A lispy compiler

Yves Müller

Source language

inspired by Scheme

Scheme is:

- **functional**
- **dynamic** typed
- **strong** typed
- **lexical** scoped

```
(define (fib n)
  (if (< n 2)
      n
      (+
         (fib (- n 1))
         (fib (- n 2))
      )
   )
)
(fib 20)
```

(Source: http://en.literateprograms.org/Fibonacci_numbers_(Scheme))

Source language

my language is:

- **functional**
- **static** typed
- **lexical** scoped
- **inferred** typed

my language should be:

- all above
- + dynamic typed
- mixed typed

```
(lambda (x)
   (let ((f (lambda (n)
      (if (< n 2)
         n
      (+
         (fib (- n 1))
         (fib (- n 2))
      ))))

      (isNumber x
         (f x)
         error
      )
   )
) 20
```

Compile process

0. (parse s-expressions)
1. **preprocess**

before:

```
(
    (let ((x 4) (y 3))
       (lambda(x)
          ( + x y 1)
       )
    ) 2
)
```

after:

```
(
    ((lambda (x y) (
       (lambda(x)
          ( + x (+ y 1))
       )
    ) 4 3) 2
)
```

Compile process

0. (parse s-expressions)
1. preprocess
2. **detect bindings**

before:

```
(
    ((lambda (x y) (
       (lambda(x)
          ( + x (+ y 1))
       )
    ) 4 3) 2
)
```

after:

```
(
    ((lambda (e1x e1y) (
       (lambda(e2x)
          ( + e2x
             (+ e1y 1))
       )
    ) 4 3) 2
)
```

Compile process

0. (parse s-expressions)
1. preprocess
2. detect bindings
3. **close names**

before:

```
(
    ((lambda (e1x e1y) (
       (lambda(e2x)
          ( + e2x
             (+ e1y 1))
       )
    ) 4 3) 2
)
```

after:

```
(
    ((lambda (e1x e1y) (
       (closure(e2x e1y)
          ( + e2x
             (+ e1y 1))
       (e1y) )
    ) 4 3) 2
)
```

Compile process

0. (parse s-expressions)
1. preprocess
2. detect bindings
3. close names
4. **infer typed**

before:

```
(
    ((lambda (e1x e1y) (
       (lambda(e2x)
          ( + e2x
             (+ e1y 1))
       )
    ) 4 3) 2
)
```

after with int and (int x int -> int):

```
(
    ((lambda (e1x e1y) (
       (closure(e2x e1y)
          ( + e2x
             (+ e1y 1))
       (e1y) )
    ) 4 3) 2
)
```

Compile process

0. (parse s-expressions)
1. preprocess
2. detect bindings
3. close names
4. infer typed
5. **translate to llvm-ic**

before:

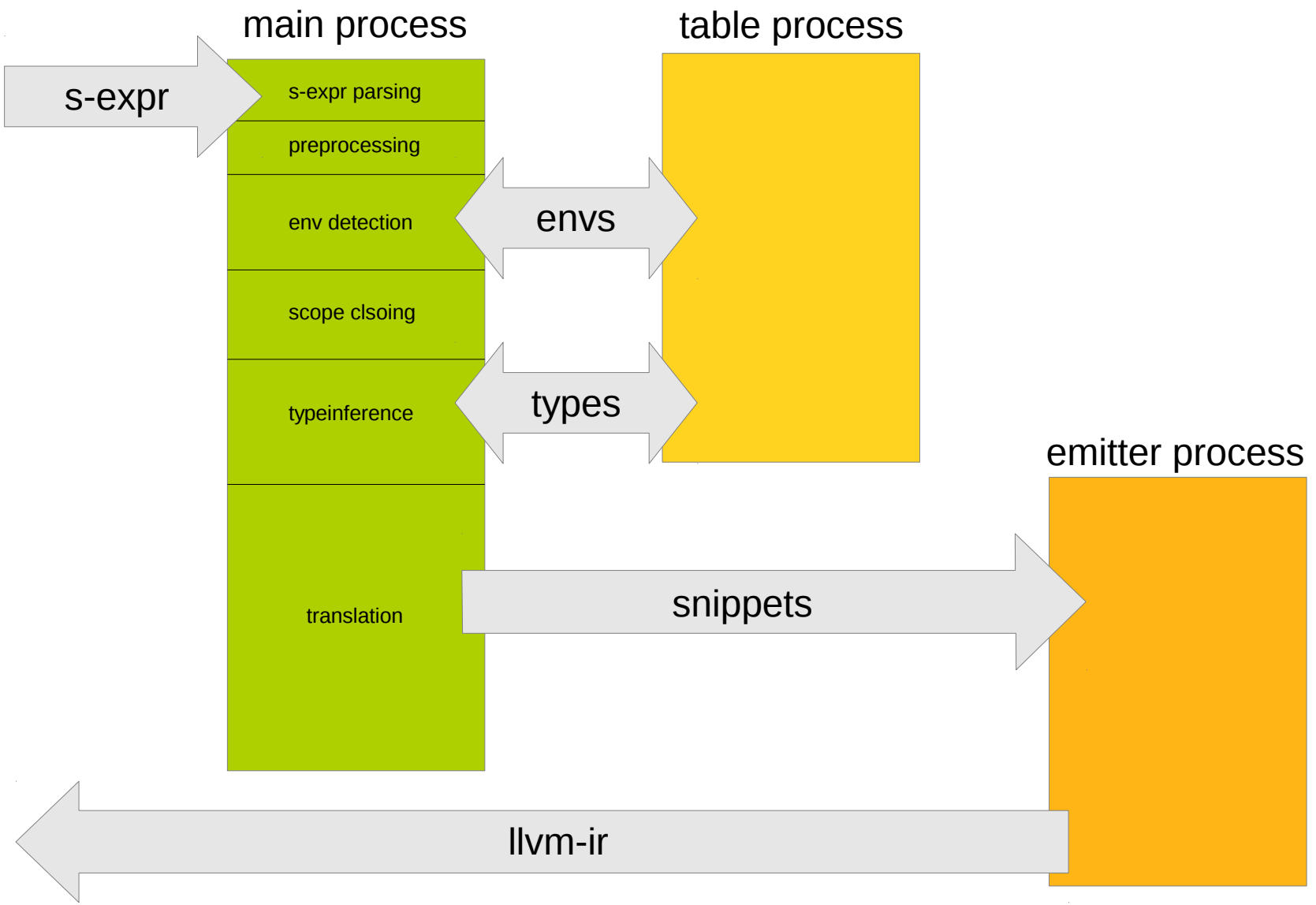
```
(
    ((lambda (e1x e1y) (
       (closure(e2x e1y)
          ( + e2x
             (+ e1y 1))
       (e1y) )
    ) 4 3) 2
)
```

after:

```
define i32 @env_2 ( i32 %env_2_x, i32 ...
        %tmp_0 = add i32 %env_2_x , ...
        ret i32 %tmp_0
}

%clojure_env_2 = type { i32(i32, i32)*,
i32 }
define %clojure_env_2* @env_1 ( i32 ...
        %tmp_1 = alloca %clojure_env_2
        %tmp_2 = getelementptr ...
        store i32(i32, i32)* @env_2, ...
        %tmp_3 = getelementptr ...
        store i32 %env_1_y, i32* %tmp_3
        ret %clojure_env_2* %tmp_1
}
```

Application Structure



On completeness

Lisp (J. McCarthy)

- head
- tail
- cons
- cond ✓
- quote
- eq ✓
- lambda ✓

other features

- static types ✓
- type inference ✓
- passable closures ✓
- basic types ✓
- and operations ✓
- let-Bindings ✓

Major Problem I

type inference isn't always working

```
( lambda (f x y)
      (f x
            ( + y 3)
      )
)
```

solution:

- add dynamic type support

as intened, but i had no time yet

- add optinal type anotations

```
( letType (( f ( function (int int) int)))
      ( lambda (f x y)
            (f x
                  + y 3)
            )
      )))
```

Major Problem II

no way to express recursion

⇒not touring complete

solutions:

- global define
- letrec
- fixpoint operator

```
( let
      ((f
            (lambda (x)
                  (cond (= x 0)
                        1
                        (f (+ x 1))
                  )
            )
      ))

      (f 3)
)
```

doesn't work, because f unknown in definition of f

todo overview

- enable recursion
- documentation

- improve type interference
- implement heap structures
  - lists ⇒ implement quote
  - store closures on head
- implement dynamic-typed code generation

Questions?

```
(print ((lambda () ( “kthxbye” )))
```