Last updated: January 27, 2022

## 1. Introduction

This manual describes the $\lambda^+$ programming language, which is a simple, functional language small enough that a complete interpreter for $\lambda^+$ can be implemented in a quarter-long course. The programming language is similar in nature to the untyped $\lambda$-calculus, but extends the $\lambda$-calculus with constructs such as let bindings and named functions to make it more convenient to program in $\lambda^+$. The language is also very similar to real-world functional programming languages, such as Lisp and OCaml.

This manual gives an informal overview of the language, describes its syntax, and gives precise semantics to the language. At the beginning of the semester, students should only focus on the overview discussion of $\lambda^+$ in Section 2. Additional sections will be added to this manual as we progress through the course.

## 2. Overview

A $\lambda^+$ program is simply an expression, and executing the program is equivalent to evaluating the expression. For example, the simple expression

```
8
```

is a valid $\lambda^+$ program, and the value of this program is the integer 8.

The most basic expressions in $\lambda^+$ are integer constants and binary arithmetic operators, such as `+, *, -`. For example,

```
(3 + 6 - 1) * 2
```

is a valid $\lambda^+$ expression with value 16.

### 2.1. Let Bindings.

Let bindings in $\lambda^+$ allow us to name and reuse expressions. Specifically, an expression of the form

```
let x = e1 in e2
```

binds the value of e1 to identifier x and evaluates e2 under this binding. The expression e2 is referred to as the body of the let expression, and e1 is called the *initializer*. The value of the let expression is the result of evaluating e2. For example,

```
let x = 3+5 in x-2
```

evaluates to 6, while the expression

```
let x = 3+5 in x+y
```

yields the run-time error:

```
Unbound variable y
```

since the identifier y is not bound (i.e. has not been “defined”) in the body of the let expression.

Let expressions in $\lambda^+$ can be arbitrarily nested. For example, consider the nested let expressions:

```
let x = 3+5 in
let y = 2*x in
y+x
```

This is a valid $\lambda^+$ expression and evaluates to 24. Observe that the body of the first let expression is `let y = 2*x in y+x` while the body of the second (nested) let expression is y+x. As another example, consider the nested let expression:

```
let x = let x = 3 in
x+1 in x
```

evaluates to 4. The initializer for the first (outer-level) let expression is `let x=3 in x+1`, which evaluates to 4. Thus, the value of x in the body of the outer let expression is 4. As a final example of let expressions, consider:

```
let x = 2 in
let x = 3 in
x
```

evaluates to 3, since each identifier refers to the most recently bound value.

2.2. **"Boolean" operations.** For simplicity, $\lambda^+$ does not have the typical boolean values of true and false. Instead, $\lambda^+$ treats 0 as false and non-zero values as true. The supported boolean operations are &&, || as well as the comparison operators $=, >, <$. For example, the following expression evaluates to 1:

```
((1 = 1) || 3 = 4) && 1
```

$\lambda^+$ also supports *if-then-else expressions*, which evaluate to the then-expression when the condition expression is non-zero or the else-expression otherwise.

```
if 1 then 2 + 3 else 3 * 4
```

This will evaluate to 5.

Note that it is possible to simulate "else if" clauses by nesting multiple if-then-else. For example,

```
let x = 1 in
if x = 0
  then 3
  else if x = 1 then 5
  else 7
```

is understood as

```
let x = 1 in
if x = 0
  then 3
  else (if x = 1 then 5 else 7)
```

and evaluates to 5.

2.3. **Lambda Expressions and Applications.** As in $\lambda$-calculus, $\lambda^+$ also provides lambda expressions of the form:

```
lambda x1, ..., xn. e
```

For example, the $\lambda^+$ expression

```
lambda x, y. x+y
```

corresponds to an unnamed (anonymous) function that takes two arguments x and y and evaluates their sum. The above $\lambda^+$ expression is equivalent to the $\lambda^+$ expression:

```
lambda x. lambda y. x+y
```

The transformation from the first lambda expression `lambda x, y. x+y` to the second expression `lambda x. (lambda y. x+y)` is known as *currying*. We could remove multi-argument lambdas from the language without reducing its expressive power; in fact, implementations of $\lambda^+$ only need to implement single-argument lambdas, with multi-argument lambdas treated as "syntactic sugar" in $\lambda^+$. That is, multi-argument lambdas merely provide a more convenient way to write expressions that can already be expressed using other constructs in the language (namely single-argument lambdas, here).

