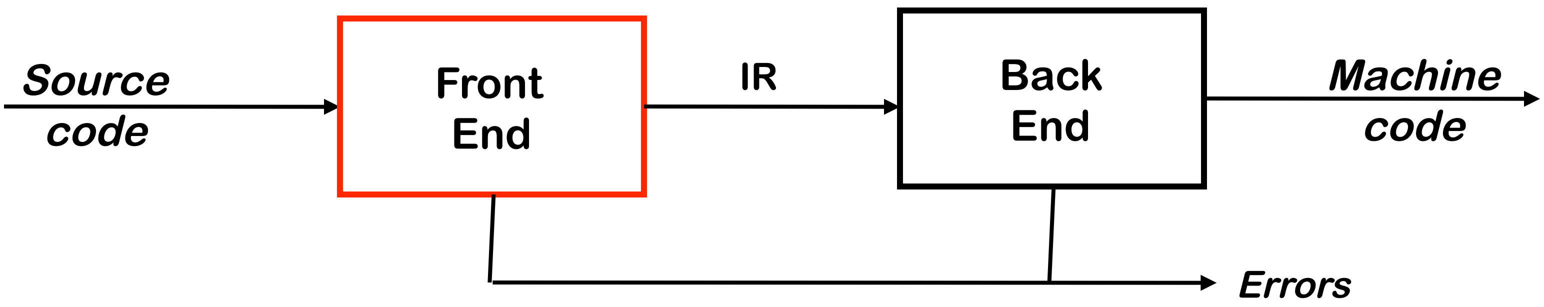
Lexical Analysis - An Introduction

The Front End

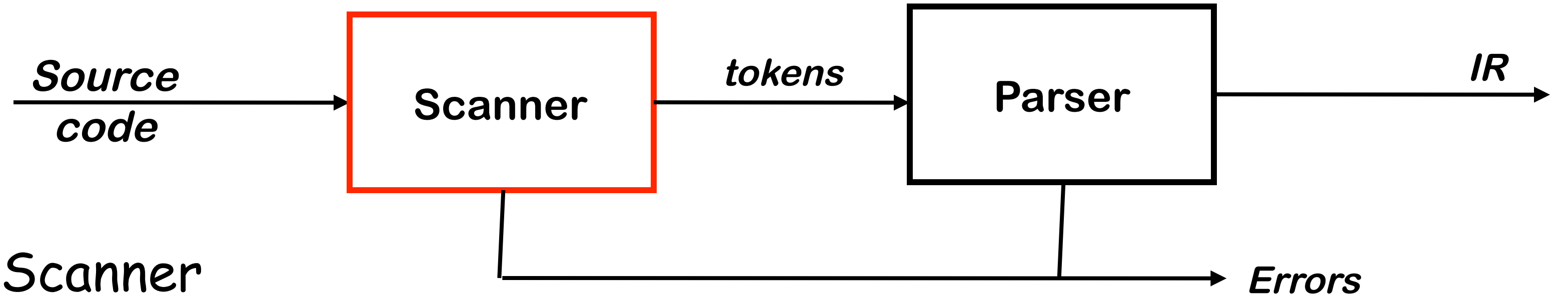


The purpose of the front end is to deal with the input language

- Perform a membership test: code $\in$ 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

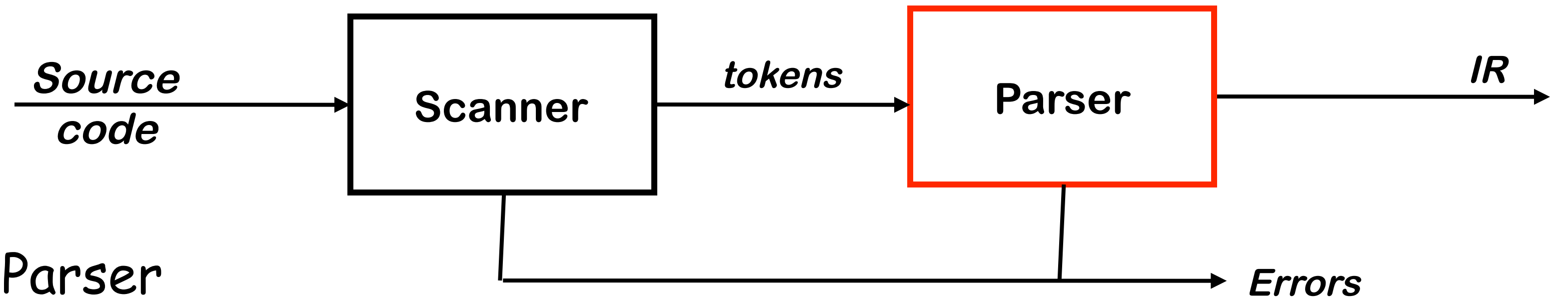
- Maps stream of characters into words
  - → Basic unit of syntax
  - → x = x + y ; *becomes*

