INF4820

Part 1: Non-Deterministic FSAs
Part 2: Lisp variables, binding and scope. And macros.

Erik Velldal

University of Oslo

Sep. 15, 2009

Topics for Today

Wrapping up FSAs

- Machines as transition tables
- State-space search and NFSA recognition
- Depth-first vs breadth-first search

Topics for Today

Wrapping up FSAs

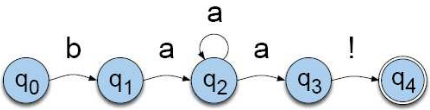
- Machines as transition tables
- State-space search and NFSA recognition
- Depth-first vs breadth-first search

Common Lisp variables

- Scope, binding and shadowing
- Lexical vs dynamic scope. (Local vs global variables)
- `let`/`let*`
- Closures
- Assignment
- Macros (maybe)

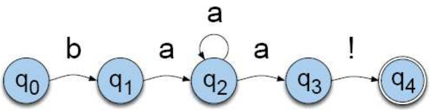
Transition Matrix for an NFSA

An NFSA for the sheep language (“baa!”, “baaa!”, “baaaa!”, “baaaaa!”,...).



Transition Matrix for an NFSA

An NFSA for the sheep language (“baa!”, “baaa!”, “baaaa!”, “baaaaa!”,...).



Alternatively, the machine can be represented as a matrix or a table:

Transition Table

	Input		
State	!	a	b
0	∅	∅	1
1	∅	2	∅
2	∅	2,3	∅
3	4	∅	∅

Non-Deterministic Recognition: State-Space Search

- With NFSAs there might exist several paths through the machine for a given string.
- Successful recognition of a string means that there exists at least one path through the machine that leads to an accept state.
- Failure occurs when none of the possible paths leads to an accept state.
- We'll walk through an example of a simple recognition task (based on depth-first search and backtracking).

```
function ND-RECOGNIZE(tape, machine) returns accept or reject

 agenda ← {(Initial state of machine, beginning of tape)}
 current-search-state ← NEXT(agenda)
 loop
   if ACCEPT-STATE?(current-search-state) returns true then
     return accept
   else
     agenda ← agenda ∪ GENERATE-NEW-STATES(current-search-state)
   if agenda is empty then
     return reject
   else
     current-search-state ← NEXT(agenda)
 end

function GENERATE-NEW-STATES(current-state) returns a set of search-states

 current-node ← the node the current search-state is in
 index ← the point on the tape the current search-state is looking at
 return a list of search states from transition table as follows:
   (transition-table[current-node,ϵ], index)
   ∪
   (transition-table[current-node, tape[index]], index + 1)

function ACCEPT-STATE?(search-state) returns true or false

 current-node ← the node search-state is in
 index ← the point on the tape search-state is looking at
 if index is at the end of the tape and current-node is an accept state of machine
 then
   return true
 else
   return false
```

Some Terminology (from Seibel 2005)

Binding form

A form introducing a variable such as a function definition or a `let` expression.

Scope

The area of the program where the variable name can be used to refer to the variable's binding. Lexically scoped variables can be referred to only by code that is textually within their binding form.

Shadowing

When binding forms are *nested* and introduce variables of the same name, the innermost binding "shadows" the outer bindings.

Bindings, Scope and Shadowing

```
(setq foo 24)
(let ((foo 42)
      (bar foo))
  (print bar))
```

⇝ 24

Bindings, Scope and Shadowing

```
(setq foo 24)
(let ((foo 42)
      (bar foo))
  (print bar))
```

⇝ 24

```
(let* ((foo 42)
       (bar foo))
  (print bar))
```

⇝ 42

Bindings, Scope and Shadowing

```
(setq foo 24)
(let ((foo 42)
      (bar foo))
  (print bar))
```

⇝ 24

```
(let* ((foo 42)
       (bar foo))
  (print bar))
```

⇝ 42

$\Longleftrightarrow$

```
(let ((foo 42))
  (let ((bar foo))
    (print bar)))
```

⇝ 42

Bindings, Scope and Shadowing (cont'd)

```
(defun foo ()
  (let ((foo 42)
        (values))
    (push foo values)
    (flet ((foo ()
             (push foo values)))
      (foo)
      (let ((foo 21))
        (push foo values)
        (foo)))
    values))
```

- What is returned by a call to `(foo)`?