Of course, for lambda expressions to be useful, we also need to be able to apply arguments to lambda abstractions. Application in $\lambda^+$ is the same as application in $\lambda$-calculus, with the form `e1 e2 ... en`. As in $\lambda$-calculus, application is left-associative, so that `e1 e2 ... en` is read as `(((e1 e2) e3) ... ) en)`.

The expression `(lambda x. e1) e2` evaluates e2 with e1 bound to x. For example, the application expression

```
((lambda x. (lambda y. x + y)) 6) 7
```

evaluates to 13. Note that this can be conveniently written in $\lambda^+$ as

```
(lambda x, y. x + y) 6 7
```

Keep in mind that lambdas are right-associative while application is left-associative, i.e. the following are equivalent:

```
lambda x. lambda y. x + y      <==>     lambda x. (lambda y. x + y)
f e1 e2                        <==>     (f e1) e2
```

As a more interesting example, consider the application expression

```
(lambda x, y. x + y) 6
```

which evaluates to the lambda expression

```
lambda y. 6 + y
```

This example illustrates an interesting feature of $\lambda^+$: Expressions in $\lambda^+$ do not have to evaluate to constants; they can be *partially evaluated* functions, such as `lambda y. (6 + y)` in this example.

Here, we highlight two possible mistakes one can make using application expressions in $\lambda^+$. First, someone may write

```
lambda x. x 4
```

to try to apply a function `lambda x. x` to 4, but what this will really do is create a function that accepts an argument `x` and then apply `x` to 4. The correct way of writing this expression is

```
(lambda x. x) 4
```

As a second caveat, the application expression

```
((let x = 2 in x) 3)
```

is a syntactically valid $\lambda^+$ expression but will yield the run-time error:

```
Run-time error in expression (let x = 2 in x 3)
Only lambda expressions can be applied to other expressions
```

The problem here is that the first expression e1 in the application (e1 e2) must evaluate to a lambda expression. Only functions can be applied to arguments. On the other hand, the following expression

```
let x = lambda y. y in
(x 3)
```

is both syntactically and semantically valid and evaluates to 3.

2.4. **Named Function Definitions.** In addition to `lambda` expressions, which correspond to anonymous function definitions, the $\lambda^+$ language also makes it possible to define named functions using the syntax:

```
fun f with x1, ..., xn = e1 in e2
```

Here f is the name of the function being defined, x1, ...xn are the arguments of function f, and e1 is the body of function f. The value of the fun expression is the result of evaluating e2 under this definition of f.

Using named function definitions, we can now define recursive functions. The following program defines a recursive function for computing factorial, and the expression f 4 in the body evaluates to 4!, i.e., 24.

```
fun f with n =
  if n = 0
  then 1
  else n * (f (n-1))
in f 4
```

Like multi-argument lambdas, named function definitions are also "syntactic sugar" in $\lambda^+$. Specifically, the function definition

```
fun f with x = e1 in e2
```

is equivalent to the following let expression:

```
let f = fix (lambda f. lambda x. e1)
in e2
```

where the fix operator models *fixed-point combinators* in untyped lambda calculus.

Here is another example that illustrates the use of named functions in $\lambda^+$:

```
fun even with x =
  if x = 0 then 1
  else if x = 1 then 0
  else even (x - 2)
in
fun odd with x = even (x + 1)
in
odd 7
```

This $\lambda^+$ program evaluates to 1, as expected. To see why, observe that the sequence of function calls is:

```
  odd 7
= even 8
= even 6
= even 4
= even 2
= even 0
= 1
```

2.5. **Lists.** In addition to integers, the $\lambda^+$ language also supports linked lists. A list in $\lambda^+$ is a general data structure consisting of either:

- The empty list constant $\mathsf{Nil}$
- The *cons cell* $e_1 \,@\, e_2$, where $e_1$ is the *head* of the list and $e_2$ is the *tail* of the list.

