Lec 8: Static Scope & Tail recursion

- Names
- Variables
- The Concept of Binding
- Scope
- Tail recursion → Iteration
  - factorial
  - sum of f(n)
  - Fibonacci
  - Euclids GCD

Variables

- A variable is an abstraction of a memory cell
- Variables can be characterized as a sextuple of attributes:
  - Name
  - Address
  - Value
  - Type
  - Lifetime
  - Scope

Variables Attributes

- Name – not all variables have them
- Address – the memory address with which it is associated
  - A variable may have different addresses at different times during execution
- Type – determines the range of values of variables and the set of operations that are defined for values of that type;
- Value – the contents of the location with which the variable is associated

Static Scope

- Based on program text —

Example 3.15 Consider the following indefinite integral

$$\int_0^z \left(\int_0^y f(x)dx + \int_0^y g(u)du \right) dy$$

It contains as *free* the names z, f and g. The other names x, u and y are *bound*. The scopes of the *bound* variables are shown below.

$$\underbrace{\int_0^z \left(\underbrace{\int_0^y f(x)dx}_{x} + \underbrace{\int_0^y g(u)du}_{u} \right) dy}_{y}$$

- To connect a name reference to a variable, the sml interpreter must find the declaration
- Search process: search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name

Scope Example

Big

```
 - declaration of X
    Sub1
     - declaration of y -
     ...
     call Sub2
     . reference to X, y

    Sub2
     ...
     - reference to X -
     ...

 ...
 call Sub1
 ...
```

Big calls Sub1

Sub1 calls Sub2

Sub2 uses X

Scope (continued)

- Variables can be hidden (shadowed) from a unit by having a variable with the same name

Scope Example

Big

- declaration of X

Sub1

- declaration of X -

...

call Sub2

...

Sub2

...

- reference to X -

...

...

call Sub1

...

Scope Example

```
Big
      - declaration of X
          Sub1
            - declaration of X -
            ...
            call Sub2
            ...

          Sub2

            ...
            - reference to X -
            ...

      ...
      call Sub1
      ...
```

– Reference to X is to Big's X

Scope Example

```
fun perfect (n) =
    let fun add_factors (n) =
            let fun f (i) =
                    if n mod i = 0 then i
                    else 0;
                 fun sum (a, b) =
                     if a > b then 0
                     else f(b) + sum (a, b-1);
             in sum (1, n div 2)
             end;
     in  n = add_factors (n)
    end;
```

Example 3.16 Now consider the complete ML code of Example 3.13 (perfect numbers).

- The name perfect is bound and has a scope which extends beyond the definition.
- The name add-factors is bound and has a scope which begins with its definition and extends right up to the end of the definition of perfect (n) but no further.
- Similarly the name f is bound and has a scope that extends up to the end of the definition of add-factors and no further. The name sum also has a scope similar to that of f.
- The variables a and b are bound and have scopes beginning at their first occurrence in the definition of sum and ending with the same definition.

Fibonacci numbers

$$F_1 = 1,$$
$$F_2 = 1,$$
$$F_n = F_{n-1} + F_{n-2} \text{ for all } n \geq 3.$$

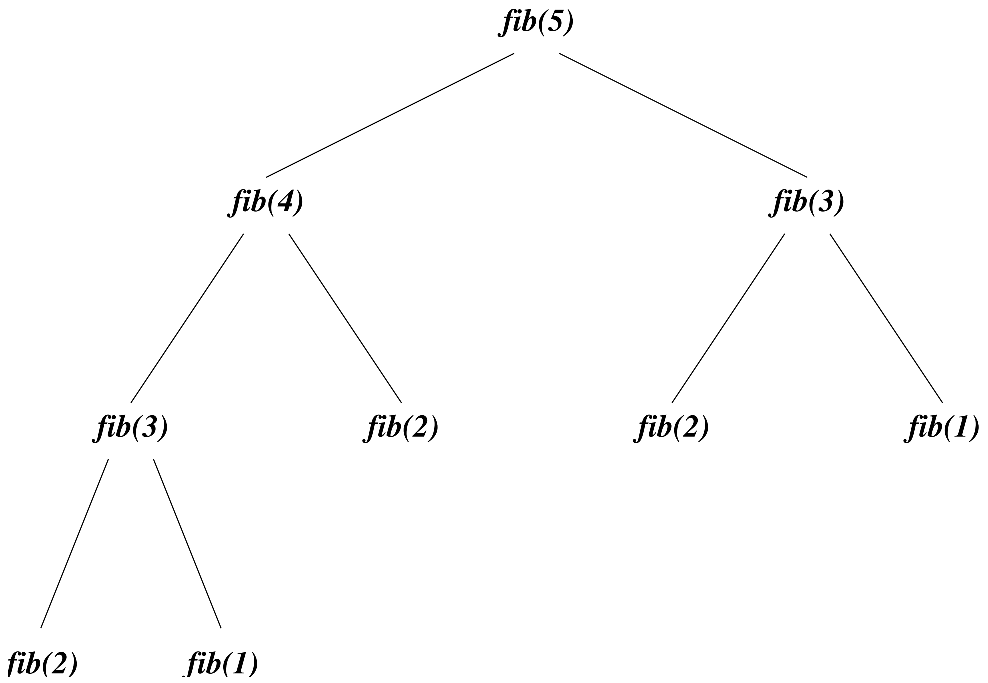
$F_1 = 1$
$F_2 = 1$
$F_3 = 2$
$F_4 = 3$
$F_5 = 5$
$F_6 = 8$
$F_7 = 13$
$F_8 = 21$
$\vdots$

Naive algorithm:

$$fib(n) = \begin{cases} 1 & \text{if } n \leq 2, \\ fib(n-1) + fib(n-2) & \text{otherwise.} \end{cases}$$

Why does it turn out to be so slow?

$$fib(n) = \begin{cases} 1 & \text{if } n \le 2, \\ fib(n-1) + fib(n-2) & \text{otherwise.} \end{cases}$$



Example: *factorial*

$$factorial(n) = \begin{cases} 1 & \text{if } n = 0, \\ factorial(n-1) \times n & \text{otherwise} \end{cases}$$

$$\begin{aligned} &factorial(3) \\ &= factorial(2) \times 3 \\ &= (factorial(1) \times 2) \times 3 \\ &= ((factorial(0) \times 1) \times 2) \times 3 \\ &= ((1 \times 1) \times 2) \times 3 \\ &= (1 \times 2) \times 3 \\ &= 2 \times 3 \\ &= 6 \end{aligned}$$

Suppose each multiplication takes the same amount of time.
(True when multiplying `int`s!)

Total time
$\propto$ number of multiplications
= ?

$$factorial(n) = \begin{cases} 1 & \text{if } n = 0, \\ factorial(n-1) \times n & \text{otherwise} \end{cases}$$

We also need space to:

- keep track of deferred operations
- or, stack up frames for function calls

Similarly, show that #frames = n + 1: *space complexity*

$$\begin{aligned} & factorial(3) \\ &= factorial(2) \times 3 \\ &= (factorial(1) \times 2) \times 3 \\ &= ((factorial(0) \times 1) \times 2) \times 3 \\ &= ((1 \times 1) \times 2) \times 3 \\ &= (1 \times 2) \times 3 \\ &= 2 \times 3 \\ &= 6 \end{aligned}$$

$\boxed{factorial(3)}$

$\boxed{factorial(2)}$

$\boxed{factorial(1)}$

$\boxed{factorial(0)}$

Iterative Process

- Represent state of the computation with auxiliary variables and maintain some key invariant property with the variables
    - count=1, product=1, target=n=6
    - count=2, product=2, target=n
    - count=3 product=6, target=n
    - …..
    - count=6 product=720, target=6
        - >>>> count==target result=product=720
- Obtain the final result from final state of these variables

