Lecture 10

COP3402 FALL 2015 – DR. MATTHEW GERBER – 10/7/2015

FROM EURIPIDES MONTAGNE, FALL 2014

Tonight

The Scanner, Part 2

Review and Expansion: Alphabets and Strings

Definition (Alphabet):	An ***alphabet*** is *any finite, defined set of characters.*
Definition (String):	A ***string*** is a *finite sequence of characters drawn from an alphabet*.

For example:

◦ Σ = { 0, 1 }	The binary alpbahet
◦ **s** = 1011	A binary string **s**

Any sequence of 0s and 1s is a string over this alphabet.

We use |s| to denote the length of a string - that is, the number of symbols in it.

- If s = “while”, |s| = 5
- s = ε iff |s| = 0

Review and Expansion: Languages

Definition (Language): A ***language*** is *any countable set of strings over a fixed alphabet.*

- Note that a language does *not* have to be finite.

Let **L** be the alphabet of letters and **D** be the alphabet of digits:

- **L** = { A, B, …, Z, a, b, …, z }
- **D** = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }

Oddity: Alphabets are all *already languages.*

- Their strings just happen to all be of length 1

Review and Expansion: Languages (2)

Operation	Name	Observation
$L \cup D$	Union	62 strings of length one
LD	Concatenation	520 strings of length two Each is a letter followed by a digit
L^3	Repetition	The set of all three-letter strings
L^*	Closure	The set of all letter strings of any length; includes ε Formally called the *Kleene closure* of **L** Star means “zero or more”
L^+	Closure	The set of all digit strings of length ≥ 1
These operations can be combined. For example…		
$L(L \cup D)^*$		The set of all strings that begin with a letter and contain letters and digits

Starting to look familiar, isn't it?

Regular Expressions, In Brief

A *regular expression* **r** is a notation describing a language L(**r**) over an alphabet Σ.

- ε is a regular expression denoting the language $L(\varepsilon) = \{\varepsilon\}$.
- $\forall\ \alpha \in \Sigma$, α is a regular expression denoting the language $L(\alpha) = \{\alpha\}$.
- Given two regular expressions **r** and **s**, **rs** denotes the language L(**r**)L(**s**).
- Given two regular expressions **r** and **s**, **r|s** denotes the language L(**r**) $\cup$ L(**s**).
- Given a regular expression **r**, $\mathbf{r}^*$ is a regular expression.
- Given a regular expression **r**, $\mathbf{r}^+$ is a regular expression.
- Given a regular expression **r**, (**r**) is a regular expression.

Regular Expression Examples

Consider the alphabet Σ = { A, B, …, Z, a, b, …, z, 0, 1, 2, … 9 }. Then:

- ε is a regular expression denoting the language L(ε) = {ε}.
- $\forall\ \alpha \in \Sigma$, α is a regular expression denoting the language L(α) = {α}.
- “a” and “b” are regular expressions.
- “a|b” denotes the language { a, b }.
- “(a|b)(a|b)” denotes the language { aa, ab, ba, bb }.
- “a*” denotes the language { ε, a, aa, aaa, aaaa, … }
- “(a|b)*” denotes the language { ε, a, b, aa, ab, ba, bb, aaa, … }
- “a|a*b” denotes the language {a, b, ab, aab, aaab, aaaab, … }

Common Extensions to Regular Expressions

Plus closure: The + operator.

- $\mathbf{r}^{+} = \mathbf{rr}^{*}$

Zero or one instances: The ? operator.

- $\mathbf{r}? = (\varepsilon|\mathbf{r})$

Bracketed character ranges:

- [a-z] = (a | b | c | … | z)

Regular expressions in the Scanner

An excerpt from the PL/0 reference document:

◦ Identifiers:	letter (letter | digit)*
◦ Literals:	(digit)+

Does this look familiar?

◦ Identifiers:	[a-zA-Z][a-zA-Z0-9]*
◦ Literals:	[0-9]+

And *you already know how to find these*.

DFAs in the Scanner

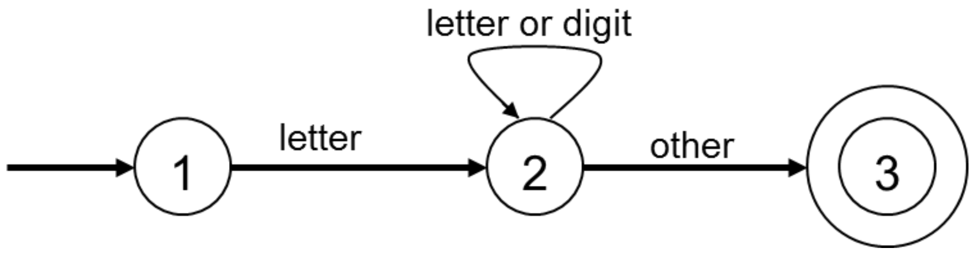
Recall that DFAs have:

- States
- Actions
- Start States
- Final States

This is the DFA to recognize identifiers.

- We have some bookkeeping to do once we arrive at the final state, but we'll get to that in a couple of slides.

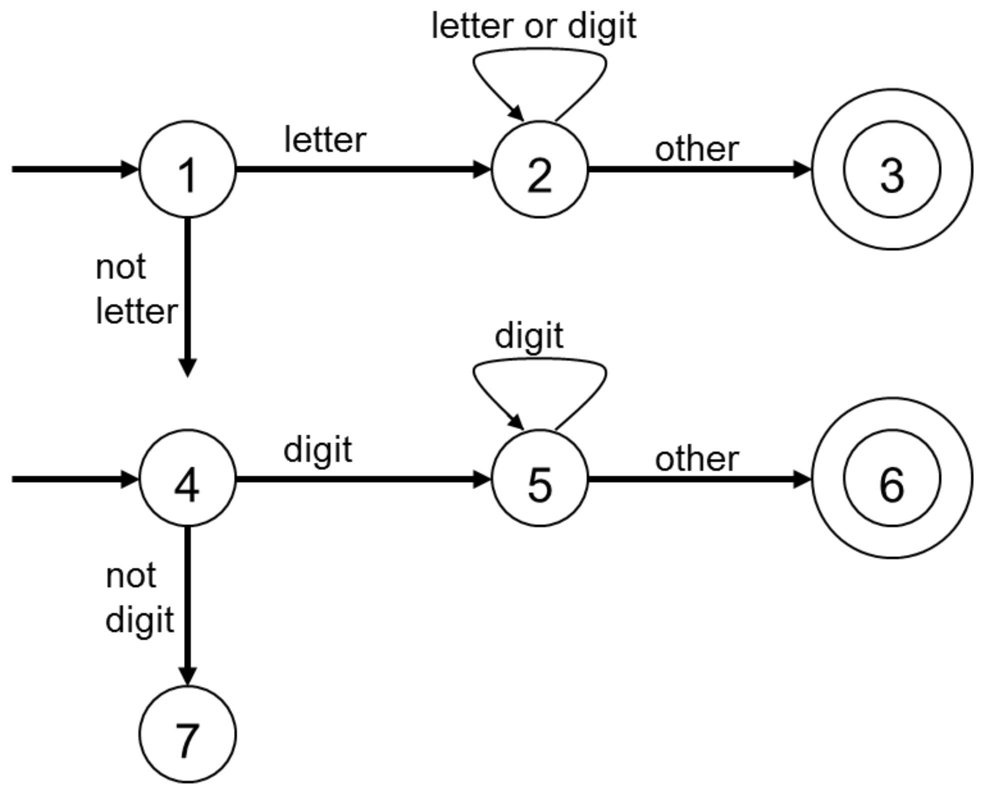
But how do we distinguish identifiers from literals?



Chained DFAs in the Scanner

We just chain the DFAs together!

- If we reach state 3, we've found an identifier
- If we reach state 6, we've found a literal integer
- If we reach state 7, whatever we've found isn't either of them
  - It might be an error or it might not – we'll have to figure that out in subsequent states

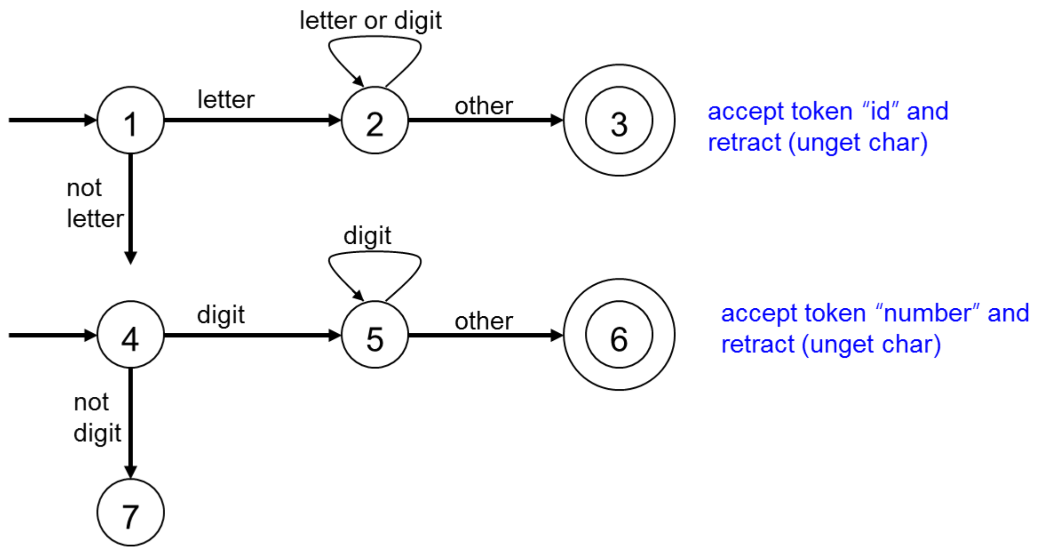


Remember Lookahead?

The character that ends our identifier or integer isn't actually part of it.

That means that after recognizing and tokenizing the identifier or integer, we need to *unget* the character that finished it for us.

We then go back to the beginning of our whole recognizer, and process that "ungot" character as the beginning of the next token (or whitespace, or whatever).



letter or digit
letter
other
1
2
3
accept token "id" and retract (unget char)
not letter
digit
digit
other
4
5
6
accept token "number" and retract (unget char)
not digit
7

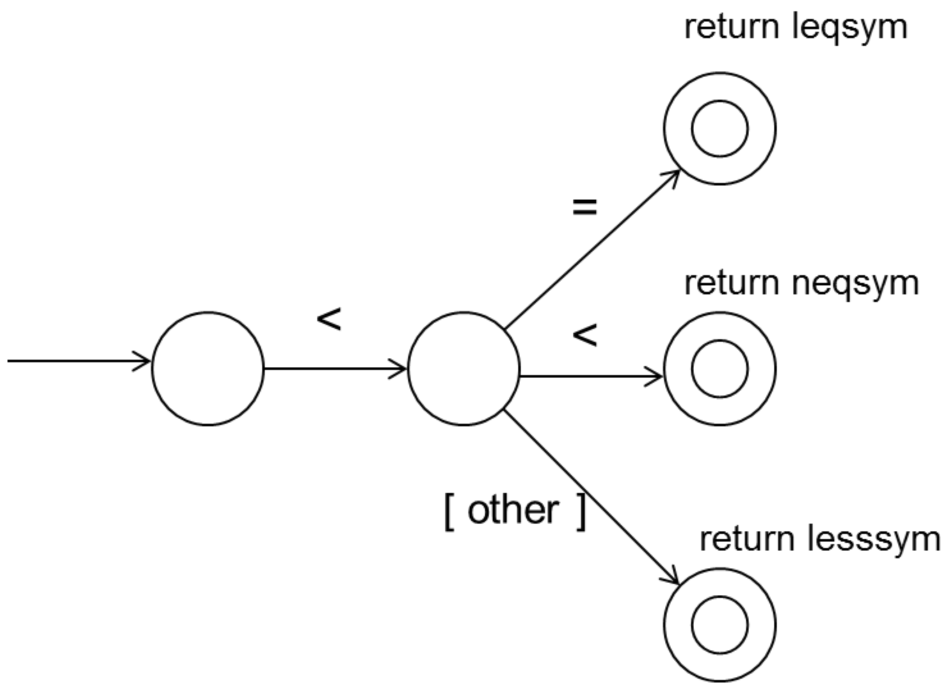
DFA Pseudocode

```
/* state 1 */  ch = getchar
               if isletter (ch) then {
/* state 2 */    while isletter(ch) or isdigit(ch) do{
                   ch := getchar;
                 }
/* state 3 */    ungetch /* Back up for next token */
                 create token
                 accept
                 return(token)
               } else {
                 reject /* Not an identifier */
                 ungetch /* Back up for next check */
               }
```

```
/* state 4 */  ch = getchar
               if isdigit(ch) then {
                 value := convert (ch)
/* state 5 */    ch = getchar
                 while isdigit (ch) do {
                   value := 10 * value + convert (ch)
                   ch := getchar
                 }
/* state 6 */    ungetch /* Back up for next token */
                 create token
                 accept
                 return (token)
               } else {
                 reject /* Not a number */
                 ungetch /* Back up for next check */
               }
/* state 7 */  /* Check for something else */
```

Another Example: Some Comparisons

```
ch := getch;
If ch = '<' then
   begin
      ch := getch;
      if ch = '=' then
         begin
            token := leqsym;
            ch := getch
         end
      else
         if ch = '>' then
            begin
              token := neqsym;
              ch := getch
            end
         else token := lessym
   end;
```



One More Example: C Comments

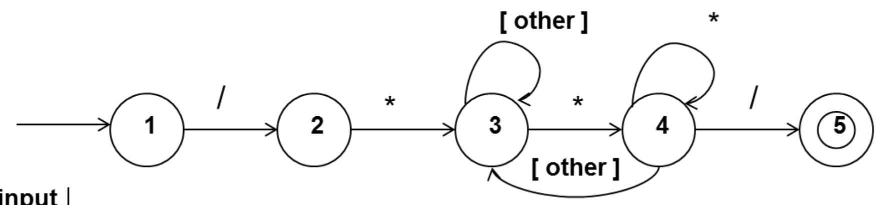
C comments are simple conceptually:

- /* (Anything goes here) */

We’re not going to try to write the regular expression.

- The comments actually *involve the star character*
- It would be more confusing than helpful

However, the DFA is still clear enough – and that means turning it into code would be clear enough as well.



input / state	/	*	[other]	accepting
1	2			no
2		3		no
3	3	4	3	no
4	5	4	3	no
5				yes

Transition table

Next Time:
Parser Preliminaries