Writing a Lexical Analyzer in Haskell (part II)

Today

- Regular languages and lexicographical analysis part II
- Some of the slides today are from Dr. Saumya Debray and Dr. Christian Colberg

This week

- PA1: It is due in 4 days!
- PA2 has been posted. We are starting to cover concepts needed for PA2.
- Recitation tomorrow will be on implementing lexers in Haskell using a table-driven approach.

General Approach for Lexical Analysis

Regular Languages

Finite State Machines

- –DFAs: Deterministic Finite Automata
- –***Complications when doing lexical analysis***
- – NFAs: Non Deterministic Finite State Automata

From Regular Expressions to NFAs

From NFAs to DFAs

Complications

1. "1234" is an NUMBER but what about the "123" in "1234" or the "23", etc. Also, the scanner must recognize many tokens, not one, only stopping at end of file.

2. "if" is a keyword or reserved word IF, but "if" is also defined by the reg. exp. for identifier ID. We want to recognize IF.

3. We want to discard white space and comments.

4. "123" is a NUMBER but so is "235" and so is "0", just as "a" is an ID and so is "bcd", we want to recognize a token, but add attributes to it.

Complications 1 (longest match)

1. "1234" is an NUMBER but what about the "123" in "1234" or the "23", etc. Also, the scanner must recognize many tokens, not one, only stopping at end of file. So:

- recognize the largest string defined by some regular expression,
- only stop getting more input if there is no more match.
- This introduces the need to reconsider a character, as it is the first of the next token

