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” → op1:”add1”
ID: “sub1” → op1:”sub1”
ID: “iszero” → op1:”iszero”
ID: “…” → var:”…”
NUM: “…” → num:”…”
OP: ”+” → op2:”+”
OP: “-” → op2:”-”
OP:” “*” → op2:”*”
OP: “^” → 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:

$$
\begin{aligned}
E = &\ \mathrm{var} \\
&|\ (\mathrm{lambda\ var}\ E) \\
&|\ (\mathrm{app}\ E\ E) \\
&|\ (\mathrm{op1}\ E) \\
&|\ (\mathrm{op2}\ E\ E) \\
&|\ \mathrm{num}
\end{aligned}
$$

E : (only) nonterminal

$$
\begin{aligned}
T = &\ \langle \mathrm{var} \rangle \\
&|\ \langle \mathrm{lambda}\ bvar\!: \mathrm{var}\ body\!: T \rangle \\
&|\ \langle \mathrm{app}\ fun\!: T\ arg\!: T \rangle \\
&|\ \langle \mathrm{op1}\ arg11\!: T \rangle \\
&|\ \langle \mathrm{op2}\ arg12\!: T\ arg22\!: T \rangle \\
&|\ \langle \mathrm{num} \rangle
\end{aligned}
$$

T : type/class of AST

Value terms

$$
\begin{aligned}
V = &\ \langle \mathrm{lam\ var}\ E \rangle \\
&|\ \langle \mathrm{num}\ b \rangle
\end{aligned}
$$

- 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)

$$E(\langle \text{var}, rem \rangle) = (\langle \text{var} \rangle, rem)$$
$$E(\langle \text{num}, rem \rangle) = (\langle \text{num} \rangle, rem)$$
$$\begin{aligned} E(\langle \text{lparen}, \text{lam}, \text{var}, rem_0 \rangle) = {} & \textbf{let } body, rem_1 = E(rem_0) \textbf{ in} \\ & \quad \textbf{if } rem_1 = \langle \text{rparen}, rem_2 \rangle \textbf{ then } (\langle \text{lam} \;\; bvar{:}\langle \text{var} \rangle \;\; body{:}body \rangle, rem_2) \\ & \qquad \textbf{else } "error" \end{aligned}$$
$$\begin{aligned} E(\langle \text{lparen}, \text{app}, rem_0 \rangle) = {} & \textbf{let } fun, rem_1 = E(rem_0) \textbf{ in} \\ & \textbf{let } arg, rem_2 = E(rem_1) \textbf{ in} \\ & \quad \textbf{if } rem_2 = \langle \text{rparen}, rem_3 \rangle \textbf{ then } (\langle \text{app} \;\; fun{:}fun \;\; arg{:}arg \rangle, rem_3) \\ & \qquad \textbf{else } "error" \end{aligned}$$
$$\ldots$$
$$\begin{aligned} Parse(input) = {} & \textbf{let } ast, rem = E(input) \textbf{ in} \\ & \textbf{if } rem = \langle\, \rangle \textbf{ then } ast \textbf{ else } "error" \end{aligned}$$

The Evaluator

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

- The CEK machine
  - Start state: $\langle\langle M, \emptyset\rangle, \mathrm{mt}\rangle$
    - 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 $\longmapsto_{cek}$ (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:

$\varkappa \quad = \text{mt}$ // "empty"

$\quad | \langle sC_v ; \varkappa \rangle$

$sC_v =$

$\quad | \text{ fun } v$

$\quad | \text{ arg } c$

$\quad | \text{ arg 11 } o$

$\quad | \text{ arg12 } o \; c$

$\quad | \text{ arg22 } o \; c$

$\varkappa$: (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

$$\left\langle \langle \langle \text{app}\ M\ N \rangle, e \rangle\ , \varkappa \right\rangle$$
$$\longmapsto_{cek}$$
$$\langle \langle M, e \rangle\ , \langle \text{arg}\ \langle N, e \rangle ; \varkappa \rangle \rangle$$

