Basic Exercises on CafeOBJ

- basic commands, parsing, debugging, tracing, modeling, specification -

CafeOBJ Team of JAIST

Topics

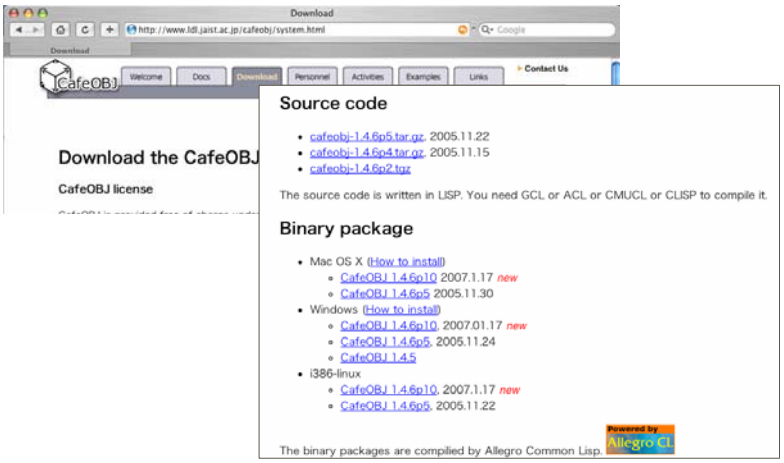
- **Basic commands of CafeOBJ system**
- **Inputting specifications, parsing, debugging, tracing**
- **A simple example of modeling and specification: STACK**

Basic Top level commands

How to get CafeOBJ System

- Source code (in CommonLisp) and binary code for UNIX/Linus, MacOS, and Windows are available
- URL of CafeOBJ official home page : http://www.ldl.jaist.ac.jp/cafeobj/

Several Kinds of CafeOBJ Systems



Starting System

- **CafeOBJ system is invoked by typing “cafeobj” (or clicking CafeOBJ.bat or CafeOBJ.exe files), and the system waits for your input with a “prompt”.**

```
d:/localSystems/cafeobjSystem> cafeobj
% -- loading standard prelude
; Loading c:¥localPrograms¥cafeobj148p2-071109¥system148p2¥prelude¥std.bin

          -- CafeOBJ system Version 1.4.8(PigNose0.99,p2) --
                   built: 2007 Nov 8 Thu 0:16:58 GMT
                         prelude file: std.bin
                                ***
                      2007 Nov 30 Fri 4:22:48 GMT
                           Type ? for help
                                ***
                  -- Containing PigNose Extensions
                                ---
         built on International Allegro CL Enterprise Edition
                   8.1 [Windows] (Nov 8, 2007 9:16)
processing input : d:/home/.cafeobj
cafeOBJ>
```

.cafeobj file is loaded

“Prompt” from CafeOBJ system shows the current module

Quitting System

- By typing “quite” or “q” (or typing control-D; this depends on the OS you are using), you can quite from the CafeOBJ system

```
CafeOBJ> quit
[Leaving CafeOBJ]
%
```


Inputting moudles or selecting modules

- In the top level of CafeOBJ system, the commands for inputting modules and selecting modules are available.

```
CafeOBJ> mod TEST { [ Elt ] }                    input
  -- defining module TEST_* done.                output
CafeOBJ>
```

After a inputting module, the module is available by typing “selecting <the module name>”

```
CafeOBJ> select TEST

TEST>
```

After selecting the module “ModName”, the prompt is changed to “ModName>” .

System Error - error should be eliminated – - warning needs not be eliminated, but recommended to be eliminated -

- CafeOBJ system report an error as follows:

```
CafeOBJ> mod ERROR }
[Error]: was expecting the symbol `{' not `}' .
CafeOBJ>
```

- "[Error]…" reports a serious error like syntax error in inputted CafeOBJ code. CafeOBJ system may go down to CHAOS level for some special errors.

```
CafeOBJ> ^C
Error: Received signal number 2 …
[1c] CHAOS(1):
```

- From "CHAOS" level, CafeOBJ system will be recovered by typing ":q" in almost all cases.

```
[1c] CHAOS(1): :q
CafeOBJ>
```