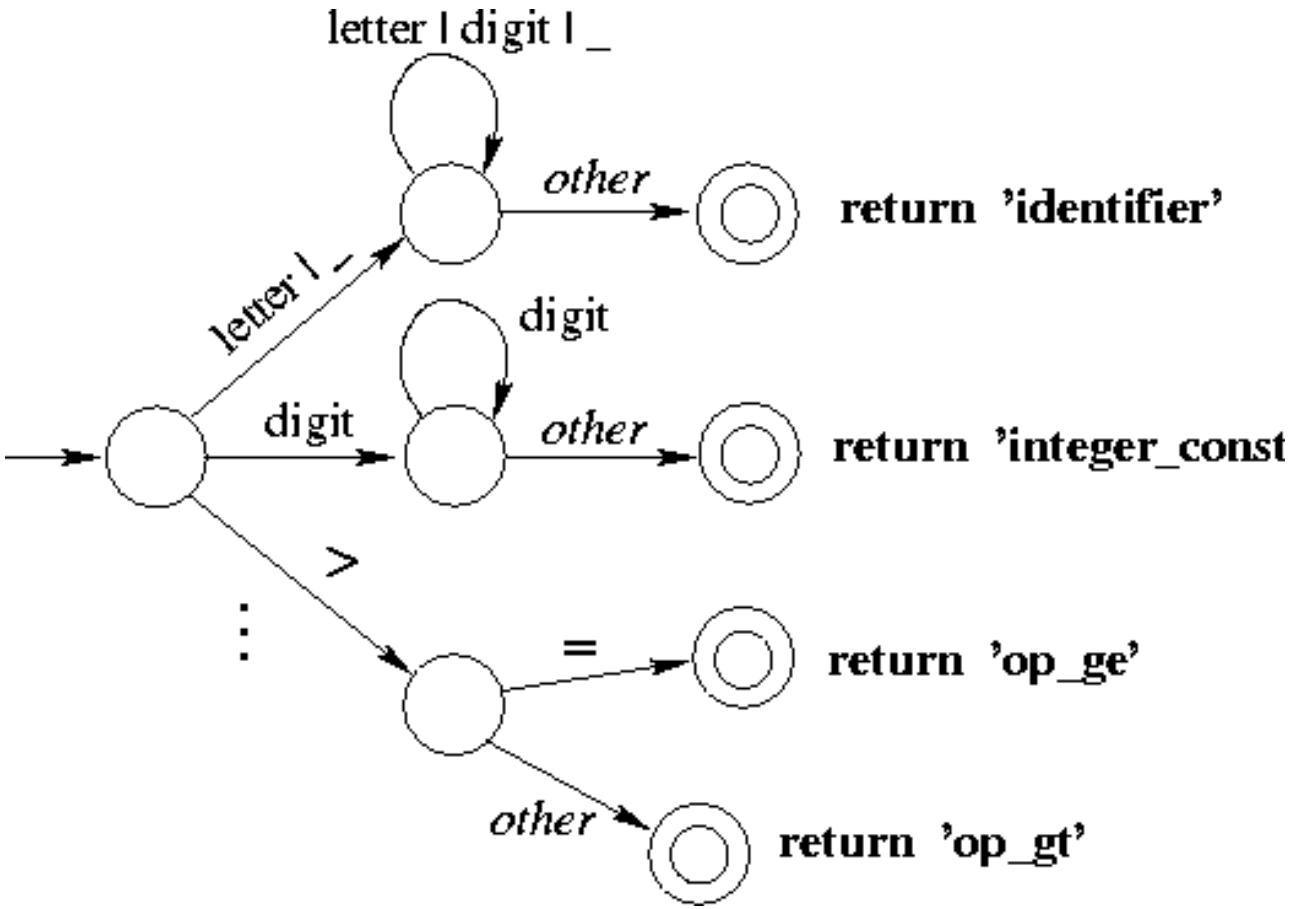
e.g. *fname(a,bcd);*

- would be scanned as

(TokenID "fname") OPEN (TokenID "a") COMMA ... SEMI EOF

- scanning *fname* would peek at (, which would be put back and then recognized as OPEN

Structure of a Scanner Automaton



Implementing finite state machines

Table-driven FSMs (e.g., *lex*, *flex*):

- Use a table to encode transitions:

  next_state = *T*(curr_state, next_char);

- Use one bit in state no. to indicate whether it's a final (or error) state. If so, consult a separate table for what action to take.

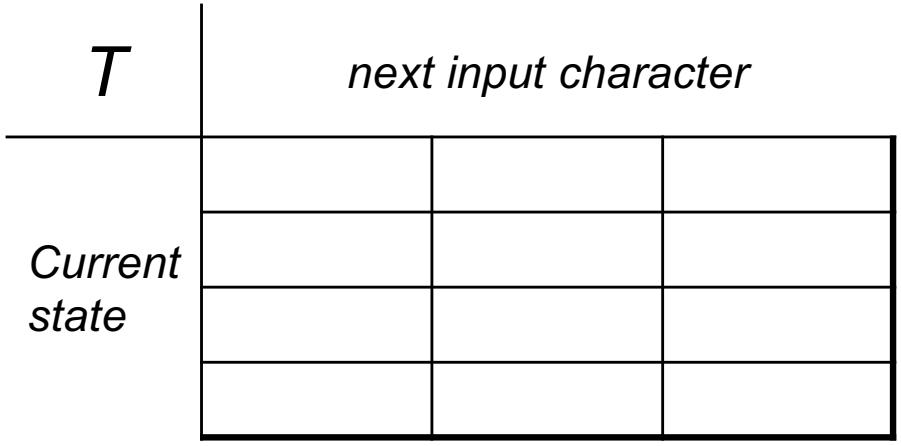
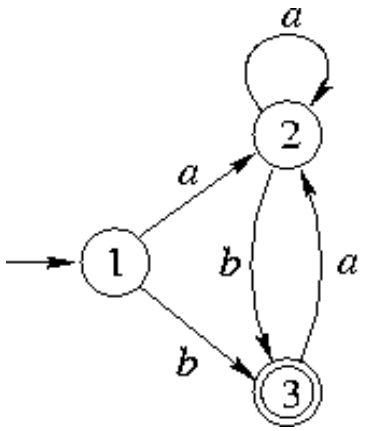


Table-driven FSMs: Example

```
int acceptString()
{  char ch;
   int currState = 1;  ch = nextChar();

   while (ch!=EOF) {
     currState= T [currState, ch];
   } /* while */
   if (IsFinal(currState)) {
      return 1; /* success */
   }
}
```