By convention, we take @ to be right-associative, so that $1@2@3@\,\mathsf{Nil}$ is read as $1@(2@(3@\,\mathsf{Nil}))$. $\lambda^+$ supports the following list operations:

- $\mathsf{isnil}\, e$: The expression $(\mathsf{isnil}\, e)$ evaluates to 1 if $e$ is $\mathsf{Nil}$, or 0 otherwise.
- $e_1 \,@\, e_2$: The expression $e_1 \,@\, e_2$ evaluates to a new list with head corresponding to the value of $e_1$ and tail corresponding to the value of $e_2$.
- $!\, e$: The expression $!\, e$ yields the head of the list if $e$ evaluates to a cons cell, or causes a runtime error otherwise. For example, $!(2@3@\,\mathsf{Nil})$ evaluates to 2.
- $\#\, e$: The expression $\#\, e$ yields the tail of the list if $e$ evaluates to a cons cell, or causes a runtime error otherwise. For example, $\#(2@3@\,\mathsf{Nil})$ evaluates to $3@\,\mathsf{Nil}$, and $\#(1@2@3@\,\mathsf{Nil})$ evaluates to $2@3@\,\mathsf{Nil}$ and $\#(2@\,\mathsf{Nil})$ evaluates to $\mathsf{Nil}$.

Consider the following example of a function on lists:

```
fun length with list  =
  if isnil list
    then 0
    else (length #list) + 1
in
length (1 @ 2 @ 2 @ 1 @ Nil)
```

Here, we define a function length to compute the number of elements in a list and use this function on the list $(1@2@2@1@\,\mathsf{Nil})$. The above expression evaluates to 4 (exercise: try expanding this expression out by hand).

As another example, here is a program that adds n to each element in a given list:

```
fun add with l, n =
  if isnil l
  then Nil
  else (!l + n) @ (add #l n)
in add (1 @ 2 @ 3 @ Nil) 2
```

This program evaluates to the list $3 \,@\, 4 \,@\, 5 \,@\, \mathsf{Nil}$.

# 3. Syntax

In any programming language, there are two types of "syntax" that the language designer is concerned with. The first type is *concrete syntax*, the syntax used for the source code that the programmer will type in. The concrete syntax will specify how the linear sequence of *tokens* making up the source code should be converted into a data structure called a *parse tree*. The parse tree will then be converted into an *abstract syntax tree*, which is a data structure consisting of the core parts of the language with all irrelevant notational constructs (e.g. parentheses, braces, etc.) removed. Interpreters and compilers will operate on abstract syntax trees.

3.1. **Concrete Syntax.** The complete *concrete syntax* of $\lambda^+$ is specified by the context-free grammar presented in Figure 1.

$$
\begin{array}{rcl}
Program & ::= & Expr \\
Expr & ::= & let\ ID\ =\ Expr\ in\ Expr \\
 & | & fun\ ID\ with\ Idlist\ =\ Expr\ in\ Expr \\
 & | & lambda\ Idlist.\ Expr \\
 & | & fix\ Expr \\
 & | & if\ Expr\ then\ Expr1\ else\ Expr2 \\
 & | & Expr1 \oplus Expr2 \qquad (\oplus \in \{+, -, *, =, >, <, \&\&, ||\}) \\
 & | & !\,Expr \\
 & | & \#\,Expr \\
 & | & Expr1\ Expr2 \\
 & | & \mathsf{Nil} \\
 & | & Expr1\,@\,Expr2 \\
 & | & \mathsf{isnil}\ Expr \\
 & | & \mathsf{INT\_CONST} \\
 & | & ID
\end{array}
$$

Figure 1. Concrete syntax of the $\lambda^+$ language.

