CS344 – Compilers: Scanning

Note Title 1/20/2012

Announcements

- First HW due next Wed. (an essay)
- No class next Wednesday
- Make sure book is in by end of next week (ish)
- Next HW won't need book - will be up by Monday

Compilers

The process by which programming languages are turned into assembly or machine code is important in programming languages.

We'll spend some time on these compilers, although it isn't a focus of this class.

Compilers

Compilers are essentially translaters, so must semantically understand the code

Output: either assembly, machine code or some other output

Java → bytecode

Compilers begin by preprocessing:

- remove white space ← comments
- include macros or libraries
- group characters into tokens

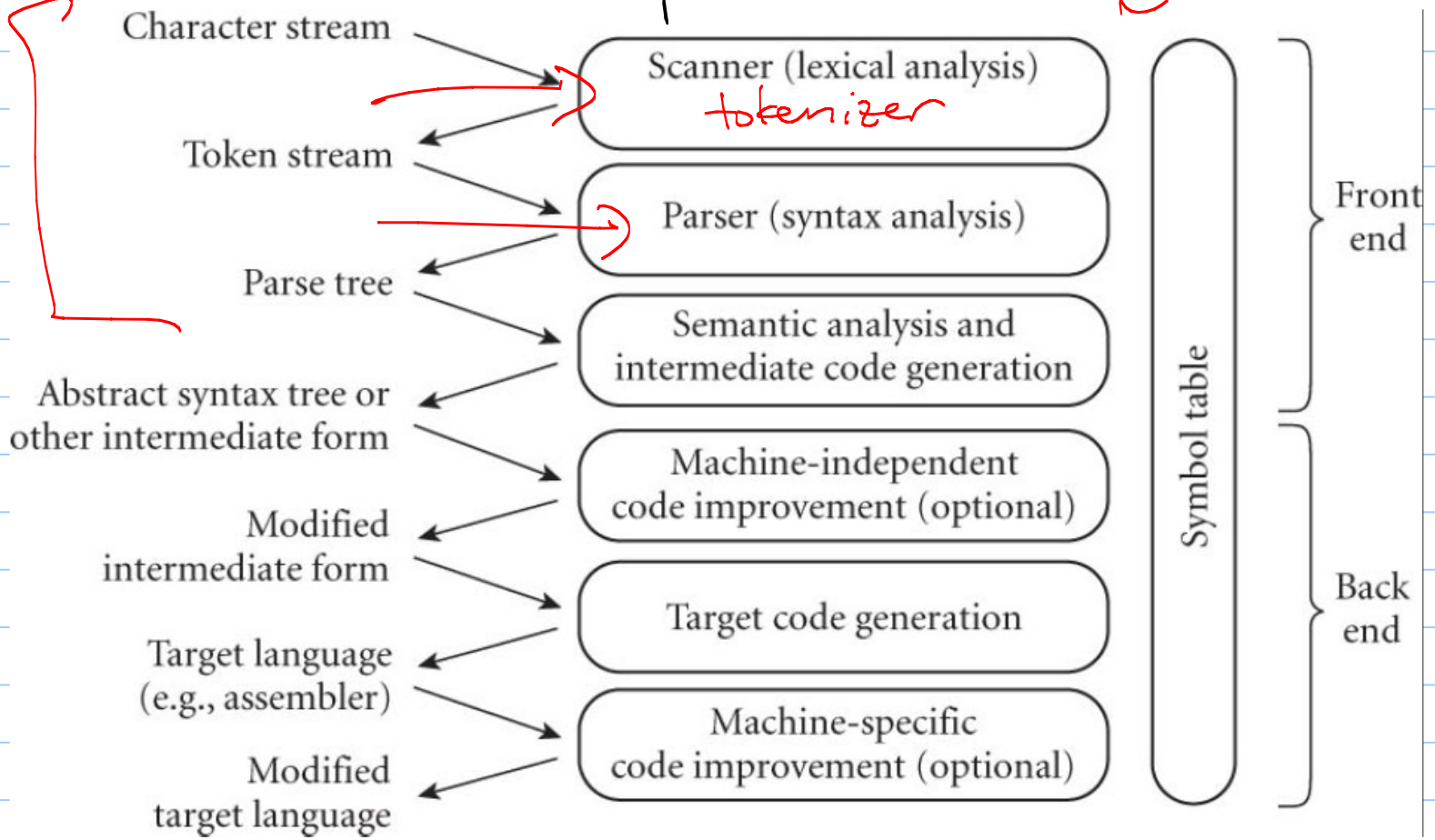
ex: for (int i=0; i< x; i++)
{
. . .
}

- identify high-level syntactical structures

ex: if ( ) i=i+5; cout << ";

Overview of Compilation

Ch 2 of book



The steps:

Front end:

(A) Scanner

(B) Parser

(C) Semantic Analysis

Let's dive into these first...