    <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

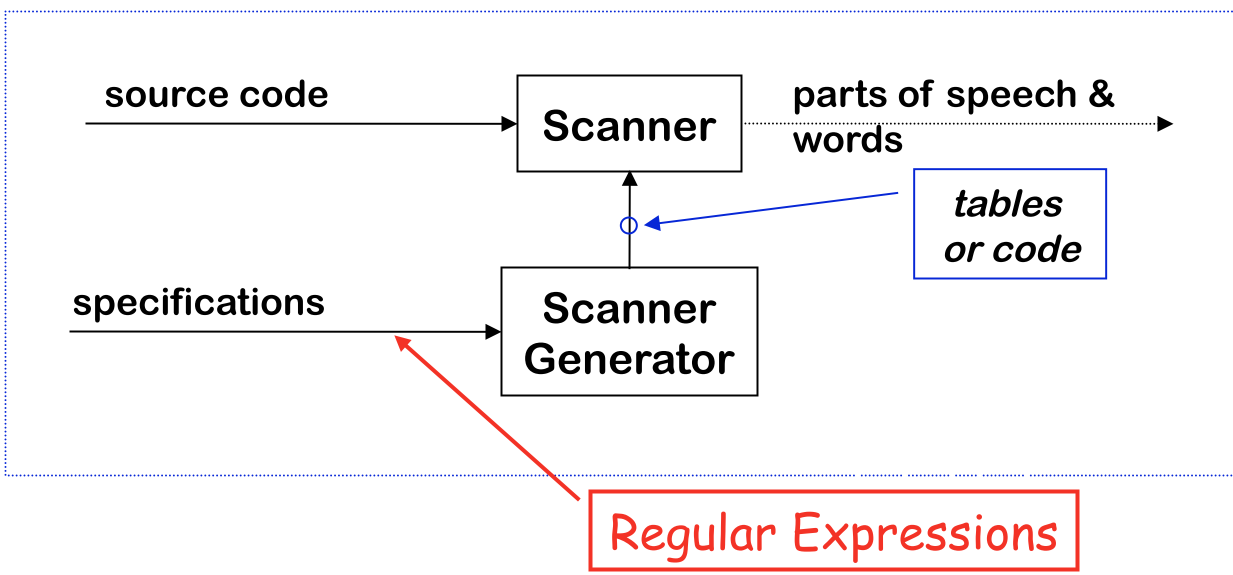
The Big Picture

Why study lexical analysis?

- We want to avoid writing scanners by hand
- We want to harness the theory from classes like CISC 303

Goals:

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



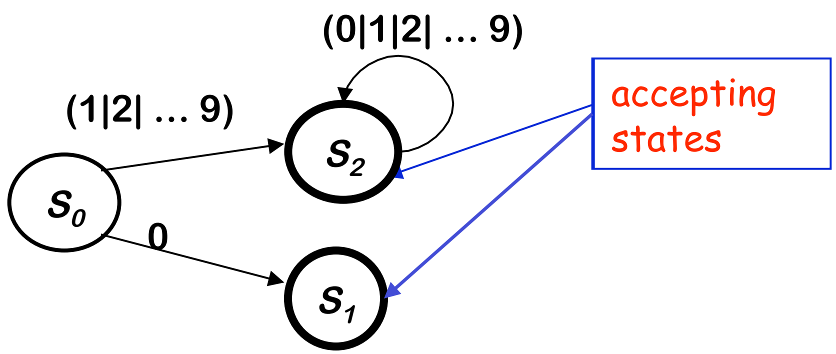
Where is Lexical Analysis Used?

For traditional languages but where else…

- Web page "compilation"
  - Lexical Analysis of HTML, XML, etc.
- Natural Language Processing
- Game Scripting Engines
- OS Shell Command Line
- GREP
- Prototyping high-level languages
- JavaScript, Perl, Python

Recognizing Words

Finite Automaton (FA) – recognizers that can scan a stream of symbols to find words



Transition Diagram for *Number*

- An FA is a five-tuple $(S,\Sigma,\partial,s_0,S_F)$ where
  - S is the set of states
  - Σ is the alphabet
  - ∂ a set of transition functions where each takes a state and a character and returns another state
  - s_0 is the start state
  - S_F is the set of final states

Regular Expressions

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
  - → *Closure:* x^* is an RE denoting $L(x)^*$
  - → *Concatenation:* xy is an RE denoting $L(x)L(y)$
  - → *Alternation:* $x \mid y$ is an RE denoting $L(x) \cup L(y)$

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

Set Operations (review)

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 by a lexical analyzer

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

You may have seen this construction in a Automata Course

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

Regular Expression Class Problem?

