Last updated: February 10, 2024 at 9:31pm

1. INTRODUCTION

This manual describes the λ^+ programming language, which is a simple, functional language small enough that a complete interpreter for λ^+ can be implemented in a quarter-long course. The programming language is similar in nature to the untyped λ-calculus, but extends the λ-calculus with constructs such as integer and boolean arithmetic, let-bindings, and named functions to make it more convenient to program in λ^+. The language is also very similar to real-world functional programming languages, such as 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 λ^+ in Section 2. Additional sections will be added to this manual as we progress through the course.

2. OVERVIEW

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

```
8
```

is a valid λ^+ program, and the value of this program is the integer 8.

The most basic expressions in λ^+ are integer constants and binary arithmetic operators, such as +, *, −. For example,

```
(3 + 6 − 1) * 2
```

is a valid λ^+ expression with value 16.

2.1. Let Bindings.

Let bindings in λ^+ 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 λ^+ 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 λ^+ 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. **Lambda Expressions and Applications.** As in λ-calculus, λ^+ also provides lambda expressions of the form:

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

For example, the λ^+ 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 λ^+ expression is equivalent to the λ^+ 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 λ^+ only need to implement single-argument lambdas, with multi-argument lambdas treated as "syntactic sugar" in λ^+. 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 λ^+ is the same as application in λ-calculus, with the form `e1 e2 ... en`. As in λ-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 λ^+ 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 λ^+: Expressions in λ^+ 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 λ^+. 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 λ^+ 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.

2.3. **Named Function Definitions.** In addition to `lambda` expressions, which correspond to anonymous function definitions, the λ^+ 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. Named functions are also syntactic sugars, and the λ^+ implementation translates it into a let-binding and an anonymous function:

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

2.4. **Boolean expressions.** λ^+ has two introduction forms for constructing boolean values – true and false – and one elimination form for consuming boolean values: if e_1 then e_2 else e_3. It also provides integer comparison operators $=, >, <$.

For example,

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

will evaluate to 12, while

```
if 1 = 1 then 3 < 4 else true
```

will evaluate to false.

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.

2.5. **Recursive Functions.** By default, named functions are non-recursive. To define recursive functions, you can add keyword `rec` after `fun`. 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 rec f with n =
  if n = 0
  then 1
  else n * (f (n-1))
in f 4
```

Recursive function definitions are also "syntactic sugar" in λ^+. Specifically, the function definition

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

is equivalent to the following let expression:

```
let f = fix f is lambda x. e1
in e2
```

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

Here is another example that illustrates the use of named functions in λ^+:

```
fun rec 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 λ^+ 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
```

2.6. **Lists.** In addition to integers, the λ^+ language also supports linked lists. A list in λ^+ is a general data structure that can be constructed from:

- the empty list constant Nil, or
- the *cons* constructor $e_1 :: e_2$, 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 $1 :: 2 :: 3 :: \text{Nil}$ is read as $1 :: (2 :: (3 :: \text{Nil}))$.

To use/consume a given list, λ^+ supports pattern-matching on lists:

```
match e1 with
| Nil -> e2
| x::y -> e3
end
```

The meaning of the above expression is:

(1) Evaluate e1 to some valuet v
(2) If v is Nil, evaluate expression e2.
(3) If v is $v_1 :: v_2$, then bind v1 to variable x, v2 to variable y, and then evaluate e3.

Consider the following example of a function on lists:

```
fun rec length with l  =
    match l with
    | Nil -> 0
    | h::t -> (length t) + 1
    end
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 $(1 :: 2 :: 2 :: 1 :: Nil)$. 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 rec add with l, n =
  match l with
  | Nil -> Nil
  | h::t -> (h + n) :: (add t n)
  end
in add (1 :: 2 :: 3 :: Nil) 2
```

This program evaluates to the list $3 :: 4 :: 5 :: \text{Nil}$.

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.** If you'd like to write down λ^+ programs in a text editor, please refer to the context-free grammar presented in Figure 1.

$$
\begin{array}{lll}
Expr & ::= & let\ ID\ =\ Expr\ in\ Expr \\
& | & fun\ ID\ with\ Idlist\ =\ Expr\ in\ Expr \\
& | & fun\ rec\ ID\ with\ Idlist\ =\ Expr\ in\ Expr \\
& | & lambda\ Idlist.\ Expr \\
& | & fix\ ID\ is\ Expr \\
& | & if\ Expr\ then\ Expr1\ else\ Expr2 \\
& | & Expr1 \oplus Expr2 \quad (\oplus \in \{+, -, *, =, >, <\}) \\
& | & \mathsf{Nil} \\
& | & Expr1 :: Expr2 \\
& | & match\ Expr\ with\ |\ Nil \rightarrow Expr\ |\ ID :: ID\ \rightarrow Expr\ end \\
& | & \mathsf{INT\_CONST} \\
& | & ID
\end{array}
$$

FIGURE 1. Concrete syntax of the λ^+ 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 λ^+ 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 parsed as (2*3)+4, instead of 2*(3+4). Similarly, since ::has higher precedence than anything else, this means the expression x::y+1 should be understood as (x ::y) + 1 rather than (x ::y) + 1.

Observe that precedence declarations are not sufficient to resolve all ambiguities concerning these operators; we also need associativity declarations. For example, precedence declarations

$*$
$+, -$
$<, >, =$
$::$

FIGURE 2. Operator precedence

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 λ^+ 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 abstract syntax is a mathematical description of how to inductively construct an abstract syntax tree (AST). The abstract syntax of λ^+ is specified by the context-free grammar presented in Figure 3. Note that syntactic sugars, such as multi-argument lambdas, are no longer present.

Syntactic construct	Description
$\oplus \in ArithOp$ ::= $+ \mid - \mid *$	
$\odot \in Pred$ ::= $= \mid < \mid >$	
$\diamond \in BinOp$ ::= $\oplus \mid \odot$	
$e \in Expr$ ::= i	integer
$\mid x$	variable
$\mid e_1 \diamond e_2$	binary op
$\mid$ if e_1 then e_2 else e_3	if-then-else
$\mid$ lambda $x.\ e$	lambda abstraction
$\mid$ let $x = e_1$ in e_2	let binding
$\mid$ fix f is e	fixed-point operator
$\mid e_1\ e_2$	function application
$\mid$ Nil	empty list
$\mid e_1 :: e_2$	cons cell
$\mid$ match e_1 with Nil $\rightarrow e_2 \mid x::y \rightarrow e_3$ end	list pattern-match

FIGURE 3. Abstract syntax of the λ^+ language.

For example, the expression "if $(1 = 0)$ then $(1 + (2 * 3))$ else $(1 + 3)$" corresponds to the following AST:
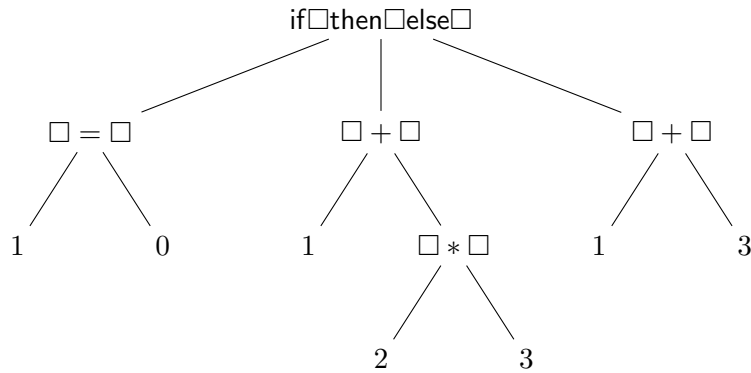
Note that there is inherently no ambiguity in an AST, because it is a data structure that encodes the order of operations.

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 λ^+ defines the meaning of every expression in the λ^+. 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 λ^+ 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 lambda $x.\ e$ is a value.
- Any boolean constant is a value
- Nil is a value.
- If v_1, v_2 are values, then $v_1 :: v_2$ are values.
- No other expression is a value.

