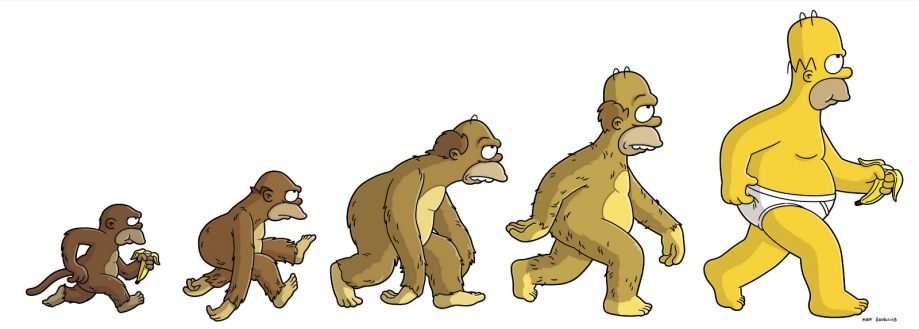
COMP302: Programming Languages and Paradigms

Week 11: Introduction to Programming Language:

Growing the toy language to include variables, let-expressions, etc.

Prof. Brigitte Pientka <bpientka@cs.mcgill.ca>

School of Computer Science, McGill University



Three key questions

- What are the syntactically legal expressions? Grammar
  What expressions does the parser accept?
- What are well-typed expressions? Static semantics
  What expressions does the type-checker accept?

  $e : T$ "Expression e has type T"

- How is an expression executed? Dynamic semantics

  $e \Downarrow v$ "Expression e evaluates to value v".

Let's extend our language with variables, let-expressions, and functions!

Adding variables and let-expressions

bound variable names can be renamed.

Backus-Naur Form (BNF):

$$
\begin{array}{lll}
\text{Operations op} & ::= & + \mid - \mid * \mid < \mid = \\
\text{Expressions } e & ::= & n \mid e_1 \text{ op } e_2 \mid \texttt{true} \mid \texttt{false} \mid \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \\
 & & \mid x \mid \texttt{let } x = e_1 \texttt{ in } e_2 \texttt{ end}
\end{array}
$$

exp exp

Which of the following well-formed expressions are well-formed?

- `let` $x = x$ `in` $x + 1$ `end` ✓ (free)
- `let` $x = y$ `in` $x + 1$ `end` ✓
- `let` $x = 3$ ✗
- `let` $x =$ (`let` $y = x$ `in` $y + x$ `end`) `in` $x + 1$ `end` ✓ (free, free)
- `let` $x = x + 1$ `in` $x + 1$ `end` ✓
- `let` $y = x + 1$ `in` $y + x$ ✗

Let's understand better variables ... and their scope!

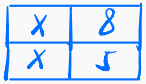
Variable names should not matter

All the same to me ...

$$\texttt{let } x = 5 \texttt{ in } (\texttt{let } y = x + 3 \texttt{ in } y + y \texttt{ end}) \texttt{ end}$$

or

overshadow



$$\texttt{let } x = 5 \texttt{ in } (\texttt{let } x = x + 3 \texttt{ in } x + x \texttt{ end}) \texttt{ end}$$

or

$$\texttt{let } v = 5 \texttt{ in } (\texttt{let } w = v + 3 \texttt{ in } w + w \texttt{ end}) \texttt{ end}$$

When is a variable free? When is it bound?

x is free bound by x = 5

x is free bound by x = x+3

$$\texttt{let } x = 5 \texttt{ in } (\texttt{let } x = \overbrace{x + 3}^{FV(x+3)=\{x\}} \texttt{ in } \overbrace{x + x}^{FV(x+x)=\{x\}} \texttt{ end}) \texttt{ end}$$

$\Rightarrow FV(\texttt{let } x = x+3 \texttt{ in } x+x) = \{x\}$

$FV(x+3) \cup (FV(x+x) \setminus \{x\})$

$\{x\} \qquad \phi$

$\Rightarrow FV(\texttt{let } x = 5 \texttt{ in } (FV(\texttt{let } x = x+3 \texttt{ in } x+x) = \{x\}))$

$= FV(5) \cup (FV(\texttt{let } x = x+3 \texttt{ in } x+x) \setminus \{x\})$

$= \phi$

Computing Free Variables

$\mathrm{FV}(e)$ returns the set of the free variable names occurring in the expression e.

$$
\begin{array}{lcl}
\mathrm{FV}(x) & = & \{x\} \\
\mathrm{FV}(e_1 \text{ op } e_2) & = & \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \\
\mathrm{FV}(\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2) & = & \mathrm{FV}(e) \cup \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \\
\mathrm{FV}(\texttt{let } x = e_1 \texttt{ in } e_2 \texttt{ end}) & = & \mathrm{FV}(e_1) \cup (\mathrm{FV}(e_2) \setminus \{x\})
\end{array}
$$

Substitution

Let's define it as:

$[e'/x]e$ Replace all free occurrences of the variable x in the expression e with expression e'.