Bindings, Scope and Shadowing (cont'd)

```
(defun foo ()
  (let ((foo 42)
        (values))
    (push foo values)
    (flet ((foo ()
             (push foo values)))
      (foo)
      (let ((foo 21))
        (push foo values)
        (foo)))
    values))
```

- What is returned by a call to `(foo)`?
- → `(42 21 42 42)`

Bindings, Scope and Shadowing (cont'd)

```
(defun foo ()
  (let ((foo 42)
        (values))
    (push foo values)
    (flet ((foo ()
             (push foo values)))
      (foo)
      (let ((foo 21))
        (push foo values)
        (foo)))
    values))
```

- What is returned by a call to `(foo)`?
- → (42 21 42 42)
- The inner `flet`-defined `foo` is actually also an example of a closure: free variables that are "closed over" by a function object.

Closures

- So far we've focused on the idea that local variables in Lisp are based on lexical scoping.

Closures

- So far we've focused on the idea that local variables in Lisp are based on lexical scoping.
- But, in Lisp the concept of *closures* still makes possible the use of variable references in functions that are called in code outside the scope of the binding form that introduced the variables.

Closures

- So far we've focused on the idea that local variables in Lisp are based on lexical scoping.
- But, in Lisp the concept of *closures* still makes possible the use of variable references in functions that are called in code outside the scope of the binding form that introduced the variables.
- Confused yet?

Closures. An Example.

```
(let ((c 0))
  (defun counter (action &optional (n 1))
    (case action
      (add (incf c n))
      (sub (decf c n))
      (print (format t "Current count = ~d.~%" c)))))
```

- When a function is defined in a non-null lexical environment, we say that it "closes over" and captures the bindings of its free variables.

Closures. An Example.

```
(let ((c 0))
  (defun counter (action &optional (n 1))
    (case action
      (add (incf c n))
      (sub (decf c n))
      (print (format t "Current count = ~d.~%" c)))))
```

- When a function is defined in a non-null lexical environment, we say that it "closes over" and captures the bindings of its free variables.

(counter 'sub 11) $\longrightarrow$ -11

(counter 'add) $\longrightarrow$ -10

(counter 'print) $\rightsquigarrow$ Current count = -10.

Closures. Another Example.

- An example of a function that returns an anonymous function, implementing (very simple) memoization through lexical closure:

```
(defun memoize (fn)
  (let ((cache (make-hash-table :test #'equal)))
    #'(lambda (&rest args)
        (multiple-value-bind
            (val stored-p) (gethash args cache)
          (if stored-p
              val
            (setf (gethash args cache)
              (apply fn args)))))))
```

Closures. Another Example.

- An example of a function that returns an anonymous function, implementing (very simple) memoization through lexical closure:

```
(defun memoize (fn)
  (let ((cache (make-hash-table :test #'equal)))
    #'(lambda (&rest args)
        (multiple-value-bind
            (val stored-p) (gethash args cache)
          (if stored-p
              val
            (setf (gethash args cache)
              (apply fn args)))))))

(setf mem-ccc (memoize #'complex-costly-computation))
(funcall mem-ccc arguments)
```

Dynamic Variables

AKA Special Variables

- Lisp's global variables have dynamic scope.
- `defparameter` / `defvar`
- Global variables are handy for environment variables and parameters that we don't want to pass around (such as standard I/O streams).

Dynamic Variables

AKA Special Variables

- Lisp's global variables have dynamic scope.
- `defparameter` / `defvar`
- Global variables are handy for environment variables and parameters that we don't want to pass around (such as standard I/O streams).

Dynamic how?

- So far we've only looked at lexically scoped local variables...
- When binding a dynamic variable in e.g. a `let`, the new binding can be seen by *all code invoked during the execution of form* (not just within the textual bounds).

