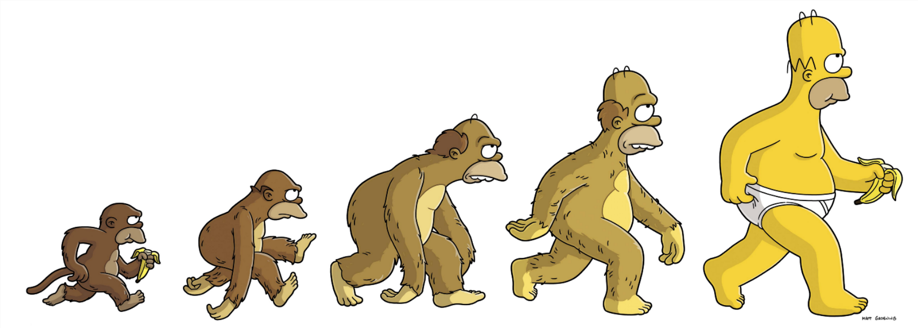
# COMP302: Programming Languages and Paradigms

## Week 10: Introduction to Programming Language:

## How to define your own language?

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

School of Computer Science, McGill University



## The four main goals of COMP 302

1. Provide a thorough introduction to fundamental concepts in programming languages
   **Higher-order functions, State-full vs state-free computation, Modelling objects and closures, Exceptions to defer control, Continuations to defer control, Polymorphism, Partial evaluation, Lazy programming, ...**
2. Show different ways to reason about programs
   **Type checking, Induction, Operational semantics, ...**
3. Introduce fundamental principles in programming language design
   **Grammars and parsing, Operational semantics and interpreters, Type checking, polymorphism, and subtyping**
4. Expose students to a different way of thinking about problems
   **It's like going to the gym; it's good for you!**

## 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?

- How is an expression executed? Dynamic semantics

# What are the syntactically legal expressions?

**Definition**

The set of expressions is defined inductively by the following clauses

- A number $n$ is an expression.
- The booleans `true` and `false` are expressions.
- If $e_1$ and $e_2$ are expressions, then $e_1$ op $e_2$ is an expression where op $= \{+, =, -, *, <\}$.
- If $e$, $e_1$ and $e_2$ are expressions, then `if` $e$ `then` $e_1$ `else` $e_2$ is an expression.

**Alternative – Backus-Naur Form (BNF)**:

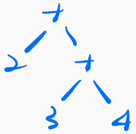
$$\begin{array}{lcl} \text{Operations op} & ::= & + \mid - \mid * \mid < \mid = \\ \text{Expressions } e & ::= & n \mid e_1 \text{ op } e_2 \mid \texttt{true} \mid \texttt{false} \mid \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \end{array}$$

# What are the syntactically legal expressions? – Examples

**Alternative – Backus-Naur Form (BNF)**:

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

| Syntactically Legal Expression (accepted by parser) | Not Syntactically Legal Expressions minus (not accepted by parser |
|---|---|
| 2+3 | −2 ? |
| 2+true | if true then 3 |
| if 2 then true else 7 | |
| 2+(3+4) | |



# How to implement expressions in OCaml?

**Backus-Naur Form (BNF):**

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

Representation in OCaml

```ocaml
type primop = Equals | LessThan | Plus | Minus | Times

type exp =
| Int of int                 (* 0 | 1 | 2 | ... *)
| Bool of bool               (* true | false *)
| If of exp * exp * exp      (* if e then e1 else e2 *)
| Primop of primop * exp list (* e1 <op> e2  or  <op> e *)
```

allow negative. negation

| Expression on paper | Encoded in OCaml |
|---|---|
| $\boxed{\texttt{if } 3 < 0 \texttt{ then } 1 \texttt{ else } 0}$ | `If (Primop (LessThan, [Int 3 ; Int 0]) , Int 1, Int 0)` |

## How to evaluate an expression?

A better question ... How to describe evaluation of expressions?

We want to say:

"Expression $e$ evaluates to a value $v$."

Hm . . . what are values?

$$\text{Values } v \quad ::= \quad n \mid \texttt{true} \mid \texttt{false}$$

# Expression $e$ evaluates to a value $v$.

**Definition**

Evaluation of the expression $e$ to a value $v$ is defined inductively by the following clauses:

- A value $v$ evaluates to itself.
- If expression $e$ evaluates to the value `true`
  and expression $e_1$ evaluates to a value $v$,
  then `if` $e$ `then` $e_1$ `else` $e_2$ evaluates to the value $v$.
- If expression $e$ evaluates to the value `false`
  and expression $e_2$ evaluates to a value $v$,
  then `if` $e$ `then` $e_1$ `else` $e_2$ evaluates to the value $v$.