For example, the following expressions are all values:

$$10$$
$$\mathsf{lambda}\ x.\ 1+2$$
$$\mathsf{true}$$
$$\mathsf{Nil}$$
$$10 :: \mathsf{lambda}\ y.\ y$$

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

$$1+2$$
$$(\mathsf{lambda}\ x.\ 1+2)10$$
$$\mathsf{if\ Nil\ then}\ 10\ \mathsf{else}\ 20$$
$$(1+2) :: \mathsf{Nil}$$

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 λ^+ expressions. Let us start by defining the *evaluation rule* for the simplest of λ^+ expressions, namely integer constants:

$$\frac{}{i \Downarrow i} \text{ INT}$$

The evaluation rule 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:

$$\frac{e_1 \Downarrow i_1 \qquad e_2 \Downarrow i_2}{e_1 + e_2 \Downarrow i_1 + i_2} \text{ ADD}$$

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:

$$\frac{\dfrac{\overline{1 \Downarrow 1} \text{ INT} \qquad \overline{2 \Downarrow 2} \text{ INT}}{1 + 2 \Downarrow 3} \text{ ADD} \qquad \overline{4 \Downarrow 4} \text{ INT}}{(1 + 2) + 4 \Downarrow 7} \text{ ADD}$$

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 ADD rule, so then we recursively proceed to evaluate $1 + 2$. But again only the ADD rule matches, so we evaluate 1 to 1 by the INT rule (and similarly for 2). Overall, $1 + 2$ evaluates to 3 under ADD, so then we continue through with the use of the 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 λ^+:

$$\frac{e_1 \Downarrow i_1 \qquad e_2 \Downarrow i_2}{e_1 \oplus e_2 \Downarrow i_1 \oplus i_2} \text{ ARITH}$$

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 false if the predicate does not hold and to true otherwise, as stated in the following operational semantic rules:

$$\frac{e_1 \Downarrow i_1 \qquad e_2 \Downarrow i_2 \qquad i_1 \odot i_2 \text{ holds}}{e_1 \odot e_2 \Downarrow \text{true}} \text{ PREDTRUE}$$

$$\frac{e_1 \Downarrow i_1 \qquad e_2 \Downarrow i_2 \qquad i_1 \odot i_2 \text{ does not hold}}{e_1 \odot e_2 \Downarrow \text{false}} \text{ PREDFALSE}$$

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 λ^+.

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 true, and another in which it evaluates to false:

$$\frac{e_1 \Downarrow \text{true} \qquad e_2 \Downarrow v}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v} \text{ IFTRUE}$$

$$\frac{e_1 \Downarrow \text{false} \qquad e_3 \Downarrow v}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v} \text{ IFFALSE}$$

In the first rule IFTRUE, if the expression e_1 evaluates to true, then the if-then-else expression should evaluate e_2 which is the then branch, and yields the value v. If e_1 evaluates to false, the expression e_3 is never evaluated. The second rule IFFALSE is similar to first one, except that when e_1 evaluates to false, then evaluation proceeds by evaluating e_3.

4.3. **Functions and Application.** As in λ-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:

$$\frac{}{\text{lambda } x. \; e \Downarrow \text{lambda } x. \; e} \text{ LAMBDA}$$

Now consider the application rule.

$$\frac{e_1 \Downarrow \text{lambda } x. \; e_1' \qquad e_2 \Downarrow v \qquad [x \mapsto v]e_1' \Downarrow v'}{(e_1 \; e_2) \Downarrow v'} \text{ APP}$$

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 $\text{lambda } x. \; e_1'$. 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 $\text{lambda } x. \; e_1'$, next we evaluate the argument e_2 to v. Lastly, we substitute x with v in the function body $e1' \; x$ as in β-reduction in lambda calculus.