Dynamic Variables (cont'd)

(Example from an interactive ACL session in Emacs)

```
CL-USER(32): (defparameter *base* (exp 1))
*BASE*
CL-USER(33): *base*
2.7182817
CL-USER(34): (defun my-exp (x)
               (expt *base* x))
MY-EXP
CL-USER(35): (my-exp 4)
54.598145
CL-USER(36): (let ((*base* 2))
               (my-exp 4))
16
CL-USER(37):
```

Assignment

- `setq` and `setf`
  - `(setf goo "abc")`

Assignment

- `setq` and `setf`
  - `(setf goo "abc")`
- Through so called modify macros, specialized variants of `setf` are defined for access to places in lists, arrays, hash-tables, structs, etc.

```
(setf (aref goo 1) #\n)
(incf bar 5)
(incf (gethash 'key table 0) 1)
```

Assignment

- `setq` and `setf`
  - `(setf goo "abc")`
- Through so called modify macros, specialized variants of `setf` are defined for access to places in lists, arrays, hash-tables, structs, etc.

```
(setf (aref goo 1) #\n)
(incf bar 5)
(incf (gethash 'key table 0) 1)
```

- When thinking about assignment, don't confuse named variables and other places that can hold values...
- Or, in other words; binding is not always the same as assignment.

  (Cf. First Assignment Part B, exercise 2b on functional variants of `push` / `pop`).

Macros

- With `defmacro` we can write Lisp code that generates Lisp code.
- Macro expansion time vs runtime

Important operators when writing macros

- `'`: Quota suppresses evaluation.
- `` ` ``: Backquote also suppresses evaluation, but..
- `,`: A comma inside a backquoted form means the following subform should be evaluated.
- `@`: “Explodes” lists.

Macros. A Rather Silly Example

```
CL-USER(53): (defmacro dolist-reverse ((e list) &rest body)
               `(let ((r (reverse ,list)))
                  (dolist (,e r)
                    ,@body)))
DOLIST-REVERSE
```

Macros. A Rather Silly Example

```
CL-USER(53): (defmacro dolist-reverse ((e list) &rest body)
               `(let ((r (reverse ,list)))
                  (dolist (,e r)
                    ,@body)))
DOLIST-REVERSE

CL-USER(54): (dolist-reverse (x (list 1 2 3))
              (print x))

3
2
1
NIL
```

Macros. A Rather Silly Example

```
CL-USER(53): (defmacro dolist-reverse ((e list) &rest body)
               `(let ((r (reverse ,list)))
                  (dolist (,e r)
                    ,@body)))
DOLIST-REVERSE

CL-USER(54): (dolist-reverse (x (list 1 2 3))
              (print x))

3
2
1
NIL
```

All according to plan. Or..?

Unintended variable capture can be a pitfall...

```
CL-USER(55): (let ((r 42))
                (dolist-reverse (x (list 1 2 3))
                  (print (list x r))))
(3 (3 2 1))
(2 (3 2 1))
(1 (3 2 1))
NIL
CL-USER(56):
```

Not quite what we wanted...

gensym to the rescue!

```
CL-USER(56): (defmacro dolist-reverse ((e list) &rest body)
               (let ((r (gensym)))
                 `(let ((,r (reverse ,list)))
                    (dolist (,e ,r)
                      ,@body))))
DOLIST-REVERSE
CL-USER(57): (let ((r 42))
               (dolist-reverse (x (list 1 2 3))
                 (print (list x r))))

(3 42)
(2 42)
(1 42)
NIL
```

All according to plan. (No, really!)

A more useful example, do-set-permutations

Macro definitions can even be recursive!

```
(defmacro do-set-permutations (lists &body body)
  (if (null (cdr lists))
      `(dolist ,(first lists)
         ,@body)
    `(dolist ,(first lists)
       (do-set-permutations ,(cdr lists)
         ,@body))))
```

Usig our new macro, do-set-permutations

```
CL-USER(190): (do-set-permutations ((x '(a b))
                                    (y '(1 2 3))
                                    (z '(foo)))
                (print (list x y z)))

(A 1 FOO)
(A 2 FOO)
(A 3 FOO)
(B 1 FOO)
(B 2 FOO)
(B 3 FOO)
NIL
```