You can read it as a function:

Input: expression e', variable x, and expression e

Output: an expression where all free occurrences of the variable x in expression e have been replaced by e'

Substitution is defined by considering different cases for e.

Example: $[5/x](2 + x) = 2 + 5$

Substitution Defined

$[e'/x]e$ Replace all free occurrences of the variable x in the expression e with expression e'.

$$
\begin{array}{lcl}
[e'/x](x) & = & e' \\
[e'/x](e_1 \text{ op } e_2) & = & [e'/x]e_1 \text{ op } [e'/x]e_2 \\
[e'/x](\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2) & = & \texttt{if } [e'/x]e \texttt{ then } [e'/x]e_1 \texttt{ else } [e'/x]e_2 \\
[e'/x](\texttt{let } y = e_1 \texttt{ in } e_2 \texttt{ end}) & = &
\end{array}
$$

How about substitution for let-expressions?

$$[e'/x](\texttt{let}\ \ y = e_1\ \texttt{in}\ e_2\ \texttt{end}) \quad =$$

Example:

$$[5/x](\texttt{let}\ \ y = x + 3\ \texttt{in}\ y + x\ \texttt{end}) =$$
$$\texttt{let}\ \ y = [5/x](x + 3)\ \texttt{in}\ [5/x](y + x)\ \texttt{end} =$$
$$\texttt{let}\ \ y = 5 + 3\ \texttt{in}\ y + 5\ \texttt{end}$$

This seems to suggest:

$$[e'/x](\texttt{let}\ \ y = e_1\ \texttt{in}\ e_2\ \texttt{end}) \quad = \quad (\texttt{let}\ \ y = [e'/x]e_1\ \texttt{in}\ [e'/x]e_2\ \texttt{end})$$

Let's look at some more examples ...

Attempt :

$$[e'/x](\texttt{let}\ \ y = e_1\ \texttt{in}\ e_2\ \texttt{end}) \quad = \quad (\texttt{let}\ \ y = [e'/x]e_1\ \texttt{in}\ [e'/x]e_2\ \texttt{end})$$

Example:

free

$$\begin{aligned} & [y+1/x](\texttt{let}\ \ y = x+3\ \texttt{in}\ y+x\ \texttt{end}) \\ = \quad & \texttt{let}\ \ y = [y+1/x](x+3)\ \texttt{in}\ [y+1/x](y+x)\ \texttt{end} \\ = \quad & \texttt{let}\ \ y = (y+1)+3\ \texttt{in}\ y+(y+1)\ \texttt{end} \end{aligned}$$

Anything wrong here?

YES! y gets captured ... it used to be free, now it is bound

Substitution isn't stable under renaming of bound variable names.

Capture-Avoiding Substitution

$[e'/x]e$ Replace all free occurrences of the variable x in the expression e with expression e'.

$$
\begin{array}{lcl}
[e'/x](x) & = & e' \\
[e'/x](e_1 \text{ op } e_2) & = & [e'/x]e_1 \text{ op } [e'/x]e_2 \\
[e'/x](\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2) & = & \texttt{if } [e'/x]e \texttt{ then } [e'/x]e_1 \texttt{ else } [e'/x]e_2 \\
[e'/x](\texttt{let } y = e_1 \texttt{ in } e_2 \texttt{ end}) & = & \texttt{let } y = [e'/x]e_1 \texttt{ in } [e'/x]e_2 \texttt{ end} \\
 & & \text{provided } x \neq y \text{ and } y \notin \text{FV}(e')
\end{array}
$$

