINUS**⟩ → `-`
   ⟨**MULTIPLY**⟩ → `*`
   ⟨**NOT**⟩ → `!`
   ⟨**NOTEQUAL**⟩ → `!=`
   ⟨**OR**⟩ → `||`
   ⟨**PLUS**⟩ → `+`
   ⟨**RB**⟩ → `}`
   ⟨**RBK**⟩ → `]`
   ⟨**RP**⟩ → `)`

4. Comments are delimited by a `//`. All characters following the `//`on the same line are part of the comment.

5. Identifiers are written with upper and lowercase letters and are defined as follows:

   $\langle\textbf{ALPHA}\rangle \rightarrow \texttt{a} \mid \texttt{b} \mid \texttt{c} \mid \cdots \mid \texttt{z} \mid \texttt{A} \mid \texttt{B} \mid \cdots \mid \texttt{Z}$
   $\langle\textbf{DIGIT}\rangle \rightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \cdots \mid \texttt{9}$
   $\langle\textbf{IDENTIFIER}\rangle \rightarrow \langle\textbf{LETTER}\rangle\ (\langle\textbf{LETTER}\rangle \mid \langle\textbf{DIGIT}\rangle)^*$

6. Constants are defined as follows:

   $\langle\textbf{POSITIVE}\rangle \rightarrow \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \ldots \mid \texttt{9}$
   $\langle\textbf{INTNUM}\rangle \rightarrow \langle\textbf{POSITIVE}\rangle\ \langle\textbf{DIGIT}\rangle^* \mid \texttt{0}$
   $\langle\textbf{EXPONENT}\rangle \rightarrow (\texttt{e} \mid \texttt{E})\ (\texttt{+} \mid \texttt{-})?\ (\langle\textbf{DIGIT}\rangle)^+$
   $\langle\textbf{FLOATNUM}\rangle \rightarrow \langle\textbf{INTNUM}\rangle\ (\langle\textbf{DOT}\rangle\ (\langle\textbf{DIGIT}\rangle)^+\ (\langle\textbf{EXPONENT}\rangle)? \mid \langle\textbf{EXPONENT}\rangle)$
   $\langle\textbf{STRING}\rangle \rightarrow {}'\ (\sim[\,'\,])^*\ '$

## 1.2 Context-free Grammar

The following grammar describes the context-free syntax of L4850:

$$
\begin{array}{lcl}
\textbf{program} & \rightarrow & (\ \textbf{functionDef} \\
 & | & \quad \textbf{classDef} \\
 & | & \quad \textbf{expression} \\
 & | & \quad \textbf{loadFile}\ )^{+} \\[1em]
\textbf{functionDef} & \rightarrow & \langle\textbf{DEFUNC}\rangle\ \langle\textbf{ID}\rangle\ \langle\textbf{LP}\rangle\ (\ \textbf{idList}\ )?\ \langle\textbf{RP}\rangle\ \textbf{expressionList} \\[1em]
\textbf{idList} & \rightarrow & \langle\textbf{ID}\rangle\ (\ \langle\textbf{COMMA}\rangle\ \langle\textbf{ID}\rangle\ )^{*} \\[1em]
\textbf{classDef} & \rightarrow & \langle\textbf{DEFCLASS}\rangle\ \langle\textbf{ID}\rangle\ \langle\textbf{LB}\rangle\ (\ \textbf{classVars}\ )?\ (\ \textbf{methods}\ )?\ \langle\textbf{RB}\rangle \\[1em]
\textbf{classVars} & \rightarrow & \langle\textbf{VARS}\rangle\ \textbf{idList} \\[1em]
\textbf{methods} & \rightarrow & (\ \langle\textbf{METHOD}\rangle\ \langle\textbf{ID}\rangle\ \langle\textbf{LP}\rangle\ (\ \textbf{idList}\ )?\ \langle\textbf{RP}\rangle\ \textbf{expressionList}\ )^{+} \\[1em]
\textbf{loadFile} & \rightarrow & \langle\textbf{LOAD}\rangle\ \langle\textbf{STRING}\rangle \\[1em]
\textbf{expressionList} & \rightarrow & \langle\textbf{LB}\rangle\ (\ \textbf{expression}\ )^{+}\ \langle\textbf{RB}\rangle \\[1em]
\textbf{expression} & \rightarrow & \textbf{compExpr}\ (\ \textbf{logOp}\ \textbf{compExpr}\ )^{*} \\
 & | & \langle\textbf{NOT}\rangle\ \textbf{compExpr} \\[1em]
\textbf{logOp} & \rightarrow & \langle\textbf{OR}\rangle\ \mid\ \langle\textbf{AND}\rangle \\[1em]
\textbf{compExpr} & \rightarrow & \textbf{addExpr}\ (\ \textbf{compOp}\ \textbf{addExpr}\ )^{*} \\[1em]
\textbf{compOp} & \rightarrow & \langle\textbf{EQUAL}\rangle\ \mid\ \langle\textbf{NOTEQUAL}\rangle\ \mid\ \langle\textbf{LESS}\rangle\ \mid\ \langle\textbf{LESSEQUAL}\rangle \\
 & | & \langle\textbf{GREATER}\rangle\ \mid\ \langle\textbf{GREATEREQUAL}\rangle \\[1em]
\textbf{addExpr} & \rightarrow & \textbf{mulExpr}\ (\ \textbf{addOp}\ \textbf{mulExpr}\ )^{*} \\[1em]
\textbf{addOp} & \rightarrow & \langle\textbf{PLUS}\rangle\ \mid\ \langle\textbf{MINUS}\rangle \\[1em]
\textbf{mulExpr} & \rightarrow & \textbf{factor}\ (\ \textbf{mulOp}\ \textbf{factor}\ )^{*} \\[1em]
\textbf{mulOp} & \rightarrow & \langle\textbf{MULTIPLY}\rangle\ \mid\ \langle\textbf{DIVIDE}\rangle
\end{array}
$$