Very verbose – need some better more compact notation

# Step 1: Turning an informal description into a formal one

Let's write

$$e \Downarrow v$$

for

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

**Definition**

$e \Downarrow v$ is defined inductively by the following clauses:

- $v \Downarrow v$
- If $e \Downarrow \texttt{true}$ and $e_1 \Downarrow v$, then $\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Downarrow v$.
- If $e \Downarrow \texttt{false}$ and $e_2 \Downarrow v$, then $\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Downarrow v$.

# Step 2: Turning an informal description into a formal one

$$\frac{\text{premise}_1 \quad \ldots \quad \text{premise}_n}{\text{conclusion}}\ \text{name}$$

Read as:

If $\text{premise}_1$ and $\text{premise}_2$ and $\ldots$ and $\text{premise}_n$ then conclusion.

## Step 2: Turning an informal description into a formal one

### Definition

$e \Downarrow v$ is defined inductively by the following clauses:

- $v \Downarrow v$
- If $e \Downarrow \texttt{true}$ and $e_1 \Downarrow v$, then $\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \Downarrow v$.
- If $e \Downarrow \texttt{false}$ and $e_2 \Downarrow v$, then $\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \Downarrow v$.

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

$$\frac{e \Downarrow \texttt{false} \quad e_2 \Downarrow v}{\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \Downarrow v}\ \text{B-IFFALSE}$$

Evaluation rules do not impose an order on the premises.

# Extending it to primitive operators

**Definition**

$e \Downarrow v$ is defined inductively by the following clauses:

- $v \Downarrow v$
- If $e \Downarrow \texttt{true}$ and $e_1 \Downarrow v$, then $\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Downarrow v$.
- If $e \Downarrow \texttt{false}$ and $e_2 \Downarrow v$, then $\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Downarrow v$.
- If $e_1 \Downarrow v_1$ and $e_2 \Downarrow v_2$, then $e_1 \mathsf{\ op\ } e_2 \Downarrow v$ where $v = \overline{v_1 \mathsf{\ op\ } v_2}$.

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

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1 \mathsf{\ op\ } e_2 \Downarrow \overline{v_1 \mathsf{\ op\ } v_2}}\ \text{B-OP} \qquad \frac{e \Downarrow \texttt{false} \quad e_2 \Downarrow v}{\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Downarrow v}\ \text{B-IFFALSE}$$

What is the value that the expression `if` $((4-1) < 6)$ `then` $3+2$ `else` $4$ evaluates to?

$$
\dfrac{
\dfrac{
\dfrac{\dfrac{}{4 \Downarrow 4}\,\text{B-VAL} \qquad \dfrac{}{1 \Downarrow 1}\,\text{B-VAL}}{4-1 \Downarrow 3}\,\text{B-OP}
\qquad
\dfrac{}{6 \Downarrow 6}\,\text{B-VAL}
}{4-1 < 6 \Downarrow \mathit{true}}\,\text{B-OP}
\qquad
\dfrac{\dfrac{}{3 \Downarrow 3}\,\text{B-VAL} \qquad \dfrac{}{2 \Downarrow 2}\,\text{B-VAL}}{3+2 \Downarrow 5}\,\text{B-OP}
}{\text{if } ((4-1)<6) \text{ then } 3+2 \text{ else } 4 \Downarrow 5}\,\text{BIF}\ \mathit{true}
$$

# Let's see what this means in practice!

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{}{4 \Downarrow 4}\,\textsc{B-Num}
      \quad
      \dfrac{}{1 \Downarrow 1}\,\textsc{B-Num}
    }{(4 - 1) \Downarrow 3}\,\textsc{B-Op}
    \quad
    \dfrac{}{6 \Downarrow 6}\,\textsc{B-Num}
  }{((4 - 1) < 6) \Downarrow \texttt{true}}\,\textsc{B-Op}
  \quad
  \dfrac{
    \dfrac{}{3 \Downarrow 3}\,\textsc{B-Num}
    \quad
    \dfrac{}{2 \Downarrow 2}\,\textsc{B-Num}
  }{3 + 2 \Downarrow 5}\,\textsc{B-Op}
}{\texttt{if}\ ((4 - 1) < 6)\ \texttt{then}\ 3 + 2\ \texttt{else}\ 4 \Downarrow 5}\,\textsc{B-IfTrue}
$$

