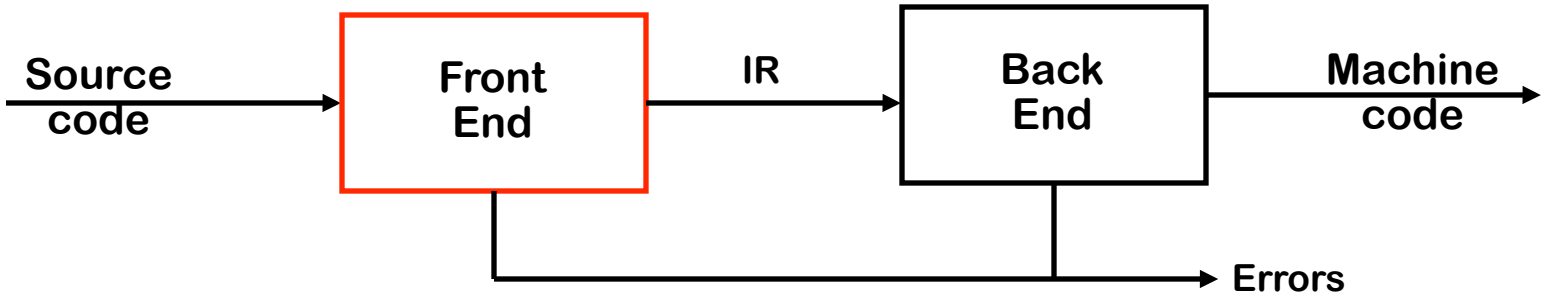
Lexical Analysis - An Introduction

Lecture 4
Spring 2005
Department of Computer Science
University of Alabama
Joel Jones

The Front End

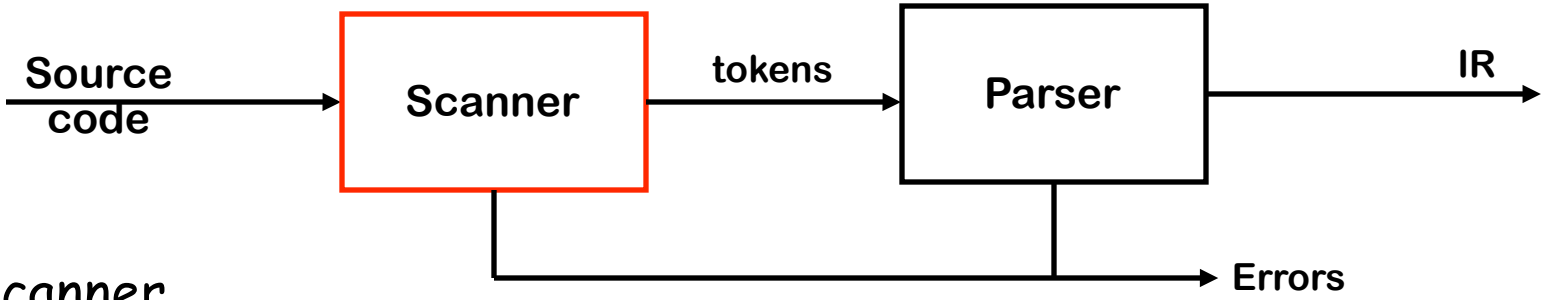


The purpose of the front end is to deal with the input language

- Perform a membership test: code ∈ 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



Scanner

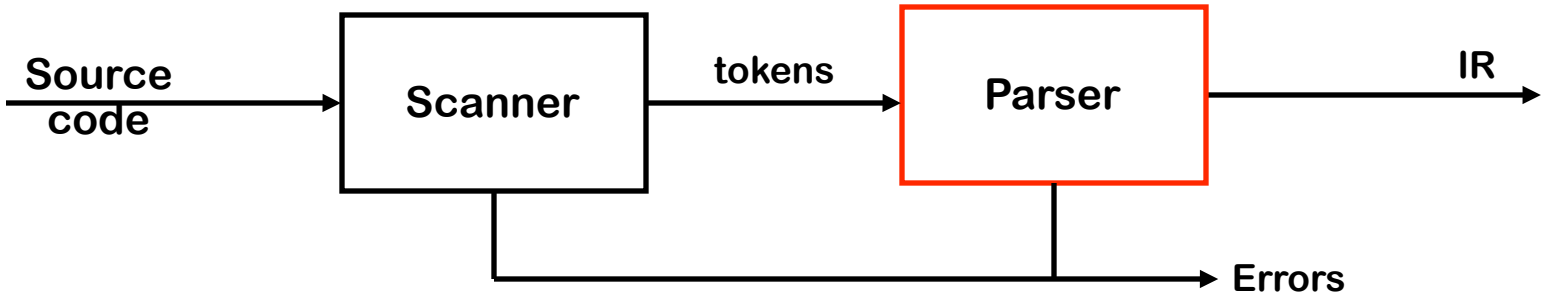
- Maps stream of characters into words
  - → Basic unit of syntax
  - → x = x + y ; becomes

