1. Expressions, variables and substitution

An *expression* (or *term*) is a piece of code, or a program. Examples include `3`, `3 + 4`, `"hello"` and `x * 3`.

We'll build up our understanding of Haskell by focussing on two questions:

- How are complex expressions *built up* out of smaller expressions?
- What (other expression) does such a complex expression *evaluate to*?

We'll write $e \Longrightarrow e'$ to say that the expression e evaluates-in-one-step to the expression e'. For example:

`3 + 4` $\Longrightarrow$ `7`

`2 * (3 + 4)` $\Longrightarrow$ `2 * 7`

`"un" ++ "lock"` $\Longrightarrow$ `"unlock"`

Assignment Project Exam Help

We'll write $e \Longrightarrow^* e'$ to say that e evaluates-in-zero-or-more-steps to e'.[1] For example:

https://powcoder.com

`3 + 4` $\Longrightarrow^*$ `7`

`2 * (3 + 4)` $\Longrightarrow^*$ `2 * (3 + 4)`

`2 * (3 + 4)` $\Longrightarrow^*$ `2 * 7`

`2 * (3 + 4)` $\Longrightarrow^*$ `14`

`("un" ++ "lock") ++ "able"` $\Longrightarrow^*$ `"unlockable"`

Add WeChat powcoder

1 `let` expressions

We can use `let` expressions to give a name to the result of some computation, so that this result can be used elsewhere (perhaps multiple times).

(1) For example:

`let x = 3 in (x + 4) * x` $\Longrightarrow$ `(3 + 4) * 3`

This is also essentially what happens when we write `x = 3` in a file, and then evaluate `(x + 4) * x`.

(2) The basic idea for understanding how `let` expressions are built up is this:

- If e_1 and e_2 are both expressions, and v is a variable, then `let` v `=` e_1 `in` e_2 is an expression.

(3) To understand how `let` expressions are evaluated, the main idea is that `let` v `=` e_1 `in` e_2 is like the expression e_2, but with occurrences of the variable v replaced with e_1. Intuitively, it may be useful to picture it like this:

`let` v `=` e_1 `in` $\ldots v \ldots v \ldots$ $\Longrightarrow$ $\ldots e_1 \ldots e_1 \ldots$

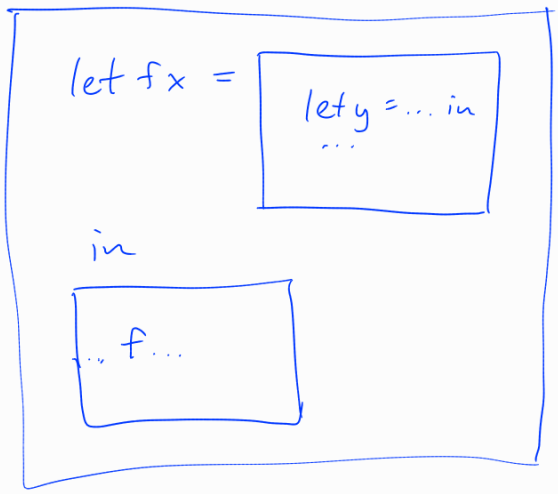
[1] This relation is defined as the reflexive, transitive closure of $\Longrightarrow$, i.e. (i) $e \Longrightarrow^* e$ and (ii) $e \Longrightarrow^* e'$ if there is an e'' such that $e \Longrightarrow e''$ and $e'' \Longrightarrow^* e'$.

let x = 3 in x + 4 ⇒ 3 + 4 ⇒ 7

① let x = 3 in 5 + 4 ⇒ 5 + 4 ⇒ 9

② let x = 3 in (let y = 2 * x in y + 4)
⇒ let y = 2 * 3 in y + 4
⇒ let y = 6 in y + 4
⇒ 6 + 4 ⇒ 10

③ let z = (let x = 3 in x + 4) in z * 2
⇒ let z = 3 + 4 in z * 2



Assignment Project Exam Help
https://powcoder.com
Add WeChat powcoder

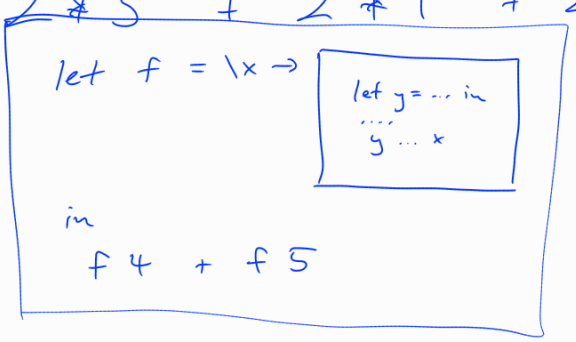
let x = 3 in x * x
⇒ 3 * 3

(λx → x * x) 3
⇒ 3 * 3

let x = 2 in
x * 3 + x * 4 + x * 5

2 * 3 + 2 * 4 + 2 * 5

let f = λx → 2 * x in
f 3 + f 4 + f 5



λy x → x - y ≡ λy → (λx → x - y)
λx y → x - y ≡ λx → (λy → x - y)

(λy → (λx → x - y)) 3 ⇒ λx → x - 3
(λx → (λy → x - y)) 3 ⇒ λy → 3 - y

(((λx → (λy → x - y)) 3) 4 ⇒ (λy → 3 - y) 4
⇒ 3 - 4

x + 3	OPEN
λx → x + 3	CLOSED
λx → x + y	OPEN
λy → (λx → x + y)	CLOSED

Slightly more formally, we can write

$$\texttt{let } v \texttt{ = } e_1 \texttt{ in } e_2 \Longrightarrow [e_1/v]e_2$$

where $[e_1/v]e_2$ means the expression like e_2 but with all free occurrences of the variable v replaced with e_1. We'll come back to a careful discussion of this later.

(4) So we can write out what's going on in the example above a bit more explicitly like this:

$$\texttt{let x = 3 in (x + 4) * x} \Longrightarrow [3/x]\texttt{(x + 4) * x} = \texttt{(3 + 4) * 3}$$

2 Lambda expressions

An expression like `x + 3` is called an *open* expression; it contains free occurrences of variables, so if you type it into `ghci` you will get a "Not in scope" error.[2] One way to build a *closed* expression out of this open expression is to use the open expression as the second part of a `let` expression, as we have seen.

Another way is to use the open expression as the body of a lambda expression.

(5) The basic idea for understanding how lambda expressions are built up is this:
- If e is an expression, and v is a variable,
  then $\texttt{\textbackslash}v \texttt{ -> } e$ is an expression.

(6) For lambda expressions to be involved in evaluation, we must also be able to combine expressions like this ("function application"):
- If e_1 and e_2 are both expressions,
  then $e_1\ e_2$ and $e_1 \texttt{ \$ } e_2$ are also expressions.

(7) Then the evaluation "recipe" for lambda expressions can be stated like this:

$$(\texttt{\textbackslash}v \texttt{ -> } e)\ e_2 \Longrightarrow [e_2/v]e$$
$$(\texttt{\textbackslash}v \texttt{ -> } e) \texttt{ \$ } e_2 \Longrightarrow [e_2/v]e$$

(8) For example:

$$\texttt{(\textbackslash x -> (x + 4) * x) 3} \Longrightarrow [3/x]\texttt{(x + 4) * x} = \texttt{(3 + 4) * 3}$$
$$\texttt{(\textbackslash x -> (x + 4) * x) \$ 3} \Longrightarrow [3/x]\texttt{(x + 4) * x} = \texttt{(3 + 4) * 3}$$

Note that an expression like `\x -> 3 + x` can "stand on its own" — it is *closed* — in the way that `let x = 5 in 3 + x` can, but `3 + x` cannot. In `\x -> 3 + x`, the variable `x` is *bound*, even though no value has been provided for it.

3 `case` expressions

3.1 Simple versions, without variables

We've seen expressions of type `Int`, such as `3` and `4 * 5`, and expressions of type `String`, such as `"hello"`. We can define a new type of our own, called `Shape`, like this:

```
data Shape = Rock | Paper | Scissors deriving Show
```

(For now, you can ignore the `deriving Show` part, and treat it as meaningless boilerplate.)

