Last updated: January 20, 2022

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 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 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 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. **"Boolean" operations.** For simplicity, λ^+ does not have the typical boolean values of true and false. Instead, λ^+ 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
```

λ^+ 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 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.

2.4. **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.

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 λ^+. 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 λ^+:

```
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 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
```

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

- The empty list constant Nil
- The *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.

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

- isnil e: The expression (isnil e) evaluates to 1 if e is Nil, or 0 otherwise.
- $e_1 \mathbin{@} e_2$: The expression $e_1 \mathbin{@} e_2$ 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@ 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@ Nil) evaluates to 3@ Nil, and #(1@2@3@ Nil) evaluates to 2@3@ Nil and #(2@ Nil) evaluates to 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@ 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 @ 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 λ^+ is specified by the context-free grammar presented in Figure 1.

$$
\begin{array}{rcl}
\mathit{Program} & ::= & \mathit{Expr} \\
\mathit{Expr} & ::= & \mathit{let}\ \mathit{ID}\ =\ \mathit{Expr}\ \mathit{in}\ \mathit{Expr} \\
& | & \mathit{fun}\ \mathit{ID}\ \mathit{with}\ \mathit{Idlist}\ =\ \mathit{Expr}\ \mathit{in}\ \mathit{Expr} \\
& | & \mathit{lambda}\ \mathit{Idlist}.\ \mathit{Expr} \\
& | & \mathit{fix}\ \mathit{Expr} \\
& | & \mathit{if}\ \mathit{Expr}\ \mathit{then}\ \mathit{Expr}1\ \mathit{else}\ \mathit{Expr}2 \\
& | & \mathit{Expr}1 \oplus \mathit{Expr}2 \qquad (\oplus \in \{+, -, *, =, >, <, \&\&, ||\}) \\
& | & !\,\mathit{Expr} \\
& | & \#\,\mathit{Expr} \\
& | & \mathit{Expr}1\ \mathit{Expr}2 \\
& | & \mathsf{Nil} \\
& | & \mathit{Expr}1\,@\,\mathit{Expr}2 \\
& | & \mathsf{isnil}\ \mathit{Expr} \\
& | & \mathsf{INT\_CONST} \\
& | & \mathit{ID}
\end{array}
$$

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

!, #
@
*
+, −
<, >, =
&&, ||

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 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 complete *abstract syntax* of λ^+ is specified by the context-free grammar presented in Figure 3.

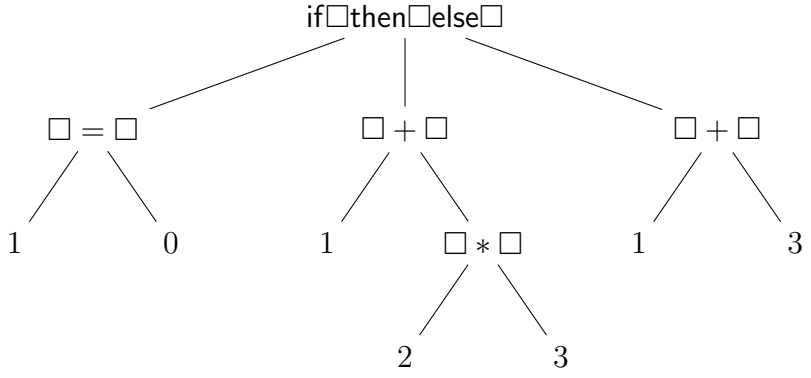
Syntactic construct			Description
$\oplus \in \mathit{ArithOp}$	::=	$+ \mid - \mid *$	
$\odot \in \mathit{Pred}$	::=	$= \mid < \mid > \mid$ && $\mid \mid\mid$	
$\diamond \in \mathit{BinOp}$	::=	$\oplus \mid \odot$	
$e \in \mathit{Expr}$	::=	i	integer
	$\mid$	x	variable
	$\mid$	$e_1 \diamond e_2$	binary op
	$\mid$	$\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$	if-then-else
	$\mid$	$\mathsf{lambda}\ x.\ e$	lambda abstraction
	$\mid$	fix	fixed-point operator
	$\mid$	$e_1\ e_2$	function application
	$\mid$	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 λ^+ 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.