RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

CS415 Compilers

Lexical Analysis
Part 2

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University

RUTGERS

Announcements

- Second homework due this Friday, February 18.

- First project (local instruction scheduler) has been posted
  Deadline for code: March 2
  Deadline for report: March 4

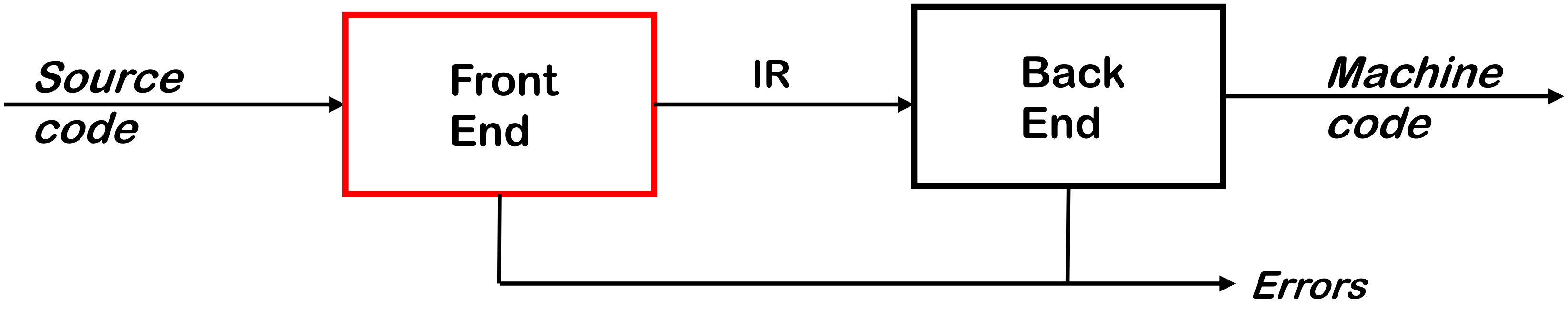
- First midterm: Wednesday, February 23
  In class exam, 60 minutes,
  Topics: ILOC, instruction scheduling, register allocation

- Spring recess: March 12 - 20

- Final exam (exam code C): Tuesday, May 10
  1:00pm – 2:00pm
  In person, location TBD

Review - The Front End

EaC Chapter 2

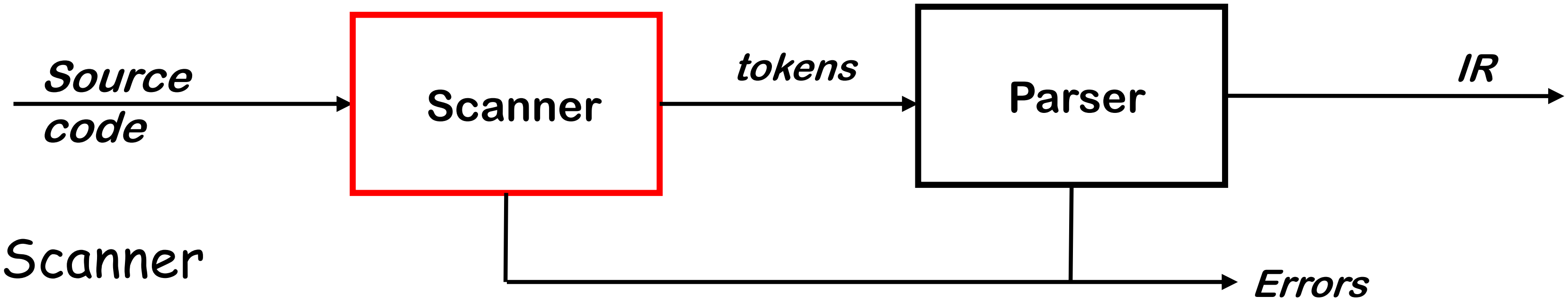


The purpose of the front end is to deal with the input language

- 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

Review - The Front End



Scanner

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

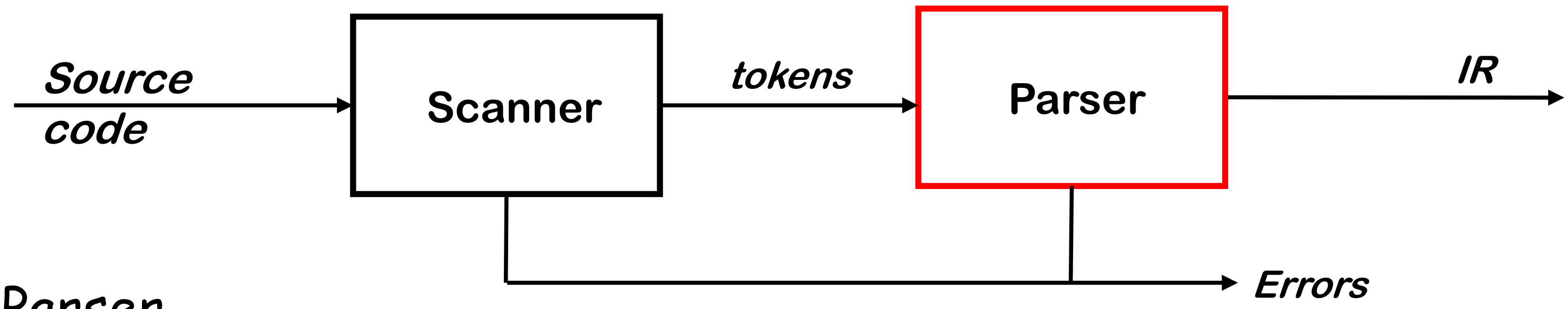
    ‹id,x› ‹eq,=› ‹id,x› ‹pl,+› ‹id,y› ‹sc,; ›

Speed is an issue in scanning

⇒ use a specialized recognizer

- Character sequence that forms a word/token is its *lexeme*
- Its *part of speech* (or *syntactic category*) is called its *token type*
- Scanner discards white space & (often) comments

Review - The Front End



Parser

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

We'll get to parsing in the next lectures

Parser - The Big Picture

- Language syntax is specified over *parts of speech* (tokens)
- Syntax checking matches *sequence of tokens* against a grammar
- Here is an example context free grammar (CFG) *G*:

1.	*goal*	$\rightarrow$	*expr*
2.	*expr*	$\rightarrow$	*expr op term*
3.		|	*term*
4.	*term*	$\rightarrow$	number
5.		|	id
6.	*op*	$\rightarrow$	+
7.		|	-

S = *goal*

T = { number, id, +, - }

N = { *goal*, *expr*, *term*, *op* }

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

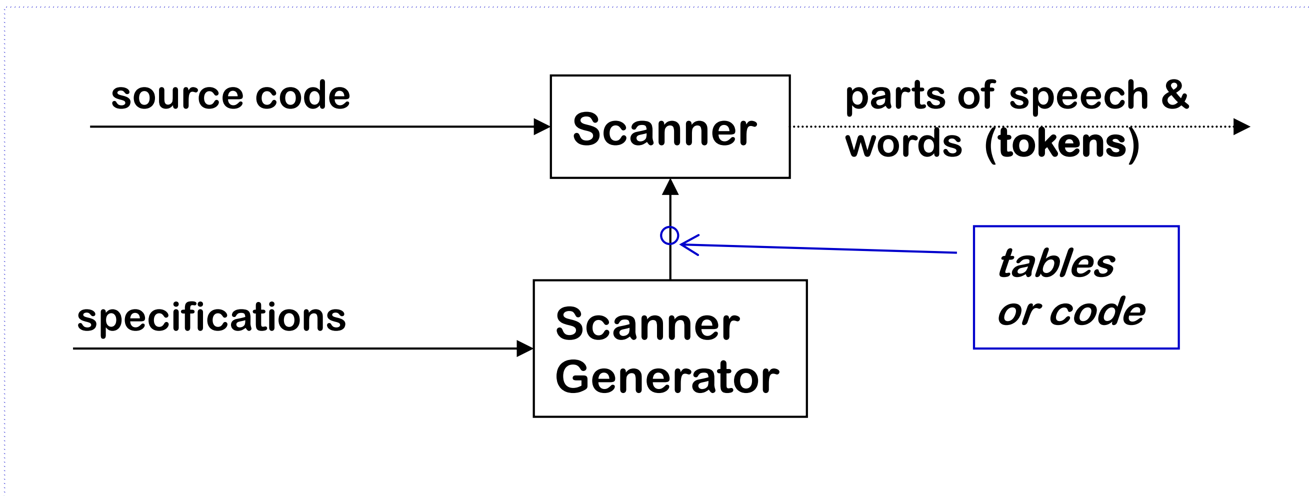
G in BNF form

G = (S, T, N, P)

Scanner - The Big Picture

Why study lexical analysis?

- We want to avoid writing scanners by hand



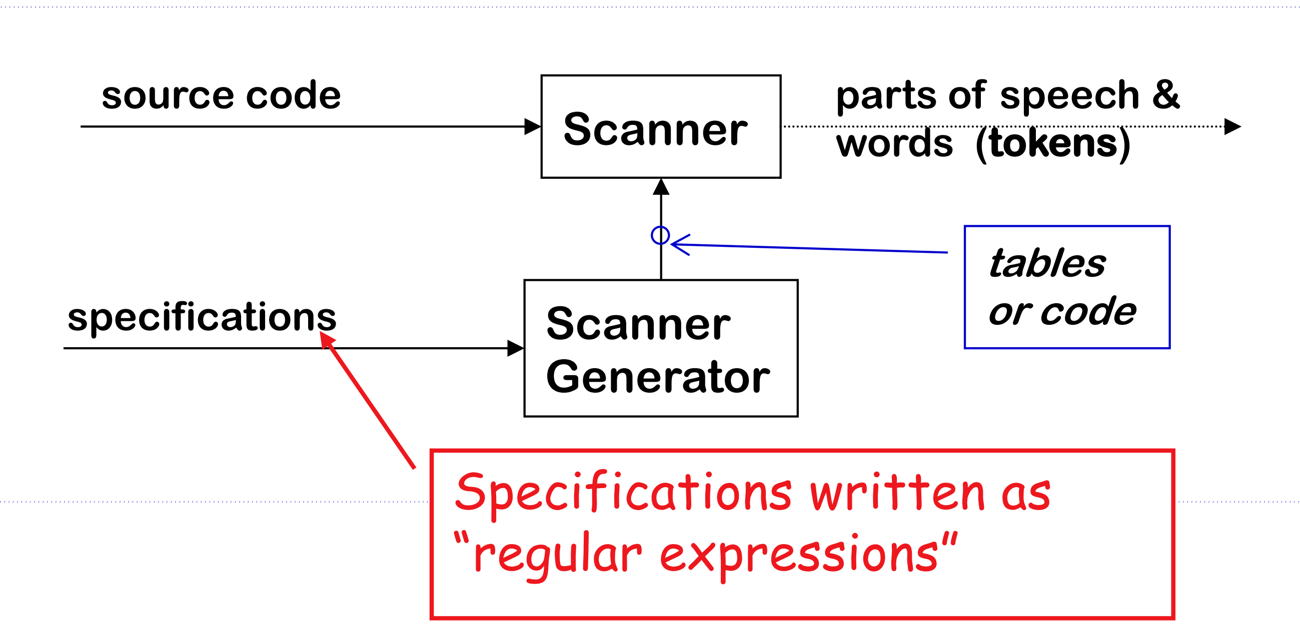
Goals:

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

The Big Picture

Why study lexical analysis?

- We want to avoid writing scanners by hand



Goals:

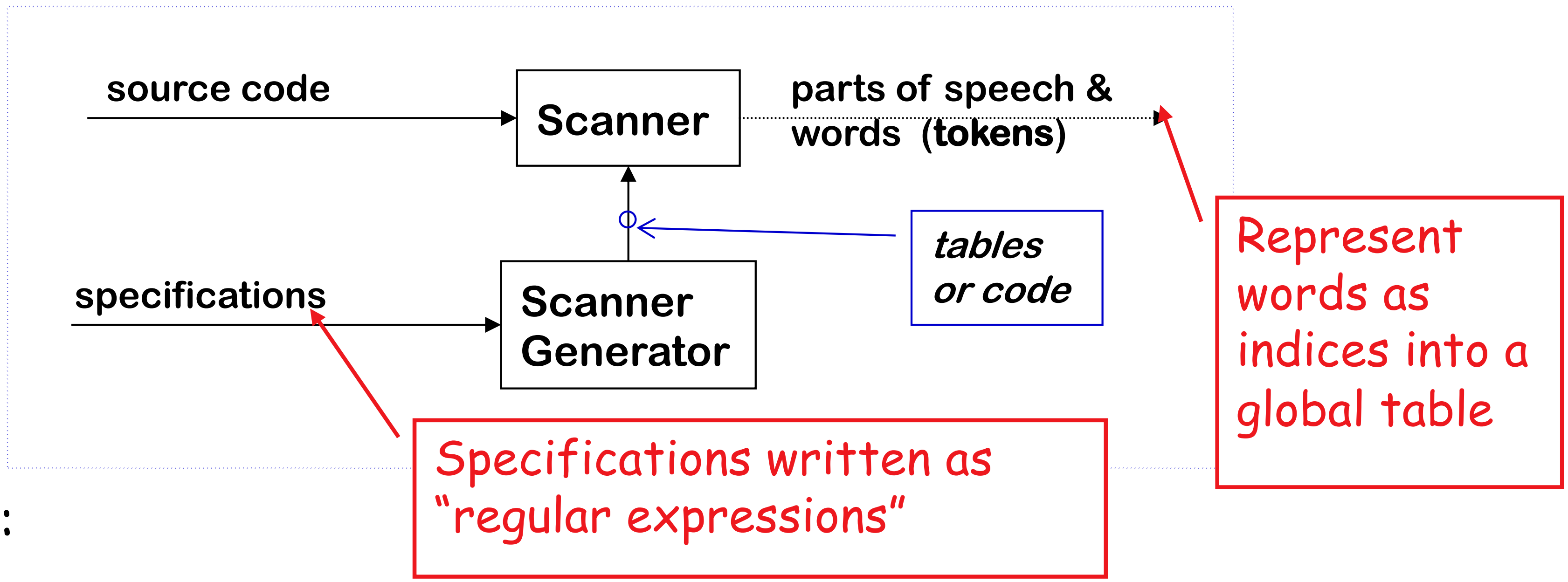
→ To simplify specification & implementation of scanners

→ To understand the underlying techniques and technologies

The Big Picture

Why study lexical analysis?

- We want to avoid writing scanners by hand



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 an alphabet Σ, a finite set of symbols):

- ε is a RE denoting the set $\{\varepsilon\}$
- If "a" is in Σ, then $\underline{a}$ is a RE denoting $\{a\}$
- If x and y are REs denoting $L(x)$ and $L(y)$ then
  - → $x \mid y$ is an RE denoting $L(x) \cup L(y)$
  - → xy is an RE denoting $L(x)L(y)$
  - → x^* is an RE denoting $L(x)^*$
  - → (x) is an RE denoting $L(x)$

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

Set Operations

Operation	Definition
Union of L and M *Written $L \cup M$*	$L \cup M = \{s \mid s \in L \text{ or } s \in M\}$
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 (tokens) by a lexical analyzer

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

$\Rightarrow$ We study REs and associated theory to automate scanner construction !

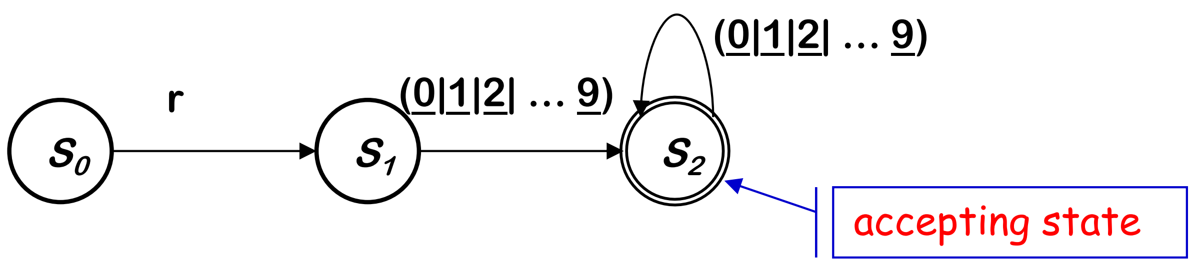
Example

Consider the problem of recognizing ILOC register names

Register → r (0|1|2| … | 9) (0|1|2| … | 9)*

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

RE corresponds to a recognizer (or DFA)



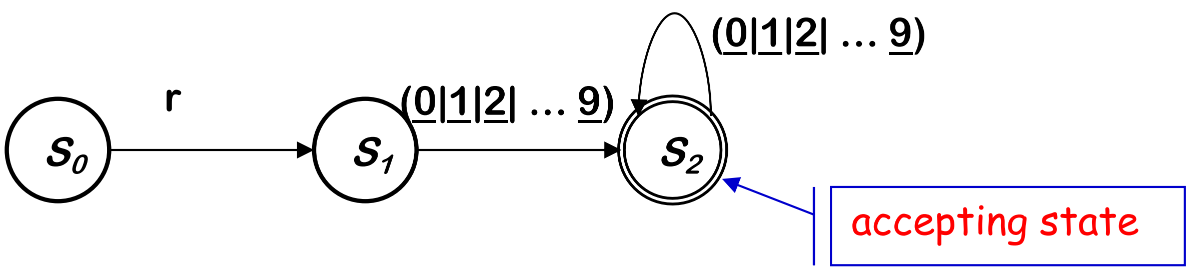
Recognizer for *Register*

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 <u>x</u> *iff* <u>x</u> 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 error state s_e (not shown here)

Example (continued)

To be useful, recognizer must turn into code

$\delta(s_x, a) = s_y$

```
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

RUTGERS

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

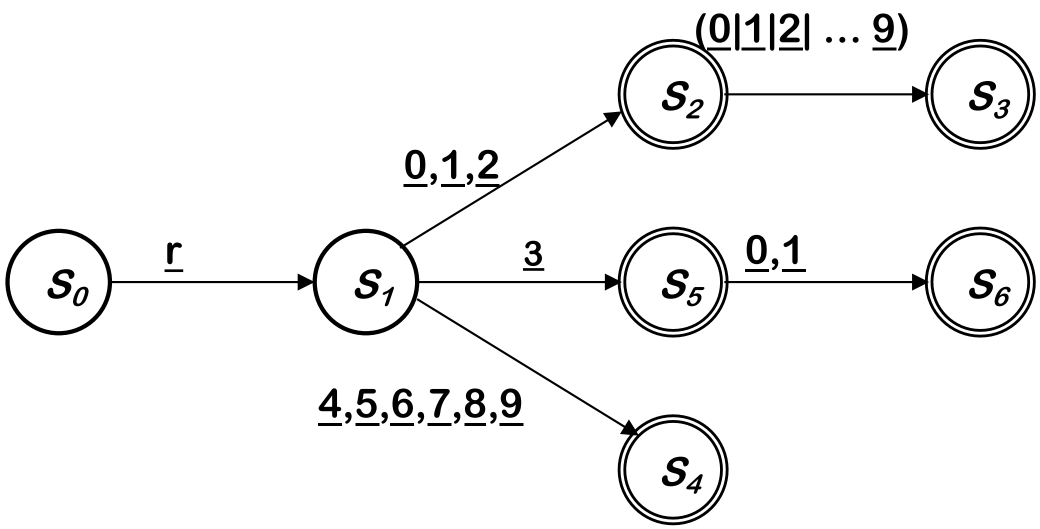
Produces a more complex DFA

- 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

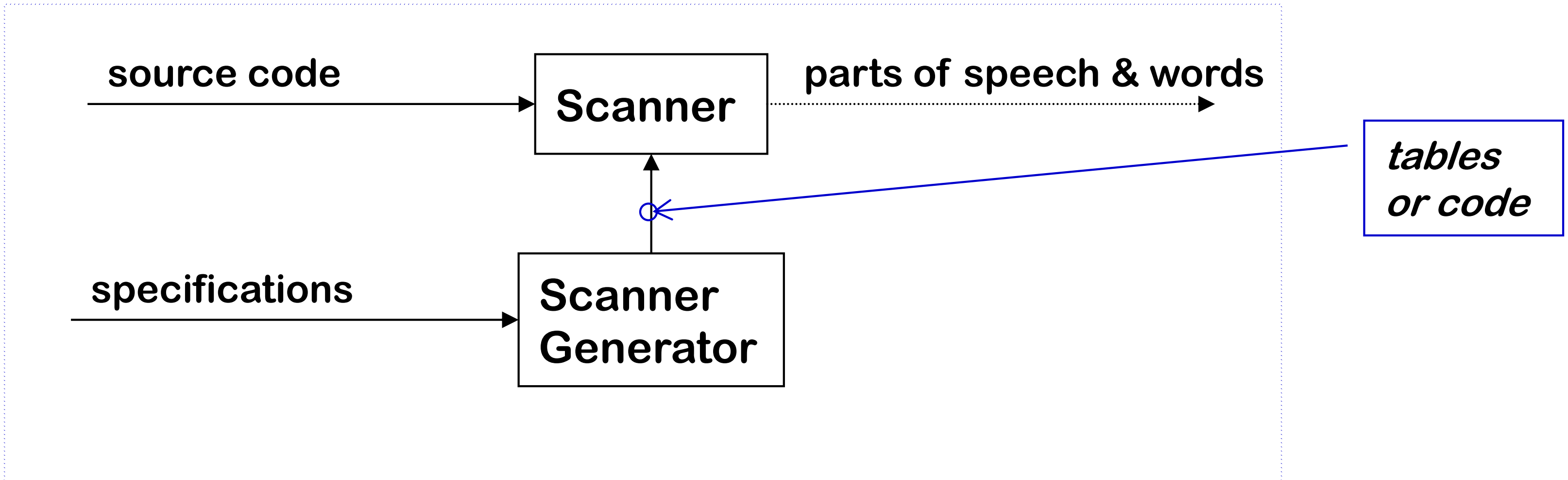
Tighter register specification (continued)

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

Constructing a Scanner - Quick Review



- The scanner is the first stage in the front end
- Specifications can be expressed using regular expressions
- Build tables and code from a DFA

Next class

More Lexical Analysis

Syntax Analysis (top-down)

Read EaC: Chapter 3.1 – 3.3