Observe that this grammar in Figure 1 is ambiguous, and we discuss the intended meaning of the ambiguous constructs. The first source of ambiguity in the grammar is binary operators. For example, the $\lambda^+$ expression 2*3+4 can be parsed in two ways: either as (2*3)+4 or as 2*(3+4). To disambiguate the grammar, we therefore need to declare the precedence and associativity of operators. Figure 2 shows the precedence of operators, where operators higher up in the figure have higher precedence than those lower down in the figure. Operators shown on the same line have the same precedence.

To illustrate how precedence declarations allow us to resolve ambiguities, consider again the expression 2*3+4. Since * has higher precedence than +, this means the expression should be

| |
|---|
| !, # |
| @ |
| * |
| +, − |
| <, >, = |
| &&, \|\| |

Figure 2. Operator precedence

parsed as (2*3)+4, instead of 2*(3+4). Similarly, since ! has precedence than @, this means the expression !x@y should be understood as (!x)@y rather than !(x@y).

Observe that precedence declarations are not sufficient to resolve all ambiguities concerning these operators; we also need associativity declarations. For example, precedence declarations alone are not sufficient to decide whether the expression 1+2+3 should be parsed as (1+2)+3 or as 1+(2+3). To resolve this issue, we also need to specify the associativity of the binary operators.

In the $\lambda^+$ language, the binary operators +, -, *, /, =, <, and > are all left-associative; the only right-associative operator is @. This indicates that the expression 1+2+3 should be parsed as (1+2)+3, while the expression 1@2@3 should be parsed as 1@(2@3).

3.2. **Abstract Syntax.** The complete *abstract syntax* of $\lambda^+$ is specified by the context-free grammar presented in Figure 3.

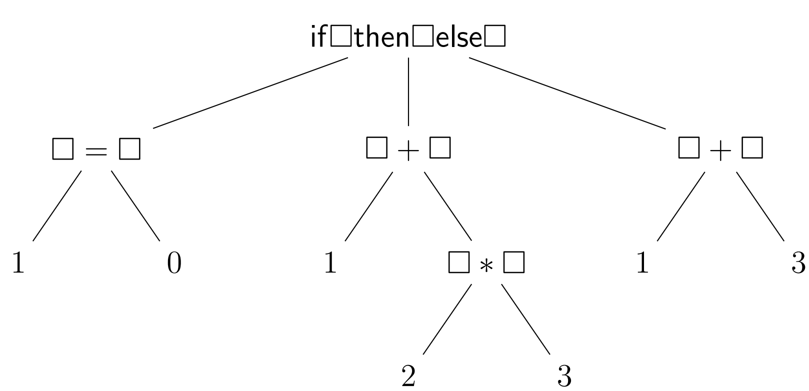
| Syntactic construct | | | Description |
|---|---|---|---|
| $\oplus \in \mathit{ArithOp}$ | ::= | $+ \mid - \mid *$ | |
| $\odot \in \mathit{Pred}$ | ::= | $= \mid < \mid > \mid \ \&\& \ \mid \ \|\|$ | |
| $\diamond \in \mathit{BinOp}$ | ::= | $\oplus \mid \odot$ | |
| $e \in \mathit{Expr}$ | ::= | $i$ | integer |
| | $\mid$ | $x$ | variable |
| | $\mid$ | $e_1 \diamond e_2$ | binary op |
| | $\mid$ | $\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$ | if-then-else |
| | $\mid$ | $\mathsf{lambda}\ x.\ e$ | lambda abstraction |
| | $\mid$ | $\mathsf{fix}$ | fixed-point operator |
| | $\mid$ | $e_1\ e_2$ | function application |
| | $\mid$ | $\mathsf{Nil}$ | empty list |
| | $\mid$ | $e_1 @ e_2$ | cons cell |
| | $\mid$ | $!\,e$ | list head |
| | $\mid$ | $\#\,e$ | list tail |
| | $\mid$ | $\mathsf{isnil}\ e$ | is nil predicate |

FIGURE 3. Abstract syntax of the $\lambda^+$ language.