| | | |
|---|---|---|
| **factor** | → | **operand ( call )?** |
| **operand** | → | **varRef** \| **constant** \| **newExpr** \| **ifExpr** |
| | \| | **funcExpr** \| **assignExpr** \| **condExpr** |
| | \| | **withExpr** \| **⟨LP⟩ expression ⟨RP⟩** |
| **call** | → | **⟨INVOKE⟩ ⟨LP⟩ ( paramList )? ⟨RP⟩** |
| **paramList** | → | **expression ( ⟨COMMA⟩ expression)**$^*$ |
| **varRef** | → | **⟨ID⟩ ( ⟨DOT⟩ ⟨ID⟩ )?** |
| **constant** | → | **⟨INTUM⟩** \| **⟨FLOATNUM⟩** \| **list** \| **⟨STRING⟩** |
| | \| | **⟨TRUE⟩** \| **⟨FALSE⟩** |
| **list** | → | **⟨LBK⟩ constantList ⟨RBK⟩** |
| **constantList** | → | **constant ( ⟨COMMA⟩ constant )**$^*$ |
| **newExpr** | → | **⟨NEW⟩ ⟨ID⟩** |
| **ifExpr** | → | **⟨IF⟩ expression ⟨THEN⟩ expression ⟨ELSE⟩ expression ⟨FI⟩** |
| **funcExpr** | → | **⟨FUNC⟩ ⟨LP ⟩ ( idList )? ⟨ RP⟩ expressionList** |
| **assignExpr** | → | **⟨ASSIGN⟩ expression ⟨TO⟩ ⟨ID⟩** |
| **condExpr** | → | **⟨COND⟩ condClauses** |
| **condClauses** | → | **( ⟨LB⟩ expression expression ⟨RB⟩ )**$^+$ |
| **withExpr** | → | **⟨WITH⟩ ⟨LP⟩ variableDefs ⟨RP⟩ expressionList** |
| **variableDefs** | → | **( ⟨LBK⟩ ⟨ID⟩ expression ⟨RBK⟩ )**$^*$ |

# 2 L4850 Notes

## 2.1 Function Declarations

The semantics of function definition cause the variable to be added to the environment with a closure as a value.

```
Example:

defunc even (n)  {
    if (n == 0) then
        true
   else
       odd->(n - 1)
   fi
 }

defunc odd(n) {
    if (n == 0) then
        false
    else
        even->(n - 1)
    fi
 }
```

In the above program, the variables `even` and `odd` are added to the environment with closures for values. The value of a function definition is a closure.

## 2.2 Assignment Expressions

A variable is given a new value using an `assign` expression.

```
assign 2+3 to x
```

In the previous expression, `x` is given the value 5 and 5 is the return value of the entire `assign` expression.

