Implementation of Regular Expression Recognizers

CS164
Lecture 6
Outline

• Testing for membership in a “regular” language.
• Specifying lexical structure using regular expressions. A FORMAL high-level approach.
• Could be automatically programmed from spec.

• Finite automata: a “machine” description
  - Deterministic Finite Automata (DFAs)
  - Non-deterministic Finite Automata (NFAs)
  - Implemented in software (but could be in hardware!)

• Implementation of regular expressions as programs
  RegExp => NFA => DFA => Tables or programs
Common Notational Extensions

- There are various extensions used in regular expression notation; this uses up more meta characters but we can generally manage it by escape/quotes when we need them...

- **Union:** \( A \mid B \equiv A + B \)

- **Optional:** \( A + \varepsilon \equiv A? \)

- **Sequence:** \( AB \equiv AB \)

- **Kleene Star:** \( A^* \equiv A^* \)

- **Parens used for grouping:** \( (A+B)C \equiv AC+BC \)

- **Range:** ‘a’+’b’+…+’z’ \( \equiv [a-z] \)

- **Excluded range:**
  
  complement of \([a-z]\) \( \equiv [^a-z] \)
Examples of REs

• $R := (0+1)*ab*a$
• $S := [a-z]([a-z]+[0-9])*$

• Described in English:
  • an element of $R$ starts optionally with a string of any combination of the digits 0 or 1 of any length, followed by exactly one $a$ then optionally some number of $b$ characters and then an $a$.
• What is $S$?
Let’s get real

- Do we want yet another language to parse, the language of regular expressions, where $A|BC$ has to be disambiguated? {Is this $(A|B)C$ or $A|(BC)$? Is $ab^*$ the same as $(ab)^*$ or $a(b^*)$?}
- What a mathematician can complicate with notation, we can make more easily constructive by using computer notation.
- What notation is that??
Notation extensions

- We can use lisp...
- Union: \( A \mid B \) \( \equiv \) (union \( A \) \( B \))
- Option: \( A + \varepsilon \) \( \equiv \) (union \( A \) \( \varepsilon \))
- Range: ‘a’+‘b’+...+‘z’ \( \equiv \) alphachar
- Sequence: \( A \ B \) \( \equiv \) (seq \( A \) \( B \))
- Kleene Star: \( A^* \) \( \equiv \) (star \( A \))
- Excluded range:
  complement of \( A \) \( \equiv \) (not \( A \))
Notation extensions

Examples in lisp

- $(0+1)^*(ab*a)$.
  - $(\text{seq } (\text{star } (\text{union } 0 1)) (\text{seq } a (\text{star } b) a))$
  - $(\text{seq } (\text{star } (\text{union } 0 1)) a (\text{star } b) a)$
- $[a-z]([a-z]+[0-9])^*$
  - $(\text{seq } \text{alphachar } (\text{star } (\text{union } \text{alphachar } \text{digitchar}))))$
Regular Expressions in Lexical Specification

• Last lecture: a specification for the predicate $s \in L(R)$
• But a yes/no answer is not enough!
• Instead: we want to partition the input into tokens.

• Tradition is to write an algorithm based on partitioning by regular expressions.
Regular Expressions => Lexical Spec. (1)

1. Select a set of tokens
   • Number, Keyword, Identifier, ...

2. Write a rexp for the lexemes of each token
   • Number = \texttt{digit+}
   • Keyword = ‘if’ + ‘else’ + ...
   • Identifier = \texttt{letter (letter + digit)*}
   • OpenPar = ‘(‘
   • ...

Prof. Fateman  CS 164  Lecture 6
3. Construct $R$, matching all lexemes for all tokens (and a pattern for everything else..)

$$R = \text{Keyword} + \text{Identifier} + \text{Number} + \ldots$$

$$= R_1 + R_2 + \ldots + R_n=rathole$$

Facts: If $s \in L(R)$ then $s$ is a lexeme
- Furthermore $s \in L(R_i)$ for some “$i$”
- This “$i$” determines the token that is reported
4. Let input be \( x_1 \ldots x_n \), a SEQUENCE of CHARS
   - \( x_1 \ldots x_n \) are individual characters
   - For \( 1 \leq k \leq n \) check
     \[ x_1 \ldots x_k \in L(R) \]?