The abstract syntax is a mathematical description of how to inductively construct an abstract syntax tree (AST). For example, the expression

$$\mathsf{if}\,(1 = 0)\,\mathsf{then}\,(1 + (2 * 3))\,\mathsf{else}\,(1 + 3)$$

corresponds to the following AST:



Note that there is inherently no ambiguity in an AST, because it is a data structure that encodes the order of operations.

The remaining sections in the manual will only use the abstract syntax, so that we can precisely reason about the expressions in our program.

## 4. Operational Semantics

Recall that a *semantics* of $\lambda^+$ defines the meaning of every expression in the $\lambda^+$. In *operational semantics*, we mathematically model a machine that can execute a program using some *state*. Some different methods to operational semantics are:

- A type of operational semantics called *small-step operational semantics* is based on defining *transitions* between states. To evaluate an expression, one constructs an initial state from the program and then takes transitions until a *final state* is reached. This is similar to the automata that you learned about in CS 138.
- Another type is *big-step operational semantics*, which is based on defining (often recursively or inductively) how a program can be directly executed from its expression down to a final *value*.

Although small-step is much more popular among programming language theorists, big-step is usually more intuitive to most people and how languages are typically implemented in practice. Thus, we will be using (and you will be implementing) a big-step operational semantics for $\lambda^+$ in this course.

### 4.1. The Machine Model.

Before we begin, we must understand what our machine model will require. We will need some notion of when our machine is "done" executing a program; let us use the term *value* to refer to the final expression obtained by executing a program.

We formally define a value using the following inductive definition:

- Any integer $i$ is a value.
- Any lambda expression $\mathsf{lambda}\, x.\ e$ is a value.
- $\mathsf{Nil}$ is a value.
- If $v_1, v_2$ are values, then $v_1 \mathbin{@} v_2$ are values.
- No other expression is a value.

For example, the following expressions are all values:

$$\begin{array}{l} 10 \\ \mathsf{lambda}\, x.\ 1+2 \\ \mathsf{Nil} \\ 10 \mathbin{@} \mathsf{lambda}\, y.\ y \end{array}$$

The following expressions are **<u>not</u>** values:

$$\begin{array}{l} 1+2 \\ (\mathsf{lambda}\, x.\ 1+2)10 \\ \mathsf{if\ Nil\ then}\ 10\ \mathsf{else}\ 20 \\ (1+2) \mathbin{@} \mathsf{Nil} \end{array}$$

We denote values by variations of the symbol $v$.

#### 4.1.1. *The Evaluation Relation.*

We can now define how program states are related to final values:

**Definition 4.1** (Big-step evaluation relation)**.** The big-step evaluation relation has the form

$$e \Downarrow v$$

that asserts that $e$ will evaluate to $v$.

The idea is that whenever this relation holds, then $e$ will evaluate to the final value $v$. As the reader shall see shortly, this will allow us to both 1) precisely define the meaning of "evaluation"; and 2) provide us an algorithm to perform evaluation.

4.2. **Arithmetic and Boolean Operations.** Now, we finally have all of the tools required for us to be able to specify the semantics of $\lambda^+$ expressions. Let us start by defining the *evaluation rule* for the simplest of $\lambda^+$ expressions, namely integer constants:

$$\frac{}{i \Downarrow i}\ \textsc{Int}$$

The evaluation rule $\textsc{Int}$ says that any integer constant $i$ will evaluate to itself.

This rule is not very useful by itself, so let's include an evaluation rule for addition:

$$\frac{e_1 \Downarrow i_1 \qquad e_2 \Downarrow i_2}{e_1 + e_2 \Downarrow i_1 + i_2}\ \textsc{Add}$$

which says that if $e_1$ and $e_2$ both evaluate to integers, then $e_1 + e_2$ evaluates to the sum of those integers. Note that we abuse notation here: the $+$ on the left-hand side of the conclusion is merely a symbol in the AST, while we take the $+$ on the right-hand side of the conclusion to mean integer addition.

