Min-Exercise - Tail Recursion

Consider the following definition of the `length` function.

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

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

Min-Exercise - Solution

```ml
fun length xs = 
  let fun acc_length (_::[]) = 0
      | 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 \( \text{fun } f(x) = e \), \( e \) is in tail position.
- If \( \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \) is in tail position, then \( e_2 \) and \( e_3 \) are in tail position (not \( e_1 \)). (Similar for case).
- If \( \text{let } b_1 \ldots \text{ in } e \text{ end} \) is in tail position, then \( e \) is in tail position (not any binding expressions).
- Function arguments are not in tail position.
- ...

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., \( \text{C } p \)) (matches a value if the value “is a \( \text{C} \)” and \( p \) matches the value it carries)
- A pair of patterns (\( \langle p_1, p_2 \rangle \)) (matches a pair if \( p_1 \) matches the first component and \( p_2 \) matches the second component)
- A record pattern...
- An integer constant...
- ...

Inexhaustive matches may raise exceptions and are bad style.

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.

Arguments to functions

Interesting fact: Every ML function takes exactly one argument!

- \( \text{fun } f_1() = 34 \)
- \( \text{fun } f_2(x,y) = x + y \)
- \( \text{fun } f_3 p = \text{let } \text{val } (x,y) = p \text{ in } x + y \text{ end} \)

There isn’t any difference to callers between \( f_2 \) and \( f_3 \).

In most languages, “argument lists” are syntactically separate, second-class constructs.

Can be useful: \( f_2 (\text{if } e_1 \text{ then } (3,2) \text{ else } p) \)
Mini-Exercise - Patterns

Given these definitions:

\[
\begin{align*}
\text{fun pat1 (x::y::zs)} &= (x, y, zs) \\
\text{fun pat2 (x, (y, z))} &= (x, y, z)
\end{align*}
\]

What is the result of evaluating each of these expressions?

\[
\begin{align*}
\text{pat1 [1,2,3,4,5,6]} \\
\text{pat2 ((4,5), (10,11))}
\end{align*}
\]

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?