5. It must be that
    \[ x_1 \ldots x_k \in L(R_j) \] for some \( j \), so it is a type-\( j \) token
   Remove \( x_1 \ldots x_k \) from input and go to (4)
How to Handle Spaces and Comments?

1. We could create a token **Whitespace**
   
   ```
   Whitespace = (' ' + '\n' + '\t')+
   ```
   
   - We could also add comments in there
   - An input "    \t\n   5555 " is transformed into
     Whitespace Integer Whitespace

2. Alternatively, Lexer skips spaces (preferred)
   
   - Modify step 5 from before as follows:
     It must be that $x_k \ldots x_i \in L(R_j)$ for some \( j \) such
     that $x_1 \ldots x_{k-1} \in L(Whitespace)$
   
   - Parser is not bothered with (extra) spaces
Ambiguities (1)

• There are ambiguities in the algorithm

• How much input is used? What if
  • $x_1...x_i \in L(R)$ and also
  • $x_1...x_k \in L(R)$ for $k>i$

  - One possible Rule: Pick the longest possible substring
  - The “maximal munch”
Ambiguities (2)

- Which token is used? What if
  - \( x_1 \ldots x_i \in L(R_j) \) and also
  - \( x_1 \ldots x_i \in L(R_k) \)
  - Another possible rule: use rule listed first (j if \( j < k \))

- Example:
  - \( R_1 = \text{Keyword} \) and \( R_2 = \text{Identifier} \)
  - “if” matches both.
  - Treats “if” as a keyword not an identifier (many languages just tell user: don’t use keyword as identifier.)
Error Handling

• What if
  No rule matches a prefix of input?
• Problem: Can’t just get stuck ...
• Solution:
  - Write a rule matching all “bad” strings
  - Put it last (remember, $R_n = \text{rathole\ldots}$)
• Lexer tools allow the writing of:
  $R = R_1 + \ldots + \text{Error}$
  - Token Error matches if nothing else matches
Summary

- Regular expressions provide a concise notation for string patterns
- Use in lexical analysis requires small extensions
  - To resolve ambiguities
  - To handle errors
- Good algorithms known (e.g. r.e. → lexer)
  - Require only single pass over the input
  - Few operations per character (table lookup)
Finite Automata

- Regular expressions = specification
- Finite automata = closer to implementation
- *(Singular: automaton. Plural: automata.)*
- A **finite automaton** or (D)FA is an abstraction consisting of
  - An input alphabet $\Sigma$
  - A set of states $S$
  - A start state $n$
  - A set of **accepting** states $F \subseteq S$
  - A set of transitions $\text{state}_1 \rightarrow^{\text{input}} \text{state}_2$
Finite Automata

- **Transition**
  
  \[ s_1 \rightarrow^a s_2 \]

- **Is read**

  In state \( s_1 \) on input \( a \) go to state \( s_2 \)

- **If end of input (or no transition possible)**
  - If in accepting state \( \Rightarrow \) accept
  - Otherwise \( \Rightarrow \) reject
Finite Automata State Graphs

- A state
- The start state
- An accepting state
- A transition
A Simple Example

- A finite automaton that accepts only “1"
Another Simple Example

• A finite automaton accepting any number of 1’s followed by a single 0
• Alphabet: {0,1}; as a RegExp: 1*0
And Another Example

- Alphabet \( \{0,1\} \)
- What language does this recognize?
And Another Example

- Alphabet still \{ 0, 1 \}

- The operation of the automaton is not completely defined by the input
  - On input “11” the automaton could be in either state
Epsilon Moves

• Another kind of transition: \(\varepsilon\)-moves

\[ A \xrightarrow{\varepsilon} B \]