Equipped with these two rules, we can now show how they can be used. Consider the following *proof tree* or *derivation* of how $(1+2)+4$ evaluates to 7:

$$\dfrac{\textsc{Add}\ \dfrac{\textsc{Int}\ \dfrac{}{1 \Downarrow 1} \qquad \dfrac{}{2 \Downarrow 2}\ \textsc{Int}}{1 + 2 \Downarrow 3} \qquad \dfrac{}{4 \Downarrow 4}\ \textsc{Int}}{(1+2)+4 \Downarrow 7}\ \textsc{Add}$$

This says that in order to evaluate $(1+2)+4$, first we must match against some evaluation rule. The only rule that matches is the $\textsc{Add}$ rule, so then we recursively proceed to evaluate $1+2$. But again only the $\textsc{Add}$ rule matches, so we evaluate 1 to 1 by the $\textsc{Int}$ rule (and similarly for 2). Overall, $1+2$ evaluates to 3 under $\textsc{Add}$, so then we continue through with the use of the $\textsc{Add}$ rule on $(1+2)+4$ to evaluate to 7. Note that this can be easily turned into an algorithm which can evaluate the expression.

The addition rule can be generalized to all of the arithmetic operators in $\lambda^+$:

$$\frac{e_1 \Downarrow i_1 \qquad e_2 \Downarrow i_2}{e_1 \oplus e_2 \Downarrow i_1 \oplus i_2}\ \textsc{Arith}$$

where $\oplus \in \{+, -, *, \}$. Note that we are abusing the notation slightly here: the $\oplus$ in $i_1 \oplus i_2$ is meant as the actual operation on the integers, whereas the $\oplus$ in $e_1 \oplus e_2$ is used as part of a tree of symbols.

The predicate operators $\odot \in \{=, <, >\}$ evaluate to 0 if the predicate does not hold and to 1 otherwise, as stated in the following operational semantic rules:

$$\frac{e_1 \Downarrow i_1 \qquad e_2 \Downarrow i_2 \qquad i_1 \odot i_2}{e_1 \odot e_2 \Downarrow 1} \textsc{PredTrue}$$

$$\frac{e_1 \Downarrow i_1 \qquad e_2 \Downarrow i_2 \qquad \neg(i_1 \odot i_2)}{e_1 \odot e_2 \Downarrow 0} \textsc{PredFalse}$$

Again, we are abusing notation here, where the $i_1 \odot i_2$ are taken to mean the actual predicate on integers, not on symbols in $\lambda^+$.

Lastly, we present the semantics for if-then-else expressions. The operational semantics for this expression consists of two inference rules, one in which the conditional evaluates to a non-zero value, and another in which it evaluates to zero:

$$\frac{e_1 \Downarrow i \qquad i \neq 0 \qquad e_2 \Downarrow v}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Downarrow v} \textsc{IfTrue}$$

$$\frac{e_1 \Downarrow 0 \qquad e_3 \Downarrow v}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Downarrow v} \textsc{IfFalse}$$

In the first rule IfTrue, if the expression $e_1$ evaluates to a non-zero integer $i$, then the if-then-else expression overall evaluates to the expression $e_1$ in the then branch, which yields the value $v$. Observe that in the case where $e_1$ evaluates to a non-zero value, the expression $e_3$ is never evaluated. The second rule IfFalse is similar to first one, except that when $e_1$ evaluates to 0, then evaluation proceeds by evaluating $e_3$.

4.3. **Functions and Application.** As in $\lambda$-calculus, the lambda abstractions and applications follow the substitution model of evaluation. However, we present them here for completeness.