Note that we evaluate e_2 first before binding it to x; thus, λ^+ 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 λ^+ 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 let expressions is defined as as follows:

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

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 let expression.

4.5. **The Fixed-point Operator.** Giving a precise meaning to recursive functions can be quite tricky. In λ^+, recursive function definitions are desugared to use the fixed-point operator fix, so the meaning of recursive functions depends on the meaning of fix. The fix operator behaves like various fixed-point operators in untyped lambda calculus: it evaluates a recursive function by unrolling its definition, while replacing any recursive call with a copy of itself:

$$\frac{e[f \mapsto \mathsf{fix}\, f\ \mathsf{is}\ e] \Downarrow v}{\mathsf{fix}\, f\ \mathsf{is}\ e \Downarrow v}\ \text{FIX}$$

That is, to evaluate a fixed-point expression $\mathsf{fix}\, f\ \mathsf{is}\ e$, we substistute any occurrence of f in e – which you can think of as recursive calls – with the $\mathsf{fix}\, f\ \mathsf{is}\ e$ itself. Essentially, we're unrolling the body of the recursive function once. We then evaluate the result of the substitution into a value v.

4.6. **List Operators.** Recall that a list is either the empty list Nil, or it is a *cons cell* $(e_1 :: e_2)$ 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:

$$\frac{}{\mathsf{Nil} \Downarrow \mathsf{Nil}}\ \text{NIL} \qquad\qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1 :: e_2 \Downarrow v_1 :: v_2}\ \text{CONS}$$

Now that we have defined the meaning of Nil, we can now state the semantics of the pattern-matching on lists. Since any list value can either be Nil or a $(::)$ constructor, we have two cases:

$$\frac{e_1 \Downarrow \mathsf{Nil} \qquad e_2 \Downarrow v}{\mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{Nil} \to e_2 \mid x :: y \to e_3\ \mathsf{end} \Downarrow v}\ \text{MATCHNIL}$$

$$\frac{e_1 \Downarrow v_1 :: v_2 \qquad e_3[x \mapsto v_1][y \mapsto v_2] \Downarrow v_3}{\mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{Nil} \to e_2 \mid x :: y \to e_3\ \mathsf{end} \Downarrow v_3}\ \text{MATCHCONS}$$

The $\mathsf{MatchNil}$ rule says that if e_1 evaluates to Nil, then we branch to the Nilcase, and evaluate e_2. The $\mathsf{MatchCons}$ rule says that if e_1 evaluates to a $(::)$ constructor whose head value is v_1 and tail value is v_2, then we branch into the $(::)$-case of the match, and evaluate e_3. Since inside e_3, we may have referred to the head using name x and the tail using the name y, we substistute x with v_1 and y with v_2 in e_3 before evaluating it.

4.7. **Summary.** The evaluation rules in the big-step operational semantics for λ^+ 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.

$$\frac{}{i \Downarrow i} \text{ INT} \qquad\qquad \frac{e_1 \Downarrow i_1 \qquad e_2 \Downarrow i_2}{e_1 \oplus e_2 \Downarrow i_1 \oplus i_2} \text{ ARITH}$$

$$\frac{e_1 \Downarrow i_1 \qquad e_2 \Downarrow i_2 \qquad i_1 \odot i_2 \text{ holds}}{e_1 \odot e_2 \Downarrow \text{true}} \text{ PREDTRUE}$$

$$\frac{e_1 \Downarrow i_1 \qquad e_2 \Downarrow i_2 \qquad i_1 \odot i_2 \text{ does not hold}}{e_1 \odot e_2 \Downarrow \text{false}} \text{ PREDFALSE}$$

$$\frac{e_1 \Downarrow \text{true} \qquad e_2 \Downarrow v}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v} \text{ IFTRUE} \qquad \frac{e_1 \Downarrow \text{false} \qquad e_3 \Downarrow v}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v} \text{ IFFALSE}$$

