Trefoil v3 Language Spec

In this document we will only describe what is **different** from Trefoil v2.

Tokens

As in Trefoil v2.

Parenthesized Symbol Trees

As in Trefoil v2.

Abstract Syntax Tree

A *program* is a sequence of bindings, as in Trefoil v2.
A *binding* is one of the following:

- Variable
- Top-level expression
- Test
- **Function definition (new for Trefoil v3)**

An *expression* is one of the following:

- Value (int, bool, **symbol (new for Trefoil v3)**, nil, cons)
- Variable
- Arithmetic operation (addition, subtraction, multiplication, **equality (new for Trefoil v3)**)
- If expression
- **Let expression (changed in Trefoil v3)**
- Cons expression
- **Print expression (new for Trefoil v3)**
- List operation (nil?, cons?, car, cdr)
- **Cond expression (new for Trefoil v3)**
- **Function call (new for Trefoil v3)**

Semantics

The meaning of a Trefoil v3 program depends on the dynamic environment.

The meaning of an expression is its *value* in the current dynamic environment.
The meaning of a binding is the new dynamic environment produced by evaluating the binding in the current environment. Evaluating a binding can *fail*.
The meaning of a program is the dynamic environment produced by evaluating each binding in order, where each binding is evaluated in the dynamic environment produced by

evaluating the previous binding. If evaluating a binding fails, the binding is ignored, and the program continues with the next binding.

The Dynamic Environment

The dynamic environment maps string to entries. An `entry` is either a `VarEntry` containing a `value` or a `FnEntry` containing a `(fn_info * env)`.
It supports two operations:

- `bind`: Takes as input a `name`, an `entry`, and an `environment`, and produces a new environment where the input environment is extended with a mapping from `name` to `entry`.
- `lkup`: Takes as input a `name` and an `environment`, and returns the `entry` that `name` is bound to in the environment (or `None` if `name` is not bound in the environment).

Detailed Syntax and Semantics

Bindings

- Variable
  - Syntax: `(define x e)` where `x` is any Atom and `e` is any expression.
    - Example: `(define (x (+ 1 2))`
  - Semantics:
    - Evaluate `e` in the current dynamic environment to a value `v`.
    - Print the string `x = v`.
    - Return the new dynamic environment, which is the current environment extended with `x` maps to `(VarEntry v)`.
- Function
  - Syntax: `(define (f_name` a_1 a_2 … a_n`) body)` where `f_name` is an Atom that is not a Trefoil keyword, a_1, …, a_n are Atoms, and `body` is any expression.
    - Example: `(define (f x y) (+ x y))`
  - Semantics:
    - If `f_name` is the same as any of the argument names, or if any of the argument names are the same as each other, raise a syntax error.
    - Create a `fn_info` record containing the function name, argument names, and the body of the function.
    - Return the new dynamic environment, which is the current environment extended with a mapping from `f_name` to a `FnEntry` containing the `fn_info` record and the current environment (called the defining environment of the function).

Expressions

- Value
  - Syntax:
    - Integers, booleans, Nil, and Cons, as in Trefoil v2
    - Symbols: An Atom consisting of `'` followed by a non-empty sequence of characters
  - Semantics: All values evaluate to themselves
- Variables
  - Syntax: As in Trefoil v2
  - Semantics:
    - Look up `x` in the current environment.
    - If it maps to a `VarEntry` with value `v`, the variable expression evaluates to `v`.
    - If it maps to a `FnEntry` or it is not bound in the environment, signal a runtime error.
- Print
  - Syntax: `(print e)` where `e` is any expression
  - Semantics:
    - Evaluate `e` to a value, `v`
    - Print `v` (The semantics do not specify exactly how to print `v`, it is up to the implementation)
    - The result of evaluating a print expression is `Nil`.
- Let
  - Syntax: `(let ((a1 e1) (a2 e2) … (an en)) body)` where `a1`, ..., `an` are Atoms and `e1`, ..., `en` are expressions, and `body` is an expression.
  - Semantics:
    - Ensure all of the names `a1`, ..., `an` are unique.
    - Evaluate all expressions `e1`, ..., `en` in the current dynamic environment to values `v1`, ..., `vn`.
      - Note that all e_i are evaluated in the same dynamic environment, so earlier expressions are not bound when evaluating later expressions in the list of binding expressions.
    - Return the result of evaluating `body` in the current dynamic environment extended with mappings `ai -> vi` for all `i` in `1, …, n`.
- Cond
  - Syntax: `(cond c1 c2 … cn)` where each `ci` has the form `(e1 e2)` where `e1` and `e2` are expressions. Any number of cond clauses (including zero) is permitted.
    - Example: The body of this function is a cond expression `(define (sum l)`

```
(cond
  ((nil? l) 0)
  (true (+ (car l) (sum (cdr l))))))
```

    - Semantics:
        - Each clause in the (possibly empty) list of cond clauses has the form `(pi bi)`.
        - Iterate over the clauses in order. For each clause:
            - Evaluate `pi` in the current dynamic environment to a value `vi`.
            - If `vi` is `false`, continue the loop.
            - If `vi` is any other value, evaluate `bi` in the current dynamic environment to a value `v`. The result of the entire cond expression is `v`.
            - If the loop reaches the end of the sequence of clauses without finding a `pi` that evaluates to something besides `false`, signal a `RuntimeError`.
- Function Calls
    - Syntax: `(f arg1 arg2 … argn)` where `f` is an Atom that is not a Trefoil keyword and `arg1`, `arg2`, ..., `argn` are expressions.
        - Example: `(f (+ 1 2) true)` calls the function names `f` with two arguments.
    - Semantics:
        - Call the current dynamic environment `callenv`.
        - Look up the function name in the `callenv`.
        - If it is not bound, or it is not bound to a function entry, signal a runtime error.
        - Otherwise, it is bound to a function entry, which contains the function's name, the param names, the body, and the defining environment.
        - If the number of params in the function definition is different from the number of arguments passed to the function call, signal a runtime error.
        - Evaluate the arguments `arg1`, `arg2`, ..., `argn` in left-to-right order in `callenv` to values `v1`, `v2`, ..., `vn`.
        - The function call evaluates to the result of evaluating the function body in the function's defining environment, extended with mappings `pi -> vi` for each param name `pi` and corresponding value `vi`.
- Equality
    - Syntax: `(= e1 e2)` where `e1` and `e2` are expressions, as in Trefoil v2.
    - Semantics:
        - Evaluate `e1` to value `v1` in the current dynamic environment.
        - Evaluate `e2` to value `v2` in the current dynamic environment.

- If `v1` and `v2` are both integers, the result is `true` if they are the same and `false` otherwise.
- If `v1` and `v2` are both booleans, the result is `true` if they are the same and `false` otherwise.
- If `v1` and `v2` are both symbols, the result is `true` if they are the same and `false` otherwise.
- If `v1` and `v2` are both `Nil`, the result is `true`.
- If `v1 = (cons a b)` for some values `a` and `b`, and `v2 = (cons c d)` for some values `c` and `d`, the result is `true` if `a = c` and `b = d` and `false` otherwise.
- In any other case, the result is `false`.