First, consider lambda abstractions. Since they are essentially function definitions, we cannot really evaluate them until they are "called", i.e., they are applied to some value. Thus, lambda abstractions just evaluate to themselves:

$$\frac{}{\mathsf{lambda}\ x.\ e \Downarrow \mathsf{lambda}\ x.\ e} \textsc{Lambda}$$

Now consider the application rule.

$$\frac{e_1 \Downarrow \mathsf{lambda}\ x.\ e_1' \qquad e_2 \Downarrow v \qquad [x \mapsto v]e_1' \Downarrow v'}{(e_1\ e_2) \Downarrow v'} \textsc{App}$$

To evaluate the application $(e_1\ e_2)$, we first evaluate the expression $e_1$. Note that application is semantically nonsensical if the expression $e_1$ is not a lambda abstraction; thus, the operational semantics "get stuck" if $e_1$ is not a lambda abstraction of the form $\mathsf{lambda}\ x.\ e_1'$. This notion of "getting stuck" in the operational semantics corresponds to having a runtime error. Assuming the expression $e_1$ evaluates to a lambda expression $\mathsf{lambda}\ x.\ e_1'$, next we evaluate the argument $e_2$ to $v$. Lastly, we substitute $x$ with $v$ in the function body $e1'$ $x$ as in $\beta$-reduction in lambda calculus.

Note that we evaluate $e_2$ first before binding it to $x$; thus, $\lambda^+$ uses a *call-by-value* evaluation order. An alternate semantics may choose not to evaluate $e_2$, instead directly substituting it into the function body; this alternate evaluation order is called *call-by-name*.

4.4. **Let Bindings and Variables.** The meaning of let bindings in $\lambda^+$ can be defined in terms of lambda abstractions and applications[1]: a let expression $\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2$ is equivalent to $(\mathsf{lambda}\, x.\ e_2)e_1$, so the rule for $\mathsf{let}$ expressions is defined as as follows:

$$\frac{e_1 \Downarrow v_1 \qquad [x \mapsto v_1]e_2 \Downarrow v_2}{\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 \Downarrow v_2}\ \textsc{Let}$$

To evaluate a let expression $\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2$, we first evaluate the initial expression $e_1$, which yields value $v_1$. Then, to evaluate the body $e_2$, we substitute occurrences of identifier $x$ in $e_2$ with value $v_1$, and evaluate the substituted expression, which yields value $v_2$, the result of evaluating the entire $\mathsf{let}$ expression.

4.5. **The Fixed-point Operator.** The $\mathsf{fix}$ operator behaves like various fixed-point operators in untyped lambda calculus: it approximates a recursive function by unrolling the generator of the recursive function on demand:

$$\frac{e \Downarrow \mathsf{lambda}\, f.\ e' \qquad [f \mapsto \mathsf{fix}\ (\mathsf{lambda}\, f.\ e')]e' \Downarrow v}{\mathsf{fix}\ \ e \Downarrow v}\ \textsc{Fix}$$

That is, to evaluate a fixed-point expression $\mathsf{fix}\ \ e$, we first evaluate $e$ to a lambda expression $\mathsf{lambda}\, f.\ e'$, which is the generator of a recursive function that refers to itself as $f$. We then apply the lambda expression to a copy of the fixed-point expression (i.e. substituting $\mathsf{fix}\,(\mathsf{lambda}\, f.\ e')$ for $f$ in $e'$), essentially unrolling the body of the recursive function once.

4.6. **List Operators.** Recall that a list is either the empty list $\mathsf{Nil}$, or it is a *cons cell* $(e_1 \mathbin{@} e_2)$ where $e_1$ is the head of the list and $e_2$ is the tail of the list. The evaluation rules for constructing lists are shown below:

$$\frac{}{\mathsf{Nil} \Downarrow \mathsf{Nil}}\ \textsc{Nil} \qquad\qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1 \mathbin{@} e_2 \Downarrow v_1 \mathbin{@} v_2}\ \textsc{Cons}$$

Now that we have defined the meaning of $\mathsf{Nil}$, we can now state the semantics of the $\mathsf{isnil}$ operator. Since any list value can either be $\mathsf{Nil}$ or a cons cell, we have two cases:

$$\frac{e \Downarrow \mathsf{Nil}}{\mathsf{isnil}\, e \Downarrow 1}\ \textsc{IsNilTrue} \qquad\qquad \frac{e \Downarrow v_1 \mathbin{@} v_2}{\mathsf{isnil}\, e \Downarrow 0}\ \textsc{IsNilFalse}$$

Here, which rule matches triggered will depend on whether $e$ evaluates to $\mathsf{Nil}$ or not. If $e$ is not a list, then the evaluation will get stuck.

