CSE 341 AB

Section 2
(4/11)

Questions?

Agenda

1. Intros

2. Small Things
   a. Syntactic Sugar
   b. Function Tracing

3. Types!
   a. Type Synonyms
   b. Parametric Polymorphism
   c. Type Generality
   d. Equality Types

4. Variants
   e. Syntactic Sugar
   f. A Note on Patterns
   g. Tracing

Intros

Please introduce yourself to someone you haven’t talked to yet!

E.g.

- What’s your name?
- Why are you taking 341?
- What do you do for fun?
- What’s your favorite programming language?

Syntactic Sugar

Sometimes we don't change our core language to add new language constructs.

```
x andalso y
```

Syntactic Sugar

Sometimes we don’t change our core language to add new language constructs.

```
x andalso y
```

↓

```
if x then y else false
```

Syntactic Sugar

Sometimes we don't change our core language to add new language constructs.

```
x andalso y

↓

if x then y else false

↓

case x of
     true  => y
   | false => false
```

Syntactic Sugar

Sometimes we don’t change our core language to add new language constructs.

```
x orelse y

↓

if x then true else y

↓

case x of
    true  => true
  | false => y
```

Function Tracing

Function Tracing

- Function tracing is simplified (for now!).

- In Unit 3 we will look at a more complex, *but more accurate*, representation.

Function Tracing - Function Binding

When you visit a function binding, just map its name to `fn`.

```
fun foo (x: int) = x + 2;

foo 2
```

id	val
RES	
foo	fn

Function Tracing - Function Call

```
fun foo (x: int) = x + 2;

foo 2
```

id	val
RES	
foo	fn

Function Tracing - Function Call

Visit the left- and right-hand sides of the function call.

```
fun foo (x: int) = x + 2;

foo 2
```

id	val
RES	
foo	fn

<u>foo</u> 2

foo <u>2</u>

foo 2

Function Tracing - Function Call

Once we've determined the function we need to call, create a *new* environment!

Extend it with the arguments to `foo`.

```
fun foo (x: int) = x + 2;
```

```
foo 2
```

<u>foo</u> 2

foo <u>2</u>

foo 2

id	val
RES	
foo	fn

foo	
id	val
RES	
x	2

Function Tracing - Function Call

Evaluate the function body.

```
fun foo (x: int) = x + 2;

foo 2
```

```
x + 2
2 + 2
2 + 2
4
```

id	val
RES	
foo	fn

foo	
id	val
RES	
x	2

Function Tracing - Function Call

Save the result in RES.

```
fun foo (x: int) = x + 2;

foo 2
```

```
x + 2
2 + 2
2 + 2
4
```

id	val
RES	
foo	fn

foo	
id	val
RES	4
x	2

Function Tracing - Function Call

We now know the value of the original call.

Destroy the environment and pass the value back.

```
fun foo (x: int) = x + 2;
```

```
foo 2
```

```
x + 2
2 + 2
2 + 2
4
```

id	val
RES	4
foo	fn

foo	
id	val
RES	4
x	2

Function Tracing - But What About...

variables bound outside a function body?

```
val y = 2;
fun foo (x: int) = x + y;
val y = 3;
foo 2
```

Function Tracing - But What About...

nested functions?

```
fun foo (x: int) =
    let fun bar (y: int) = y * y
    in
        bar (x * x)
    end;
foo 2
```

Function Tracing - But What About...

Find out next week!

Types

Type Synonyms

```
datatype suit = Club | Diamond | Heart | Spade
datatype rank = Jack | Queen | King | Ace
      | Num of int

type card = suit * rank
```

A synonym doesn't add a new type name.
What's the type of `(Club, Jack)`? Try it out!

In a World Without Parametric Polymorphism...

```
fun append_ints (xs : int list, ys : int list) =
    case xs of
        []          => ys
      | x::xs       => x::append(xs, ys)
```

```
fun append_strings (xs : string list, ys : string list) =
    case xs of
        []          => ys
      | x::xs       => x::append(xs, ys)
```

The code is the same, but every new data type requires a new function!

(Notice that we only use the inputs' *structures*, not their *values*. This will become important in future weeks.)

What If… NOT VALID SML!!!

```
fun append ('a) (xs : 'a list, ys : 'a list) =
    case xs of
        []        => ys
      | x::xs     => x::append(xs, ys)
```

What If… NOT VALID SML!!!

```
fun append ('a) (xs : 'a list, ys : 'a list) =
    case xs of
        []        => ys
      | x::xs     => x::append(xs, ys)
```

```
append : forall 'a, 'a list * 'a list -> 'a list
```

What If… NOT VALID SML!!!

```
fun append ('a) (xs : 'a list, ys : 'a list) =
    case xs of
        []        => ys
      | x::xs     => x::append(xs, ys)
```

```
append : forall 'a, 'a list * 'a list -> 'a list
```

```
val append_ints = append(int)
val append_strings = append(string)
```

```
append_ints       : int list * int list -> int list
append_strings    : string list * string list -> string list
```

What If… NOT VALID SML!!!

```
fun append ('a) (xs : 'a list, ys : 'a list) =
    case xs of
        []        => ys
      | x::xs     => x::append(xs, ys)
```

```
append : forall 'a, 'a list * 'a list -> 'a list
```

```
val append_ints = append(int)
val append_strings = append(string)
```

Types in our expressions?!?! Take me back!