[cek1]

$$\langle \langle X, e \rangle\ , \varkappa \rangle$$
$$\longmapsto_{cek} \qquad \text{where } e(X) = \mathrm{c}$$
$$\langle c, \varkappa \rangle$$

[cek7]

$$\langle \langle V, e \rangle , \langle \text{arg}\ \langle N, e' \rangle ; \varkappa \rangle \rangle$$
$$\longmapsto_{cek} \qquad V \text{ value},\ V \neq X$$
$$\langle \langle N, e' \rangle , \langle \text{fun}\ \langle V, e \rangle ; \varkappa \rangle \rangle$$

[cek4]

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

[cek3]

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

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

CEK Machine – State transition relation (2)

$$\langle\langle\langle \mathrm{op1{:}\,o}\ M\rangle, e\rangle, \varkappa\rangle \longmapsto_{cek} \langle\langle M, e\rangle, \langle \mathrm{arg11}\ o; \varkappa\rangle\rangle$$ [cek2a]

$$\langle\langle\langle \mathrm{op2{:}\,o}\ M\ N\rangle, e\rangle, \varkappa\rangle \longmapsto_{cek} \langle\langle M, e\rangle, \langle \mathrm{arg12}\ o\ \ \langle N, e\rangle; \varkappa\rangle\rangle$$ [cek2b]

$$\langle\langle b, e\rangle, \langle \mathrm{arg11}\ o; \varkappa\rangle\rangle \longmapsto_{cek} \langle\langle V, \emptyset\rangle, \varkappa\rangle \quad \text{where } V = \delta(o\ b)$$ [cek5a]

$$\langle\langle V, e\rangle, \langle \mathrm{arg12}\ o\ \langle N, e'\rangle; \varkappa\rangle\rangle \longmapsto_{cek} \langle\langle N, e'\rangle, \langle \mathrm{arg22}\ o\ \langle V, e\rangle; \varkappa\rangle\rangle \quad V \neq X$$ [cek6b]

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

$$\langle\langle b, e\rangle, \langle \mathrm{arg22}\ o\ \langle b_1, e'\rangle; \varkappa\rangle\rangle \longmapsto_{cek} \langle\langle V, \emptyset\rangle, \varkappa\rangle \quad \text{where } V = \delta(o\ b_1\ b)$$ [cek5b]

$$\langle\langle V_1, e\rangle, \langle \mathrm{arg11}\ o; \varkappa\rangle\rangle \longmapsto_{cek} \langle\langle V, \emptyset\rangle, \varkappa\rangle \quad \text{where } V = \delta(o\ V_1)$$ [cek5a’]

$$\langle\langle V_2, e\rangle, \langle \mathrm{arg22}\ o\ \langle V_1, e'\rangle; \varkappa\rangle\rangle \longmapsto_{cek} \langle\langle V, \emptyset\rangle, \varkappa\rangle \quad \text{where } V = \delta(o\ V_1\ V_2)$$ [cek5b’]

- **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**: $\left(\text{app}\ \left(\text{app}\ \left(\text{lam}\ x1\ (\text{lam}\ x2\ (+x1\ x2))\right)\ 42\right)\ (\text{sub1}\ 42)\right)$
- **Scanner output**:
  (ID:"app" WS
  (ID"app" WS
  (ID:"lam" WS ID:"x1" WS (ID:"lam" WS ID:"x2" WS (OP:"+" WS ID:"x1" WS ID:"x2")) WS
  NUM:"42") WS
  (ID:"sub1" WS NUM:"42"))
- **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**:
  <app
  fun: <app
  fun: <lam *bvar*: var:"x1"
  body: <lam *bvar*: var:"x2"
  body: <op2:"+" arg12: var:"x1" arg22: "var:x2" >
  >
  >
  arg: num:"42":42
  >
  arg: < op1:"sub1" *arg11*: num:"42":42 >
  >