• Machine can move from state \(A\) to state \(B\) without reading input. Which state is it really in?
Deterministic and Nondeterministic Automata

• Deterministic Finite Automata (DFA)
  - One transition per input per state
  - No ε-moves

• Nondeterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have ε-moves

• Either kind of finite automaton has finite memory
  - Need only to encode the current state(s)
Execution of Finite Automata

• A DFA can take only one path through the state graph
  - Completely determined by input
• One could think that NFAs can “choose”
  - Whether to make $\varepsilon$-moves
  - Which of multiple transitions for a single input to take

Actually, NFAs do not have free will. It would be more accurate to say an execution of an NFA marks “all” choices from a set of states to a new set of states.
Acceptance of NFAs

• An NFA can be “in multiple states”

• Input: 1 0 1

• Rule: NFA accepts if at least one of its current states is a final state
NFA vs. DFA (1)

- NFAs and DFAs have the same abstract power to recognize languages. Namely the same set of regular languages.

- DFAs are easier to implement naively as a program
- NFAs can always be converted to DFAs
NFA vs. DFA (2)

• For a given language the NFA can be simpler than the DFA

NFA

DFA

• DFA can be exponentially larger than NFA (n states in a NFA could require as many as $2^n$ states in a DFA)
Regular Expressions to Finite Automata

• High-level sketch

NFA

Regular expressions

Lexical Specification

DFA

Table-driven Implementation of DFA
Regular Expressions to NFA (1)

• For each kind of rexp, define an NFA
  - Notation: NFA for rexp M

```
  M
  ε
  a
```

• For ε

• For input a
Regular Expressions to NFA (2)

• For $AB$

• For $A + B$
Regular Expressions to NFA (3)

- For $A^*$
Example of RegExp -> NFA conversion

- Consider the regular expression\[(1+0)^*1\]
- The NFA is
NFA to DFA. The Trick

• Simulate the NFA
• Each state of DFA
  = a non-empty subset of states of the NFA
• Start state
  = the set of NFA states reachable through $\varepsilon$-moves from NFA start state
• Add a transition $S \xrightarrow{a} S'$ to DFA iff
  - $S'$ is the set of NFA states reachable from any state in $S$ after seeing the input $a$
    • considering $\varepsilon$-moves as well
**NFA to DFA. Remark**

- An NFA may be “in many states” at one time

- How many different states?

- If there are N states, the NFA must be in some subset of those N states

- How many subsets are there (at most)?
  - $2^N - 1 = \text{finitely many, but usually much more than } N$
NFA -> DFA Example
Implementation

- A DFA can be implemented by a 2D table $T$
  - One dimension is “states”
  - Other dimension is “input symbols”
  - For every transition $S_i \xrightarrow{a} S_k$ define $T[i,a] = k$
- DFA “execution”
  - If in state $S_i$ and input $a$, read $T[i,a] = k$ and skip to state $S_k$
  - Very efficient
Table Implementation of a DFA

<table>
<thead>
<tr>
<th>state</th>
<th>inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>U</td>
<td>T</td>
</tr>
</tbody>
</table>
Implementation (Cont.)

• NFA -> DFA conversion is at the heart of tools such as flex.

• But, DFAs can be huge.

• In practice, flex-like tools trade off speed for space in the choice of NFA and DFA representations.

• Oh, there can be many extra states, and usually are, in an auto-generated DFA. Can be mechanically reduced to a minimum number of states, but still may be huge.
Writing a DFA in Lisp

;;; -*- Mode: Lisp; Syntax: Common-Lisp -*-

;;; A simple finite state machine (fsm) simulator
;;; Note FSM is the same as a DFA (deterministic finite automaton).

;;; Reference to MCIJ is "Modern Compiler Implementation in Java"
;;; by Andrew Appel.

;;; First we show a deterministic finite state machine fsm, then a
;;; non-deterministic fsm: nfsm then a version of nfsm allowing
;;; "epsilon" transitions.