Scanning (lexical analysis)

- Divide program into tokens, or smallest meaningful units

  Ex: for, recognize keywords, group operations, names, etc.

- Scanning & tokenizing makes parsing much simpler!

- While parsers can work character by character, it is slow.

- Note: Scanning is recognizing a regular language, eg via DFA

Parsing

- Recognizing a context-free language, e.g via PDA
- Finds the structure of the program (or the syntax)

Ex: iteration-statement →
while (expression) statement

statement → compound_statement

Outputs a parse tree

Semantic Analysis (after parsing)

This discovers the meaning of the commands.

Actually only does static semantic analysis, consisting of all that is known at compile time.

(Some things - eg array out of bounds - are unknowable until run time.)

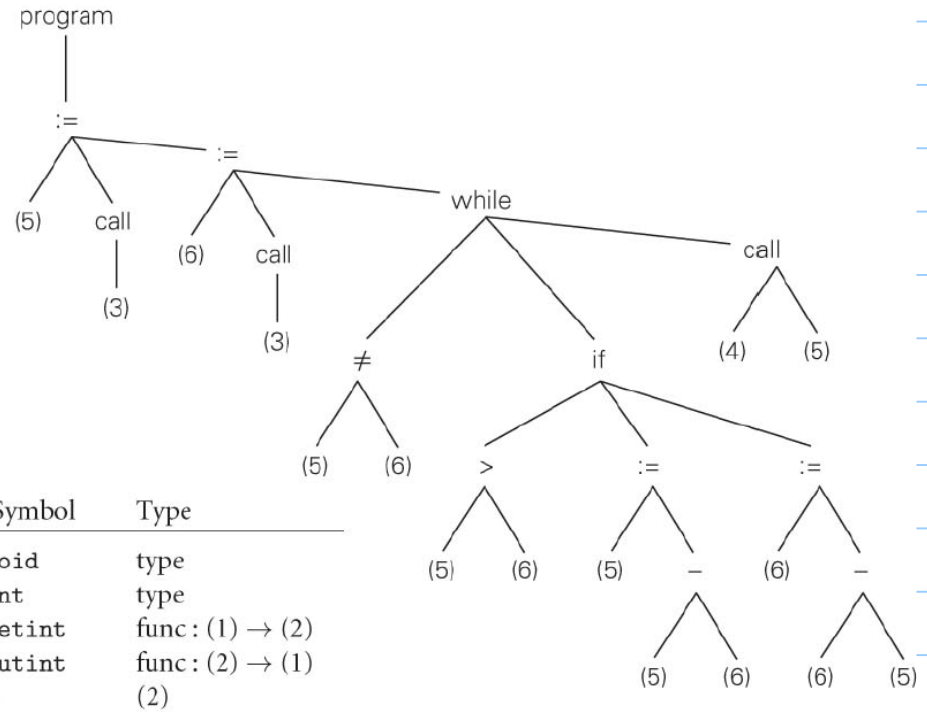
Ex: (semantic analysis)

- Variables can't be used before being declared.
- Type checking.
- Identifiers are used in proper context.
- Functions have correct inputs & returns.

etc... (very language dependent)

