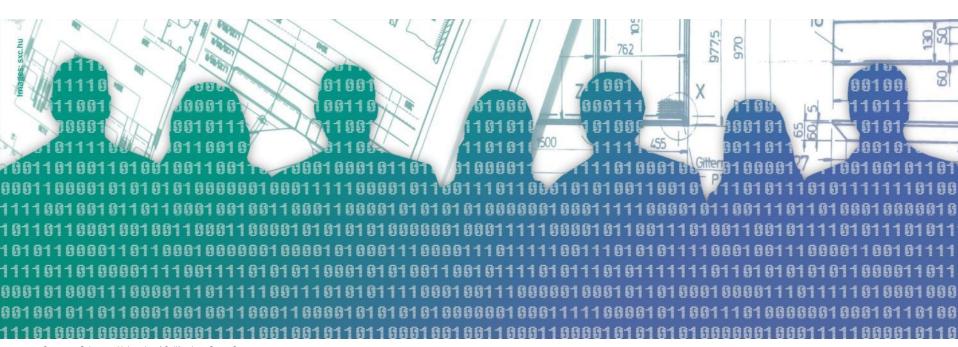


Homework 4 – CEK Machine implementation

CMPS203 Programming Languages – Spring 2016 Richard Jullig



Homework 4: CEK machine implementation



The assignment is to implement the CEK machine for ISWIM together with a front-end.

Homework submission:

- Programming language: Python
- To make grading a feasible task you must submit
 - The source code of your implementation
 - Test data (as specified below) for your system.
- Where to submit: details forthcoming (probably on eCommons)
- Due date: Sunday, 05 June 2016

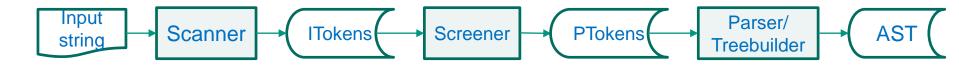
System Overview



The overall structure of the system to be implemented is as follows:



The front-end has the following components:



- Scanner: maps the input string into a sequence of input tokens
 - Essentially a finite state machine
- Screener: filters input tokens and converts them into parser tokens
- Parser/Treebuilder: maps the sequence of parser tokens into an AST
 - Recommended approach: recursive descent parser (more below)

The Scanner



- The scanner will accept strings over the alphabet including
 - Lower and upper case letters
 - Digits
 - Parentheses
 - The operator symbols +, -, *, ^
- The scanner will output a sequence of input tokens; each input token specifies its
 - Class (specified below)
 - Instance: the substring of the input string consumed by the token

```
Scanner classes

ID = Letter (Letter | Digit)*

NUM = Digit+

OP = '+' | '-' | '*' | '^'

Lparen = '('

Rparen = ')'

WS = (<blank> | <newline>)+
```

The Screener



- The screener filters and maps input tokens to parser tokens
 - Input tokens dropped: WS (white space tokens)
 - Input tokens passed through: Lparen, Rparen
 - Input tokens transformed: see table below

Input to Parser token map

```
ID: "lam" → lam
```

ID: "app" → app

ID: "add1" \rightarrow op1:"add1"

ID: "sub1" \rightarrow op1:"sub1"

ID: "iszero" → op1:"iszero"

ID: "..." → var:"..."

NUM: "..." → num:"..."

OP: "+" → op2:"+"

OP: "-" → op2:"-"

OP:" "*" → op2:"*"

OP: "^" \rightarrow op2:"^"

The screener

- Recognizes the reserved words "lam", "app", "add1", "sub1", "iszero"
- All other identifiers are variable names.
- Note: the parser only uses the token class (of variables, numbers, operators); however, the evaluator needs access to specific variable names, number values, and operator names, and must therefore be preserved in the AST.

The Parser



Construct a recursive descent parser for the following context-free grammar:

```
T = \langle \text{var} \rangle
| \langle \text{lambda } bvar : \text{var } body : T \rangle
| \langle \text{app } fun : T \ arg : T \rangle
| \langle \text{op1 } arg11 : T \rangle
| \langle \text{op2 } arg12 : T \ arg22 : T \rangle
| \langle \text{num} \rangle
```

Value terms

```
V = \langle \text{lam var } E \rangle| \langle \text{num } b \rangle
```

- Recursive descent parser
 - Collection of recursive functions corresponding to nonterminals
 - Constructs leftmost derivation
- AST builder: while parsing, construct abstract syntax tree
- Tree grammar: specifies the set of ISWIM abstract syntax trees
 - In a functional language, interpret as a data type
 - In OO language, interpret as abstract class T with concrete subclasses (lambda, app, etc.)