;;; First with no data abstractions. We decide on the representation
;;; and program away. The correspondence of (state,input) --> next
;;; state is recorded in an association list, as illustrated below.

(defun fsm (state input)
  (getf (assoc (adjoin (state) input) transitions) final)
)

(defun nfsm (state input)
  (cond ((epsilon input)
           (found final state)
         (t
           (for each transition do
             (fseq ( NFsm (state) input) done)
            (if (found final state) state)
          (error "state not found" state))
        1))

(defun epsilon (state)
  (getf (assoc state (state)) transitions))

(defun transitions (state)
  (getf (assoc state (state)) transitions))

(defun found (state)
  (equal state final))

(defun adjoin (state input)
  (cons (car state) (cdr state))
)

(defun fseq (state input)
  (if (null input)
    state
    (fseq (state) (cdr input))
  )
)

(defun fseq (state)
  (state)
)

(defun assoc (state transitions)
  (getf transitions state)
)

(defun defstruct (state (:type list)) transitions final)

;;; first use of defstruct
Set up Mach1 with 3 states

(setf Mach1 (make-array 3))

;; The first machine, with 3 states we will denote 0,1,2 will be stored
;; in an array called Mach1. This machine accepts \((c+d)c^*\) and that's all

(setf (aref Mach1 0) ; initial state
  (make-state :transitions
    '(((#\c 1) ;; if you read a c go to state 1
      (#\d 1)) ;; if you read a d go to state 1
    ;; if you read anything else it is a error
      :final nil))

(setf (aref Mach1 1)
  (make-state :transitions
    '(((#\c 1)
      (#\d 2))
    :final t))

(setf (aref Mach1 2) ;; dead end state. no way out
  (make-state :transitions
    '(((#\c 2) ;
      (#\d 2))
    :final nil))

\[\text{Diagram of the machine states and transitions}\]
FSM program in lisp

;; fsm simulates a deterministic finite state machine.
;; given a state number 0,1,2,... returns t for accept, nil for reject.

(defun fsm (state state-table input)
  (cond ((string= input "")
            (state-final (aref state-table state))
          (t(let ((trans
                    (assoc
                      (elt input 0)
                      (state-transitions
                       (aref state-table state))))
              (and trans (fsm (cadr trans) state-table (subseq input 1)))))))

;; that’s all. See file fsm.cl for many fluffed-up abstractions,
;; comments, and extensions to NFA
Actually, we can write lexers rather simply

- Although RegExps / DFAs/ NFAs are neat, and we teach them in CS164, we are writing lexers on digital computers with memory.
- These are more powerful than DFAs.
- An entirely reasonable lexer can be written using (what amounts to) recursive descent parsing, (later in course!) but in such a simple form that it hardly needs explanation.
- If we insist on automated tools, we can compile patterns into programs simply, too.
Writing stuff in Lisp

• I’d feel bad if too much of this course is specifically about details of Lisp (or for that matter about any particular language)
• But there are features and design issues raised by how Lisp works.
• Some details are inevitably needed... how to read, print, stop loops.
• File: readprintrex (mostly text); iterate.cl
RegExps in Lisp. A recipe for matchers

• Say we want to write a clear metalanguage for RegExps so we can automatically build specific recognizer programs. Like flex. But we will write it in 2 pages of Lisp you can read.

• Step one: Come up with a formal “grammar” for regexps that can be “parsed”.

• Step two: Write a parser than produces as output a Lisp program that implements the recognizer.
A data language for constructing REs

• “abc” is the language {“abc”}
• stwildcard matches any string. { [a-z,A-Z]∗}
• If r1, r2, … rn are REs then so are
  - (union r1 r2)
  - (star r1)
  - (star+ r1)
  - (sequence r1 r2 …)
  - (assign r1 name) same as r1 with side effect
  - (eval r1 expression) same as r1 with eval side effect
Important: So far we are talking about data not operations

- We are not computing union etc etc. We are merely constructing Lisp lists.
- For example, type '(union "a" "b")
- Or (list 'union "a" "b")
The only interesting operations we need are matching RegExps.