(9) This definition of the `Shape` type has the consequence that `Rock`, `Paper` and `Scissors` are all expressions, and also:

[2] Unless you've loaded a file that provides a definition for `x`.

Assignment Project Exam Help
https://powcoder.com
Add WeChat powcoder

- If e, e_1, e_2 and e_3 are expressions,[3]
  then `case` e `of {Rock ->` e_1`; Paper ->` e_2`; Scissors ->` e_3`}` is an expression.

(10) Here are the evaluation rules for these `case` expressions:

$$\texttt{case Rock of \{Rock ->}\ e_1\texttt{; Paper ->}\ e_2\texttt{; Scissors ->}\ e_3\texttt{\}} \Longrightarrow e_1$$
$$\texttt{case Paper of \{Rock ->}\ e_1\texttt{; Paper ->}\ e_2\texttt{; Scissors ->}\ e_3\texttt{\}} \Longrightarrow e_2$$
$$\texttt{case Scissors of \{Rock ->}\ e_1\texttt{; Paper ->}\ e_2\texttt{; Scissors ->}\ e_3\texttt{\}} \Longrightarrow e_3$$

(11) A straightforward example is this:

$$\texttt{case Paper of \{Rock -> 0; Paper -> 5; Scissors -> 2\}} \Longrightarrow \texttt{5}$$

That's pretty boring on its own, but usually the `Shape`-type thing being "matched" will only arise from other evaluation steps. To give you a sense of what more meaningful examples will look like, notice that:

$$\texttt{let myShape = Paper in (case myShape of \{Rock -> 0; Paper -> 5; Scissors -> 2\})} \Longrightarrow \texttt{case Paper of \{Rock -> 0; Paper -> 5; Scissors -> 2\}} \Longrightarrow \texttt{5}$$

3.2 More interesting versions, with variables

The `case` expressions for more interesting types (what we might call "compound types") involve a third instance of variable substitution, in addition to `let` expressions and lambda expressions.

We can define a new type `Result` like this:

```
data Result = Draw | Win Shape deriving Show
```

Whereas `Shape` is a type with three "options", `Result` is a type with two "options". But furthermore, one of those options, namely `Win`, comes with some extra information, namely a `Shape`.

(12) This definition of the `Result` type has the consequence that `Draw` is an expression, and also:

- If e is an expression (of type `Shape`),
  then `Win` e is an expression (of type `Result`).
- If e, e_1, and e_2 are expressions,[4] and v is a variable,
  then `case` e `of {Draw ->` e_1`; Win` v `->` e_2`}` is an expression.

(13) Here are the evaluation rules for these `case` expressions:

$$\texttt{case Draw of \{Draw ->}\ e_1\texttt{; Win}\ v\ \texttt{->}\ e_2\texttt{\}} \Longrightarrow e_1$$
$$\texttt{case (Win}\ e\texttt{) of \{Draw ->}\ e_1\texttt{; Win}\ v\ \texttt{->}\ e_2\texttt{\}} \Longrightarrow [e/v]e_2$$

(14) If we imagine for the moment that we have an appropriate `toString` function, then an example illustrating variable substitution would be:

```
case (Win Rock) of {Draw -> "No comment"; Win x -> "Congratulations " ++ toString x}
```
$$\Longrightarrow [\texttt{Rock}/\texttt{x}]\texttt{"Congratulations " ++ toString x} = \texttt{"Congratulations " ++ toString Rock}$$
$$\Longrightarrow^* \texttt{"Congratulations " ++ "Rock"}$$
$$\Longrightarrow \texttt{"Congratulations Rock"}$$

[3] In addition, (i) e must be of type `Shape`, and (ii) e_1, e_2 and e_3 must all be of the same type.
[4] In addition, (i) e must be of type `Result`, and (ii) e_1 and e_2 must both be of the same type.

Assignment Project Exam Help
https://powcoder.com
Add WeChat powcoder

Int : 0, 1, 2, -1, -2, ...
String : "hello", "world"
Shape : Rock, Paper, Scissors
Result : Draw, Win Rock, Win Paper, Win Scissors

case of { Rock → 0; Paper → 5; Scissors → 2 }

case f x of { R → _; P → _; S → _ }
case x of { ... }

data Result = Draw | Win Shape deriving Show

\x → ...
let x = ... in ...
Win x → ...

