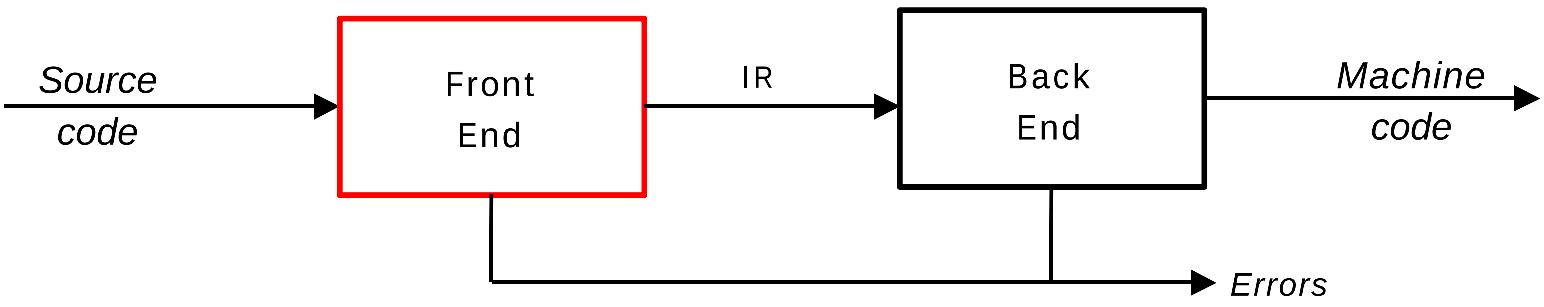
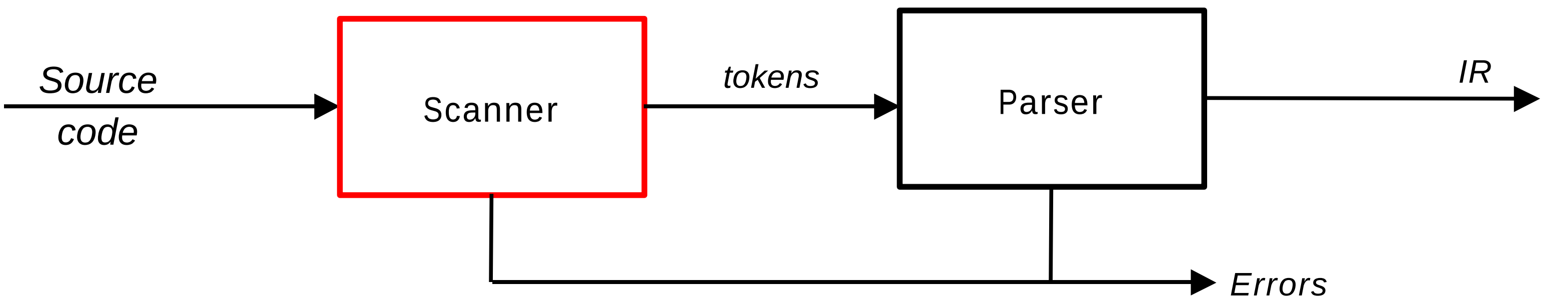
The Front End



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 (syntactically) ?
- Build an IR version of the code for the rest of the compiler



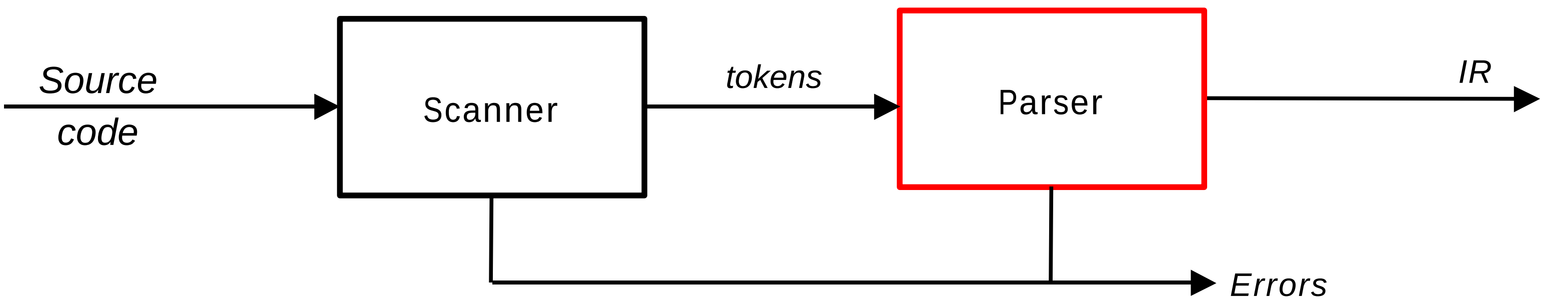
Scanner

- Maps stream of characters into words
  - > Basic unit of syntax
  - > x = x + y ; *becomes* <id,x> <assignop,=> <id,x> <arithop,+> <id,y> ;
- Characters that form a word are its *lexeme*
- Its *part of speech* (or *syntactic category*) is called its *token*
- 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

In natural languages, *word* ® *part of speech* is idiosyncratic

> Based on connotation & context

> Typically done with a table lookup

In formal languages, *word* ® *part of speech* is syntactic

> Based on denotation

> Makes this a matter of syntax, or *micro-syntax*

> We can recognize this micro-syntax efficiently

> Reserved keywords are critical (no context!)

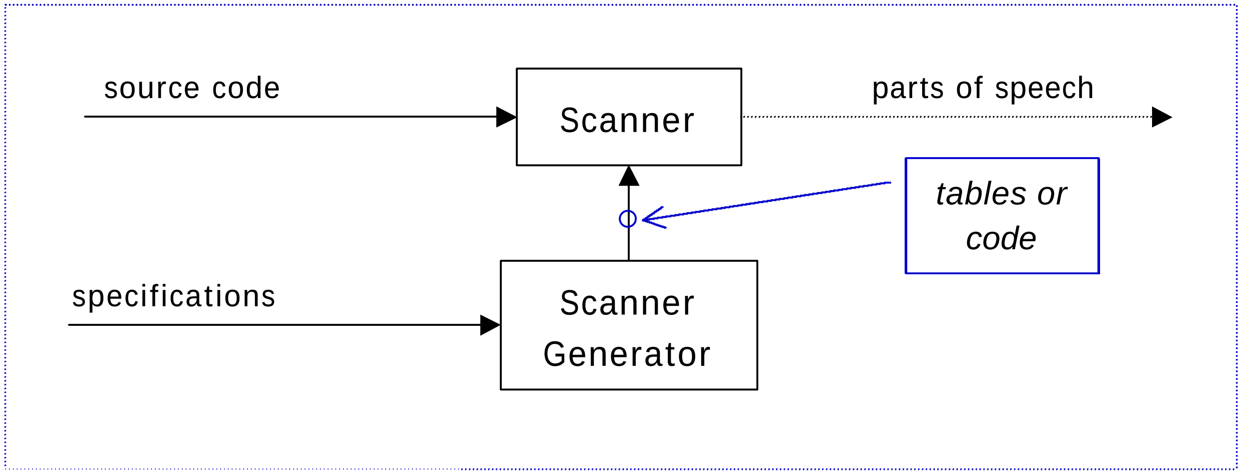
⟹ Fast recognizers can map *words* into their *parts of speech*

⟹ Study formalisms to automate construction of recognizers

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

Specifying Lexical Patterns *(micro-syntax)*

A scanner recognizes the language's parts of speech

Some parts are easy

- White space
  - *WhiteSpace* → blank | tab | *WhiteSpace* blank | *WhiteSpace* tab
- Keywords and operators
  - Specified as literal patterns: if, then, else, while, =, +, ...
- Comments
  - Opening and (*perhaps*) closing delimiters
  - /* *followed by* */ in C
  - // in C++
  - % in LaTeX

Specifying Lexical Patterns *(micro-syntax)*

A scanner recognizes the language's parts of speech

Some parts are more complex

- Identifiers
  - > Alphabetic followed by alphanumerics + _, &, $, ...
  - > May have limited length
- Numbers
  - > Integers: 0 *or* a digit from 1-9 followed by digits from 0-9
  - > Decimals: integer . digits from 0-9, *or* . digits from 0-9
  - > Reals: (integer or decimal) E (+ *or* -) digits from 0-9
  - > Complex: (real , real)

We need a notation for specifying these patterns

We would like the notation to lead to an implementation

Regular Expressions

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 a is in Σ, then a is a RE denoting {a}
- If *x* and *y* are REs denoting *L(x)* and *L(y)* then
  - > *x* is a RE denoting *L(x)*
  - > *x* |*y* is a RE denoting *L(x)* È *L(y)*
  - > *xy* is a RE denoting *L(x)L(y)*
  - > x^* is a RE denoting *L(x)**

Precedence is
closure, then
concatenation, then
alternation

Set Operations *(refresher)*

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

You need to know these definitions

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)*

To make scanning tractable, programming languages differentiate between parts of speech by controlling their spelling (as opposed to dictionary lookup)

Difference between *Identifier* and *Keyword* is entirely lexical

> While is a *Keyword*
>
> Whilst is an *Identifier*

The lexical patterns used in programming languages are regular

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

⟹ We study REs to automate scanner construction !

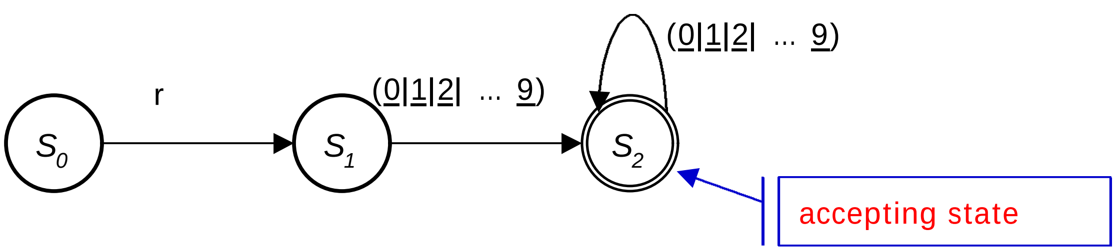
Example

Consider the problem of recognizing register names

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

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

RE corresponds to a recognizer (or DFA)



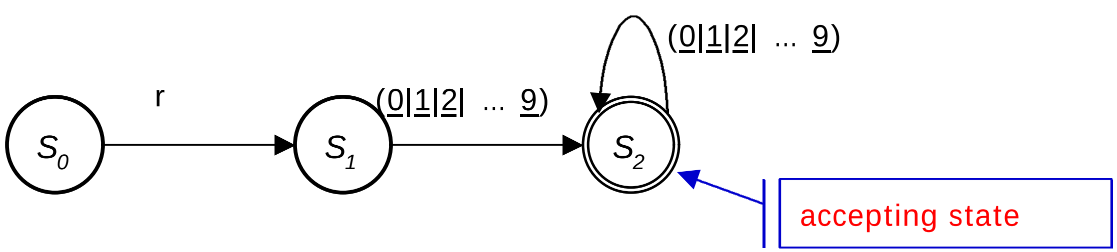
Recognizer for *Register*

With implicit transitions on other inputs 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)

```
char ¬ next character;
state ¬ s0 ;
call action(state,char);
while (char ¹ eof)
  state ¬ d(state,char);
  call action(state,char);
  char ¬ next character;

if T(state) = final then
  report acceptance;
else
   report failure;
```

```
action(state,char)
  switch(T(state) )
    case start:
      word ¬ char;
       break;
    case normal:
      word ¬ word + char;
       break;
    case final:
      word ¬ char;
       break;
    case error:
       report error;
       break;
    end;
```

T	action
s_0	start
s_1	normal
s_2	final
s_e	error

δ	r	0,1, 2,3, 4,5, 6, 7,8, 9	other
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

- *The recognizer translates directly into code*
- *To change DFAs, just change the tables*

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

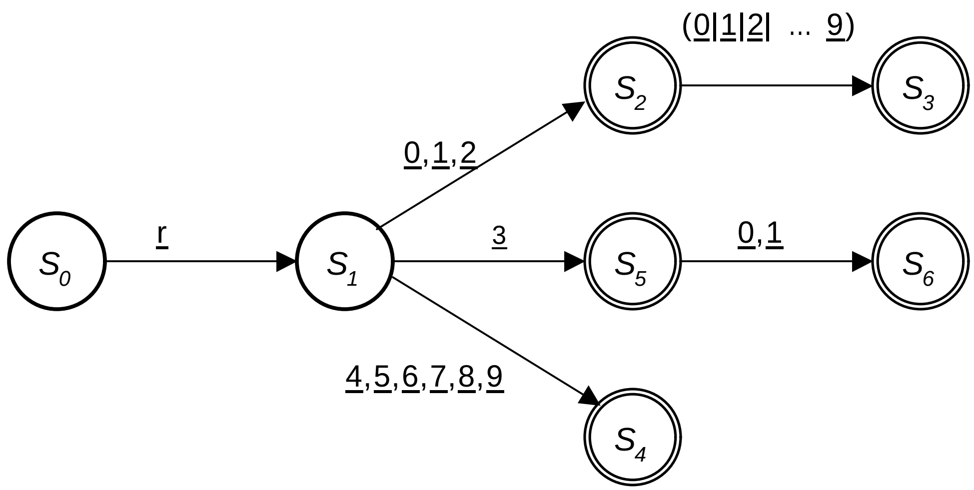
Produces a more complex DFA

- 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)*

To implement the recognizer

- Use the same code skeleton
- Use transition and action tables for the new RE

δ	r	0,1	2	3	4,5,6 7,8,9	*other*
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

T	*action*
S_0	start
S_1	normal
$S_{2,3,4,5,6}$	final
S_e	error

- Bigger tables, more space, same asymptotic costs
- Better (micro-)syntax checking at the same cost