TDDA69 Data and Program Structure
Declarative Computation Implementation

Cyrille Berger

LINKÖPING UNIVERSITY

Lectures

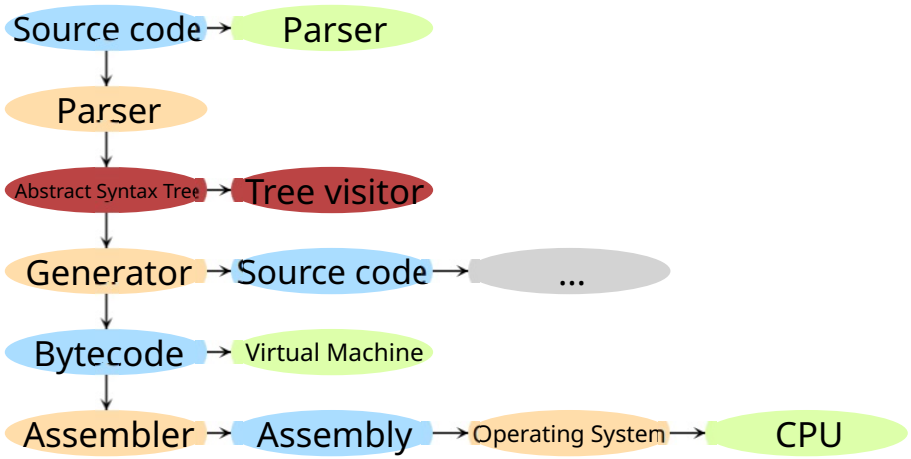
Lecture content

- Practical Normal Order Evaluator
- Practical Applicative Order Evaluator

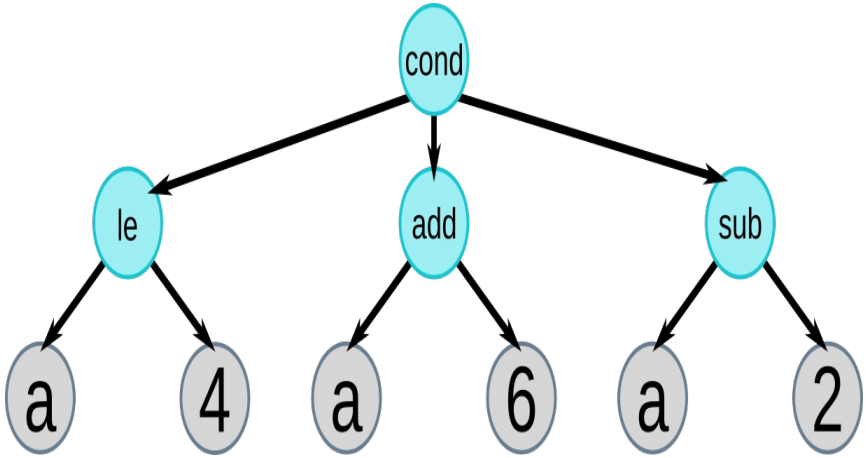
Applicative vs Normal

- Recursively evaluate all the subexpressions, but in what order?
  - *Applicative order*: evaluate the arguments and then apply. Evaluate the procedure body with every formal parameter replaced with the corresponding argument value
  - *Normal order*: fully expand and then reduce. Recursively expand every procedure until only primitive operators are left. Evaluate (reduce) the result.

How is a program interpreted?

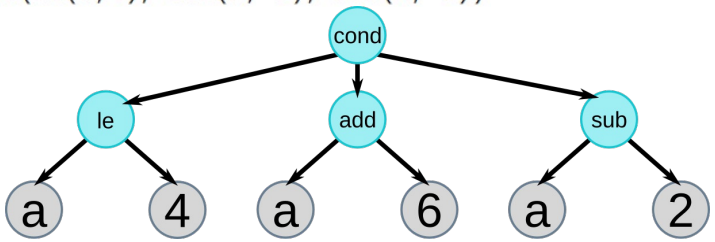


AST Execution



AST as List of List

```
cond(le(a,4), add(a, 6), sub(a, 2))
```



- As array:

```
['cond', ['le', 'a', 4], ['add', 'a', 6], ['sub',
'a', 2]
```

In Lisp:

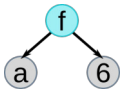
```
(cond ((le a 4) (add a 6)) (T (sub a 2)))
```