## 2.3 If Expression

If the first expression in an `if` is `true`, then the expression in the `then`-part is evaluated and returned. Otherwise the expression in the `else`-part is evaluated and returned.

```
if x == 5 then 5 else 4 fi
```

The value returned by this expression is `5`.

## 2.4 Cond Expression

The `cond` expression is a concise format for a sequence of `if-then-else-if` expressions. For example,

```
if x == 5 then
  5+2
else
  if x == 6 then
    5+3
  else
    if x == 7 then
      5+4
    else
      5+5
    fi
  fi
fi
```

can equivalently be expressed as

```
cond
  {x == 5  5+2}
  {x == 6  5+3}
  {x == 7  5+4}
  {true    5+5}
```

## 2.5 With Expressions

Variables are given scope in L4850 using a `with` expression. For example,

```
with ([x 5]) {
  x + 10
}
```

gives scope to the variable `x` in the body of the `with` expression and gives it an initial value of `5`. The expression evaluation returns the value `10`.

## 2.6 Function Invocation

A function invocation is done with the `->` operator. Thus, the expression `even->(n-1)` in the previous declaration of `odd` calls the function `even` with the value of `n-1` as an argument. L4850 also defines the language-supported functions in Table 1.

| Function Prototype | Parameter Types | | Description |
|---|---|---|---|
| `first->(L)` | `L` | a list | Return the first element of a `L` |
| `rest->(L)` | `L` | a list | Return a list contain all elements of `L` except the first |
| `insert->(e,L)` | `e`<br>`L` | any value<br>a list | Insert `e` onto the front of `L` |
| `list->(e,...)` | `e,...` | one or more values | create a list containing all parameters |
| `empty?->(L)` | `L` | a list | Return true iff `L` is an empty list |
| `pair?->(L)` | `L` | a list | true iff `L` is a non-empty list |
| `list?->(L)` | `L` | a list | Return true iff `L` is a list |
| `equal?->(L1,L2)` | `L1,L2` | lists | Return true iff `L1` and `L2` are equivalent lists |
| `length->(L)` | `L` | a list | Return the length of `Lt` |
| `number?->(e)` | `e` | any value | Return true iff `e` is a number |
| `exit->()` | | | exit the interpreter |

Table 1: L4850 Language-supported Functions

## 2.7 Function Expression

L4850 supports first-class functions. Thus, functions are values just as numbers are values. The expression

```
func (x) { x + 2 }
```

declares a nameless function that adds `2` to its argument. A function expression can be put anywhere in a program that any other expression can be put. Thus, it can be asssigned to a variable in an assignment expression or called in an invocation as in

```
(func (x) { x + 2 })->(3)
```

which returns the value `5`.

## 2.8 Expressions

L4850 arithmetic expressions compute simple values of type integer or float For both integer and floating point numbers, arithmetic and comparison are defined.

***Coercion:*** If an expression contains operands of only one type, evaluation is straight forward. When an operand contains mixed types, the situation is more complex. If an arithmetic expression has an integer operand and a float operand, the integer operand should be converted to a float before the operation is performed.

Relational and logic operators always produce a boolean. To perform a comparison between an integer and a float, the integer is converted to a float. Logic operators are only defined on boolean values. In addition, boolean values may not be used in relational and arithmetic operators.

**Operator Precedence** Operator precedences in L4850 are specified in the table below. Multiplication and division have the highest priority, `&&` and `||` have the lowest.

| Operator | Precedence |
|---|---|
| `*, /` | 5 |
| `+, -` | 4 |
| `<, <=, =, >=, >, !=` | 3 |
| `!` | 2 |
| `&&, \|\|` | 1 |

Precedence has already been encoded in the grammar in Section 1.2.

## 2.9 Classes and Objects

L4850 supports simple classes without inheritance. Classes may contain instance variables and methods. All variables have private access and all methods have public access. The code below defines a class with one instance variable and getter and setter methods for that variable.

```
defclass test {
    vars x
    method setX(n) { assign n to x }
    method getX() { x }
}
```

To instantiate an object, the `new` operator is used. In the expression

```
assign new test to t
```

a new instance of `test` is assigned to the variable `t`. Methods are accessed using the `.` operator and invoked using the `->` operator. As an example, the following code returns the value `0`.

```
defunc f (x) {
    assign new test to t
    t.setX->(x)
    t.getX->()
}

f->(0)
```