CS 3110

Lecture 5: Pattern Matching

Prof. Clarkson
Fall 2014

Today’s music: “Puff, the Magic Dragon” by Peter, Paul & Mary
Review

Features so far: variables, operators, let expressions, if expressions, functions (higher-order, anonymous), datatypes, records, lists, options

Today:
• Pattern matching
• A mind-altering experience
• Polymorphic datatypes
Question #1

How much of PS1 have you finished?
A. None
B. About 25%
C. About 50%
D. About 75%
E. I’m done!!!
Review

Algebraic datatype we saw last time:

```haskell
type suit = Club | Diamond | Heart | Spade

type rank = Jack | Queen | King
           | Ace   | Num of int
```

Here’s a card:

```haskell
let two_clubs : suit*rank = (Club, Num 2)
```

• *Type annotation*: `two_clubs` has type `suit*rank`
• Wouldn’t it be nice to write something more meaningful (say, `card`) instead of `suit*rank`?
  – Would prevent (e.g.) having to remember whether `suit` comes first or `rank`
Type synonym

A *type synonym* is a new kind of declaration

\[
\text{type name} = t
\]

– Creates another name for a type
– The type and the name are *interchangeable in every way*
Why have type synonyms?

• For now: convenience and style
  – (makes code self-documenting!)

```haskell
type card = suit*rank
let two_clubs : card = (Club, Num 2)
```

– Write functions of type (e.g.)
  ```haskell
card -> bool
```

– Note: okay if REPL says your function has type
  ```haskell
  suit * rank -> bool
  ```

• Later: other uses related to modularity
Datatypes: Syntax and semantics

• Syntax:

\[
\text{type } t = C_1 \text{ of } t_1 \mid C_2 \text{ of } t_2 \mid \ldots \mid C_n \text{ of } t_n
\]

• Type checking:
  – If \( t_1 \ldots t_n \) are types, then \( t \) is a type
  – And \( t_1 \ldots t_n \) are allowed to mention \( t \)
Datatypes: Syntax and semantics

• Syntax:

```plaintext
type t = C1 of t1 | C2 of t2 | ... | Cn of tn
```

• Evaluation:

  – For declaration itself, none. Types aren’t evaluated

  – Building:

    • Ci v is a value

    • If e --> v then Ci e --> Ci v

  – Accessing...?
Match expressions

• **Syntax**

```plaintext
match e with p1 -> e1 | p2 -> e2 | ... | pn -> en
```

• **Evaluation:**

  – Evaluate `e` to a value `v`
  
  – If `pi` is the first pattern to match `v`, then evaluate `ei` to value `vi` and return `vi`
    
    • Note: pattern itself is not evaluated
Match expressions

• **Syntax**

```
match e with p1 -> e1 | p2 -> e2 | ... | pn -> en
```

• **Evaluation (cont’d):**
  
  — Pattern *matches* value if it “looks like” the value
    
    — Pattern \( \text{Ci}(x_1, \ldots, x_n) \) matches value \( \text{Ci}(v_1, \ldots, v_n) \)
    
    — *Wildcard* pattern \( _\) (i.e., underscore) matches any value

  — When evaluating \( e_i \), *pattern variables are bound to corresponding values “inside” \( v \). More soon...
Match expressions

• **Syntax**

```match e with p1 -> e1 | p2 -> e2 | ... | pn -> en```

• **Type-checking:**
  - If `e, p1..pn` have type `ta` and `e1..en` have type `tb` then entire match expression has type `tb`
  - *Do you see how this generalizes type-checking of if expressions? Hmm...*
Enhanced pattern syntax

• Patterns can nest arbitrarily deep
  – (Just like expressions)
  – Easy-to-read, nested patterns can replace hard-to-read, nested \texttt{match} expressions

• Examples:
  – Pattern \texttt{a::b::c::d} matches all lists with \texttt{>= 3} elements
  – Pattern \texttt{a::b::c::[]} matches all lists with \texttt{3} elements
  – Pattern \texttt{((a,b),(c,d))::e} matches all non-empty lists of pairs of pairs
Useful example: zip/unzip 3 lists