Inputting files

Commands.mod

- It is always recommended that CafeOBJ codes is prepared in some file and the file is inputted into CafeOBJ system by typing "in <fileName>" or "input <fileName>". The file extension of CafeOBJ file is ".mod" .

  example: inputting "test.mod" file

```
% more test.mod
mod TEST { [Elt] }
select TEST
%
```

contents of the file "test.mod"

```
CafeOBJ> in test.mod
processing input : /…/test.mod
-- defining module TEST.* done.
TEST>
```

?, show, describe, show ? commands

- By typing "?", you can see the list of commands which are available at the level.
- "show" command shows varieties of information
  - "show <module name>", "show sorts", "show ops"
  - "show ?" gives you a list of show commands available
- "show" can be shortened into "sh"

```
CafeOBJ> ?
-- CafeOBJ top level commands :
-- Top level definitional forms include `module'(object, theory),
-- `view', and `make'
  ?                 print out this help
  quit -or-
  q                 exit from CafeOBJ interpreter
  select <Modexp>   set the <Modexp> current
  show -or-
  describe          print various info., for further help, type `show ?'
  ...
```


set, set ? and show switches command

set command set switches of CafeOBJ system. By changing switches you can customize CafeOBJ system get different behaviors of the system.

```
CafeOBJ> set auto context on
CafeOBJ> mod TEST { [ Elt] }
-- defining module TEST._* done.
TEST>
```

making system select the last entered module automatically

```
TEST> set ?
TEST> ...
```

Shows all **set** commands

```
TEST> show switches
TEST> ...
```

Shows all switches

Comments

Comments.mod

A line beginning with "- -" (or "") is ignored, and**
A line beginning with "- ->" (or ">") is echoed back.**

```
CafeOBJ> -- this is a comment
CafeOBJ>
```

```
CafeOBJ> ** this is a comment
CafeOBJ>
```

```
CafeOBJ> --> this is a comment
--> this is a comment
CafeOBJ>
```

```
CafeOBJ> **> this is a comment
**> this is a comment
CafeOBJ>
```

It is very important to write as much appropriate comments as possible for explaining specifications and verifications/proofs.

Parsing error, debugging

Parse command

nat+.mod

```
...> select NAT+
NAT+> parse s 0 + 0 .

((s 0) + 0):Nat
NAT+> set verbose on
NAT+> parse s 0 + 0 .

((s 0) + 0):Nat
     _+_:Nat
    /        ¥
s_:NzNat  0:Zero
    |
 0:Zero

NAT+>
```


Parsing errors

- CafeOBJ system issues an error message if an equation can not be parsed

```
CafeOBJ> mod! NAT+ {...
  eq (s M) + N = s(M, N) .
}
-- defining module! NAT+
[Error] bad axiom (ignored):
...
No possible parse for RHS
-- failed to evaluate the form:
axiom declaration(equation):
 lhs = (( s M ) + N)
 rhs = s ( M , N )
CafeOBJ>
```

a paring error message

Debugging specifications

nat+err.mod

Loading the codes above the point of error and using show and/or parse command to fix the error. “open” command is also effective in debugging modules.

```
NAT+> mod! NAT+DEBUG {
  [Zero NzNat < Nat]
  op 0 : -> Zero
  op s_ : Nat -> NzNat
  op _+_ : Nat Nat -> Nat

  vars M N : Nat
  eq    0  + N = N .
}

-- defining module!
   NAT+DEBUG....._.* done.
NAT+> select NAT+DEBUG
NAT+DEBUG>
```

```
NAT+DEBUG> parse s(M, N) .
[Error] no successfull parse
  (s ( M , N ))
("s" "(" "M" "," "N" ")") : SyntaxErr
NAT+DEBUG> parse M .
M : Nat
NAT+DEBUG> parse N .
N : Nat
NAT+DEBUG> parse s(M + N) .
(s (M + N)) : NzNat
NAT+DEBUG>
```


Trace commands

- **When some reduction in a proof score does not returns “`true`”, the trace commands help us to detect the reason.**