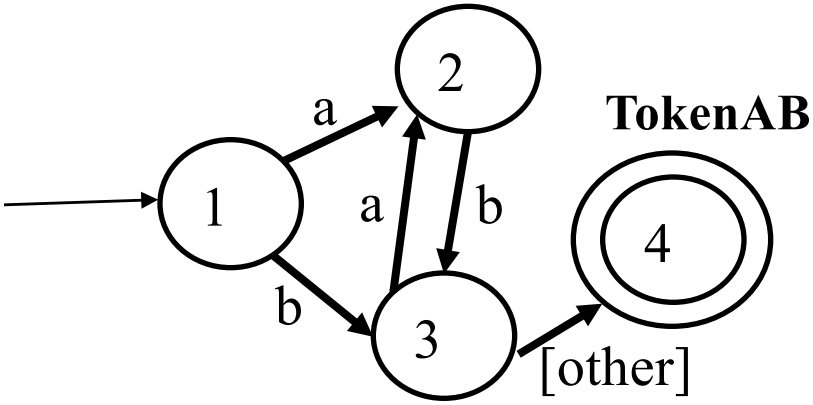
T

state \ input	a	b
1	2	3
2	2	3
3(final)	2	

Table-driven FSMs: Determines if full string is in language

```
Token scanner()
{  char ch;
   int currState = 1;  ch = nextChar();

   while (not IsFinal(currState)) {
     nextState = T [currState, ch];
     if (consume(currState,ch)) {
       ch = NextChar( );
     }
     if (ch == EOF) { return 0; } /* fail */
     currState = nextState;
   } /* while */
   if (IsFinal(currState)) {
     return finalToken(currState); /* success */
   }
}
```



T

state \ *input*	*a*	*b*
1	2	3
2	2	3
3	2	[other]
4(final)		

Go see http://www.cs.arizona.edu/classes/cs453/fall16/Recit/LexerStart-take2.hs

Table-Driven FSM for Numbers

```
-- Produce tokens until the input string
-- has been completely consumed.
lexer :: String -> [Token]
lexer []    = []
lexer input =
  let (tok,remaining) = driveTable 0 “” input
  in if tok==WhiteSpace then lexer remaining
     else tok : lexer remaining

-- From given state consume characters
-- from the string until token is found.
driveTable :: Int->String->String->(Token,String)
driveTable curr [] = (UnexpectedEOF, "")
driveTable curr (c:rest) =
  let (next,consume)  = nextState curr c
      (nextTokStr,remaining)= nextStrings ...
      (done,tok)      = final next nextTokStr
  in if done then (tok,remaining)
     else driveTable next nextTokStrnremaining
```

Draw FSM on board

- State 0
  - Digit goto state 1
- State 1
  - Digit goto state 1
  - Other goto state 2
- State 2 is a final state for TokenNUM

How should we define nextState and final functions?

Complication 2 (priority combined with longest match)

2. "if" is a keyword or reserved word IF, but "if" is also defined by the reg. exp. for identifier ID, we want to recognize IF, so

Have some way of determining which token (IF or ID) is recognized.

This can be done using priority, e.g. in scanner generators an earlier definition has a higher priority than a later one.

By putting the definition for IF before the definition for ID in the input for the scanner generator, we get the desired result.

What about the string "ifyouleavemenow"?

Complication 3

3. we want to discard white space and comments and not bother the parser with these. So:

In scanner generators, we can

- specify, using a regular expression, white space e.g. [\t\n] and return no token, i.e. move to the next
- specify comments using a (NASTY) regular expression and again return no token, move to the next

When writing scanner by hand

```
lexer (c:rest) = if isSpace c
                 then lexer rest
                 else TokenUnknownChar c : lexer rest
```

Complication 4 (Information Associated with Token)

4. "123" is a NUMBER but so is "235" and so is "0", just as "a" is an ID and so is "bcd", we want to recognize a token, but add attributes to it. So,

Want to associate some data with some Token types:

```
data Token
    = TokenIfKW
    | TokenID String
    -- ...
    deriving (Show,Eq)
```

Often more information is added to a Token, e.g. line number and position

(Non) Deterministic Finite State Automata

A Deterministic Finite State Automaton (DFA) has disjoint character sets on its edges, i.e. the choice "which state is next" is deterministic.

A Non-deterministic Finite State Automaton (NFA) does NOT, i.e. it can have character sets on its edges that overlap (non empty intersection), and empty sets on the some edges (labeled ε).

NFAs are used in the translation from regular expressions to FSAs. E.g. when we combine the reg. exp for IF with the reg.exp for ID by just merging the two Transition graphs, we would get an NFA.

NFAs are a first step in creating a DFA for a scanner.

The NFA is then transformed into a DFA.