$[y+1/x](\texttt{let } y = x+3 \texttt{ in } y+x)$

$= [y+1/x](\texttt{let } y_0 = x+3 \texttt{ in } y_0+x)$

$[y+1/x](\texttt{let } x = x+3 \texttt{ in } x+2)$

$= [y+1/x](\texttt{let } x_0 = x+3 \texttt{ in } x_0+2)$

Implementing variables and let-expressions

Backus-Naur Form (BNF):

$$\begin{array}{lll} \text{Operations op} & ::= & + \mid - \mid * \mid < \mid = \\ \text{Expressions } e & ::= & n \mid e_1 \text{ op } e_2 \mid \texttt{true} \mid \texttt{false} \mid \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \\ & & \mid x \mid \texttt{let } x = e_1 \texttt{ in } e_2 \texttt{ end} \end{array}$$

Representation in OCaml

```
1 type name = string
2 type exp =
3 | Var of name
4 | Int of int                    (* 0 | 1 | 2 | ... *)
5 | Bool of bool                  (* true | false *)
6 | If of exp * exp * exp         (* if e then e1 else e2 *)
7 | Primop of primop * exp list   (* e1 <op> e2  or  <op> e *)
8 | Let of dec * exp              (* let dec in e end *)
9 and dec = Val of name * exp     (* val x = e *)
```

Take-Away

Understand how to add variable binding to a toy language

- What expressions are syntactically legal?
- Where are variables bound? What is their scope?
- What is a free variable vs a bound variable?
- Names of bound variables should not matter; they can always be renamed.
- Defining capture-avoiding substitution

To check your understanding:

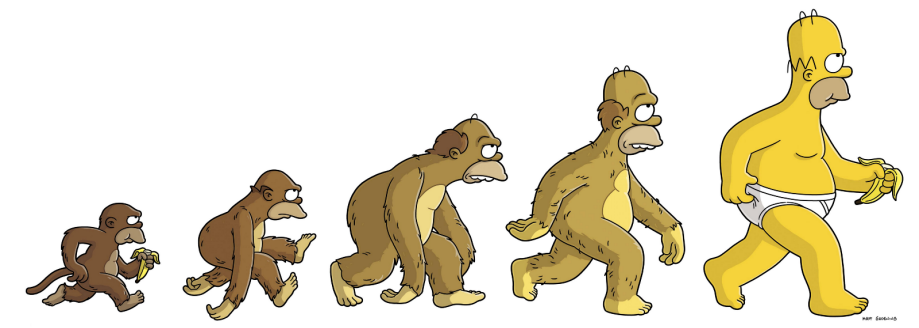
- Can you extend the substitution operation to functions, recursion, etc.?
- Can you extend the definition for free variables to also compute the free variables for functions, recursion, etc.?

COMP302: Programming Languages and Paradigms

Week 11: Building an evaluator for a toy language with let-expressions, functions, recursion, etc.

Prof. Brigitte Pientka <bpientka@cs.mcgill.ca>

School of Computer Science, McGill University



Three key questions

- What are the syntactically legal expressions? Grammar
  What expressions does the parser accept?
- What are well-typed expressions? Static semantics
  What expressions does the type-checker accept?

  $e : T$ "Expression e has type T"

- How is an expression executed? Dynamic semantics

  $e \Downarrow v$ "Expression e evaluates to value v".

Let's extend our language with variables, let-expressions, and functions!

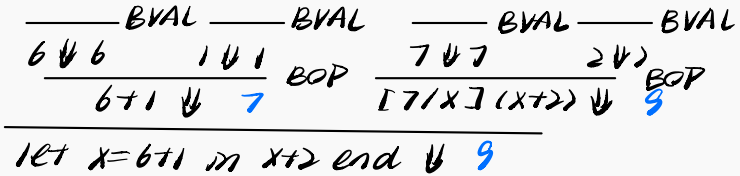
Adding variables and let-expressions

Backus-Naur Form (BNF):

$$
\begin{array}{lll}
\text{Operations op} & ::= & + \mid - \mid * \mid < \mid = \\
\text{Expressions } e & ::= & n \mid e_1 \text{ op } e_2 \mid \texttt{true} \mid \texttt{false} \mid \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \\
& & \mid x \mid \texttt{let } x = e_1 \texttt{ in } e_2 \texttt{ end}
\end{array}
$$

How to evaluate let-expressions?

Evaluation Revisited



$e \Downarrow v$

for

"Expression e evaluates to value v".

