CSE 341: Programming Languages

Autumn 2005

Lecture 6 — Tail Recursion; Bindings; Course Motivation

Two Versions of Factorial

```
(* traditional factorial - not tail-recursive *)
fun fact n = if n<1 then 1 else n*fact(n-1);


(* tail-recursive version of factorial *)
(* this version uses an auxiliary function accum_fact
   that includes an accumulator (the product so far) *)
fun fact2 n =
    let fun accum_fact (n,prod) =
        if n<1 then prod else accum_fact(n-1,n*prod)
    in
        accum_fact(n,1)
    end;
```


Min-Exercise - Tail Recursion

Consider the following definition of the `length` function.

Is it tail recursive? If not, write a tail recursive version.

```
fun length [] = 0
| length (_ :: xs) = 1 + length(xs)
```


Min-Exercise - Solution

```
fun length xs =
   let fun acc_length ([],n) = n
       | acc_length(y::ys,n) = acc_length(ys,n+1)
   in
      acc_length(xs,0)
   end;
```

Tail calls

If the result of `f(x)` is the result of the enclosing function body, then `f(x)` is a *tail call*.

More precisely, a tail call is a call in *tail position*:

- In `fun f(x) = e`, `e` is in tail position.
- If `if e1 then e2 else e3` is in tail position, then `e2` and `e3` are in tail position (not `e1`). (Similar for case).
- If `let b1 ... bn in e end` is in tail position, then `e` is in tail position (not any binding expressions).
- Function arguments are not in tail position.
- ...

Significance of Tail Recursion

Why does this matter?

- Normally, a recursive function requires space proportional to depth of function calls ("call stack" must "remember what to do next")
- But particularly for functional languages, the implementation must ensure that tail calls are implemented in a space-efficient way
- Accumulators are a systematic way to make some functions tail recursive
- "Self" tail-recursive is very loop-like because space does not grow.

Deep patterns

Patterns are much richer than we have let on. A pattern can be:

- A variable (matches everything, introduces a binding)
- `_` (matches everything, no binding)
- A constructor and a pattern (e.g., `C` p) (matches a value if the value "is a `C`" and p matches the value it carries)
- A pair of patterns (($\boldsymbol{p1}$,$\boldsymbol{p2}$)) (matches a pair if $\boldsymbol{p1}$ matches the first component and $\boldsymbol{p2}$ matches the second component)
- A record pattern...
- An integer constant...
- ...

Inexhaustive matches may raise exceptions and are bad style.

Arguments to functions

Interesting fact: Every ML function takes exactly one argument!

- `fun f1 () = 34`
- `fun f2 (x,y) = x + y`
- `fun f3 pr = let val (x,y) = pr in x + y end`

There isn't any difference to callers between `f2` and `f3`.

In most languages, "argument lists" are syntactically separate, *second-class* constructs.

Can be useful: `f2 (if e1 then (3,2) else pr)`

Mini-Exercise - Patterns

Given these definitions:

```
fun pat1 (x::y::zs) = (x,y,zs)
fun pat2 (x,(y,z)) = (x,y,z)
```

What is the result of evaluating each of these expressions?

```
pat1 [1,2,3,4,5,6]
pat2 ((4,5), (10,11))
```


A question?

What's the best car?

What are the best kind of shoes?

Aren't all languages the same?

Yes: Any input-output behavior you can program in language X you can program in language Y

- Java, ML, and a language with one loop and three infinitely-large integers are "equal"
- This is called the "Turing tarpit"

Yes: Certain fundamentals appear in most languages (variables, abstraction, each-of types, *inductive definitions*, ...)

- Travel to learn more about where you're from

No: Most cars have 4 tires, 2 headlights, ...

- Mechanics learn general principles and what's different

Aren't these academic languages worthless?

In the short-term, maybe: Not many summer internships using ML?

But:

- Knowing them makes you a better Java, C, and Perl programmers (affects your idioms)
- Java did not exist in 1993; what does not exist now?
- Do Java and Scheme have anything in common? (Hint: check the authors)
- Eventual vindication: garbage-collection and generics

Aren't the semantics my least concern?

Admittedly, there are many important considerations:

- What libraries are available?
- What does my boss tell me to do?
- What is the de facto industry standard?
- What do I already know?

Technology *leaders* affect the answers to these questions.

Sound reasoning about programs, interfaces, and compilers *requires* knowledge of semantics.

Aren't languages somebody else's problem?

If you design an *extensible* software system, you'll end up designing a (small?) programming language!

Examples: VBScript, JavaScript, PHP, ASP, QuakeC, Renderman, bash, AppleScript, emacs, Eclipse, AutoCAD, ...

Another view: A language is an interface with just a few functions (evaluate, typecheck) and a sophisticated input type.

In other words, an interface is just a stupid programming language.

Summary

There is no such thing as a "best programming language". (There are good general design principles we will study.)

A good language is a relevant, crisp, and clear interface for writing software.

Software leaders should know about programming languages.

Learning languages has super-linear payoff.

- But you have to learn the semantics and idioms, not a cute syntactic trick for printing "Hello World".

End of the course: Language-design goals, mechanisms, and trade-offs

Next time: why ML, Scheme, and Smalltalk?