Intermediate Form

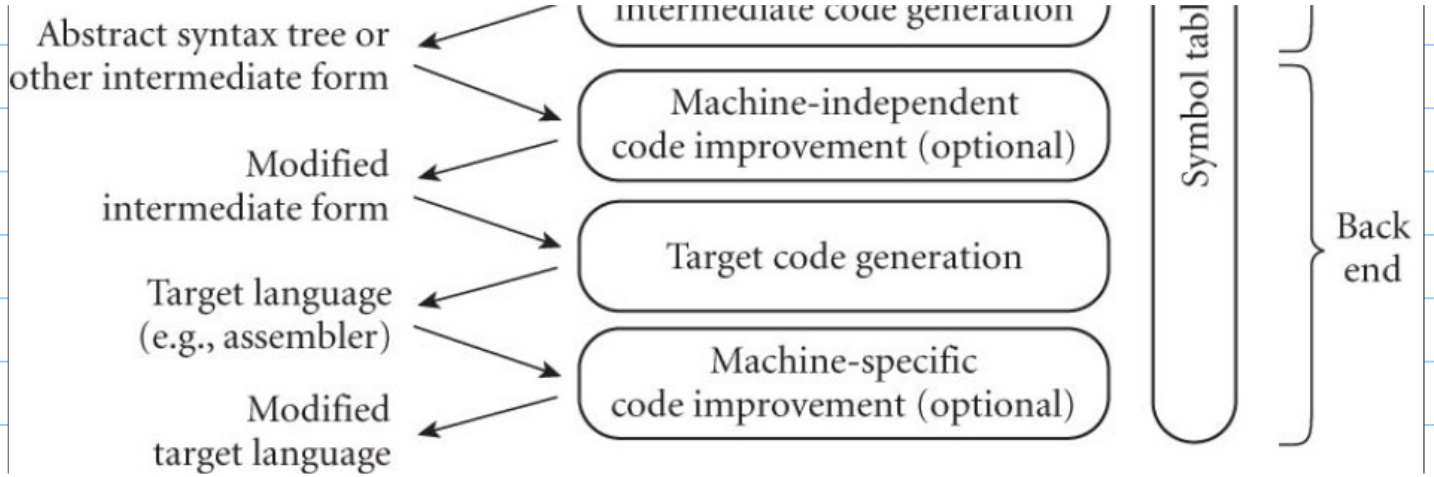
This is the output of the "front end"



Index	Symbol	Type
1	`void`	type
2	`int`	type
3	`getint`	func : (1) → (2)
4	`putint`	func : (2) → (1)
5	`i`	(2)
6	`j`	(2)

- Often, this is an abstract syntax tree — a simplified version of a parse tree
- May also be a type of assembly-like code

Back end: (Actual code generation)



Creating correct code is generally not difficult.

Optimization of that code is.

Back to front end:

(A) How is this actually done?

Input is actually a String of ASCII.

Need to find a way to scan letter by letter + decide what is a token.

Then pass the tokens on to the parser.

Regular Expressions: Some Theory

A regular expression is defined (recursively) as:

Base case:
- A character
- The empty string, ε

Recursive formulation:
- 2 regular expressions concatenated
- 2 regular expressions separated by an or (written |)
- A regular expression followed by * (Kleene star – 0 or more occurrances)

Regular Languages

The class of languages described by a regular expression.

Ex: $0^*10^* = L$

The set of 0-1 strings which contain exactly one 1.

$1 \in L$

$01 \in L$

$0100 \in L$

$11 \notin L$

$0 \notin L$

0,1 strings

Ex: Give the regular expression for $\{w \mid w$ begins with a 1 and ends with a 0$\}$