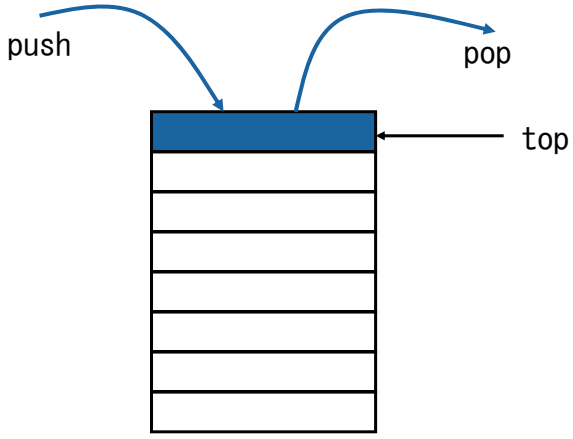
```
NAT+> set trace whole on
NAT+> red +(s(0), s(0)) .
-- reduce in NAT+ : +(s(0),s(0))
[1]: +(s(0),s(0))
---> s(+(0,s(0)))
[2]: s(+(0,s(0)))
---> s(s(0))
s(s(0)) : NzNat
(0.000 sec ...)
NAT+> set trace whole off
```

```
NAT+> set trace on
NAT+> red +(s(0), s(0)) .
-- reduce in NAT+ : +(s(0),s(0))
1>[1] rule: eq +(s(M:Nat),N:Nat) = s(+(M,N))
    { M:Nat |-> 0, N:Nat |-> s(0) }
1<[1] +(s(0),s(0)) --> s(+(0,s(0)))
1>[2] rule: eq +(0,N:Nat) = N
    { N:Nat |-> s(0) }
1<[2] +(0,s(0)) --> s(0)
s(s(0)) : NzNat
(0.000 sec ...)
NAT+> set trace off
```

An Example of Modeling/Specifying in CafeOBJ

1. By understanding a problem to be modeled/specified, determine several sorts of objects (entities, data, agents, states) and operations (functions, actions, events) over them for describing the problem
2. Define the meanings/functions of the operations by declaring equations over expressions composed of the operations

Modeling STACK in CafeOBJ



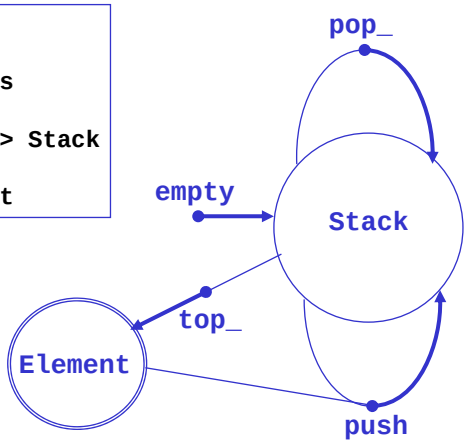
Signature of STACK (1)

sorts of objects Element Stack
operations empty : returns empty stack without argument
push : push an element to a stack and returns a new stack
top : get the top element of a stack and returns the element
pop : removes the top element of a stack and returns the stack

```
-- sorts
[ Element Stack ]
-- operations or operators
op empty : -> Stack
op push : Element Stack -> Stack
op pop_ : Stack -> Stack
op top_ : Stack -> Element
```


Signature of STACK (2)

```
-- sorts
[ Element Stack ]
-- operations or operators
op empty : -> Stack
op push : Element Stack -> Stack
op pop_ : Stack -> Stack
op top_ : Stack -> Element
```



Terms/expressions generated by the signature

```
Element =  { e1, e2, e3, … }
        ∪ { top S | S ∈ Stack }
```

```
Stack = { empty }
      ∪{ push(E,S) |  E ∈ Element ∧  S ∈ Stack }
      ∪{ pop S | S ∈ Stack }
```

Examples:

```
top push(e1,empty) ∈ Element
pop push(e1,empty) ∈ Stack
top pop push(e1,empty) ∈ Element
pop pop push(e1,empty) ∈ Stack
top push(e2, pop pop push(e1,empty)) ∈ Element
```