```ocaml
let rec zip3 lists =
  match lists with
  (null, null, null) -> null
  | (hd1::tl1, hd2::tl2, hd3::tl3) ->
    (hd1, hd2, hd3)::zip3(tl1, tl2, tl3)
  | _ -> raise (Failure "List length mismatch")

let rec unzip3 triples =
  match triples with
  [] -> ([], [], [])
  | (a,b,c)::tl ->
    let (11, 12, 13) = unzip3 tl
    in (a::11, b::12, c::13)
```
Match expressions

Evaluation:

Given a pattern $p$ and a value $v$, decide

- Does pattern match value?
- If so, what variable bindings are introduced?

Let’s give an evaluation rule for each kind of pattern...
Precise definition of pattern matching

• If \( p \) is a variable \( x \), the match succeeds and \( x \) is bound to \( v \)

• If \( p \) is \( _ \), the match succeeds and no bindings are introduced

• If \( p \) is a constant \( c \), the match succeeds if \( v \) is \( c \). No bindings are introduced.
Precise definition of pattern matching

• If $p$ is $C$, the match succeeds if $v$ is $C$. No bindings are introduced.

• If $p$ is $C\; p_1$, the match succeeds if $v$ is $C\; v_1$ (i.e., the same constructor) and $p_1$ matches $v_1$. The bindings are the bindings from the sub-match.
Precise definition of pattern matching

- If $p$ is $(p_1, \ldots, p_n)$ and $v$ is $(v_1, \ldots, v_n)$, the match succeeds if $p_1$ matches $v_1$, and ..., and $p_n$ matches $v_n$. The bindings are the union of all bindings from the sub-matches.
  - The pattern $(x_1, \ldots, x_n)$ matches the tuple value $(v_1, \ldots, v_n)$

- If $p$ is $\{f_1=p_1; \ldots; f_n=p_n\}$ and $v$ is $\{f_1=v_1; \ldots; f_n=v_n\}$, the match succeeds if $p_1$ matches $v_1$, and ..., and $p_n$ matches $v_n$. The bindings are the union of all bindings from the sub-matches.
  - (and fields can be reordered)
  - The pattern $\{f_1=x_1; \ldots; f_n=x_n\}$ matches the record value $\{f_1=v_1; \ldots; f_n=v_n\}$
Match expressions

• Syntax
• Type checking
• Evaluation

...mission accomplished!
Are you ready for a mind-altering experience?

![Image of a colorful graphic with the text "Blow your Mind" and a person's face in the center.](image-url)
1. If expressions are just matches

- *if* expressions exist only in the *surface syntax* of the language
- Early pass in compiler can actually replace *if* expression with *match* expression, then compile the *match* expression instead

```
if e0 then e1 else e2
```

becomes...

```
match e0 with true -> e1 | false -> e2
```

because...

```
type bool = false | true
```
Syntactic sugar

• *Syntactic*: Can describe the *semantics* entirely by another piece of syntax

• *Sugar*: They make the language sweeter 😊
  • There are fewer semantics to worry about
    – Simplify *understanding* the language
    – Simplify *implementing* the language

There are many more examples of syntactic sugar in OCaml...
Syntactic sugar

“Syntactic sugar causes cancer of the semicolon.”

First recipient of the Turing Award
for his “influence in the area of advanced programming techniques and compiler construction”

Alan J. Perlis
(1922-1990)
2. Options are just datatypes

- Options are just a predefined datatype

```ocaml
type 'a option = None | Some of 'a
```

- `None` and `Some` are constructors
- `'a` means “any type”

```ocaml
let string_of_intopt(x:int option) =
  match x with
  | None   -> ""
  | Some(i) -> string_of_int(i)
```
3. Lists are just datatypes

We could have coded up lists ourselves:

```ocaml
type my_int_list = Nil
          | Cons of int * my_int_list

let x = Cons(4,Cons(23,Cons(2008,Nil)))

let rec my_append (xs:my_int_list) (ys:my_int_list) =
    match xs with
      Nil -> ys
    | Cons(x,xs') -> Cons(x, my_append xs' ys)
```

But much better to reuse well-known, widely-understood implementation OCaml already provides
3. Lists are just datatypes

OCaml effectively does just code up lists itself:

type 'a list = [] | :: of 'a * 'a list

let rec append (xs: 'a list)(ys: 'a list) =
  match xs with
  | [] -> ys
  | x::xs' -> x :: (append xs' ys)

Just a bit of syntactic magic in compiler to use
[ ], ::, @ instead of Latin-alphabet identifiers

We’ve seen ‘a more than once... What is it really?
4. Let expressions are pattern matches

• The syntax on the LHS of = in a let expression is really a pattern

  \texttt{let } p = e

  – (Variables are just one kind of pattern)

• Implies it’s possible to do this (e.g.):

  \texttt{let } [x1;x2] = lst

  – Tests for the one variant (cons) and raises an exception if a different one is there (nil)—so it works like \texttt{hd, tl}

  – Therefore not a great idiom
5. Function arguments are patterns

A function argument can also be a pattern

– Match against the argument in a function call

```
let f p = e
```

Examples:

```
let sum_triple (x, y, z) =
  x + y + z

let sum_stooges {larry=x; moe=y; curly=z} =
  x + y + z
```
Recall this?

A function that takes one triple of type `int*int*int` and returns an `int` that is their sum:

```plaintext
let sum_triple (x, y, z) =
    x + y + z
```

A function that takes three `int` arguments and returns an `int` that is their sum:

```plaintext
let sum_triple (x, y, z) =
    x + y + z
```

See the difference? (Me neither.) 😊

*The argument is just a pattern.*
6. Functions take 1 argument

- What we think of as multi-argument functions are just functions taking one tuple argument, implemented with a tuple pattern in the function binding
  - Elegant and flexible language design

- Enables cute and useful things you can’t do in Java, e.g.,

```plaintext
let rotate_left (x, y, z) = (y, z, x)
let rotate_right t = rotate_left(rotate_left t)
```
Is your mind altered?
Is your mind altered?

“A language that doesn't affect the way you think about programming is not worth knowing.”

–Alan J. Perlis
Question #2

What’s your favorite OCaml feature so far?
A. Pattern matching
B. Lists
C. Higher-order functions
D. Datatypes
E. I miss Java :( 
Back to alpha...

Length of a list:

```
let rec len (xs: int list) =
  match xs with
  [] -> 0
  | _::xs' -> 1 + len xs'
```

```
let rec len (xs: string list) =
  match xs with
  [] -> 0
  | _::xs' -> 1 + len xs'
```

No algorithmic difference! Would be silly to have to write function for every kind of list type...
Type variables to the rescue

Use type variable to stand in place of an arbitrary type:

```ml
let rec len (xs: 'a list) =
  match xs with
  | [] -> 0
  | _::xs' -> 1 + len xs'
```

– Just like we use variables to stand in place of arbitrary values
– Creates a polymorphic function (“poly”=many, “morph”=form)
– Closely related to generics in Java
– Might look like, but is rather less related to, templates in C++
Datatypes: Syntax

• Syntax:

\[
\text{type } 'a \ t = C_1 \text{ of } t_1 \mid C_2 \text{ of } t_2 \mid \ldots \mid C_n \text{ of } t_n
\]

• Type checking:
  – If \( t_1 \ldots t_n \) are types, then \( t \) is a type
  – And \( t_1 \ldots t_n \) are allowed to mention \( t \) and \( 'a \)
Please hold still for 1 more minute

WRAP-UP FOR TODAY
Upcoming events

• PS1 is due Thursday
• Clarkson office hours this week: TR 2-4 pm

This is a mind-altering experience.

THIS IS 3110