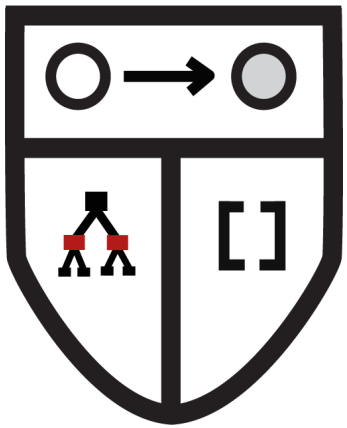


CS 3110

Expressions

Prof. Clarkson

Fall 2019

Today's music: Expression by Salt-N-Pepa

Review

Previously in 3110:

- What is a functional language?
- Why learn to program in a functional language?

Today:

- Five aspects of a language
- Expressions, values, definitions

Clicker Question

Did you bring an iClicker today?

A. Yes
B. No
C. I plead the 5th

No worries: Today is just a test run. Attendance points start for real on Thursday.

Five aspects of learning a PL

1. Syntax: How do you write language constructs?
2. Semantics: What do programs mean? (Type checking, evaluation rules)
3. Idioms: What are typical patterns for using language features to express your computation?
4. Libraries: What facilities does the language (or a third-party project) provide as "standard"? (E.g., file access, data structures)
5. Tools: What do language implementations provide to make your job easier? (E.g., top-level, debugger, GUI editor, …)

- All are essential for good programmers to understand
- Breaking a new PL down into these pieces makes it easier to learn

Our focus

We focus on semantics and idioms for OCaml

- Semantics is like a meta-tool: it will help you learn languages
- Idioms will make you a better programmer in those languages

Libraries and tools are a secondary focus: throughout your career you'll learn new ones on the job every year

Syntax is almost always boring

- A fact to learn, like "Cornell was founded in 1865"
- People obsess over subjective preferences {yawn}
- Class rule: We don't complain about syntax

Expressions

- Primary building block of OCaml programs
- Akin to statements or commands in imperative languages

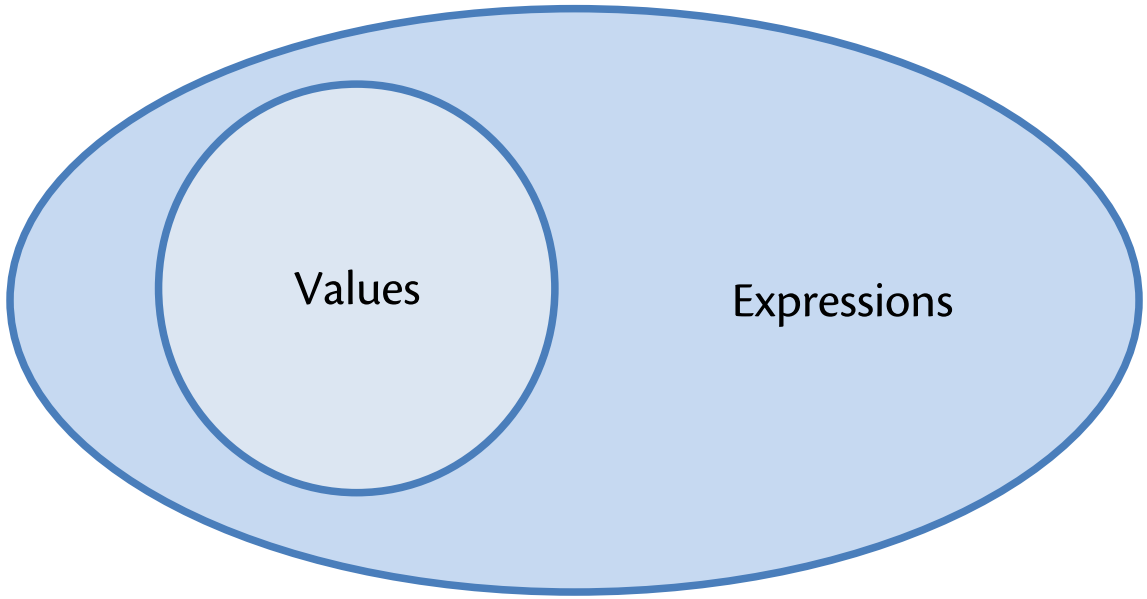
Expressions

Every kind of expression has:

- **Syntax**
- **Semantics:**
  - **Type-checking rules** (*static semantics*): produce a type, or fail with an error message
  - **Evaluation rules** (*dynamic semantics*): produce a *value,* or exception or infinite loop

Values

A **value** is an expression that does not need any further evaluation



Demo

IF EXPRESSIONS

Demo

`if` expressions

Syntax:

```
if e1
```

Write ==> to indicate evaluation
Pronounce as "evaluates to"

Evaluation:

- if **`e1`** evaluates to **`true`**, and if **`e2`** evaluates to **`v`**,
  then **`if e1 then e2 else e3`** evaluates to **`v`**
- if **`e1`** evaluates to **`fals`**
  then **`if e1 then e`**

Write colon to indicate type of expression
Pronounce colon as "has type"

Type checking:

if **`e1`** has type **`bool`** and **`e2`** has type **`t`** and **`e3`** has type **`t`**
then **`if e1 then e2 else e3`** has type **`t`**

`if` expressions

Syntax:

```
if e1 then e2 else e3
```

Evaluation:

- if `e1 ==> true` and `e2 ==> v`,
  then `if e1 then e2 else e3 ==> v`
- if `e1 ==> false` and `e3 ==> v`,
  then `if e1 then e2 else e3 ==> v`

Type checking:

if `e1 : bool` and `e2 : t` and `e3 : t`
then `if e1 then e2 else e3 : t`

`if` expressions

Syntax:

```
if e1 then e2 else e3
```

Evaluation:

- if `e1 ==> true` and `e2 ==> v`,
  then `(if e1 then e2 else e3) ==> v`
- if `e1 ==> false` and `e3 ==> v`,
  then `(if e1 then e2 else e3) ==> v`

Type checking:

if `e1 : bool` and `e2 : t` and `e3 : t`
then `(if e1 then e2 else e3) : t`

Type inference and annotation

- OCaml compiler infers types
  - Compilation fails with type error if it can't
  - Hard part of language design: guaranteeing compiler can infer types when program is correctly written

- You can manually annotate types anywhere
  - Replace `e` with `(e : t)`
  - Useful for diagnosing type errors

Demo

Clicker Question

Did you come up with a WHY from last lecture?
Your own personal motivation for being here?

A. Yes
B. No
C. I plead the 5th

No worries: Today is just a test run. Attendance points start for real on Thursday.

LET DEFINITIONS

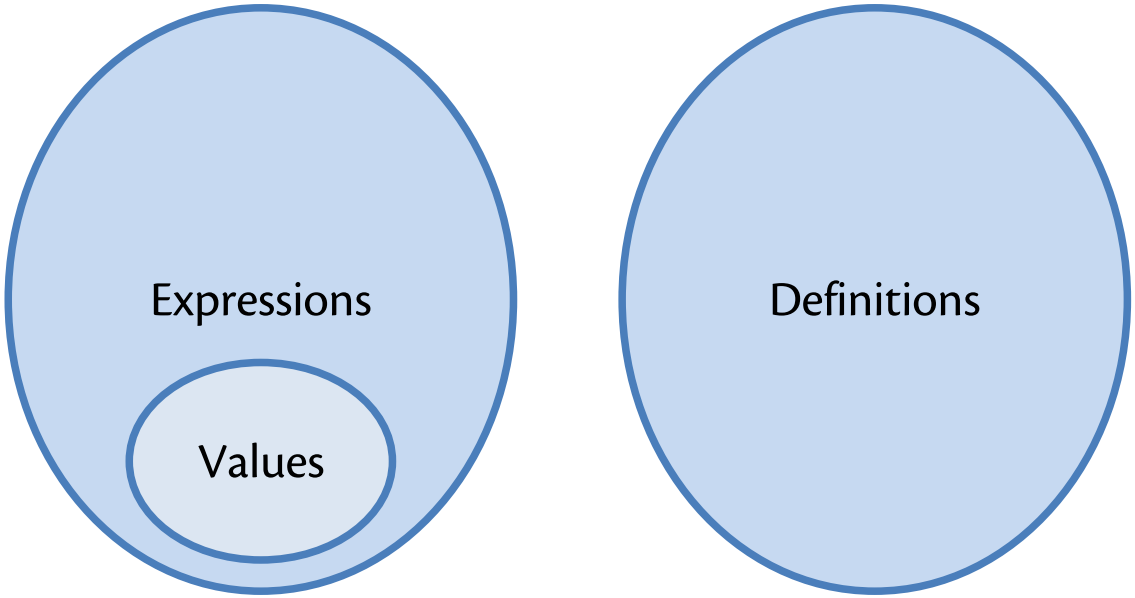
Demo

Definitions

A **definition** gives a name to a value

Definitions are not expressions, or vice-versa

But definitions syntactically contain expressions



`let` definitions

Syntax:

```
let x = e
```

where **x** is an *identifier*

Evaluation:

- Evaluate **e** to a value **v**
- Bind **v** to **x**: henceforth, **x** will evaluate to **v**
  (under the hood: there is a memory location named **x** that contains **v**)

The let definition is not an expression itself

LET EXPRESSIONS

Demo

`let` expressions

Syntax:

```
let x = e1 in e2
```

x is an *identifier*

e1 is the *binding expression*

e2 is the *body expression*

`let x = e1 in e2` is itself an expression

`let` expressions

```
let x = e1 in e2
```

Evaluation:

- Evaluate **`e1`** to a value **`v1`**
- Substitute **`v1`** for **`x`** in **`e2`**, yielding a new expression **`e2'`**
- Evaluate **`e2'`** to **`v2`**
- Result of evaluation is **`v2`**

Example

`let` expressions

```
let x = e1 in e2
```

Type-checking:

If `e1:t1` and `x:t1` and `e2:t2`

then `(let x = e1 in e2) : t2`

VARIABLE EXPRESSIONS

Variable expressions

How to evaluate just

```
x
```

at the toplevel?

`let` definitions in toplevel

`let x = e`

is implicitly, "**`in`** *rest of what you type*"

E.g., you type:

```
let a="big";;
let b="red";;
let c=a^b;;
```

Toplevel understands as

```
let a="big" in
let b="red" in
let c=a^b in…
```

Variable expressions

How to evaluate just

```
x
```

at the toplevel?

Answer: substitution from that giant nested **`let`** expression

Upcoming events

- [Wed] Consulting hours specially devoted to any remaining OCaml install issues
- [Thu] A0 released

This is expressive.

THIS IS 3110