Luckily, SML has a restriction that means we don't have to write this way:
`forall` can only appear at the beginning of a type.

But it's useful to think about what's going on under the hood.

What If… VALID SML!!!

```
fun append      (xs : 'a list, ys : 'a list) =
    case xs of
        []       => ys
      | x::xs    => x::append(xs, ys)
```

```
append :             'a list * 'a list -> 'a list
```

What If… VALID SML!!!

```
fun append      (xs : 'a list, ys : 'a list) =
    case xs of
        []       => ys
      | x::xs    => x::append(xs, ys)
```

```
append :             'a list * 'a list -> 'a list
```

You can use `append` with any type of list *as long as both lists have the same type!*

SML will do the right thing under the hood and insert type arguments for you.

Type Generality

Types with 0 or more type parameters are called *type schemes*.

For now, to get a concrete type from a type scheme, replace ALL instances of a type parameter with a concrete type.

A type scheme, A, is **more general** than another type scheme, B, if every concrete instantiation of B is also one of A.

We write A $\sqsubseteq$ B.

Don’t worry, we will refine this in the coming weeks!

Type Generality Examples

```
'a = int
'a list * 'a list -> 'a list => int list * int list -> int list

'a = string
'a list * 'a list -> 'a list => string list * string list -> string list

'a = int, b' = bool
'a * 'b -> 'b => int * bool -> bool
```

```
'a list * 'a list -> 'a list ⊑ int list * int list -> int list

'a list * 'a list -> 'a list !⊑ int list * string list -> int list

'a list * 'b list -> 'a list ⊑ 'a list * 'a list -> 'a list
```

Equality Types

Write a list contains function…

Equality Types

- The double quoted variable arises from use of the = operator
- We can use = on most types like int, bool, string, tuples (that contain only “equality types”)
- Functions and real are not ”equality types”
- Generality rules work the same, except substitution must be some type which can be compared with =

!!! You can ignore warnings about “calling polyEqual”

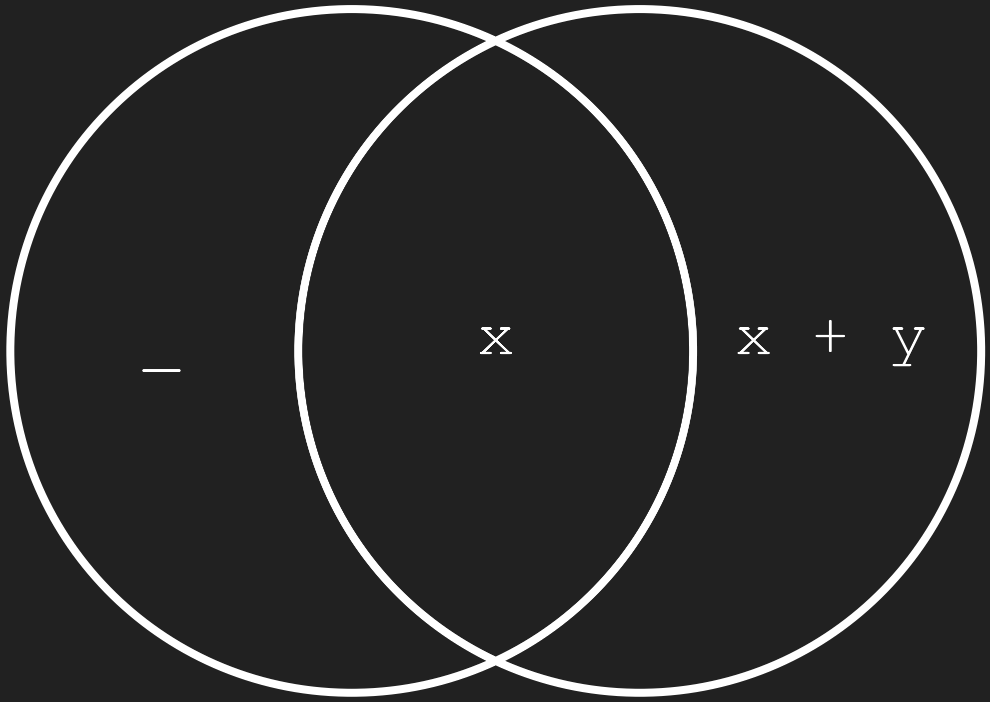
Variants

Pattern Matching Syntactic Sugar

Demo!

PATTERNS ≠ EXPRESSIONS

Patterns vs Expressions Examples



Patterns vs Expression Semantics Example

The pattern `x` *adds* a binding *to* the dynamic environment.

The expression `x` *looks up* a binding *from* the dynamic environment.

Tracing Pattern Matching