$$\frac{}{\text{lambda } x.\ e \Downarrow \text{lambda } x.\ e} \text{ LAMBDA}$$

$$\frac{e_1 \Downarrow \text{lambda } x.\ e_1' \qquad e_2 \Downarrow v \qquad [x \mapsto v]e_1' \Downarrow v'}{(e_1 \ e_2) \Downarrow v'} \text{ APP}$$

$$\frac{e_1 \Downarrow v_1 \qquad [x \mapsto v_1]e_2 \Downarrow v_2}{\text{let } x = e_1 \text{ in } e_2 \Downarrow v_2} \text{ LET}$$

$$\frac{e[f \mapsto \text{fix } f \text{ is } e] \Downarrow v}{\text{fix } f \text{ is } e \Downarrow v} \text{ FIX}$$

$$\frac{}{\text{Nil} \Downarrow \text{Nil}} \text{ NIL} \qquad\qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1 :: e_2 \Downarrow v_1 :: v_2} \text{ CONS}$$

$$\frac{e_1 \Downarrow \text{Nil} \qquad e_2 \Downarrow v}{\text{match } e_1 \text{ with Nil} \rightarrow e_2 \mid x :: y \rightarrow e_3 \text{ end} \Downarrow v} \text{ MATCHNIL}$$

$$\frac{e_1 \Downarrow v_1 :: v_2 \qquad e_3[x \mapsto v_1][y \mapsto v_2] \Downarrow v_3}{\text{match } e_1 \text{ with Nil} \rightarrow e_2 \mid x :: y \rightarrow e_3 \text{ end} \Downarrow v_3} \text{ MATCHCONS}$$

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

5. Types

5.1. **Basic definitions.** The syntax of *types* in λ^+ is shown in Figure 5.

$$\mathsf{T} ::= \mathsf{T}_1 \to \mathsf{T}_2$$
$$| \quad \mathsf{Int}$$
$$| \quad \mathsf{Bool}$$
$$| \quad \mathsf{List}[\mathsf{T}]$$

FIGURE 5. Types of the λ^+ language.

5.2. **Language Extension.** We shall update the syntax of *expressions* in λ^+ with type annotations:

e	::=	$\ldots$	(as before)
		$\mathsf{Nil}[\mathsf{T}]$	empty list
		$\mathsf{lambda}\, x : \mathsf{T}.\, e$	lambda abstraction
		$\mathsf{fix}\, f : \mathsf{T}\, \mathsf{is}\, e$	fixed-point operator
		$(e\, @\, \mathsf{T})$	type annotation

That is, we require type annotations on the Nil constructor, function parameters, and the name bound by the fixed-point operator. We also introduce a new syntax "$(e\, @\, \mathsf{T})$" that annotates the expression e with the type T.

Although type annotations are important for type checking, they are transparent to the operational semantics of the language. In other words, the expression $(e\, @\, \mathsf{T})$ behaves exactly like e in terms of evaluation.

$$\frac{e \Downarrow v}{(e\, @\, \mathsf{T}) \Downarrow v} \; \text{Annot}$$

The same goes for $\mathsf{Nil}[\mathsf{T}]$, $\mathsf{lambda}\, x : \mathsf{T}.\, e$, and $\mathsf{fix}\, f : \mathsf{T}\, \mathsf{is}\, e$.

Definition 5.1 (Typing Environment). A typing environment (or typing context) Γ is a mapping from variables to types

$$\Gamma = x_1 : \mathsf{T}_1, x_2 : \mathsf{T}_2, \ldots, x_n : \mathsf{T}_n$$

As a shorthand, we will use $x : \mathsf{T}, \Gamma$ to mean the typing environment obtained by extending Γ with $x : \mathsf{T}$. Note that order within a typing environment does not matter if the variables are unique: $x : \mathsf{T}_1, y : \mathsf{T}_2$ is the same as $y : \mathsf{T}_2, x : \mathsf{T}_1$. However, order *does* matter if the same variable is repeated: for example, the the type of x in the context $\Gamma = x : \mathsf{T}_1, x : \mathsf{T}_2$ is T_1. That is, the type of a variable is looked up in a typing environment from left to right.

Definition 5.2 (Typing Relation). The typing relation

$$\Gamma \vdash e : \mathsf{T}$$

asserts that e has type T under the typing environment Γ.