4 Free and bound variables, and consequences for substitution

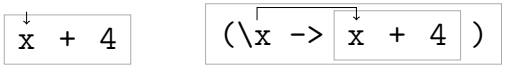
Notice that we would expect, intuitively, that `3 + 4` and `(\x -> x + 4) 3` should "behave alike" in all contexts. This means, in particular, that we would expect these two expressions to behave alike:

(15) a. `let x = 5 in x * (3 + 4)`
b. `let x = 5 in x * ((\x -> x + 4) 3)`

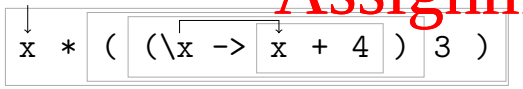
But this means that we do not want (15b) to evaluate to `5 * ((\x -> 5 + 4) 3)`! Intuitively, the occurrence of `x` that is the left operand of `+` in (15b) is none of the `let` expression's business; instead, it is being looked after by the lambda expression. The occurrence of `x` that is the left operand of `*`, in contrast, *is* the `let` expression's business.

If we "zoom in" a bit on the structure of (15b), we can illustrate these important relationships like this:

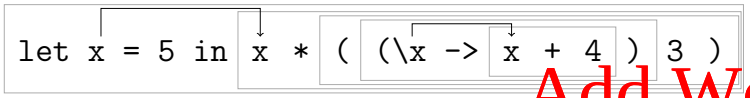
(16) The occurrence of `x` that is the left operand of `+` is *free* in the expression `x + 4`, but is *bound* in the expression `(\x -> x + 4)`.



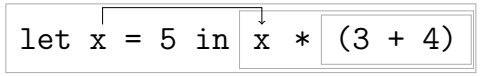
(17) In `x * ((\x -> x + 4) 3)`, the occurrence of `x` that is the left operand of `+` is *bound* (by the `\x`), and the occurrence of `x` that is the left operand of `*` is *free*.



(18) In the full expression (15b), the occurrence of `x` that is the left operand of `+` is (still) bound (by the `\x`), and the occurrence of `x` that is the left operand of `*` is *bound* (by the `let x`).



(19) So `3 + 4` and `(\x -> x + 4) 3` behave alike in the two expressions in (15) because they are both *closed expressions*; the surrounding `let` does not "get inside" either of them.



The general rules then, for the expressions we've seen so far, are:

(20) a. All free occurrences of v in e are bound by the `let` in `let` v `=` e' `in` e.[5]
b. All free occurrences of v in e are bound by the lambda in `\`v `->` e.
c. All free occurrences of v in e are bound by the `case` in `case` e' `of {...; Win` v `->` e`; ...}`.

So to ensure that things stay in sync with our intuitive expectations about what will "behave alike" as illustrated above, we must take $[e/v]e'$ to mean the result of substituting e for only the *free* occurrences of v in e'.[6]

$$[5/x]\texttt{x * (3 + 4)} = \texttt{5 * (3 + 4)}$$
$$[5/x]\texttt{x * ((\x -> x + 4) 3)} = \texttt{5 * ((\x -> x + 4) 3)}$$
$$[5/x]\texttt{x * ((\x -> x + 4) 3)} \neq \texttt{5 * ((\x -> 5 + 4) 3)}$$

[5] Actually, in Haskell, free occurrences of v are also bound in e'. This is what allows recursive definitions. But we'll put this aside until next week.

[6] There is also one more catch, which more rarely comes up in practice: $[e/v]e'$ is undefined if there are binders in e' for some variable that occurs free in e. This avoids the problem known as "variable capture".

Assignment Project Exam Help
https://powcoder.com
Add WeChat powcoder

$((\lambda y \to (\lambda x \to x + y))\ 3)\ 4$

$\Rightarrow ([3/y](\lambda x \to x + y))\ 4$

$= (\lambda x \to x + 3)\ 4$

$\Rightarrow 4 + 3$

$(\lambda v \to e_1)\ e_2$

$\Rightarrow [e_2/v]\, e_1$

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder