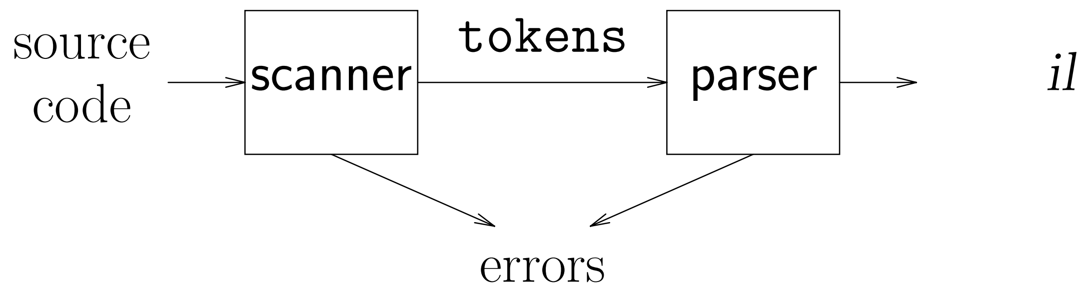
Class Information

- New accounts on ilab cluster: Interface to create new accounts is up and running.
- Last special permission numbers have been sent out this morning.
- First homework is due on Friday. Hardcopy to be handed in **before** class starts. Late submissions may or may not be graded.
- Office hours have been posted. They start this week. **You can go to any 314 office hour**.

Review - Front end of a compiler



Front End: syntax & (static) semantics analyzer, *il* code generator (syntax-directed translation)

Front End Responsibilities:

- recognize legal programs
- report errors
- produce *il*
- preliminary storage map
- shape the code for the back end

Much of front end construction can be automated

Review: Syntax and Semantics of Prog. Languages

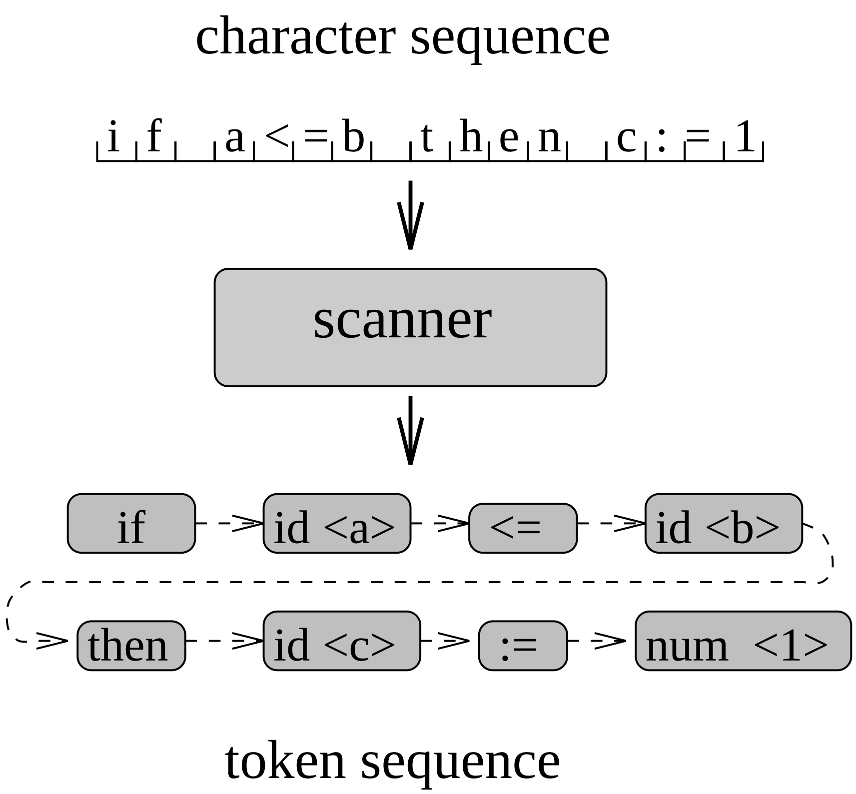
The syntax of programming languages is often defined in two layers: *tokens* and *sentences*.

- *tokens* – basic units of the language
  Question: $\boxed{\text{How to spell a token (word)?}}$
  Answer: regular expressions

- *sentences* – legal combination of tokens in the language
  Question:
  $\boxed{\text{How to build correct sentences with tokens?}}$
  Answer: (context-free) grammars (CFG) E.g.,

  Backus-Naur form (BNF) is a formalism used to express the syntax of programming languages.

Review: Lexical Analysis (Scott 2.1, 2.2)



Tokens (Terminal Symbols of CFG, Words of Lang.)

- Smallest "atomic" units of syntax
- Used to build all the other constructs
- Example, Pascal:

  keywords: `program begin if then` ...

  `= * / - < > = <= >= <>`

  `( ) [ ] ; := . ,` ...

  `number` (Example: `3.14 28` ...)

  `identifier` (Example: `b square addEntry` ...)

Review: Regular Expressions

A syntax (notation) to specify regular languages.

RE r	Language L(r)
a	$\{\mathbf{a}\}$
ϵ	$\{\epsilon\}$
r | s	$L(r) \cup L(s)$
rs	$\{rs \mid r \in L(r), s \in L(s)\}$
r^+	$L(r) \cup L(rr) \cup L(rrr) \cup \ldots$ (any number of r's concatenated)
r^* ($r^* = r^+ \mid \epsilon$)	$\{\epsilon\} \cup L(r) \cup L(rr) \cup L(rrr) \cup \ldots$
(s)	$L(s)$

(all left-assoc. in order of increasing precedence.)

$\Rightarrow$ Note: Inductive definition!

Regular Expressions for Token Definitions

Let **letter** stand for A | B | C | … | Z
Let **digit** stand for 0 | 1 | 2 | … | 9

integer constant:

identifier:

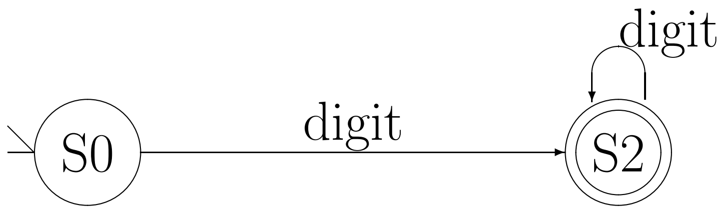
real constant:

Recognizers for Regular Expressions

Example 1: integer constant

RE: digit^+

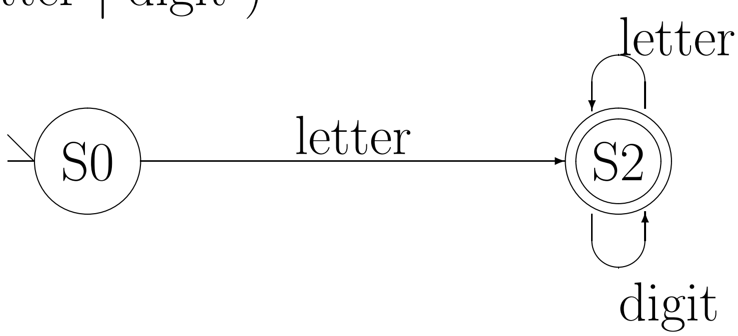
FSA:



Example 2: identifier

RE: $\text{letter} (\text{letter} \mid \text{digit})^*$

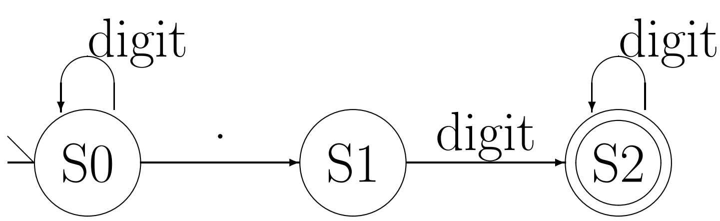
FSA:



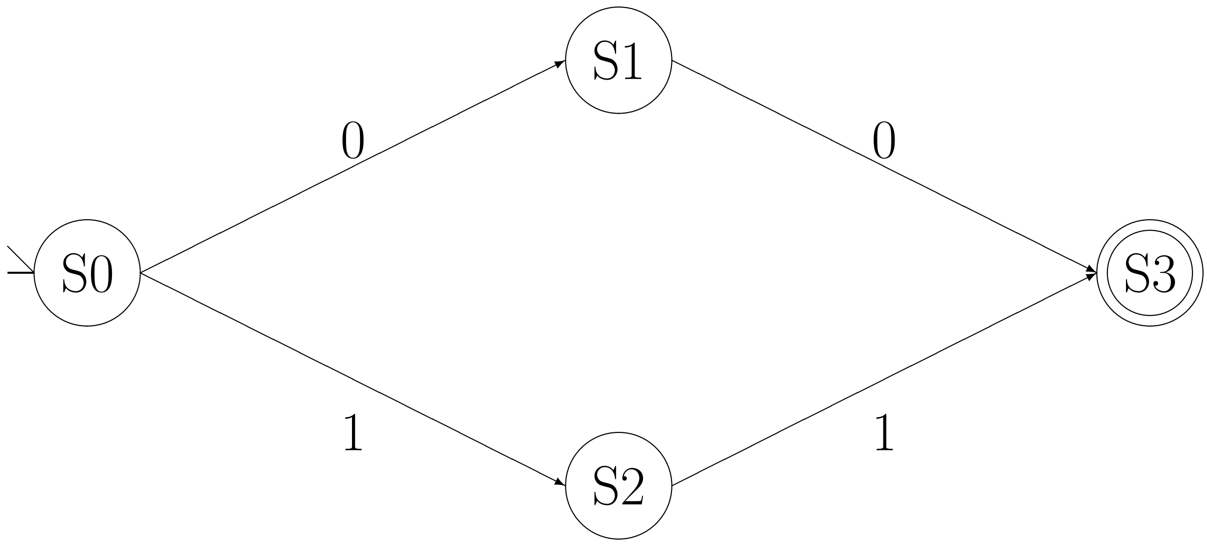
Example 3: Real constant

RE: $\text{digit}^*.\text{digit}^+$

FSA:



Finite State Automata



A Finite-State Automaton is a quadruple:
$< S, s, F, T >$

- S is a set of *states*, e.g., $\{S0, S1, S2, S3\}$
- s is the *start state*, e.g., $S0$
- F is a set of *final states*, e.g., $\{S3\}$
- T is a set of *labeled transitions*, of the form $(state, input) \mapsto state$
  [i.e., $S \times \Sigma \to S$]

Finite State Automata

Transitions can be represented using a **transition table**:

		0	1	Input
State	S0	S1	S2	
	S1	S3	-	
	S2	-	S3	

An FSA *accepts* or *recognizes* an input string iff there is some path from its start state to a final state such that the labels on the path are that string.

Lack of entry in the table (or no arc for a given character) indicates an error—reject.

Practical Recognizers

- recognizer should be a deterministic finite automaton (DFA)
- try to find longest input-string that can make up a token ($\rightarrow$ may read beyond end of token)
- report errors (error recovery?)

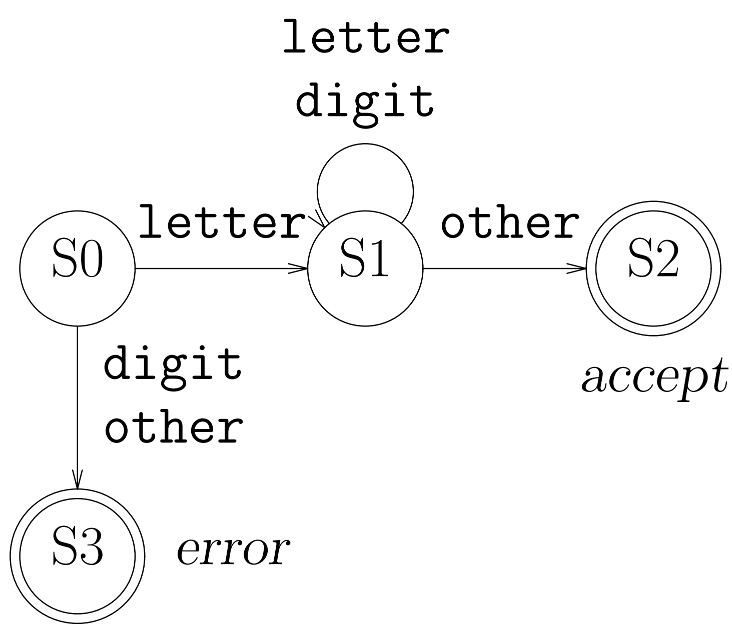
identifier

$$letter \rightarrow (a \mid b \mid c \mid ... \mid z \mid A \mid B \mid C \mid ... \mid Z)$$

$$digit \rightarrow (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)$$

$$id \rightarrow letter\ (letter \mid digit)^*$$

Recognizer for *identifier*: (transition diagram)



Implementation: Tables for the recognizer

Two tables control the recognizer.

`char_class`:

	$a - z$	$A - Z$	$0 - 9$	other
class	letter	letter	digit	other

`next_state`:

class	S0	S1	S2	S3
letter	S1	S1	—	—
digit	S3	S1	—	—
other	S3	S2	—	—

To change languages, we can just change tables.