Evaluation Rules:

$$\frac{}{v \Downarrow v}\ \text{B-VAL} \qquad \frac{e \Downarrow \texttt{true} \quad e_1 \Downarrow v}{\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \Downarrow v}\ \text{B-IFT} \qquad \frac{e \Downarrow \texttt{false} \quad e_2 \Downarrow v}{\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \Downarrow v}\ \text{B-IFF}$$

$$\frac{e_1 \Downarrow v_1 \quad [v_1/x]e_2 \Downarrow v}{\texttt{let } x = e_1 \texttt{ in } e_2 \texttt{ end} \Downarrow v}\ \text{B-LET}$$

Adding Functions and Function Application

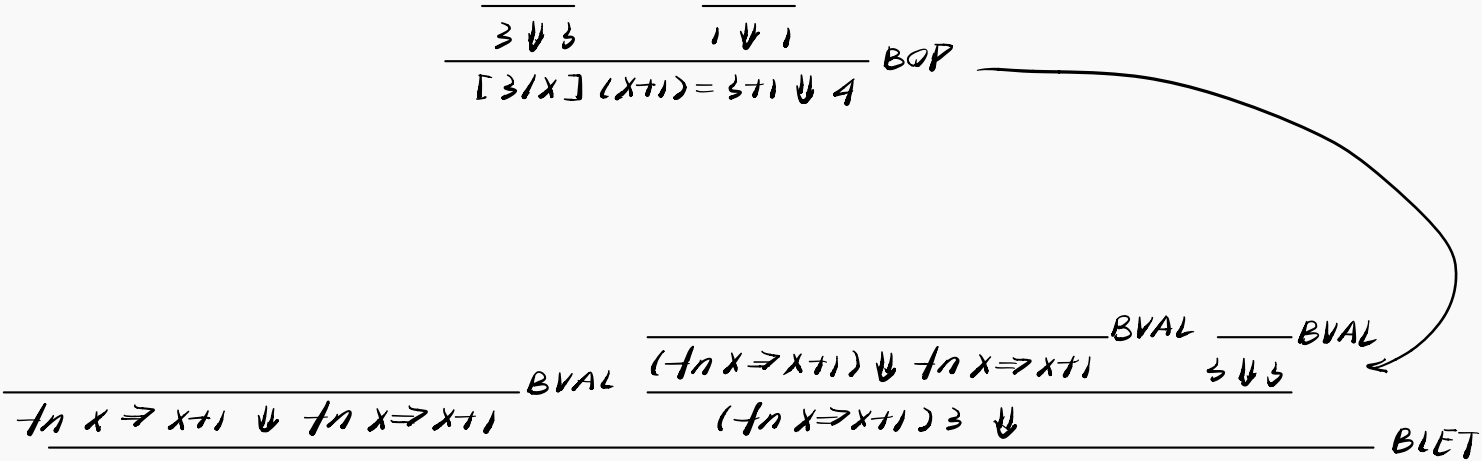
$$
\begin{array}{lll}
\text{Operations op} & ::= & + \mid - \mid * \mid < \mid = \\
\text{Expressions } e & ::= & n \mid e_1 \text{ op } e_2 \mid \texttt{true} \mid \texttt{false} \mid \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \\
 & & x \mid \texttt{fn } x \texttt{ => } e \mid e_1\ e_2 \mid \texttt{let } x = e_1 \texttt{ in } e_2 \texttt{ end} \\
\text{Values } v & ::= & n \mid \texttt{true} \mid \texttt{false} \mid \texttt{fn } x \texttt{ => } e
\end{array}
$$

$$\frac{}{v \Downarrow v}\ \text{B-VAL} \qquad \frac{e \Downarrow \texttt{true} \quad e_1 \Downarrow v}{\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \Downarrow v}\ \text{B-IFT} \qquad \frac{e \Downarrow \texttt{false} \quad e_2 \Downarrow v}{\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \Downarrow v}\ \text{B-IFF}$$

$$\frac{e_1 \Downarrow v_1 \quad [v_1/x]e_2 \Downarrow v}{\texttt{let } x = e_1 \texttt{ in } e_2 \texttt{ end} \Downarrow v}\ \text{B-LET}$$

$$\frac{e_1 \Downarrow \texttt{fn } x \texttt{ => } e \quad e_2 \Downarrow v_2 \quad [v_2/x]e \Downarrow v}{e_1\ e_2 \Downarrow v}\ \text{B-APP}$$