    <id,x> <eq,=> <id,x> <pl,+> <id,y> <sc,; >
- Characters that form a word are its lexeme
- Its part of speech (or syntactic category) is called its token type
- Scanner discards white space & (often) comments

Speed is an issue in scanning

⇒ use a specialized recognizer

The Front End



Parser

- Checks stream of classified words (parts of speech) for grammatical correctness
- Determines if code is syntactically well-formed
- Guides checking at deeper levels than syntax
- Builds an IR representation of the code

We'll come back to parsing in a couple of lectures

The Big Picture

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

1. goal → expr
2. expr → expr op term
3. | term
4. term → number
5. | id
6. op → +
7. | –

S = goal

T = { number, id, +, - }

N = { goal, expr, term, op }

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

The Big Picture

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

1. goal → expr
2. expr → expr op term
3. | term
4. term → number
5. | id
6. op → +
7. | –

S = goal

T = { number, id, +, - }

N = { goal, expr, term, op }

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

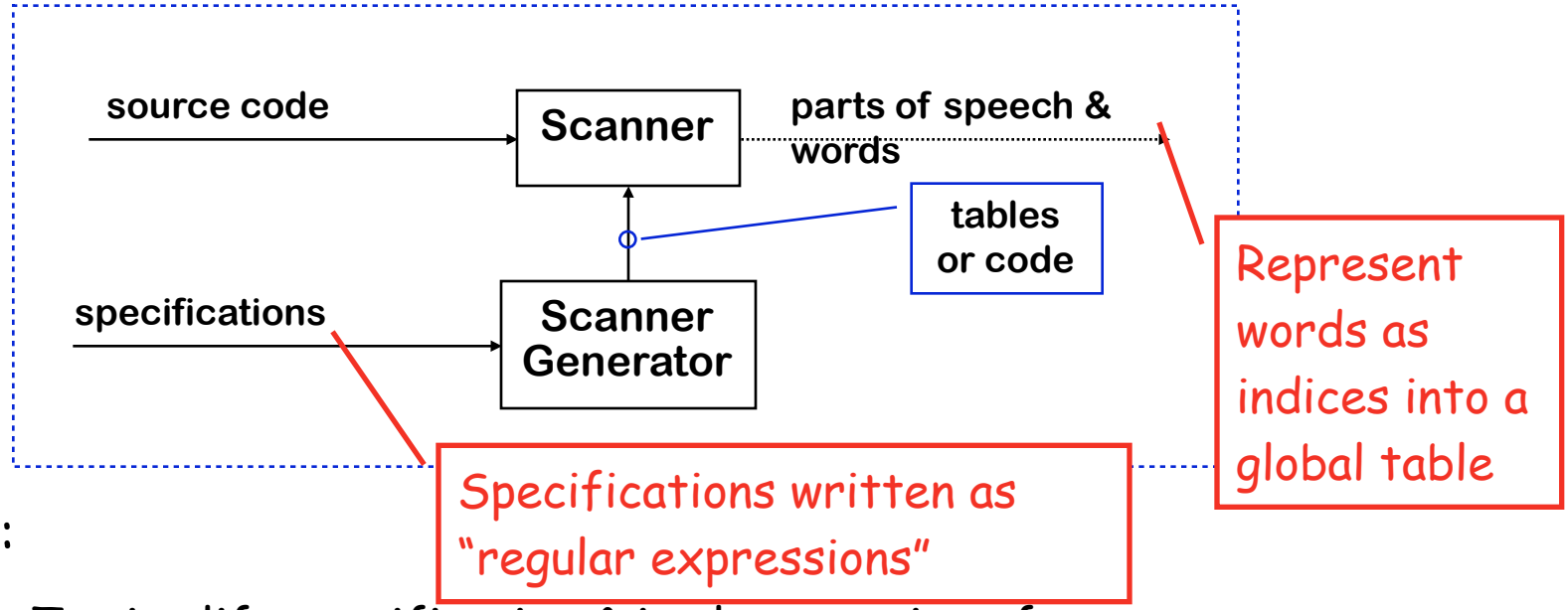
No words here!

Parts of speech, not words!

The Big Picture

Why study lexical analysis?

- We want to avoid writing scanners by han



Goals:

→ To simplify specification & implementation of scanners

→ To understand the underlying techniques and technologies

Regular Expressions

Lexical patterns form a regular language

*** any finite language is regular ***

Regular expressions (REs) describe regular languages

Ever type "rm *.o a.out" ?

Regular Expression (over alphabet Σ)

- ε is a RE denoting the set $\{\varepsilon\}$
- If $\underline{a}$ is in Σ, then $\underline{a}$ is a RE denoting $\{\underline{a}\}$
- 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

Set Operations (review)

Operation	Definition
Union of L and M Written L ∪ M	$L \cup M = \{s \mid s \in L \text{ or } s \in M\}$
Concatenation of L and M Written LM	$LM = \{st \mid s \in L \text{ and } t \in M\}$
Kleene closure of L Written L^*	$L^* = \bigcup_{0 \le i \le \infty} L^i$
Positive Closure of L Written L^+	$L^+ = \bigcup_{1 \le i \le \infty} L^i$

These definitions should be well known

Examples of Regular Expressions

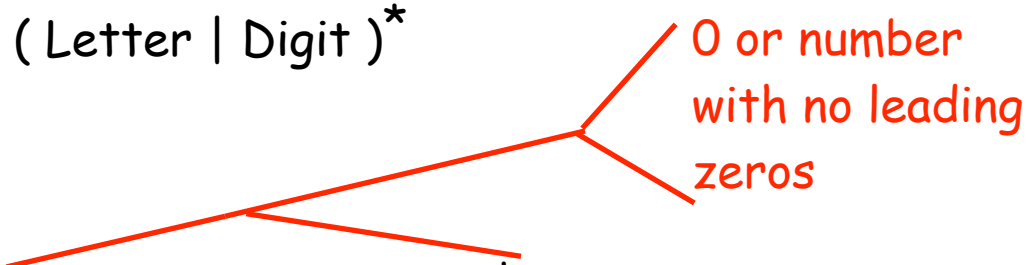
Identifiers:

Letter → (a|b|c| … |z|A|B|C| … |Z)

Digit → (0|1|2| … |9)

Identifier → Letter (Letter | Digit)*

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!

Regular Expressions (the point)

Regular expressions can be used to specify the words to be translated to parts of speech by a lexical analyzer

Using results from automata theory and theory of algorithms, we can automatically build recognizers from regular expressions

⇒ We study REs and associated theory to automate scanner construction !

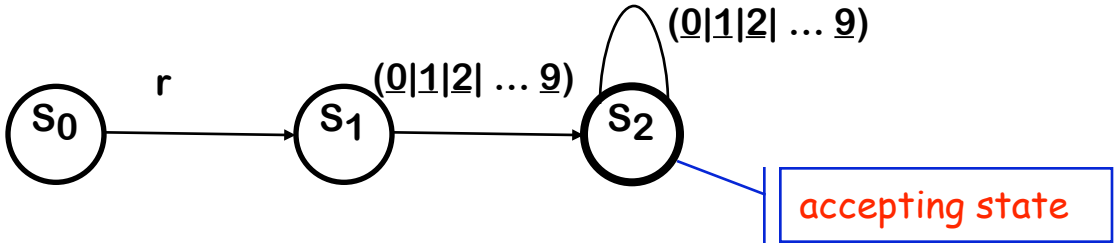
Example

Consider the problem of recognizing ILOC register names

$$\text{Register} \rightarrow r\ (\underline{0}|\underline{1}|\underline{2}|\ \ldots\ |\ \underline{9})\ \ (\underline{0}|\underline{1}|\underline{2}|\ \ldots\ |\ \underline{9})^*$$

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

RE corresponds to a recognizer (or DFA)



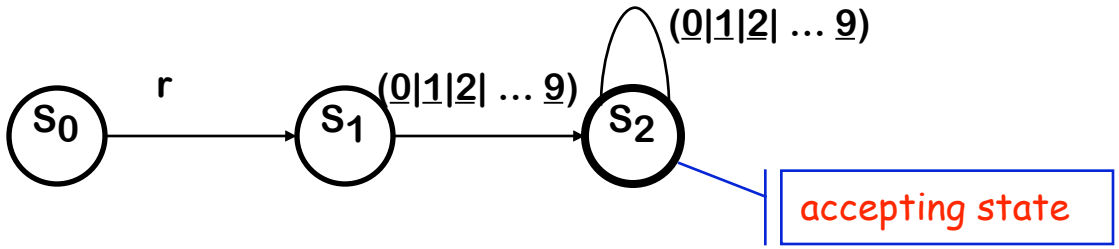
Recognizer for Register

Transitions on other inputs go to an error state, s_e

Example (continued)

DFA operation

- Start in state S_0 & take transitions on each input character
- DFA accepts a word x iff 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, recognizer must turn 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

δ	r	0,1,2,3,4, 5,6,7,8,9	All others
s_0	s_1	s_e	s_e
s_1	s_e	s_2	s_e
s_2	s_e	s_2	s_e
s_e	s_e	s_e	s_e

Table encoding RE

Example (continued)

To be useful, recognizer must turn into code

```
Char ← next character
State ← s0

while (Char ≠ EOF)
    State ← δ(State,Char)
    perform specified action
     Char ← next character

if (State is a final state )
    then report success
    else  report failure
```

Skeleton recognizer

δ	r	0,1,2,3,4, 5,6,7,8,9	All others
s_0	s_1 start	s_e error	s_e error
s_1	s_e error	s_2 add	s_e error
s_2	s_e error	s_2 add	s_e error
s_e	s_e error	s_e error	s_e error

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 | ε) | (4|5|6|7|8|9) | (3|30|31))

→ Register → r0|r1|r2| … |r31|r00|r01|r02| … |r09

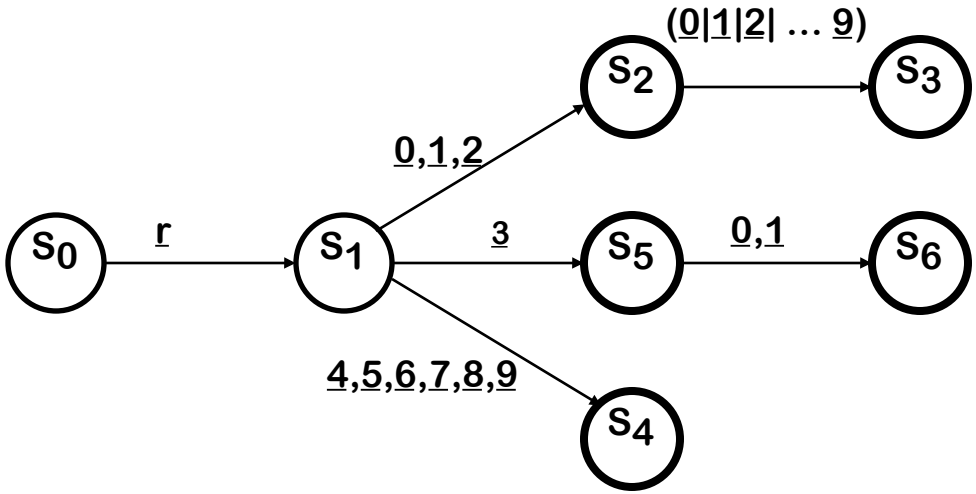
Produces a more complex DFA

- Has more states
- Same cost per transition
- Same basic implementation

Tighter register specification (continued)

The DFA for

Register → r ((0|1|2) (Digit | ε) | (4|5|6|7|8|9) | (3|30|31))



- Accepts a more constrained set of registers
- Same set of actions, more states

Tighter register specification (continued)

δ	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

Runs in the same skeleton recognizer

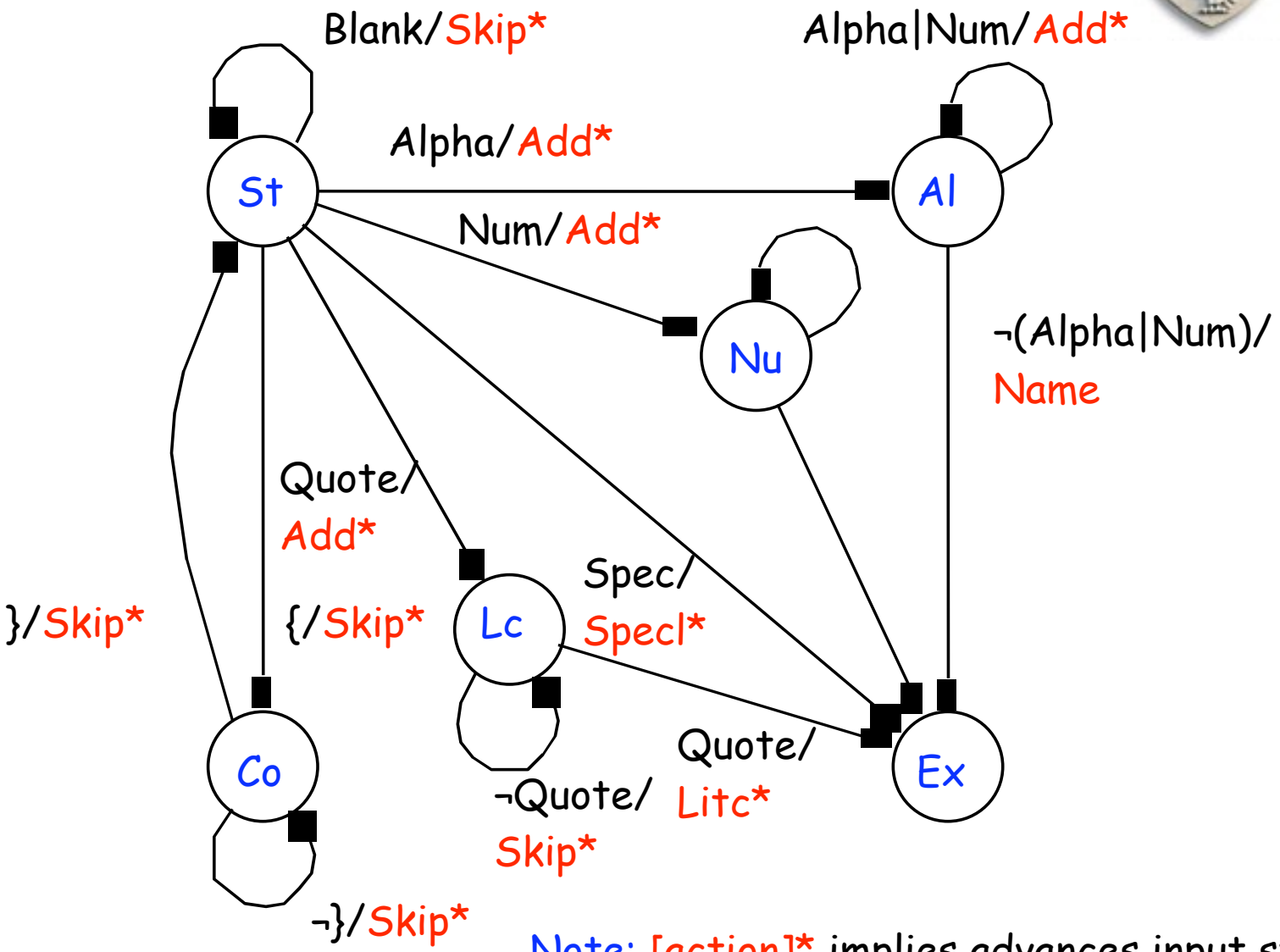
Table encoding RE for the tighter register specification

Extra Slides Start Here

Principles of Scanners

- Lexical Analysis Strategy: Simulation of Finite Automaton
  - → States, characters, actions
  - → State transition δ(state,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 δ)
    - 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]* implies advances input stream

Example 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);
}
```

Tighter register specification (continued)

state action	r	0,1	2	3	4,5,6 7,8,9	other
0	1 start	e	e	e	e	e
1	e	2 add	2 add	5 add	4 add	e
2	e	3 add	3 add	3 add	3 add	x exit
3,4	e	e	e	e	e	x exit
5	e	6 add	e	e	e	x exit
6	e	e	e	e	e	x exit
e	e	e	e	e	e	e