For example,

$$\vdash 1 : \mathsf{Int}$$

asserts that 1 has type Int under the empty typing environment.

5.3. **Explanation of the typing rules.** As with evaluation rules, we have a set of *typing rules* that allow us to construct a derivation tree (i.e. a proof) that some expression has some type.

5.3.1. *Integers and Booleans.* Integer constants have type Int:

$$\frac{}{\Gamma \vdash i : \mathsf{Int}} \; \text{T-Int}$$

and boolean constants have type Bool:

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{Bool}} \; \text{T-True} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{Bool}} \; \text{T-False}$$

An integer arithmetic operation has type Int if both its operands have type Int:

$$\frac{\Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : \mathsf{Int} \qquad \square \in \{+, -, *\}}{\Gamma \vdash e_1 \square e_2 : \mathsf{Int}} \; \text{T-Arith}$$

An integer comparison operation has type Bool if both its operands have type Int:

$$\frac{\Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : \mathsf{Int} \qquad \square \in \{<, >, =\}}{\Gamma \vdash e_1 \square e_2 : \mathsf{Bool}} \; \text{T-Rel}$$

If either operand of an arithmetic/comparison operation are not integers, then the operation will clearly get stuck during evaluation. Thus, the type system preemptively checks for this and prevents such operations from being typed. This is reflected by the fact that for those kinds of expressions, the typing will get stuck, which means that no typing derivation tree can be constructed for the expression.

For example, we can derive the following typing relation:

$$\vdash 1 > 2 * (4 - 6) : \mathsf{Bool}$$

with the following derivation tree:

$$\frac{\dfrac{}{\vdash 1 : \mathsf{Int}} \text{T-Int} \quad \dfrac{\dfrac{}{\vdash 2 : \mathsf{Int}} \text{T-Int} \quad \dfrac{\dfrac{}{\vdash 4 : \mathsf{Int}} \text{T-Int} \quad \dfrac{}{\vdash 6 : \mathsf{Int}} \text{T-Int}}{\vdash 4 - 6 : \mathsf{Int}} \text{T-Arith}}{\vdash 2 * (4 - 6) : \mathsf{Int}} \text{T-Arith}}{\vdash 1 > 2 * (4 - 6) : \mathsf{Bool}} \text{T-Rel}$$

In contrast, the relation $\vdash 1 > \mathsf{true} : \mathsf{Bool}$ is not derivable:

$$\frac{\dfrac{}{\vdash 1 : \mathsf{Int}} \text{T-Int} \quad \dfrac{}{\vdash \mathsf{true} : \mathsf{Int}} \text{???}}{\vdash 1 > \mathsf{true} : \mathsf{Bool}} \text{T-Rel}$$

because we will get stuck when trying to type true as an integer.

Finally, for if-then-else expressions, the type of the condition must be boolean, and the type of the overall expression is the same as the type of the then-branch and the else-branch:

$$\frac{\Gamma \vdash e_1 : \mathsf{Bool} \qquad \Gamma \vdash e_2 : \mathsf{T}_1 \qquad \Gamma \vdash e_3 : \mathsf{T}_2 \qquad \mathsf{T}_1 = \mathsf{T}_2}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \mathsf{T}_1} \; \text{T-If}$$

Note that we use the relation $\mathsf{T}_1 = \mathsf{T}_2$ to check if two types are equal. We elide the rules for this relation, but they are straightforward. You are encouraged to formalize them as a set of inference rules as an exercise.

5.3.2. *Lists.* The empty list has type $\mathsf{List[T]}$ if it is annotated with the type of its elements:

$$\frac{}{\Gamma \vdash \mathsf{Nil[T]} : \mathsf{List[T]}} \text{ T-N\textsc{il}}$$

Note that the type annotation associated with Nil indicates the type of the elements of the list, not the type of the list itself. For example, $\mathsf{Nil[Int]}$ has type $\mathsf{List[Int]}$, and $\mathsf{Nil[List[Int]]}$ has type $\mathsf{List[List[Int]]}$!