How expressions evaluate



`let` *inc* = `fn` x `=>` $x + 1$ `in` *inc* 3 `end` $\Downarrow$

How to add recursion?

$$
\begin{array}{lll}
\text{Operations op} & ::= & + \mid - \mid * \mid < \mid = \\
\text{Expressions } e & ::= & n \mid e_1 \text{ op } e_2 \mid \texttt{true} \mid \texttt{false} \mid \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \\
& \mid & x \mid \texttt{fn } x \texttt{ => } e \mid e_1\ e_2 \mid \texttt{let } x = e_1 \texttt{ in } e_2 \texttt{ end} \\
& \mid & \texttt{rec } f \texttt{ => } e \\
\text{Values } v & ::= & n \mid \texttt{true} \mid \texttt{false} \mid \texttt{fn } x \texttt{ => } e
\end{array}
$$

Recall that in OCaml we write

```
1 let rec sum x = if x = 0 then 1 else x + sum (x-1)
```

Using our syntax, we write :

$$\texttt{rec } sum \texttt{ => fn } x \texttt{ => if } x = 0 \texttt{ then } 1 \texttt{ else } x + sum\ (x - 1)$$

How to evaluate recursive functions?

$$\frac{[\texttt{rec}\ f\ \texttt{=>}\ e/f]e \Downarrow v}{\texttt{rec}\ f\ \texttt{=>}\ e \Downarrow v}\ \text{B-REC}$$

How does this work? Let's define some abbreviations

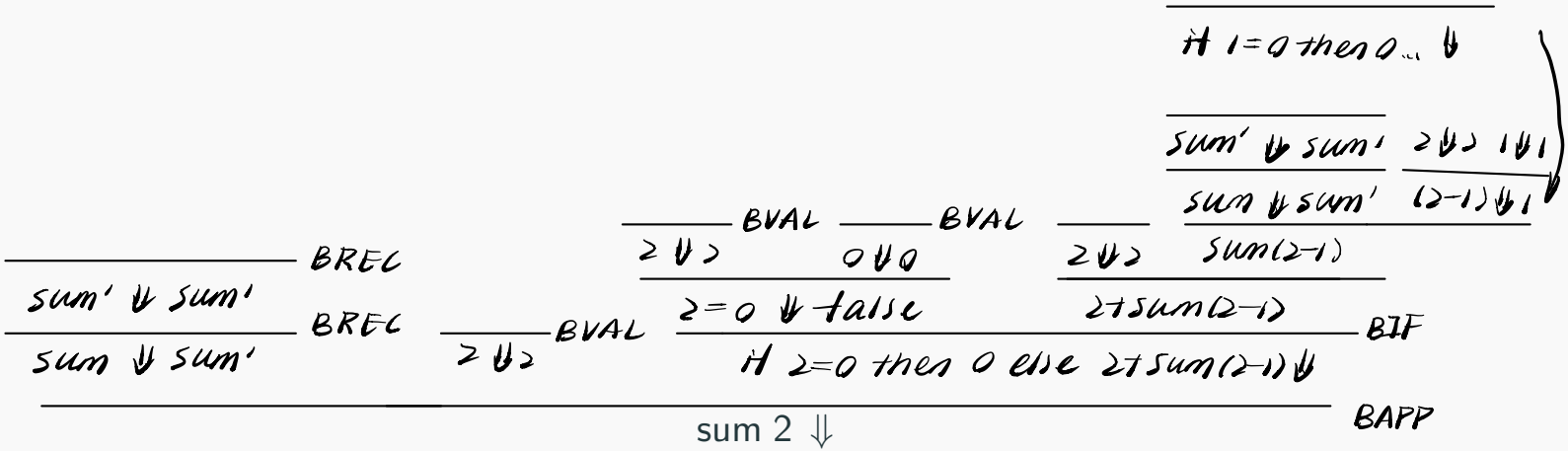
$$\text{sum} = \texttt{rec}\ f\ \texttt{=>}\ \texttt{fn}\ x\ \texttt{=>}\ \texttt{if}\ x = 0\ \texttt{then}\ 0\ \texttt{else}\ x + f(x-1)$$

We will write sum' for the first unfolding of the recursion:

$$\text{sum'} = \texttt{fn}\ x\ \texttt{=>}\ \texttt{if}\ x = 0\ \texttt{then}\ 0\ \texttt{else}\ x + \text{sum}(x-1)$$