Tail Recursion

$\langle$*Iterative factorial*$\rangle\equiv$

```
fun factorial (n) =
    let ⟨Code for fact_iter⟩
     in fact_iter (n, 1, 0)
    end;
```

$\langle$*Code for* `fact_iter`$\rangle\equiv$

```
fun fact_iter (m, f, c) =
    if c=m then f
    else fact_iter (m, f*(c+1), c+1);
```

$$
\begin{aligned}
&factorial(5)\\
&\quad = fact\_iter(5, 1, 0)\\
&\quad = fact\_iter(5, 1, 1)\\
&\quad = fact\_iter(5, 2, 2)\\
&\quad = fact\_iter(5, 6, 3)\\
&\quad = fact\_iter(5, 24, 4)\\
&\quad = fact\_iter(5, 120, 5)\\
&\quad = 120
\end{aligned}
$$

- Proof of Correctness?
    - Induction based on the invariant condition

Example – Summation of a function

$$\mathit{Iterative\ computation\ of}\ \sum_a^b f(n)$$

$$sum(a,b) = sum\_iter(a,b,0)$$

where, the auxiliary function $sum\_iter : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is given as

$$\begin{aligned} & sum\_iter(c, c_f, s) \\ & \quad = \begin{cases} s & \text{if } c = c_f + 1 \\ sum\_iter(c+1, c_f, s+f(c)) & \text{otherwise} \end{cases} \end{aligned}$$

⟨*Iterative sum*⟩≡

```
fun sum (a, b) =
    let ⟨Code for sum_iter⟩
     in sum_iter (a, b, 0)
    end;
```

⟨*Code for* `sum_iter`⟩≡

```
fun sum_iter (c, cf, s) =
    if c = cf+1 then s
    else sum_iter (c+1, cf, s + f(c));
```

»

Iterative Process – Fibonacci

- Represent state of the computation at each stage in terms of some auxiliary variables.
- Maintain a key invariant property
  - eg for Fibonacci —
  - count, a,b (a=fib[count-1], b = fib[count-2])
- Obtain the final result from the final state of these variables
  - i.e when count=n fib[n]=a+b

Iterative Fibonacci

$$(n \geq 3) \wedge (3 \leq count \leq n) \wedge (a = fib(count - 2)) \wedge (b = fib(count - 1))$$

Then, when $count = n$, the process may terminate and we may obtain the value $a + b = fib(count - 2) + fib(count - 1) = fib(n)$ as the final answer. An algorithm based on this invariant condition can be described as

$$fib(n) = \begin{cases} 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ fib\_iter(n, 1, 1, 3) & \text{otherwise} \end{cases}$$

where $fib\_iter(n, a, b, count) : \mathbb{P} \times \mathbb{P} \times \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{P}$ is an auxiliary function defined as

$$fib\_iter(n, a, b, count) = \begin{cases} a + b & \text{if } count = n \\ fib\_iter(n, b, a + b, count + 1) & \text{otherwise} \end{cases}$$

⟨*Iterative Fibonacci*⟩≡

```
fun fib (n) =
    let ⟨Code for fib_iter⟩
     in if n<=2 then 1
        else fib_iter (n, 1, 1,3)
    end;
```

⟨*Code for* fib_iter⟩≡

```
fun fib_iter (n, a, b, count) =
    if count = n then a+b
    else fib_iter (n, b, a+b, count+1);
```

GCD (Euclid)

» Greatest Common Divisor (also known as Highest Common Factor) of numbers a,b

» gcd(a,b) use the property

» **Claim:** If $a = qb + r$, $0 < r < b$, then $gcd(a, b) = gcd(b, r)$

» Euclids Algorithm

» $$Euclid\_gcd(a, b) = \begin{cases} a & \text{if } b = 0 \\ Euclid\_gcd(b, (a \ mod \ b)) & \text{otherwise} \end{cases}$$