The cons is typed as follows:

$$\frac{\Gamma \vdash e_1 : \mathsf{T} \qquad \Gamma \vdash e_2 : \mathsf{List[T]}}{\Gamma \vdash e_1 :: e_2 : \mathsf{List[T]}} \text{ T-C\textsc{ons}}$$

That is, the cons cell $e_1 :: e_2$ has type $\mathsf{List[T]}$ if the head e_1 has type T, and that the tail e_2 is another list of type $\mathsf{List[T]}$.

5.3.3. *Typing of Binding Constructs.* Recall that we use a typing environment Γ to keep track of the types of variables. This environment comes into play when we have variable declaration (as in let-expressions, lambda abstractions, pattern matches, and the fixed-point operator), or when we have variable references.

When we have a variable reference, we simply look up the type of the variable in the typing environment:

$$\frac{\Gamma(x) = \mathsf{T}}{\Gamma \vdash x : \mathsf{T}} \text{ T-V\textsc{ar}}$$

Constructs that declare variables extend the typing environment with the type of the variable being declared. For example, according to the T-L\textsc{et} rule:

$$\frac{\Gamma \vdash e_1 : \mathsf{T}_1 \qquad x : \mathsf{T}_1, \Gamma \vdash e_2 : \mathsf{T}_2}{\Gamma \vdash \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 : \mathsf{T}_2} \text{ T-L\textsc{et}}$$

a let-expression has type T_2 if the type of the expression bound to x is T_1, and that that the body of the let-expression has type T_2 under the extended typing environment $x : \mathsf{T}_1, \Gamma$.

Similarly, in the T-L\textsc{ambda} rule:

$$\frac{x : \mathsf{T}_1, \Gamma \vdash e : \mathsf{T}_2}{\Gamma \vdash (\mathsf{lambda}\, x : \mathsf{T}_1.\ e) : \mathsf{T}_1 \to \mathsf{T}_2} \text{ T-L\textsc{ambda}}$$

a lambda abstraction has type $\mathsf{T}_1 \to \mathsf{T}_2$ if the body of the lambda abstraction returns type T_2 under the extended typing environment $x : \mathsf{T}_1, \Gamma$, where T_1 is an annotation supplied by the programmer.

The rule for typing a list pattern match might look a bit daunting, but it is actually what you would expect:

$$\frac{\Gamma \vdash e_1 : \mathsf{List[T}_1] \qquad \Gamma \vdash e_2 : \mathsf{T}_2 \qquad x : \mathsf{T}_1, y : \mathsf{List[T}_1], \Gamma \vdash e_3 : \mathsf{T}_3 \qquad \mathsf{T}_2 = \mathsf{T}_3}{\Gamma \vdash \mathsf{match}\, e_1 \,\mathsf{with}\, \mathsf{Nil} \to e_2 \mid x :: y \to e_3 \,\mathsf{end} : \mathsf{T}_2} \text{ T-M\textsc{atch}}$$

To unpack this rule a bit, it says that the overall pattern match has type T_2 if

- the thing being matched is some list of type $\mathsf{List[T}_1]$
- the Nil branch e_2 and the cons branch e_3 have the same type

- the cons branch e_3 needs to be typed under the extended typing environment $x : \mathsf{T}_1, y :$ $\mathsf{List}[\mathsf{T}_1], \Gamma$, since x will be mapped to the head of the list, which has type T_1, and y will be mapped to the tail of the list, which has type $\mathsf{List}[\mathsf{T}_1]$.

In a fixed-point operator:

$$\frac{f : \mathsf{T}, \Gamma \vdash e : \mathsf{T}}{\Gamma \vdash (\mathsf{fix}\ f : \mathsf{T}\ \mathsf{is}\ e) : \mathsf{T}}\ \text{T-Fix}$$

the fixed-point operator has type T, if the body of the fixed-point operator has type T under the extended typing environment $f : \mathsf{T}, \Gamma$. In case the fixed-point operator defines a recursive function, then T will be some function type $\mathsf{T}_1 \to \mathsf{T}_2$, although in the general case there is no restriction on whether T should be a function or not.