Implementation: Code for the recognizer

```
char ← next_char();
state ← S0;          /* code for S0 */
done ← false;
token_value ← ""    /* empty string */
while( not done ) {
   class ← char_class[char];
   state ← next_state[class,state];
   switch(state) {
      case S1:      /* building an id */
         token_value ← token_value + char;
         char ← next_char();
         break;
      case S2:      /* accept state */
         token_type = identifier;
         done = true;
         break;
      case S3:      /* error */
         token_type = error;
         done = true;
         break;
   }
}
return token_type;
```

Improved efficiency

Table driven implementation is slow relative to direct code. Each state transition involves:

1. classifying the input character
2. finding the next state
3. an assignment to the state variable
4. a trip through the case statement logic
5. a branch (while loop)

We can do better by "encoding" the state table in the scanner code.

1. classify the input character
2. test character class locally
3. branch directly to next state

This takes many fewer instructions per cycle.

Implementation: Faster scanning

```
S0:  char ← next_char();
     token_value ← ""   /* empty string */
     class ← char_class[char];
     if (class != letter)
        goto S3;

S1:  token_value ← token_value + char;
     char ← next_char();
     class ← char_class[char];
     if (class != other)
        goto S1;

S2:  token_type = identifier;
     return token_type;

S3:  token_type ← error;
     return token_type;
```

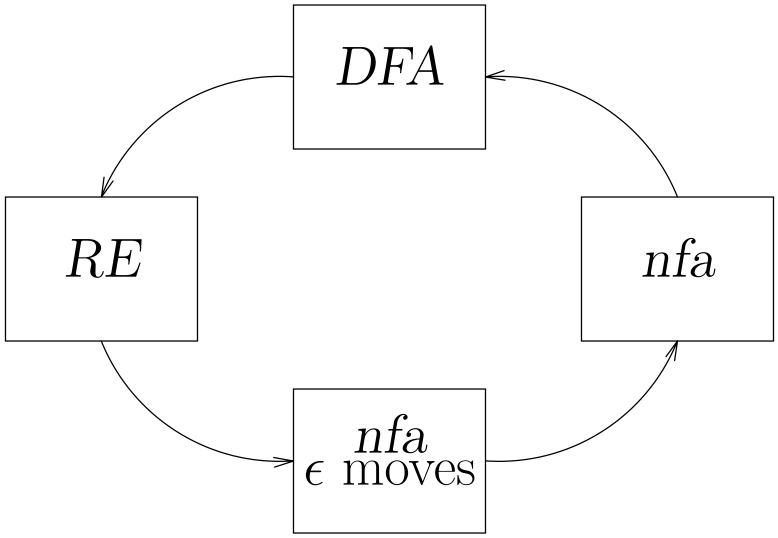
What do we want?

Ideally: The language/compiler designer specifies the tokens using a regular expression, and some automatic tool (scanner generator) produces code that implements the scanner.

$\boxed{\text{How can this be done?}}$

Note: In practice, there are a few more issues that we are not discussion here. For example, how to make sure that a keyword is not recognized as an identifier.

Constructing a *DFA* from a regular expression



regular expression (RE) →*nfa* w/ϵ moves

  build *nfa* for each term

  connect them with ϵ moves

nfa w/ϵ moves to *NFA*

  coalesce states

nfa→*DFA*

  construct the simulation ("subset" construction)

  minimize DFA (DFA with minimal number of states)

dfa→ regular expression

  construct $R_{ij}^{k} = R_{ik}^{k-1}(R_{kk}^{k-1})^{*}R_{kj}^{k-1} \cup R_{ij}^{k-1}$

Converting regular expressions to *NFA*s

Construction of *NFA* based on syntactic structure of regular expression. Each intermediate *nfa* has exactly one final state, no edge entering start state, and no edge leaving final state.

`"BASE"`: Build two-state automaton for atomic regular expression `a` (single symbol or ϵ) with `a` as the edge label. One automaton N(`a`) for each occurence of `a`.

`"INDUCTIVE STEP"`: Compose automata as follows:

- concatenate: N(`st`) – given N(`s`) and N(`t`)

- union: N(`s|t`) – given N(`s`) and N(`t`)

- Kleene closure: N($\mathtt{s}^*$) – given N(`s`)

Next Lecture

CFGs, BNF, derivations, parse tree, ambiguity, top-down parsing

Things to do:

- First homework is due Friday, January 31, **BEFORE** class
- read Scott, Ch. 2.3 - 2.5 (skip 2.3.3 Bottom-up Parsing)