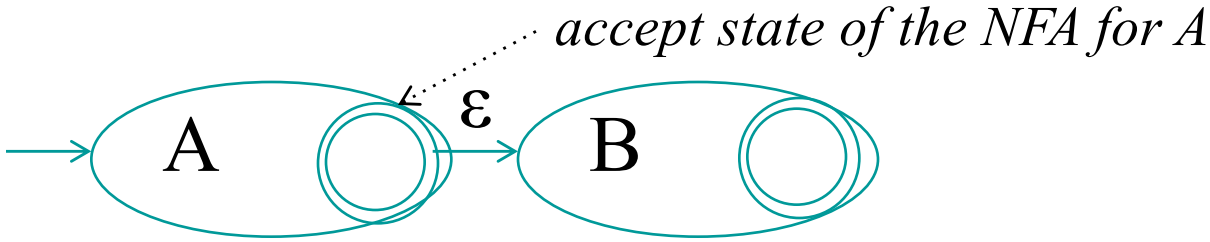
From regular expressions to NFAs

regexp

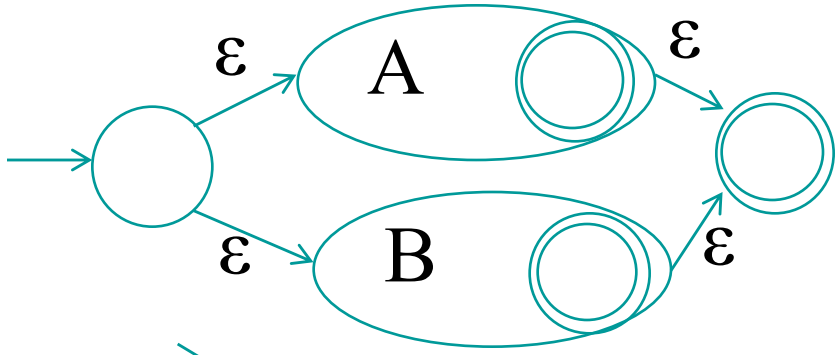
simple letter "a"

empty string

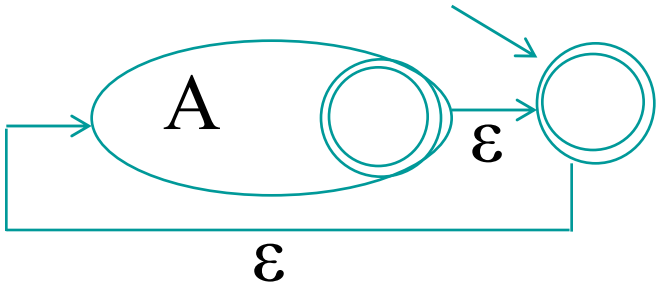
AB concat the NFAs



A|B split merge them



A* build a loop



The Problem

DFAs are easy to execute (table driven interpretation)

NFAs

- are easy to build from reg. exps,
- but hard to execute
- we would need some form of guessing, implemented by back tracking

To build a DFA from an NFA

- we avoid the back track by taking all choices in the NFA at once,
- a move with a character or ε gets us to a set of states in the NFA,
- which will become one state in the DFA.

We keep doing this until we have exhausted all possibilities.

- This mechanism is called transitive closure
- (This ends because there is only a finite set of subsets of NFA states. How many are there?)

Example IF and ID

let : [a-z]

dig : [0-9]

tok : if | id

if : “i” “f”

id : let (let | dig)*

Notes to read through later, Definitions: edge(s,c) and closure

edge(s,c): the set of all NFA states reachable from state s following an edge with character c

closure(S): the set of all states reachable from S with no chars or ε

$$closure(S) = T = S \cup (\bigcup_{s \in T} edge(s, \varepsilon))$$

T=S

repeat T' =T;

forall s in T' { T' =T; $T = T' \cup (\bigcup_{s \in T'} edge(s, \varepsilon))$

until T' ==T

This transitive closure algorithm terminates because there is a finite number of states in the NFA

DFAedge and NFA Simulation

Suppose we are in state DFA d = {s_i, s_k,s_l}

By moving with character c from d we reach a set of new NFA states, call these DFAedge(d,c), a new or already existing DFA state

$$DFAedge(d,c) = closure(\bigcup_{s \in d} edge(s,c))$$

NFA simulation:

let the input string be $c_1 \ldots c_k$

d=closure({s1}) // s_1 the start state of the NFA

for i from 1 to k

d = DFAedge(d,c_i)

Constructing a DFA with closure and DFAEdge

state d_1 = closure(s_1) the closure of the start state of the NFA

make new states by moving from existing states with a character c, using DFAEdge(d,c); record these in the transition table

make accepts in the transition table, if there is an accepting state in d, decide priority if more than one accept state.

Instead of characters we use non-overlapping (DFA)

character classes to keep the table manageable.

Suggested Exercise

Build an NFA and a DFA for integer and float literals

dot: “.”

dig: [0-9]

int-lit: dig^{+}

float-lit: dig* dot dig+