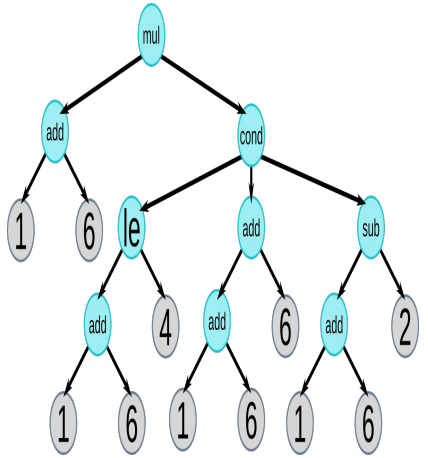

Practical Normal Order Evaluator

Practical Normal Order Evaluator

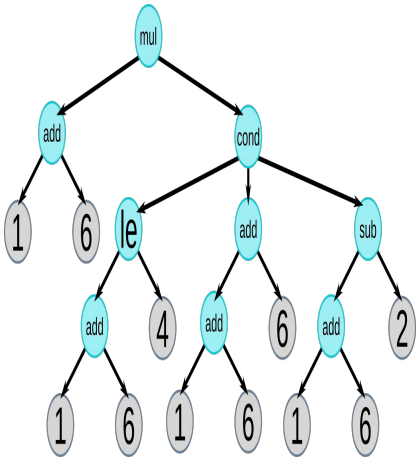
- Expansion
  - Replace *identifiers* by their values
  - Replace functions by their bodies
  - Evaluate assignements
- Reduction
  - Apply cond
  - Apply base functions

Expansion





Reduction



35

Expansion: define (1/2

```
define a = add(1, 6)
expand(['define', 'a', ['add' 1, 6]], root_env)
['add' 1, 6]
```

```
def expand(node, env):
  if not isinstance(node, list):
    return node
  elif node[0] == 'define':
    value = expand(node[2], env)
    env.define(node[1], value)
    return value
  return node
```

Expansion: define (2/2

```
define a
expand(['define', 'a'], env)
None
a = add(1, 6)
expand(['=', 'a', ['add' 1, 6]], env)
['add' 1, 6]
```

```
def expand(node, env):
  if not isinstance(node, list):
    return node
  elif node[0] == 'define':
    if len(node) == 3:
      value = expand(node[2], env)
    else:
      value = None
    env.define(node[1], value)
    return value
  elif node[0] == '=':
    value = expand(node[2], env)
    env.assign(node[1], value)
    return value
  return node
```


Expansion: variables

```
a
expand('a', root_env)
['add' 1, 6]
add(a, 6)
expand(['add', 'a', 7], root_env)
['add', ['add' 1, 6], 7]
```

```
def expand(node, env):
  if isinstance(node, string):
    return env.value(node)
  elif not isinstance(node, list):
    return node
  elif node[0] == 'define':
    if len(node) == 3:
      value = expand(node[2], env)
    else:
      value = None
    env.define(node[1], value)
    return value
  elif node[0] == '=':
    value = expand(node[2], env)
    env.assign(node[1], value)
    return value
  else:
    r = [node[0]]
    for i in range(1, len(node)):
      r.append(expand(node[i], env))
    return r
```

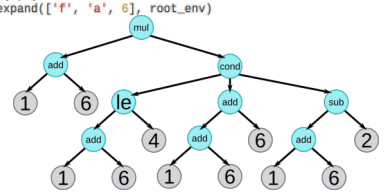

Expansion: define function

```
define f = function(x,y) -> z {
  z = mul(x, cond(le(x,4), add(x, y), sub(x, 2))
}
expand(['define', 'f', ['function' ['x', 'y'], 'z', ['='
z, [...]]], root_env)
function(root_env, ['x', 'y'], 'z', ['=' z, [...]])
```

```
def expand(node, env):
  if isinstance(node, string):
    return env.value(node)
  elif not isinstance(node, list):
    return node
  elif node[0] == 'define':
    if len(node) == 3:
      value = expand(node[2], env)
    else:
      value = None
    env.define(node[1], value)
    return value
  elif node[0] == '=':
    value = expand(node[2], env)
    env.assign(node[1], value)
    return value
  elif node[0] == 'function':
    return function(env, node[1], node[2], node[3])
  else:
    r = [node[0]]
    for i in range(1, len(node)):
      r.append(expand(node[i], env))
    return r
```


Expansion: call function

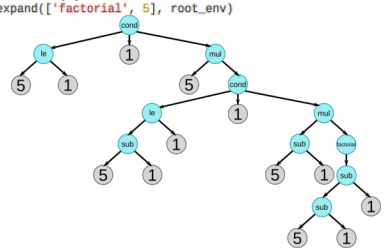
```
f(a,6)
expand(['f', 'a', 6], root_env)
```




```
def expand(node, env):
  if isinstance(node, string):
    return env.value(node)
  elif not isinstance(node, list):
    return node
  elif node[0] == 'define':
    if len(node) == 3:
      value = expand(node[2], env)
    else:
      value = None
    env.define(node[1], value)
    return value
  elif node[0] == '=':
    value = expand(node[2], env)
    env.assign(node[1], value)
    return value
  elif node[0] == 'function':
    return function(env, node[1], node[2], node[3])
  else:
    r = [node[0]]
    for i in range(1, len(node)):
      r.append(expand(node[i], env))
    if env.has(r[0]):
      func = env.value(r[0])
      nenv = environment(func.env)
      for i in range(0, len(func.arguments)):
        nenv.define(func.arguments[i], r[i+1])
      expand(func.body, nenv)
      return nenv.value(func.ret)
    else:
      return r
```