Recursive evaluation

$$\begin{aligned} \text{sum} &= \texttt{rec}\ f\ \texttt{=>}\ \texttt{fn}\ x\ \texttt{=>}\ \texttt{if}\ x = 0\ \texttt{then}\ 0\ \texttt{else}\ x + f(x-1) \\ \text{sum'} &= \texttt{fn}\ x\ \texttt{=>}\ \texttt{if}\ x = 0\ \texttt{then}\ 0\ \texttt{else}\ x + \text{sum}(x-1) \end{aligned}$$



Take-Away

Understand

- how to read a BNF grammar of a language
  What expressions are syntactically legal?
- how to read operational semantics via axioms and inference rules
  How does an expression evaluate?

Tricky issues

- Variables and scope of variables (see functions, let-expressions, recursion, etc.)
- Names of bound variables should not matter; they can always be renamed.
- Capture-avoiding substitution
  Can you extend the substitution operation to functions, recursion, etc.? Do you understand the principle behind capture-avoiding substitution?

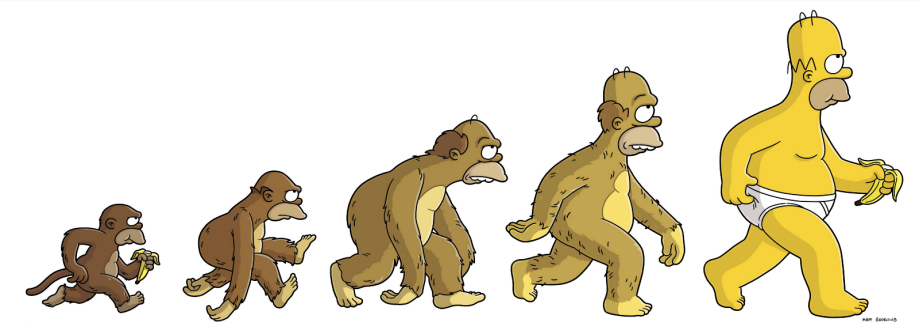
COMP302: Programming Languages and Paradigms

Week 10: Introduction to Programming Language:

Type checking for a toy language with variables and let-expressions?

Prof. Brigitte Pientka <bpientka@cs.mcgill.ca>

School of Computer Science, McGill University



How to describe well-typed expressions?

How can we statically check whether an expressions would potentially lead to a runtime error?

Static Type Checking

- Types approximate runtime behaviour
- Lightweight tool for reasoning about programs
- Detect errors statically, early in the development cycle
- Great for code maintenance
- Precise error messages
- Checkable documentation of code

Revisiting our toy language

Backus-Naur Form (BNF):

$$
\begin{array}{lll}
\text{Operations op} & ::= & + \mid - \mid * \mid < \mid = \\
\text{Expressions } e & ::= & n \mid e_1 \text{ op } e_2 \mid \texttt{true} \mid \texttt{false} \mid \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \\
 & & \mid x \mid \texttt{let } x = e_1 \texttt{ in } e_2 \texttt{ end} \\
\text{Types } T & ::= & \text{int} \mid \text{bool}
\end{array}
$$

We can write many non-sensical expressions ...

Revisiting our toy language

Backus-Naur Form (BNF):

$$
\begin{array}{lll}
\text{Operations op} & ::= & + \mid - \mid * \mid < \mid = \\
\text{Expressions } e & ::= & n \mid e_1 \text{ op } e_2 \mid \texttt{true} \mid \texttt{false} \mid \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \\
 & & \mid x \mid \texttt{let } x = e_1 \texttt{ in } e_2 \texttt{ end} \\
\text{Types } T & ::= & \text{int} \mid \text{bool}
\end{array}
$$

$\boxed{e : T}$ Expression e has type T

$$\frac{}{\texttt{true} : \text{bool}}\ \text{T-T} \qquad \frac{}{\texttt{false} : \text{bool}}\ \text{T-F} \qquad \frac{e_1 : T \quad e_2 : T}{e_1 = e_2 : \text{bool}}\ \text{T-EQ}$$

$$\frac{}{n : \text{int}}\ \text{T-NUM} \qquad \frac{e_1 : \text{int} \quad e_2 : \text{int}}{e_1 + e_2 : \text{int}}\ \text{T-PLUS} \qquad \frac{e : \text{bool} \quad e_1 : T \quad e_2 : T}{\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 : T}\ \text{T-IF}$$

How to type variables and let-expressions?