What is the regular expression for a register name?

Examples: r1, r25, r999   ← These are OK.

r, s1, a25   ← These are <u>not</u> OK.

Register Name RE Solution

Consider the problem of recognizing register names

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

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

Register Name DFA Class Problem?

Consider the problem of recognizing register names

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

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

What does the DFA look like?

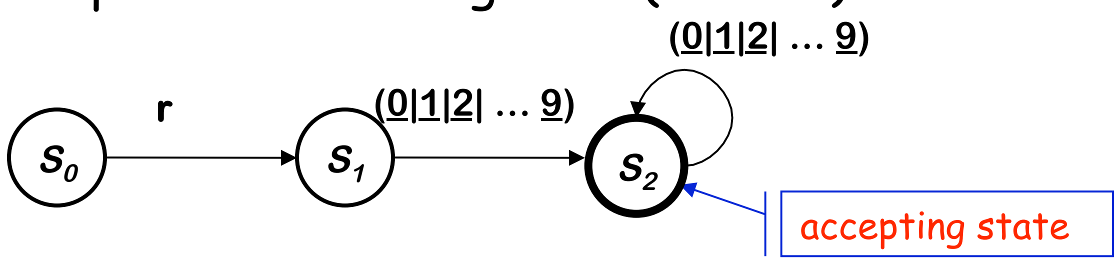
Register Name DFA Solution

Consider the problem of recognizing 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)



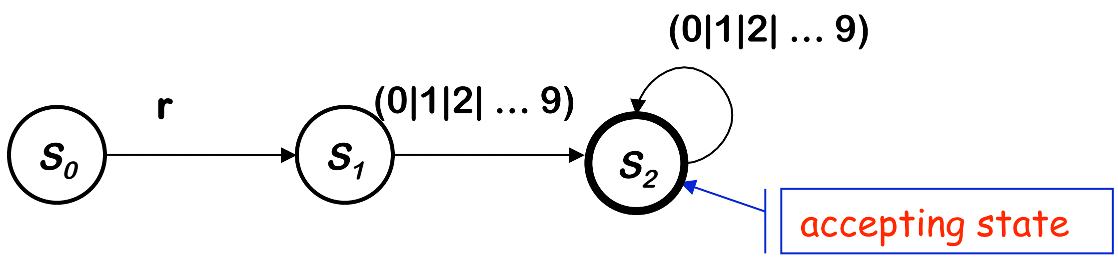
Recognizer for *Register*

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,

- <u>r17</u> takes it through s_0, s_1, s_2 and accepts
- <u>r</u> takes it through s_0, s_1 and fails
- <u>a</u> 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

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

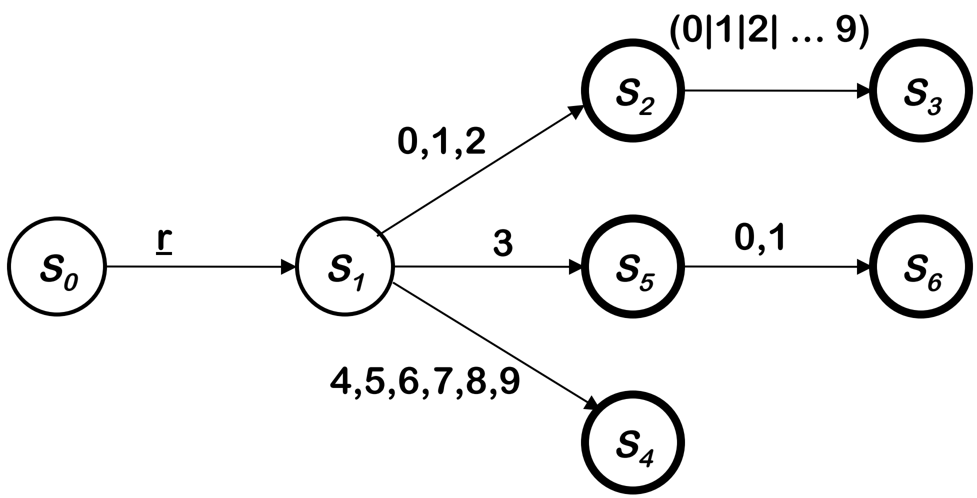
Produces a more complex DFA

- 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