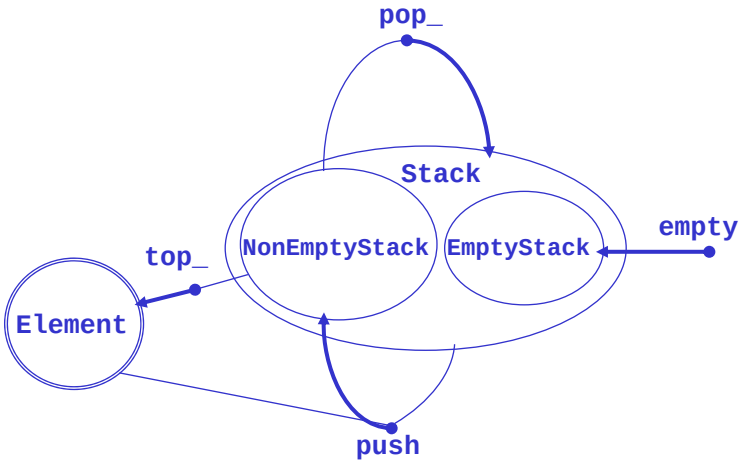

Equations

```
-- operations or operators
op empty : -> Stack
op push : Element Stack -> Stack
op pop_ : Stack -> Stack
op top_ : Stack -> Element
-- equations
eq top push(E:Element,S:Stack) = E .
eq pop push(E:Element,S:Stack) = S .
```

Examples:

```
top push(e1,empty) = e1
pop push(e1,empty) = empty
top pop push(e1,empty) = top empty
pop pop push(e1,empty) = pop empty
top push(e2, pop pop push(e1,empty)) = e2
 this is not the intended behavior
```

Revised signature of STACK (1)



Revised signature of STACK (2)

```
-- sorts and subsorts
[ Element, EmptyStack NonEmptyStack < Stack ]
-- operators
op empty : -> EmptyStack
op push : Element Stack -> NonEmptyStack
op pop_ : NonEmptyStack -> Stack
          -- only applicable to NonEmptyStack
op top_ : NonEmptyStack -> Element
          -- only applicable to NonEmptyStack
```

Terms generated by revised signature

```
Element =  { e1, e2, e3, … }
        ∪ { top S | S ∈ NonEmptyStack }
```

```
EmptyStack = { empty }
NonEmptyStack = { push(E,S) | E ∈ Element ∧  S ∈ Stack }
Stack = { pop S | S ∈ NonEmptyStack }
             ∪ EmptyStack ∪ NonEmptyStack
```

Examples:

```
top push(e1,empty) ∈ Element
push(e1,empty) ∈ NonEmptyStack
pop push(e1,empty) ∈ Stack
top pop push(e1,empty) : not well formed!
pop pop push(e1,empty) : not well formed!
top push(e2, pop pop push(e1,empty))
                        : not well formed!
```


Revised Stack -- equations --

```
[ EmptyStack NonEmptyStack < Stack ]
op empty : -> EmptyStack {constr}
op push : Element Stack -> NonEmptyStack {constr}
op pop_ : NonEmptyStack -> Stack
op top_ : NonEmptyStack -> Element
-- equations
eq top push (E:Element, S:Stack) = E .
eq pop push (E:Element, S:Stack) = S .
```

Examples:

```
top push(e1,empty) = e1
pop push(e1,empty) = empty
top pop push(e1,empty) = (top empty : ?Element)
pop pop push(e1,empty) = (pop empty : ?Stack)
top push(e2, pop pop push(e1,empty)) =
  (top push(e2,pop empty) : ?Element)
  Return value is of Error sort!
```

Revised specification of Stack in CafeOBJ

stack.mod

```
Specification of Stack
mod* ELEMENT { [Element] }

mod! STACK (X :: ELEMENT) {
  [EmptyStack NonEmptyStack < Stack ]
  op empty : -> EmptyStack {constr}
  op push : Element Stack -> NonEmptyStack {constr}
  op pop_ : NonEmptyStack -> Stack
               -- only applicable to NonEmptyStack
  op top_ : NonEmptyStack -> Element
               -- only applicable to NonEmptyStack

  eq top push (E:Element, S:Stack) = E .
  eq pop push (E:Element, S:Stack) = S . }
```