Parser/Tree builder outline



- E: recursive function
 - Input: sequence of parser tokens yet to be processed
 - Output: (abstract syntax tree, remaining token sequence

```
Sketch of recursive descent parser (functional version)
E(\langle var, rem \rangle) = (\langle var \rangle, rem)
E(\langle \text{num}, rem \rangle) = (\langle \text{num} \rangle, rem)
E(\langle lparen, lam, var, rem_0 \rangle) = let body, rem_1 = E(rem_0) in
                                             if rem_1 = \langle rparen, rem_2 \rangle then (\langle lam \ bvar: \langle var \rangle \ body: body, rem_2 \rangle)
                                                 else "error"
E(\langle lparen, app, rem_0 \rangle) = let fun, rem_1 = E(rem_0) in
                                  let arg, rem_2 = E(rem_1) in
                                      if rem_2 = \langle rparen, rem_3 \rangle then (\langle app fun: fun arg: arg \rangle, rem_3)
                                         else "error"
. . .
Parse(input) = let \ ast, rem = E(input) \ in
                          if rem = \langle \rangle then ast else "error"
```

The Evaluator



- The evaluator implements the $eval_{cek}$ function via the \mapsto_{cek}^* relation.
 - Input: abstract syntax tree of a closed ISWIM expression M.
 - Output: $eval_{cek}(M)$ together with a trace of the steps taken by the CEK machine.

The CEK machine

- Start state: ⟨⟨M, ∅⟩, mt⟩
 - Where M is the ast for a closed expression M, forming a closure with the empty environment; the initial continuation is the empty continuation.
- Reduction steps
 - In each reduction step, the machine transforms the machine state (a pair $\langle closure, continuation \rangle$) using the relation \mapsto_{cek} (details below).
- If the CEK machine halts, it should either halt in a state consisting of a value closure and the empty continuation, or report that it got stuck.
- Note: in OO implementation
 - The pattern matching on the CEK and execution of the transition step can be implemented by defining transition methods for the different ISWIM term subclasses (including a Value term subclass)

CEK Machine



CEK Continuations:

```
\mu = \text{mt}  // \text{"empty"}

|\langle sC_v; \mu \rangle

sC_v =   |\text{fun } v  |\text{arg } c  |\text{arg } 11 o  |\text{arg } 12 o c  |\text{arg } 22 o c
```

κ: (kappa) continuation

c: a closure

v: a value closure

o: operation

- Interpret the sC_v as a data type or an abstract class with concrete subclasses
 - Similar to ASTs
- View continuations as stacks of simple contexts
 - Including the empty stack
 - E.g. implemented as a linked list
- Environments
 - Stack of (variable name, closure) pairs
 - To look up a variable, search from top of stack

CEK Machine – State transition relation



$$\left\langle \left\langle (\operatorname{app} M N), e \right\rangle, \varkappa \right\rangle$$

$$\mapsto_{cek}$$

$$\left\langle \left\langle M, e \right\rangle, \left\langle \operatorname{arg} \left\langle N, e \right\rangle; \varkappa \right\rangle \right\rangle$$

[cek1]

$$\langle \langle X, e \rangle, \varkappa \rangle$$
 \mapsto_{cek} where $e(X) = c$ [cek7]
 $\langle c, \varkappa \rangle$

$$\langle \langle V, e \rangle, \langle \arg \langle N, e' \rangle; \varkappa \rangle \rangle$$

 $\mapsto_{cek} V \text{ value, } V \neq X \text{ [cek4]}$
 $\langle \langle N, e' \rangle, \langle \text{fun } \langle V, e \rangle; \varkappa \rangle \rangle$

$$\langle \langle V, e \rangle, \langle \text{fun } \langle (\text{lam } X M), e' \rangle; \varkappa \rangle \rangle$$
 $\mapsto_{cek} V \text{ value, } V \neq X \text{ [cek3]}$
 $\langle \langle M, e'[X \mapsto \langle V, e \rangle] \rangle, \varkappa \rangle$

[ck1]-[ck4]-[ck3]: Stages of application reduction

$$eval_{cek}(M) = \begin{cases} b & \text{if } \langle \langle M, \emptyset \rangle, \text{mt} \rangle \mapsto_{cek}^* \langle \langle b, e \rangle, \text{mt} \rangle \\ \text{function} & \text{if } \langle \langle M, \emptyset \rangle, \text{mt} \rangle \mapsto_{cek}^* \langle \langle \lambda X. N, e \rangle, \text{mt} \rangle \end{cases}$$

CEK Machine – State transition relation (2)



[ck2]-[ck6]-[ck5]: Stages of operation reduction

Note: rules cek5 will need to check that $\delta(o\ V_1)$ or $\delta(o\ V_1\ V_2)$ is defined for the given arguments. If one of the arguments is a lambda expression, then δ will certainly be undefined. In our case, all primitive operations are total, but for another choice of operators that may not be the case.

Test Outputs



- For each test ISWIM closed term, submit a listing of
 - Sequence of scanner tokens
 - Sequence of parser tokens
 - Pretty-printing of abstract syntax tree
 - The answer produced by the Evaluator
 - The sequence of the labels of the rules applied to compute the answer

Sample Input and Outputs



- Input: $\left(\text{app }\left(\text{lam }x1\left(\text{lam }x2\left(+x1\ x2\right)\right)\right)\ 42\right)$ (sub1 42)
- Scanner output:

Screener output:

```
(app
(app (lam var:"x1" (lam var:"x2" (op2:"+" var:"x1" var:"x2"))
num:"42":42)
(op1:"sub1" num:"42":42))
```

Parser/Tree builder output:

