



RICE

COMP 412  
FALL 2009

# *Lexical Analysis – Introduction*

## *Comp 412*

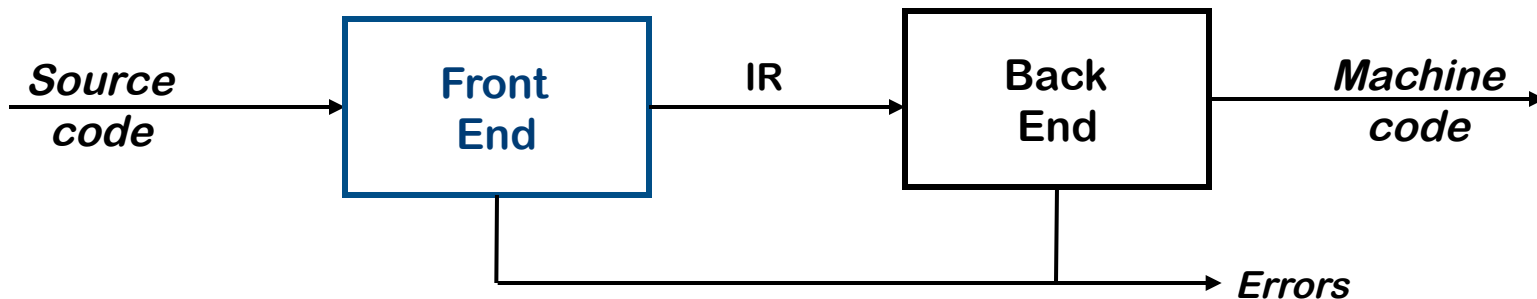
The slides assume some familiarity with finite automata.  
For a different (& more intuitive?) introduction to finite automata & recognizers, see Section 2.2 of EaC

Copyright 2009, Keith D. Cooper & Linda Torczon, all rights reserved.

Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.

Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

# The Front End

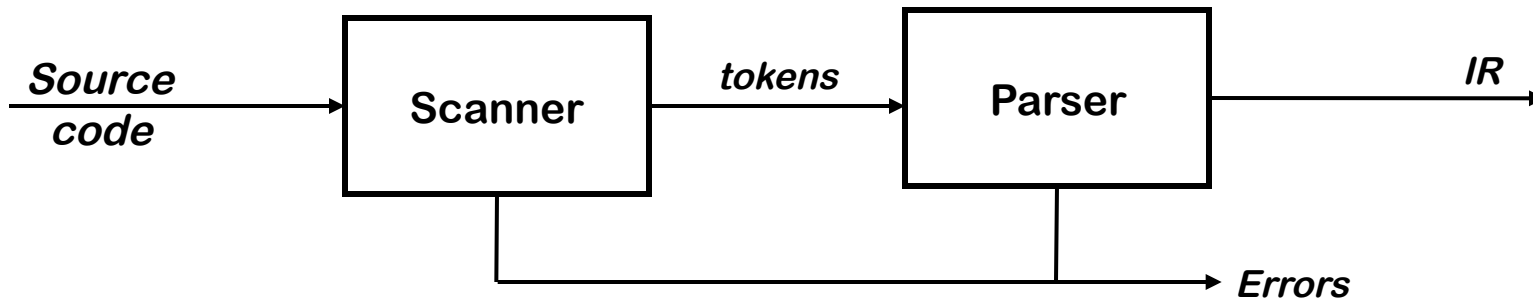


The purpose of the front end is to deal with the input language

- Perform a membership test:  $\text{code} \in \text{source language?}$
- Is the program well-formed (semantically) ?
- Build an IR version of the code for the rest of the compiler

*The front end is not monolithic*

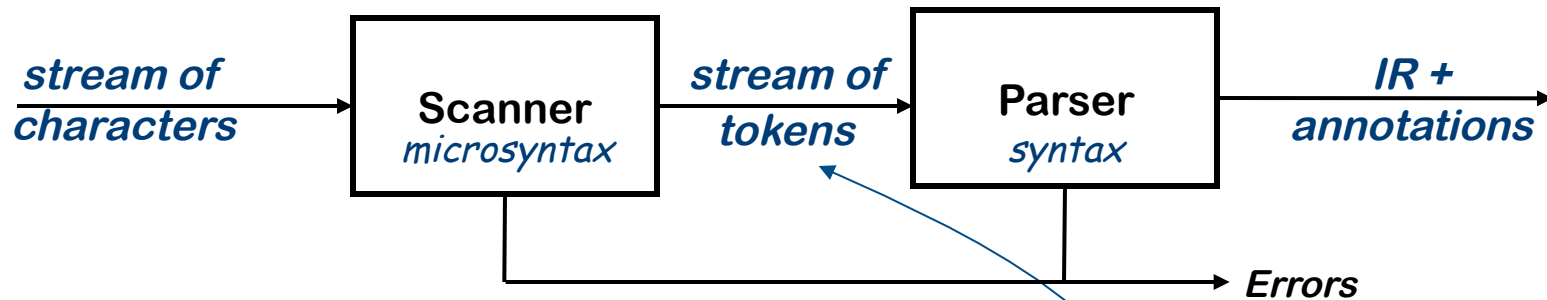
# The Front End



## Implementation Strategy

	Scanning	Parsing
Specify Syntax	regular expressions	context-free grammars
Implement recognizer	Deterministic finite automaton	Push-down automaton
Perform work	Actions on transitions in automaton	

# The Front End



## Why separate the scanner and the parser?

- Scanner classifies words
- Parser constructs grammatical derivations
- Parsing is harder and slower
- Separation simplifies implementation
  - smaller grammar for parser
  - faster front end

Scanner is only pass that touches every character of the input.

token is a pair  
<part of speech, lexeme>

also called *syntactic categories* or *tokentypes*



# The Big Picture

The front end deals with *syntax*

- Language syntax is specified with *parts of speech*, not *words*
- Syntax checking matches *parts of speech* against a grammar

Simple expression grammar from lecture 2

1. *goal* → *expr*

2. *expr* → *expr op t*

3. | *term*

4. *term* → num

5. | id

6. *op* → +

7. | -

The scanner turns a stream of characters into a stream of words, classified with their part of speech.

$N = \{ \textit{goal}, \textit{expr}, \textit{term}, \textit{op} \}$

$P = \{ 1, 2, 3, 4, 5, 6, 7 \}$

*parts of speech*  
*syntactic variables*



# The Big Picture

Why study automatic scanner construction?

- Avoid writing scanners by hand
- Harness the theory from classes like COMP 481

compile  
time

source code

Scanner

design  
time

specification

In practice, many people still write scanners by hand. Even if you intend to hand-code a scanner, writing down the specification and understanding the automata for it is extremely useful.

specifications written as "regular expressions"

Represent words as indices into a global table

Goals:

- To simplify specification & implementation of scanners
- To understand the underlying techniques and technologies

# Set Operations

(review)



Operation	Definition
<i>Union of <math>L</math> and <math>M</math> written <math>L \cup M</math></i>	$L \cup M = \{s \mid s \in L \text{ or } s \in M\}$
<i>Concatenation of <math>L</math> and <math>M</math> written <math>LM</math></i>	$LM = \{st \mid s \in L \text{ and } t \in M\}$
<i>Kleene closure of <math>L</math> written <math>L^*</math></i>	$L^* = \bigcup_{0 \leq i < \infty} L^i$
<i>Positive closure of <math>L</math> written <math>L^+</math></i>	$L^+ = \bigcup_{1 \leq i < \infty} L^i$

*These definitions should be well known*



# Regular Expressions

We constrain programming languages so that the spelling of a word always implies its part of speech *(few exceptions)*

The rules that impose this mapping form a *regular language*

*Regular expressions (REs)* describe regular languages

Regular Expression (over alphabet  $\Sigma$ )

- $\epsilon$  is a RE denoting the set  $\{\epsilon\}$
- If  $a$  is in  $\Sigma$ , then  $a$  is a RE denoting  $\{a\}$
- If  $x$  and  $y$  are REs denoting  $L(x)$  and  $L(y)$  then
  - $x | y$  is an RE denoting  $L(x) \cup L(y)$
  - $xy$  is an RE denoting  $L(x)L(y)$
  - $x^*$  is an RE denoting  $L(x)^*$

Precedence is *closure*,  
then *concatenation*,  
then *alternation*



# Regular Expressions

How do these operators help?

Regular Expression (over alphabet  $\Sigma$ )

- $\varepsilon$  is a RE denoting the set  $\{\varepsilon\}$
- If  $\underline{a}$  is in  $\Sigma$ , then  $\underline{a}$  is a RE denoting  $\{\underline{a}\}$ 
  - the spelling of any specific word is an RE
- If  $x$  and  $y$  are REs denoting  $L(x)$  and  $L(y)$  then
  - $x | y$  is an RE denoting  $L(x) \cup L(y)$ 
    - any finite list of words can be written as an RE  $(w_0 / w_1 / \dots / w_n)$
  - $xy$  is an RE denoting  $L(x)L(y)$
  - $x^*$  is an RE denoting  $L(x)^*$ 
    - we can use concatenation & closure to write more concise patterns and to specify infinite sets that have finite descriptions



# Examples of Regular Expressions

## Identifiers:

*Letter* → (a|b|c | ... | z|A|B|C | ... | Z)

*Digit* → (0|1|2 | ... | 9)

*Identifier* → *Letter* (*Letter* | *Digit*)\*

shorthand  
for

(a|b|c | ... | z|A|B|C | ... | Z) (a|b|c | ... | z|A|B|C | ... | Z) | (0|1|2 | ... | 9)\*

## Numbers:

*Integer* → (+|-|ε) (0 | (1|2|3 | ... | 9)(*Digit*\*) )

*Decimal* → *Integer* . *Digit*\*

*Real* → (*Integer* | *Decimal*) E (+|-|ε) *Digit*\*

*Complex* → (*Real* , *Real*)

*Numbers can get much more complicated!*

Using symbolic names  
does not imply recursion

underlining indicates  
a letter in the input  
stream

# Regular Expressions

*So what's the point?*



*We use regular expressions to specify the mapping of words to parts of speech for the lexical analyzer*

Using results from automata theory and theory of algorithms, we can automate construction of recognizers from REs

- ⇒ We study REs and associated theory to automate scanner construction !
- ⇒ Fortunately, the automatic techniques lead to fast scanners
  - used in text editors, URL filtering software, ...

# Example

(from Lab 1)

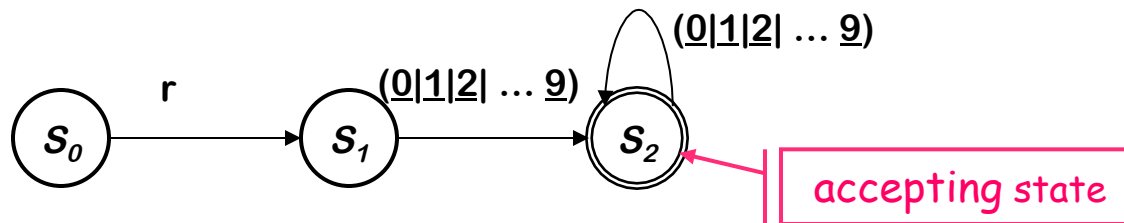


Consider the problem of recognizing ILOC register names

*Register*  $\rightarrow r (0|1|2| \dots | 9) (0|1|2| \dots | 9)^*$

- Allows registers of arbitrary number
- Requires at least one digit

RE corresponds to a recognizer (or DFA)



**Recognizer for *Register***

*Transitions on other inputs go to an error state,  $s_e$*

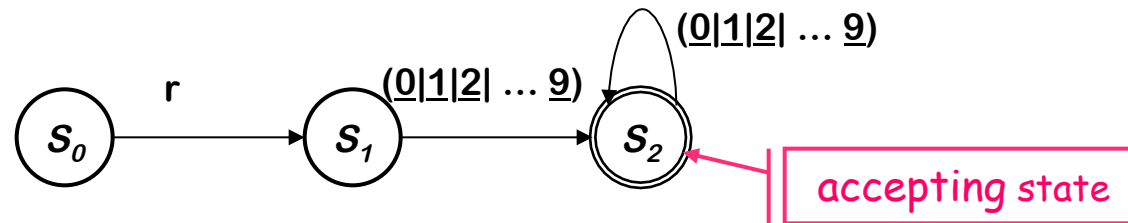
# Example

(continued)



## DFA operation

- Start in state  $S_0$  & make transitions on each input character
- DFA accepts a word  $\underline{x}$  iff  $\underline{x}$  leaves it in a final state ( $S_2$ )



## Recognizer for *Register*

So,

- r17 takes it through  $s_0, s_1, s_2$  and accepts
- r takes it through  $s_0, s_1$  and fails
- a takes it straight to  $s_e$

# Example

(continued)



To be useful, the recognizer must be converted into code

```
Char ← next character
State ← s0
while (Char ≠ EOF)
  State ← δ(State,Char)
  Char ← next character
if (State is a final state)
  then report success
  else report failure
```

*Skeleton recognizer*

$\delta$	r	0,1,2,3,4, 5,6,7,8,9	All others
s <sub>0</sub>	s <sub>1</sub>	s <sub>e</sub>	s <sub>e</sub>
s <sub>1</sub>	s <sub>e</sub>	s <sub>2</sub>	s <sub>e</sub>
s <sub>2</sub>	s <sub>e</sub>	s <sub>2</sub>	s <sub>e</sub>
s <sub>e</sub>	s <sub>e</sub>	s <sub>e</sub>	s <sub>e</sub>

*Table encoding the RE*

# Example

(continued)



We can add "actions" to each transition

```
Char ← next character
State ← s0
while (Char ≠ EOF)
  Next ← δ(State,Char)
  Act ← α(State,Char)
  perform action Act
  State ← Next
  Char ← next character
if (State is a final state)
  then report success
else report failure
```

*Skeleton recognizer*

$\delta$	$r$	0,1,2,3,4, 5,6,7,8,9	All others
$\alpha$			
$s_0$	$s_1$ <i>start</i>	$s_e$ <i>error</i>	$s_e$ <i>error</i>
$s_1$	$s_e$ <i>error</i>	$s_2$ <i>add</i>	$s_e$ <i>error</i>
$s_2$	$s_e$ <i>error</i>	$s_2$ <i>add</i>	$s_e$ <i>error</i>
$s_e$	$s_e$ <i>error</i>	$s_e$ <i>error</i>	$s_e$ <i>error</i>

*Table encoding RE*



## What if we need a tighter specification?

r *Digit Digit\** allows arbitrary numbers

- Accepts r00000
- Accepts r99999
- What if we want to limit it to r0 through r31 ?

Write a tighter regular expression

- *Register* → r ( (0|1|2) (*Digit* |  $\epsilon$ ) | (4|5|6|7|8|9) | (3|30|31) )
- *Register* → r0|r1|r2| ... |r31|r00|r01|r02| ... |r09

Produces a more complex DFA

- DFA has more states
- DFA has **same cost** per transition *(or per character)*
- DFA has same basic implementation

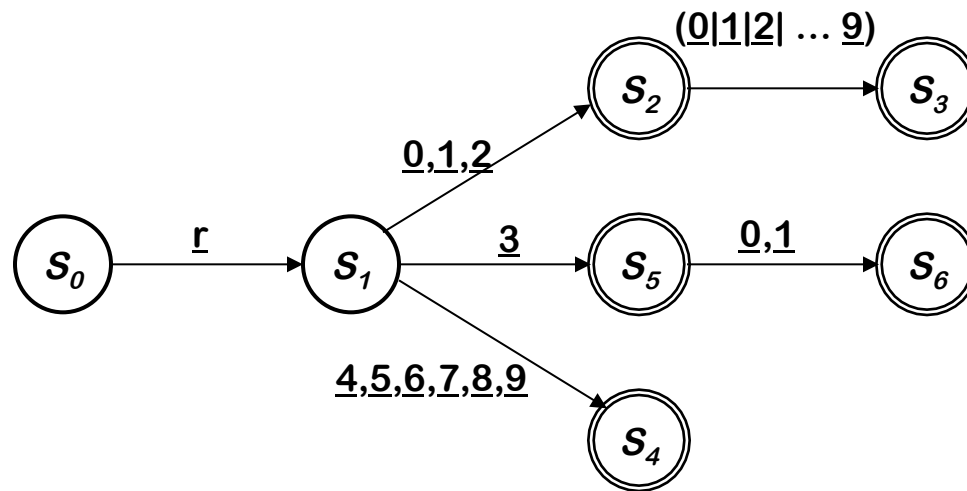
# Tighter register specification

(continued)



The DFA for

$Register \rightarrow \underline{r} ( (\underline{0}|\underline{1}|\underline{2}) (Digit | \varepsilon) | (\underline{4}|\underline{5}|\underline{6}|\underline{7}|\underline{8}|\underline{9}) | (\underline{3}|\underline{30}|\underline{31}) )$



- Accepts a more constrained set of register names
- Same set of actions, more states

# Tighter register specification

(continued)



$\delta$	$r$	0,1	2	3	4-9	All others
$s_0$	$s_1$	$s_e$	$s_e$	$s_e$	$s_e$	$s_e$
$s_1$	$s_e$	$s_2$	$s_2$	$s_5$	$s_4$	$s_e$
$s_2$	$s_e$	$s_3$	$s_3$	$s_3$	$s_3$	$s_e$
$s_3$	$s_e$	$s_e$	$s_e$	$s_e$	$s_e$	$s_e$
$s_4$	$s_e$	$s_e$	$s_e$	$s_e$	$s_e$	$s_e$
$s_5$	$s_e$	$s_6$	$s_e$	$s_e$	$s_e$	$s_e$
$s_6$	$s_e$	$s_e$	$s_e$	$s_e$	$s_e$	$s_e$
$s_e$	$s_e$	$s_e$	$s_e$	$s_e$	$s_e$	$s_e$

This table runs in the same skeleton recognizer

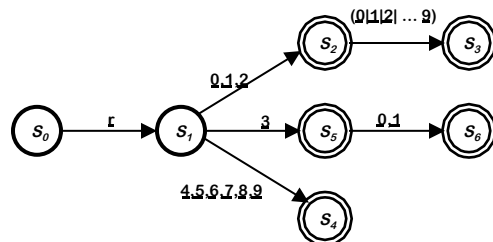
*Table encoding RE for the tighter register specification*

# Tighter register specification

(continued)



State Action	r	0,1	2	3	4,5,6 7,8,9	other
0	1 <i>start</i>	e	e	e	e	e
1	e	2 <i>add</i>	2 <i>add</i>	5 <i>add</i>	4 <i>add</i>	e
2	e	3 <i>add</i>	3 <i>add</i>	3 <i>add</i>	3 <i>add</i>	e <i>exit</i>
3,4	e	e	e	e	e	e <i>exit</i>
6	e	6 <i>add</i>	e	e	e	e <i>exit</i>
6	e	e	e	e	e	x <i>exit</i>
e	e	e	e	e	e	e



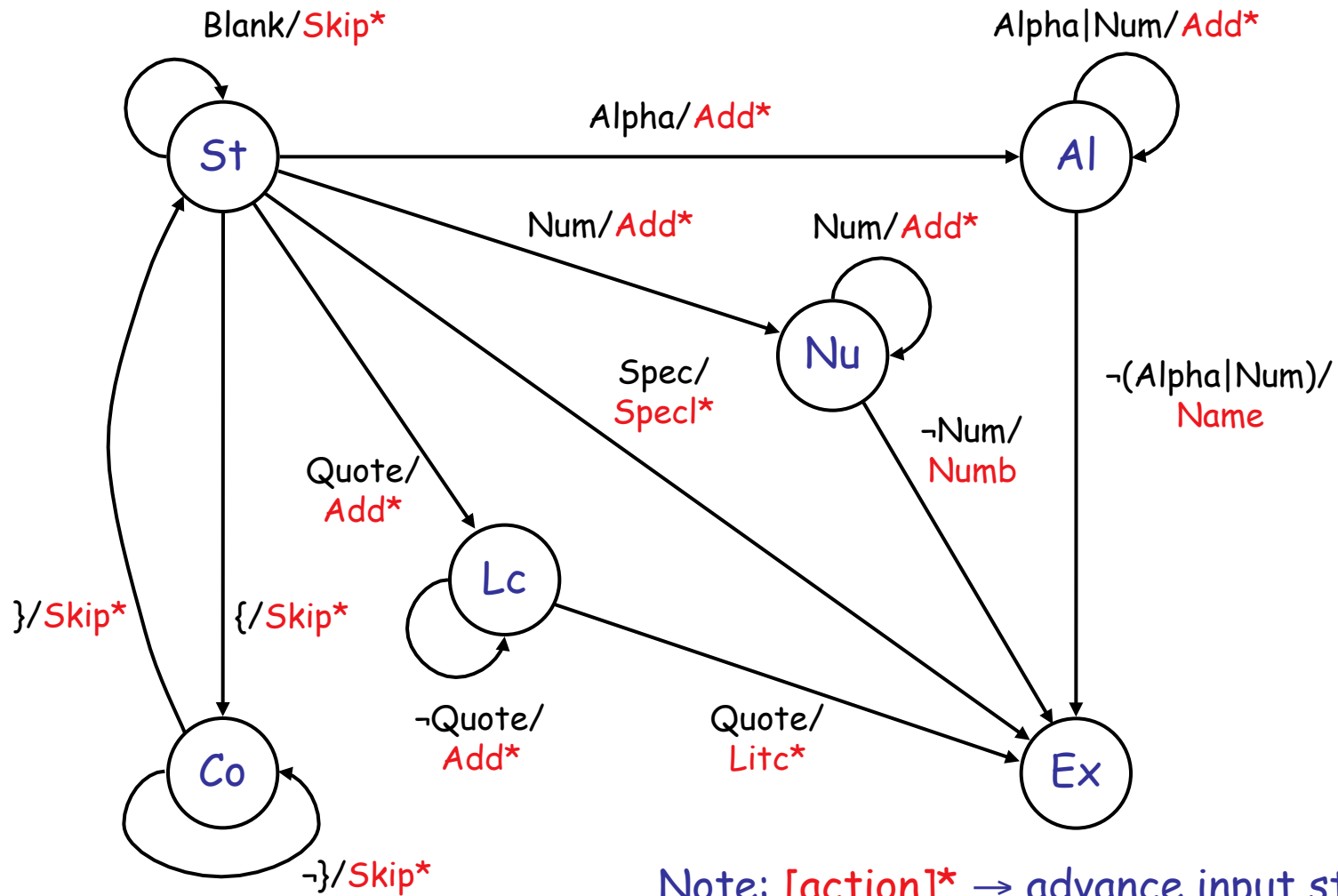
# Review of Scanners from Comp 314 (1998)

---



- Lexical Analysis Strategy: **Simulation of Finite Automaton**
  - States, characters, actions
  - State transition  $\delta(\text{state}, \text{charclass})$  determines next state
- **Next character** function
  - Reads next character into buffer
  - Computes **character class** by fast table lookup
- Transitions from state to state
  - Current state and next character determine (via  $\delta$ )
    - **Next state** and **action** to be performed
    - Some actions **preload** next character
- Identifiers distinguished from keywords by hashed lookup
  - This differs from EAC advice (discussion later)
  - Permits translation of identifiers into **<type, symbol\_index>**
    - Keywords each get their own type

# A Lexical Analysis Example



Note: [action]\* → advance input stream

# Comp 314 Lexical Scan Code

---



```
current = START_STATE;
token = "";
// assume next character has been preloaded into a buffer
while (current != EX)
{
    int charClass = inputstream->thisClass();
    switch (current->action(charClass))
    {
        case SKIP:
            inputstream->advance();break;
        case ADD:
            char* t = token; int n = ::strlen(t);
            token = new char[n + 2]; ::strcpy(token, t);
            token[n] = inputstream->thisChar(); token[n+1] = 0;
            delete [] t; inputstream->advance(); break;
        case NAME:
            Entry * e = symTable->lookup(token);
            tokenType = (e->type==NULL_TYPE ? NAME_TYPE : e->type);
            break;
        ...
    }
    current = current->nextState(charClass);
}
```