Next, let us discuss the semantics of the ! and # operators. These operators can clearly only be applied to cons cells, so the evaluation rules are simply:

$$\frac{e \Downarrow v_1 \mathbin{@} v_2}{!\, e \Downarrow v_1}\ \textsc{Head} \qquad\qquad \frac{e \Downarrow v_1 \mathbin{@} v_2}{\#\, e \Downarrow v_2}\ \textsc{Tail}$$

4.7. **Summary.** The evaluation rules in the big-step operational semantics for $\lambda^+$ are summarized in Figure 4.

[1]The reason we retain let bindings in the abstract syntax, instead of treating it as syntactic sugars, is because the static behavior (i.e. the typing rule) of a let binding is different from the static behavior of its de-sugared form, and hence requires special treatment. For more details, please refer to the next section.

$$\frac{}{i \Downarrow i}\ \textsc{Int} \qquad \frac{e_1 \Downarrow i_1 \qquad e_2 \Downarrow i_2}{e_1 \oplus e_2 \Downarrow i_1 \oplus i_2}\ \textsc{Arith}$$

$$\frac{e_1 \Downarrow i_1 \qquad e_2 \Downarrow i_2 \qquad i_1 \odot i_2}{e_1 \odot e_2 \Downarrow 1}\ \textsc{PredTrue}$$

$$\frac{e_1 \Downarrow i_1 \qquad e_2 \Downarrow i_2 \qquad \neg(i_1 \odot i_2)}{e_1 \odot e_2 \Downarrow 0}\ \textsc{PredFalse}$$

$$\frac{e_1 \Downarrow i \qquad i \neq 0 \qquad e_2 \Downarrow v}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Downarrow v}\ \textsc{IfTrue} \qquad \frac{e_1 \Downarrow 0 \qquad e_3 \Downarrow v}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Downarrow v}\ \textsc{IfFalse}$$

$$\frac{}{\mathsf{lambda}\ x.\ e \Downarrow \mathsf{lambda}\ x.\ e}\ \textsc{Lambda}$$

$$\frac{e_1 \Downarrow \mathsf{lambda}\ x.\ e_1' \qquad e_2 \Downarrow v \qquad [x \mapsto v]e_1' \Downarrow v'}{(e_1\ e_2) \Downarrow v'}\ \textsc{App}$$

$$\frac{e_1 \Downarrow v_1 \qquad [x \mapsto v_1]e_2 \Downarrow v_2}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow v_2}\ \textsc{Let}$$

$$\frac{e \Downarrow \mathsf{lambda}\ f.\ e' \qquad [f \mapsto \mathsf{fix}\ (\mathsf{lambda}\ f.\ e')]e' \Downarrow v}{\mathsf{fix}\ \ e \Downarrow v}\ \textsc{Fix}$$

$$\frac{}{\mathsf{Nil} \Downarrow \mathsf{Nil}}\ \textsc{Nil} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1 \mathbin{@} e_2 \Downarrow v_1 \mathbin{@} v_2}\ \textsc{Cons}$$

$$\frac{e \Downarrow \mathsf{Nil}}{\mathsf{isnil}\ e \Downarrow 1}\ \textsc{IsNilTrue} \qquad \frac{e \Downarrow v_1 \mathbin{@} v_2}{\mathsf{isnil}\ e \Downarrow 0}\ \textsc{IsNilFalse}$$

$$\frac{e \Downarrow v_1 \mathbin{@} v_2}{!e \Downarrow v_1}\ \textsc{Head} \qquad \frac{e \Downarrow v_1 \mathbin{@} v_2}{\# e \Downarrow v_2}\ \textsc{Tail}$$

FIGURE 4. Evaluation rules in the big-step operational semantics of the $\lambda^+$ language.