5.3.4. *Miscellaneous.* The rule for function application $(e_1 e_2)$ is as follows:

$$\frac{\Gamma \vdash e_1 : \mathsf{T}_1 \to \mathsf{T}_2 \qquad \Gamma \vdash e_2 : \mathsf{T}_3 \qquad \mathsf{T}_1 = \mathsf{T}_3}{\Gamma \vdash (e_1\ e_2) : \mathsf{T}_2}\ \text{T-App}$$

That is, the function application will have result type T_2 if the function e_1 has type $\mathsf{T}_1 \to \mathsf{T}_2$, i.e., it takes an argument of type T_1 and returns a result of type T_2, and if the actual argument e_2 has type T_1.

Finally, we have the rule for type annotations:

$$\frac{\Gamma \vdash e : \mathsf{T}_2 \qquad \mathsf{T}_1 = \mathsf{T}_2}{\Gamma \vdash (e @ \mathsf{T}_1) : \mathsf{T}_1}\ \text{T-Annot}$$

That is, the type annotation expression $(e @ \mathsf{T}_1)$ has type T_1 if we can show expression e indeed has the same type.

5.4. **Summary of Typing Rules.** The typing rules for λ^+ are summarized in Figure 6.

$$\frac{\Gamma \vdash e : \mathsf{T}_2 \qquad \mathsf{T}_1 = \mathsf{T}_2}{\Gamma \vdash (e \,@\, \mathsf{T}_1) : \mathsf{T}_1} \text{ T-Annot}$$

$$\frac{}{\Gamma \vdash i : \mathsf{Int}} \text{ T-Int} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : \mathsf{Int} \qquad \square \in \{+, -, *\}}{\Gamma \vdash e_1 \,\square\, e_2 : \mathsf{Int}} \text{ T-Arith}$$

$$\frac{\Gamma(x) = \mathsf{T}}{\Gamma \vdash x : \mathsf{T}} \text{ T-Var} \qquad \frac{\Gamma \vdash e_1 : \mathsf{T}_1 \qquad x : \mathsf{T}_1, \Gamma \vdash e_2 : \mathsf{T}_2}{\Gamma \vdash \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 : \mathsf{T}_2} \text{ T-Let}$$

$$\frac{x : \mathsf{T}_1, \Gamma \vdash e : \mathsf{T}_2}{\Gamma \vdash (\mathsf{lambda}\, x : \mathsf{T}_1.\ e) : \mathsf{T}_1 \to \mathsf{T}_2} \text{ T-Lambda}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{T}_1 \to \mathsf{T}_2 \qquad \Gamma \vdash e_2 : \mathsf{T}_3 \qquad \mathsf{T}_1 = \mathsf{T}_3}{\Gamma \vdash (e_1\ e_2) : \mathsf{T}_2} \text{ T-App}$$

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{Bool}} \text{ T-True} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{Bool}} \text{ T-False}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Bool} \qquad \Gamma \vdash e_2 : \mathsf{T}_1 \qquad \Gamma \vdash e_3 : \mathsf{T}_2 \qquad \mathsf{T}_1 = \mathsf{T}_2}{\Gamma \vdash \mathsf{if}\, e_1 \,\mathsf{then}\, e_2 \,\mathsf{else}\, e_3 : \mathsf{T}_1} \text{ T-If}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : \mathsf{Int} \qquad \square \in \{<, >, =\}}{\Gamma \vdash e_1 \,\square\, e_2 : \mathsf{Bool}} \text{ T-Rel}$$

$$\frac{f : \mathsf{T}, \Gamma \vdash e : \mathsf{T}}{\Gamma \vdash (\mathsf{fix}\, f : \mathsf{T} \,\mathsf{is}\, e) : \mathsf{T}} \text{ T-Fix}$$

$$\frac{}{\Gamma \vdash \mathsf{Nil}[\mathsf{T}] : \mathsf{List}[\mathsf{T}]} \text{ T-Nil} \qquad \frac{\Gamma \vdash e_1 : \mathsf{T} \qquad \Gamma \vdash e_2 : \mathsf{List}[\mathsf{T}]}{\Gamma \vdash e_1 :: e_2 : \mathsf{List}[\mathsf{T}]} \text{ T-Cons}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{List}[\mathsf{T}_1] \qquad \Gamma \vdash e_2 : \mathsf{T}_2 \qquad x : \mathsf{T}_1, y : \mathsf{List}[\mathsf{T}_1], \Gamma \vdash e_3 : \mathsf{T}_3 \qquad \mathsf{T}_2 = \mathsf{T}_3}{\Gamma \vdash \mathsf{match}\, e_1 \,\mathsf{with}\, \mathsf{Nil} \to e_2 \mid x :: y \to e_3 \,\mathsf{end} : \mathsf{T}_2} \text{ T-Match}$$

FIGURE 6. Typing rules in the λ^+ language.