Expansion: problem with recursion

```
define factorial = function(n) -> r {
  r = cond(le(n, 1), 1, mul(n, factorial(sub(n, 1))))
}
factorial(5)
expand(['factorial', 5], root_env)
```




```
def expand(node, env):
  if isinstance(node, string):
    return env.value(node)
  elif not isinstance(node, list):
    return node
  elif node[0] == 'define':
    if len(node) == 3:
      value = expand(node[2], env)
    else:
      value = None
    env.define(node[1], value)
    return value
  elif node[0] == '=':
    value = expand(node[2], env)
    env.assign(node[1], value)
    return value
  elif node[0] == 'function':
    return function(env, node[1], node[2], node[3])
  else:
    r = [node[0]]
    for i in range(1, len(node)):
      r.append(expand(node[i], env))
    if env.has(r[0]):
      func = env.value(r[0])
      nenv = environment(func.env)
      for i in range(0, len(func.arguments)):
        nenv.define(func.arguments[i], r[i+1])
      expand(func.body, nenv)
      return nenv.value(func.ret)
    else:
      return r
```


Expansion: fix recursion

```
define factorial = function(n) -> r {
  r = cond(le(n, 1), 1, mul(n, factorial(sub(n, 1))))
}
factorial(5)
condition(fenv, ['le', 5, 1], 1, ['mul' ... ])
define f = function(s) -> r { r = f(add(s, 1)) }
```

```
def expand(node, env):
  if isinstance(node, string):
    return env.value(node)
  elif not isinstance(node, list):
    return node
  elif node[0] == 'define':
    if len(node) == 3:
      value = expand(node[2], env)
    else:
      value = None
    env.define(node[1], value)
    return value
  elif node[0] == '=':
    value = expand(node[2], env)
    env.assign(node[1], value)
    return value
  elif node[0] == 'function':
    return function(env, node[1], node[2], node[3])
  elif node[0] == 'cond':
    t = expand(node[1], env)
    return condition(env, t, node[2], node[3])
  else:
    r = [node[0]]
    for i in range(1, len(node)):
      r.append(expand(node[i], env))
    if env.has(r[0]):
      func = env.value(r[0])
      nenv = environment(func.env)
      for i in range(0, len(func.arguments)):
        nenv.define(func.arguments[i], r[i+1])
      expand(func.body, nenv)
      return nenv.value(func.ret)
    else:
      return r
```


Reduction (1/2)

```
add(1,6)
reduce(['add', 1, 6])
 7
add(mul(2,3),6)
reduce(['add', ['mul', 2, 3], 6])
 12
```

```
def reduce(node):
  if not isinstance(node, list):
    return node
  else:
    f = self.base_func[node[0]]
    return f(*map(reduce,
                  node[1:]))
```


Reduction (2/2)

```
cond(eq(0, 0), 0, div(6, 0))
reduce(condition(cenv, ['eq', 0, 0], 0,
['div', 6, 0]))
0
```

```
def reduce(node):
  if isinstance(node, condition):
    if reduce(condition.test):
      return reduce(expand(node.true_exp,
                           condition.env))
    else:
      return reduce(expand(node.false_exp,
                           condition.env))
  elif no isinstance(node, list):
    return node
  else:
    f = self.base_func[node[0]]
    return f(*map(reduce, node[1:]))
```

Eval function

```
def eval(source):
    global root_env
    return reduce(
            expand(parse(source),
                   root_env))
```


Practical Applicative Order Evaluator

Evaluate

```
define a = add(1, 6)
evaluate(['define', 'a', ['add' 1, 6]], root_env)
7
define a
evaluate(['define', 'a'], root_env)
None
a = add(1, 6)
evaluate(['=', 'a', ['add' 1, 6]], root_env)
7
a
evaluate('a', root_env)
7
add(a, 6)
evaluate(['add', 'a', 7], root_env)
14
cond(eq(0, 0), 0, div(6, 0))
reduce(['cond', ['eq', 0, 0], 0, ['div', 6, 0]])
0
define f = function(x,y) -> z {
  z = mul(x, cond(le(x,4), add(x, y), sub(x, 2))
}
evaluate(['define', 'f', ['function' ['x', 'y'], 'z', ['=' z,
[...]]], root_env)
function(root_env, ['x', 'y'], 'z', ['=' z, [...]])
f(a,6)
evaluate(['f', 'a', 6], root_env)
35
```

```
def evaluate(node, env):
  if isinstance(node, string):
    return env.value(node)
  elif not isinstance(node, list):
    return node
  elif node[0] == 'define':
    if len(node) == 3:
      value = evaluate(node[2], env)
    else:
      value = None
    env.define(node[1], value)
    return value
  elif node[0] == '=':
    value = evaluate(node[2], env)
    env.assign(node[1], value)
    return value
  elif node[0] == 'function':
    return function(env, node[1], node[2], node[3])
  elif node[0] == 'cond':
    if eval(node[1], env):
      return eval(node[2], env)
    else:
      return eval(node[3], env)
  else:
    args = map(lambda x: evaluate(x, env), node[1:])
    if env.has(r[0]):
      func = env.value(r[0])
      nenv = environment(func.env)
      for n,v in zip(func.argument, args):
        nenv.define(n, v)
      evaluate(func.body, nenv)
      return nenv.value(func.ret)
    else:
      f = self.base_func[node[0]]
      return f(*args)
```


Conclusion

- Normal and applicative practical implementation