- Read it operationally:

  Evaluating $\texttt{if}\ ((4-1) < 6)\ \texttt{then}\ 3+2\ \texttt{else}\ 4$ returns 5

- The derivation tree above essentially describes the execution of a recursive program which computes 5 from the input $\texttt{if}\ ((4-1) < 6)\ \texttt{then}\ 3+2\ \texttt{else}\ 4$

# Dynamic semantics as a recursive program

$\boxed{\text{Dynamic semantics: } e \Downarrow v}$

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

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1\ \mathsf{op}\ e_2 \Downarrow \overline{v_1\ \mathsf{op}\ v_2}}\ \text{B-OP} \qquad \frac{e \Downarrow \texttt{false} \quad e_2 \Downarrow v}{\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Downarrow v}\ \text{B-IFFALSE}$$

```
let rec eval e = match e with
| Int _ ->  e
| Bool _ ->  e
| If(e, e1, e2) -> match (eval e) with Bool true -> eval e1
                                        | Bool false -> eval e2
| Primop (po, args) ->                  | _ -> raise Error


```

# Advantages of a formal description ...

Establish properties and formal guarantees.

- Coverage: For all expressions $e$ there exists an evaluation rule.
- Determinacy: If $e \Downarrow v_1$ and $e \Downarrow v_2$ then $v_1 = v_2$.
- Value Soundness: If $e \Downarrow v$ then $v$ is a value.
- Termination: If $e$ is well-typed, then $e \Downarrow v$.

# 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 behavior
- 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

# Basic Types

Types classify expressions according to the kinds of values they compute.

Hm ... what are values?

$$\text{Values } v \quad ::= \quad n \mid \texttt{true} \mid \texttt{false}$$

Hence, there are only two basic types.

$$\text{Types } T \quad ::= \quad \mathsf{int} \mid \mathsf{bool}$$

# Defining Typing

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

We define when an expression $e$ is well-typed inductively.

**Definition** $e : T$ is defined inductively by the following clauses:

- $n : \mathrm{int}$
- $\texttt{true} : \mathrm{bool}$ and $\texttt{false} : \mathrm{bool}$
- If $e : \mathrm{bool}$ and $e_1 : T$ and $e_2 : T$, then $\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : T$.
- If $e_1 : \mathrm{int}$ and $e_2 : \mathrm{int}$, then $e_1 + e_2 : \mathrm{int}$.
- If $e_1 : \mathrm{int}$ and $e_2 : \mathrm{int}$, then $e_1 = e_2 : \mathrm{bool}$.

# Defining Typing

**Definition** $e : T$ is defined inductively by the following clauses:

- $n$ : int
- $\texttt{true}$ : bool and $\texttt{false}$ : bool
- If $e$ : bool and $e_1 : T$ and $e_2 : T$, then $\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : T$.
- If $e_1$ : int and $e_2$ : int, then $e_1 + e_2$ : int.
- If $e_1 : T$ and $e_2 : T$, then $e_1 = e_2$ : bool.

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

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

# Two readings of typing

Type Checking $e : T$

Given the expression $e$ and the type $T$, we check that $e$ does have type $T$.

Type Inference $e : T$

Given the expression $e$, we infer its type $T$.

# Implementing type inference

```
type tp = Int | Bool

exception TypeError of string

let fail message = raise (TypeError message)
```

Wait ... Doesn't the constructor for `Int` for types clash with the constructor for expressions?

Modules to the rescue! – They provide name space management (and actually so much more...)

# Type Checking – Implemented

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

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

```
let rec check e t = match e,t with
  | E.Int _  , Int -> true
  | E.Bool _ , Bool -> true
  | E.If (e, e1, e2) , t  ->
     check e Bool && check e1 t && check e2 t
```

# Type inference – Implemented

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

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

```
let rec infer e = match e with
  | E.Int _ -> Int
  | E.Bool _ -> Bool
  | E.If (e, e1, e2) ->
    (match infer e with
  1  | Bool -> let t1 = infer e1 in
                let t2 = infer e2 in
                if t1 = t2 then t1
                else fail ("Expected " ^ typ_to_string t1 ^
  2  | _ -> fail            " - Inferred " ^ typ_to_string t2)
```