TAIL POSITION

Regular Recursion

- Consider:

```
> (define fact
    (lambda (n)
      (if (zero? n)
          1
          (* n (fact (- n 1))))))
> (trace fact)
(fact)
> (fact 4)
|(fact 4)
| (fact 3)
| |(fact 2)
| | (fact 1)
| | |(fact 0)
| | |1
| | 1
| |2
| 6
|24
24
```

Tail Recursion

- Contrast with:

```
> (define facT
    (lambda (n accu)
      (if (zero? n)
          accu
          (facT (- n 1) (* n accu)))))
> (trace facT)
(fact)
> (facT 4 1)
|(fact 4 1)
|(fact 3 4)
|(fact 2 12)
|(fact 1 24)
|(fact 0 24)
|24
24
```

Control Context

- For **`fac`**, we had to call **`eval-expression`** over and over again to determine the value of the recursive calls, letting the prior calls wait until we bottomed out.
- We had to build what we will call a *control context*.
- In other words, we were calling **`apply-proc`** on the recursive call before we were able to call it on our own call.
- For **`facT`**, we were able to call **`apply-proc`** after a simple evaluation of the arguments.

Tail Form

- Our goal is to identify a class of expressions in which no procedure call is in a position that requires control context to be built.
- We say that such an expression is in *tail form*.
- Important observation:

  It is the evaluation of actual parameters, NOT the calling of procedures, that requires creating a control context.

Tail Form

- Our goal is to define formally the notion of *tail-form* expression.
- We will identify two things:
  - We identify those places where evaluation of an expression would not require a control context to be generated. We call these the *tail positions* of the language, and say that procedure calls in these positions are *tail calls.*
  - We identify a syntactic class of expressions guaranteed not to execute any procedure calls. We say such expressions are simple.

Head and Tail Positions

- A sub-expression in *head position* is one that must be evaluated, could be evaluated first, and is evaluated in the environment of the entire expression.
- An expression may have more than one head position because our interpreter does not, for example, specify the order of evaluation of the actual parameters of a procedure.
- A sub-expression in *tail position* has the property that if it is evaluated, its value immediately becomes the value of the entire expression.
- For a sub-expression in tail position, no information need be saved, and therefore no control context need be built.
- An expression may have more than one tail position. For example, our if expressions may choose either the true or the false branch.

Head and Tail Positions

- (*prim-op* H … H)
- (if H T T)
- (let ((*v* H) … (*v* H)) T)
- (lambda (*v* … *v*) T)
- (H H … H)
- (set! *v* H)
- (letrec ((*v* H) … (*v* H)) T)
- (begin H E … E T)

Simple Expressions

- We now turn to those expressions that can never cause a procedure call.
- These are the *simple* expressions.
- The intent of this definition is that the evaluation of a simple expression is just a short sequence of primitive operations, possibly with some closure creations and tests.

Simple Expressions

- A *simple expression* is defined as follows:
- A **literal** or a **variable reference** is always simple.
- A **primitive application** is simple if and only if all of its operands are simple.
- An **if expression** is simple if and only if all three of its sub-expressions are simple.
- A **let expression** is simple if and only if all of its sub-expressions are simple.
- A **lambda** expression is always simple.
- A **letrec** expression is simple if and only if its body is simple.
- A **procedure application** is never simple.

Simple Examples

- (car x)
- (if p x (car (cdr x)))
- (f (car x))
- (car (f x))
- (if p x (f (cdr x)))
- (if (f x) x (f (cdr x)))
- (lambda (x) (f x))
- (lambda (x) (car (f x)))

Simple Examples

• (car x)	simple
• (if p x (car (cdr x)))	simple
• (f (car x))	not simple
• (car (f x))	not simple
• (if p x (f (cdr x)))	not simple
• (if (f x) x (f (cdr x)))	not simple
• (lambda (x) (f x))	simple
• (lambda (x) (car (f x)))	simple

Tail Form

- Definition. A *tail-form expression* is one in which every sub-expression in non-tail position is simple.
- To cut to the chase: All calls to non-primitive procedures are in tail position.
- In this definition, the phrase “every sub-expression” means not just the immediate sub-expressions, but all sub-expressions. So in tail-form expression, the non-tail positions contain simple expressions that are also in tail-form, and the tail-positions contain tail-form expressions.
- We say non-tail "non-tail position" because in some languages there may be sub expressions that are in neither head nor tail position.

Tail Form Examples

- (car x)
- (if p x (car (cdr x)))
- (f (car x))
- (car (f x))
- (if p x (f (cdr x)))
- (if (f x) x (f (cdr x)))
- (lambda (x) (f x))
- (lambda (x) (car (f x)))

Tail Form Examples

- Expressions in red are not simple (i.e. procedure applications) in **H**ead position.

• (car x)	tail form
• (if p x (car (cdr x)))	tail form
• (f (car x))	tail form*)
• (car (f x))	not tail form
• (if p x (f (cdr x)))	tail form
• (if (f x) x (f (cdr x)))	not tail form
• (lambda (x) (f x))	tail form
• (lambda (x) (car (f x)))	not tail form

(*) Notice that while this is a procedure application, it is an application at the top-level. Notice that *f* and *(car x)* themselves are simple.

Expressions may be in tail form but not simple, or simple but not in tail form.

• (car x)	simple	tail form
• (if p x (car (cdr x)))	simple	tail form
• (f (car x))	not simple	tail form
• (car (f x))	not simple	not tail form
• (if p x (f (cdr x)))	not simple	tail form
• (if (f x) x (f (cdr x)))	not simple	not tail form
• (lambda (x) (f x))	simple*)	tail form
• (lambda (x) (car (f x)))	simple*)	not tail form

(*) Notice that the only reason these two are simple is because of the lambda. The second one is ***not*** in tail form because there is a ***sub***-expression `(f x)` that appears in ***Head*** position of the expression `(car (f x))`. The first one is in tail form because the application `(f x)` appears in ***Tail*** position.