- To match a literal, look for it literally
- To match a sequence, do (and (match r1) (match r2) …) -- (every #'match '(r1 r2 ….))
- To match a union, do (or (match r1) (match r2) …) continues until one succeeds. - (any #'match '(r1 r2 …))
- To match (star r1), in lisp:
  - (not (do () ((not (match r1))))) ;;;;... restated more conventionally,
  - (loop indefinitely until you find a failure to match r1) then return true, for all those forms (maybe none) which matched. Problem with matching (0+1)*01 which requires backup..
Here's the matching program (most of it)

(defun mymatch (x)
  (declare (special string index end))
  (typecase x
    (list ;; either a list or something else
      (ecase (car x) ;; test the car for something we know
        (sequence (every #'mymatch (cdr x)))
        (union (some #'mymatch (cdr x)))
        (star (not (do ()((not (mymatch (cadr x))) ))))))))
  ;; it is not a list
  (t (matchitem x)))
Here's the matching program (more of it)

(defun mymatch0 (pat string)
  (declare (special string))
  (let ((index 0)
        (end (length string)))
    (declare (special index end))
    ;; this is not very nice lisp: it uses
    ;; global "special" variables instead of
    ;; lexical variables.
    ;;
    (if (and (mymatch pat) (= end index))
      'success
      `(failed after ,index chars))))) ; first use of ` backquote
    ;; (list 'failed 'after index 'chars) ..
Here's the matching program (rest of it)

(defun matchitem (x)
  (declare (special index end string))
  (cond ((>= index end) nil)
        ((characterp x) ; match a character
         (if (char= x (elt string index)) (incf index) nil))
        ((stringp x)
         (and (string= x (subseq string index (+ index (length x))))
              (incf index (length x))))
        ((eq x '?) (incf index)) ; single character wildcard
        ((eq x 'alphanumeric) (and
          (alphanumericp (elt string index))
          (incf index)))
        ;; generalize this to any predicate
        ((and (symbolp x) (get x 'chartype))
         (and (funcall (get x 'chartype) (elt string index))
              nil))
     (t nil)))
Here's the matching program (extending it)

```lisp
(setf (get 'digit 'chartype)
    #'(lambda(x)
        (and
            (member x '(#\0 #\1 #\2 #\3 #\4 #\5 #\6 #\7 #\8 #\9))
            (incf index))))

;;see matchprog.cl
```
What if you don't like (union r1 r2), (seq r1 r2)? / the META system.. (H. Baker)

- \([r1 \ r2]\) for sequence
- \(\{r1 \ r2\}\) for union
- \(R1\$\) for Kleene star
- ! For evaluation
- @ for indirect “anything of this type”

```lisp
defun parse-int (&aux (s +1) d (n 0))
  (and
    (matchit
      (\([\#\+ \[#\- !\(setq s -1\)] \]\)
       @(digit d) !(setq n (ctoi d))
       @(digit d) !(setq n (+ (* n 10) (ctoi d))))
     ($[\@\(digit d\) !\(setq n (+ (* n 10) (ctoi d))))\]])
     (* s n)))```
Pragmatic parsing (Prag-Parse.html)

• Mostly this is a tour-de-force of Lisp programming to show you can do lex/yacc Unix utilities in a few pages of Lisp. But it also suggests that with appropriate choice of data structure and a versatile language, you can scan/parsen a fairly complicated language.

• Rather sophisticated Lisp programming style.
Simpler program (pitman.cl)

• Taken off comp.lang.lisp newsgroup
• Kent Pitman’s answer to How does one do lexical analysis in lisp?
• Rather straightforward Lisp programming style.
Conclusion: Regular Expression Programs

• Easy to specify lexical structure of typical language by Regular Expressions.
• Good correspondence between intuition and implementation
• Automated tools can use the RE specs.
• Next time: more on just seat-of-pants systematic programming.