Typing in context

Let's reconsider:

$$e : T \quad \text{expression } e \text{ has type } T$$

The expresion `let` $x = e_1$ `in` e_2 `end` has type S if

1. e_1 has type T
2. Assuming x has type T, show that e_2 has type S.

Example:

The expression `let` $x = 3$ `in let` $y = 2$ `in` $x + y$ `end end` has type int.

Given assumptions $\underbrace{x{:}\text{int}, y{:}\text{int}}_{\text{TypingContext } \Gamma}$, expression $x + y$ has type int.

Typing in context

Let's reconsider:

$$e : T \quad \text{expression } e \text{ has type } T$$

The expresion `let` $x = e_1$ `in` e_2 `end` has type S if

1. e_1 has type T
2. Assuming x has type T, show that e_2 has type S.

In general:

Given assumptions $\underbrace{x_1 : T_1, \ldots, x_n : T_n}_{\text{Typing Context } \Gamma}$, expression e has type T.

How to type variables and let-expressions?

$$
\begin{array}{lll}
\text{Operations op} & ::= & + \mid - \mid * \mid < \mid = \\
\text{Expressions } e & ::= & n \mid e_1 \text{ op } e_2 \mid \texttt{true} \mid \texttt{false} \mid \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \\
 & & \mid x \mid \texttt{let } x = e_1 \texttt{ in } e_2 \texttt{ end} \\
\text{Types } T & ::= & \text{int} \mid \text{bool} \\
\text{Context } \Gamma & ::= & \cdot \mid \Gamma, x : T
\end{array}
$$

empty — new assumption

$\boxed{\Gamma \vdash e : T}$ Expression e has type T given the typing context Γ.

$$\frac{}{\Gamma \vdash \texttt{true} : \text{bool}}\ \text{T-T} \qquad \frac{}{\Gamma \vdash \texttt{false} : \text{bool}}\ \text{T-F} \qquad \frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 = e_2 : \text{bool}}\ \text{T-EQ}$$

$$\frac{}{\Gamma \vdash n : \text{int}}\ \text{T-NUM} \qquad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}}\ \text{T-PLUS}$$

Revisiting the typing rule for if-expressions

$$\frac{\Gamma \vdash e : \text{bool} \quad \Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : T}\ \text{T-IF}$$

Expression $\boxed{\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2}$ has type T given the typing context Γ, if

- Expression e has type bool given the typing context Γ
- Expression e_1 has type T given the typing context Γ
- Expression e_2 has type T given the typing context Γ

When do we collect assumptions and use them?

else, generate a new name

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T}\ \text{T-VAR} \qquad \frac{\Gamma \vdash e_1 : T_1 \quad \Gamma, x{:}T_1 \vdash e_2 : T}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end} : T}\ \text{T-LET}$$

x must be new

Read T-VAR rule as:

> Variable x has type T in a typing context Γ, if there exists a declaration $x : T$ in Γ .

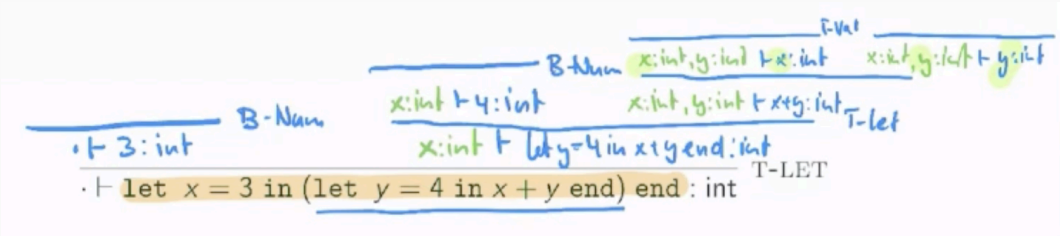
Read T-LET rule as:

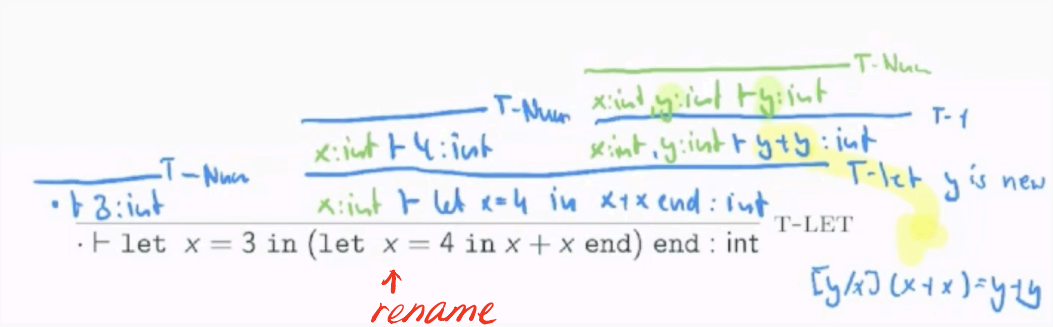
> Expression $\texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end}$ has type T in a typing context Γ, if
> - expression e_1 has type T_1 in the context Γ
> - expression e_2 has type T in the extended context $\Gamma, x : T_1$

When do we collect assumptions and use them? – Example

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T}\ \text{T-VAR} \qquad \frac{\Gamma \vdash e_1 : T_1 \quad \Gamma, x{:}T_1 \vdash e_2 : T}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end} : T}\ \text{T-LET}$$

x must be new





Two readings of typing

Type Checking $\Gamma \vdash e : T$ (input) (y/n output)

Given typing assumptions Γ, the expression e and the type T, we check that e does have type T.

Type Inference $\Gamma \vdash e : T$ (input) (output)

Given the expression e and typing context Γ, we infer its type T.

Generalizing our implementation of type inference

inference

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T}\ \text{T-VAR} \qquad \frac{\Gamma \vdash e_1 : T_1 \quad \Gamma, x{:}T_1 \vdash e_2 : T}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end} : T}\ \text{T-LET}\ \ x \text{ must be new}$$

```
1  type tp = Int | Bool
2
3  type ctx = (E.name * tp) list
4                  ctx -> exp -> tp
5  let rec infer gamma e = match e with
6  | E.Int _ -> Int
7      ....
8  | E.Var x -> List.assoc x gamma
9
10 | E.Let (E.Val (e1, x), e2) ->
```

gamma a list of $[(x_1,t_1); \ldots (x_n,t_n)]$

ordered.

let t_1 = infer gamma e_1 in infer $((x,t_1)::gamma)$ e_2

Generalizing to functions and function application (declarative reading)

$$\frac{\Gamma, x{:}T_1 \vdash e : T_2}{\Gamma \vdash \texttt{fn}\ x \texttt{=>}\ e : T_1 \to T_2}\ \text{T-FN} \qquad \frac{\Gamma \vdash e_1 : T_2 \to T \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1\ e_2 : T}\ \text{T-APP}$$

Read T-FN rule as :

> Expression `fn` x `=>` e has type $T_1 \to T_2$ in a typing context Γ, if expression e has type T_2 in the extended context $\Gamma, x : T_1$

Read T-APP rule as :

> Expression $e_1\ e_2$ has type T in a typing context Γ, if
> - expression e_1 has type $T_2 \to T$ in the context Γ
> - expression e_2 has type T_2 in the context Γ

Generalizing to functions and function application (inference)

$$\frac{\Gamma, x{:}T_1 \vdash e : T_2}{\Gamma \vdash \texttt{fn}\ x{:}T_1 \texttt{=>}\ e : T_1 \to T_2}\ \text{T-FN} \qquad \frac{\Gamma \vdash e_1 : T_2 \to T \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1\ e_2 : T}\ \text{T-APP}$$

Read T-FN rule as :

> **Infer** for expression `fn` $x{:}\ T_1$ `=>` e the type $T_1 \to T_2$ in a typing context Γ, if we can **infer** for expression e the type T_2 in the extended context $\Gamma, x : T_1$

Read T-APP rule as :

> **Infer** for expression $e_1\ e_2$ the type T in a typing context Γ, if we can
> - **Infer** for expression e_1 the type $T_2 \to T$ in the context Γ
> - **Infer** for expression e_2 has type T_2' in the context Γ
> - Check that $T_2 = T_2'$.

Take-Away

Understand

- how to read the typing rules as an algorithm (i.e. recursive program)
- Input to type inference: Assumptions Γ and the expression e
- Output to type inference: a type T (or an error)
- Type annotation on the input of a function is necessary – otherwise, we don't know what typing assumption to add!