$1(0|1)^*0$

Ex: $\{w \mid w$ starts with 0 and has an odd length$\}$

$0((0|1)(0|1))^*$

Example: Numbers in Pascal

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

unsigned_int → digit digit*

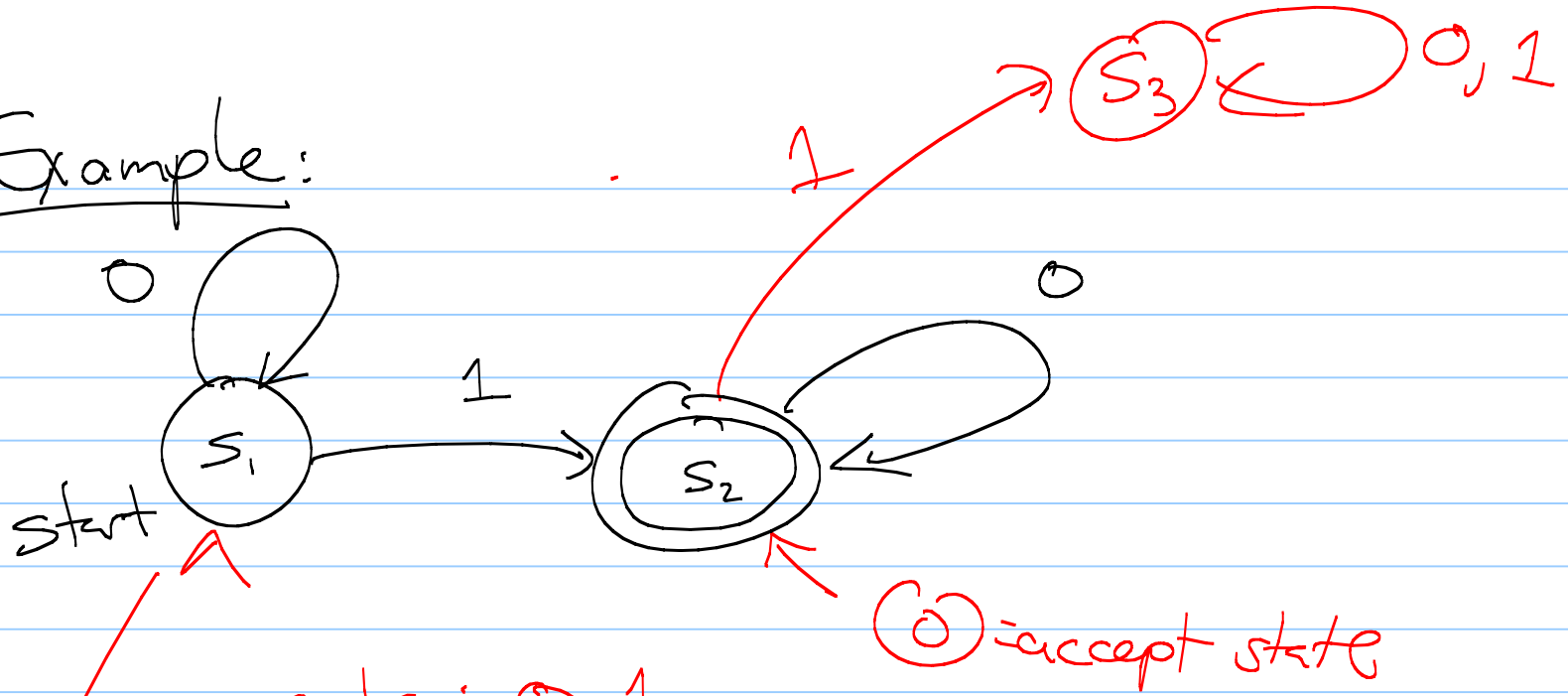
unsigned_number →
unsigned_int (ε | . unsigned_int)
(ε | ((e | E) (+ | - | ε) unsigned_int))

Deterministic Finite Automata (DFA)

Regular languages are precisely the things recognized by ~~DFAs~~.

- A set of states
- input alphabet
- A start state
- A set of accept states
- A transition function: given a state & an input, output a new state

Example:



(0) = accept state

inputs: 0, 1

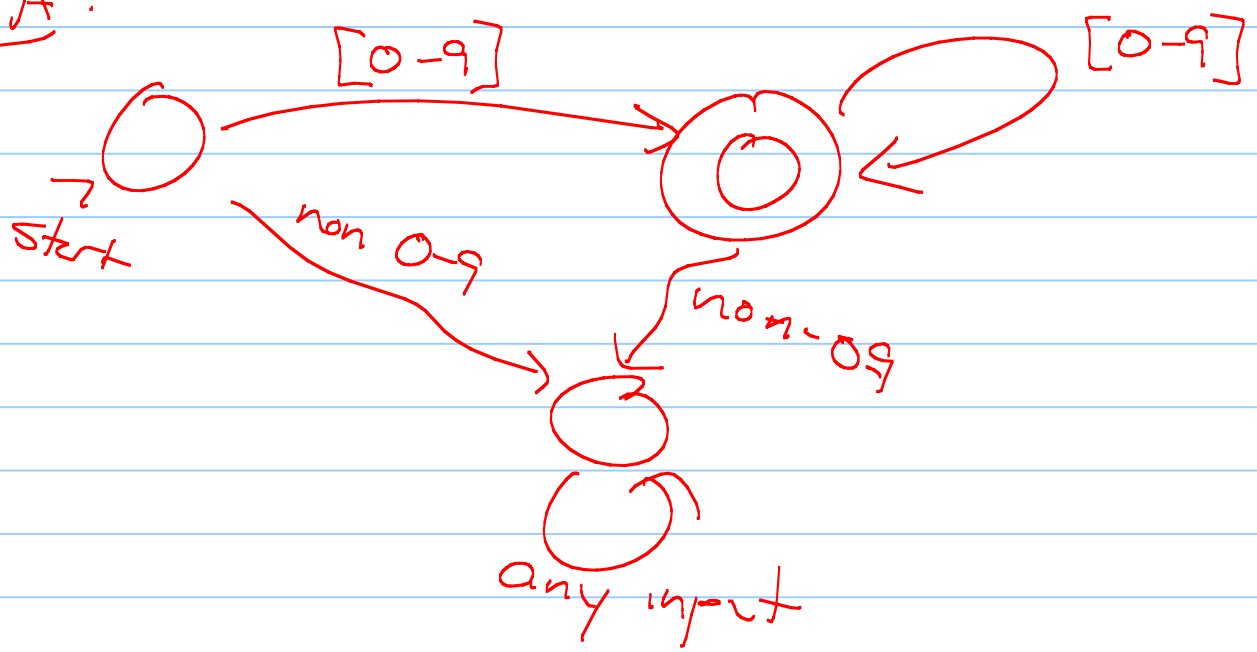
transitions – indicated w/ arrows

$0^* 1 0^*$

Ex: unsigned_int → digit digit*

digit → [0-9]

DFA:



Non-deterministic Finite Automata : NFA

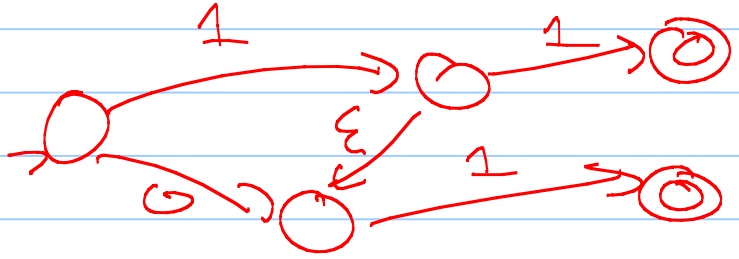
Note: No ambiguity is allowed in DFA's.

So given a state & input, can't be multiple options!

Also — no ε-transitions.

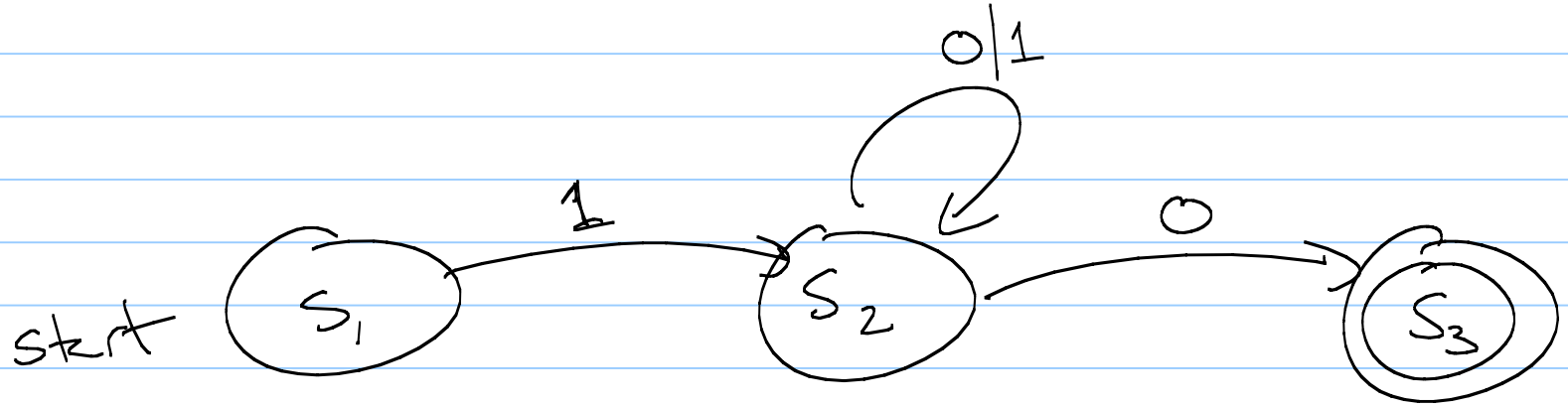
If we allow several choices to exist, this is called an NFA.

Ex:



string 11 has 2 possible paths in NFA!

Ex: $L = 1(0|1)^*0$



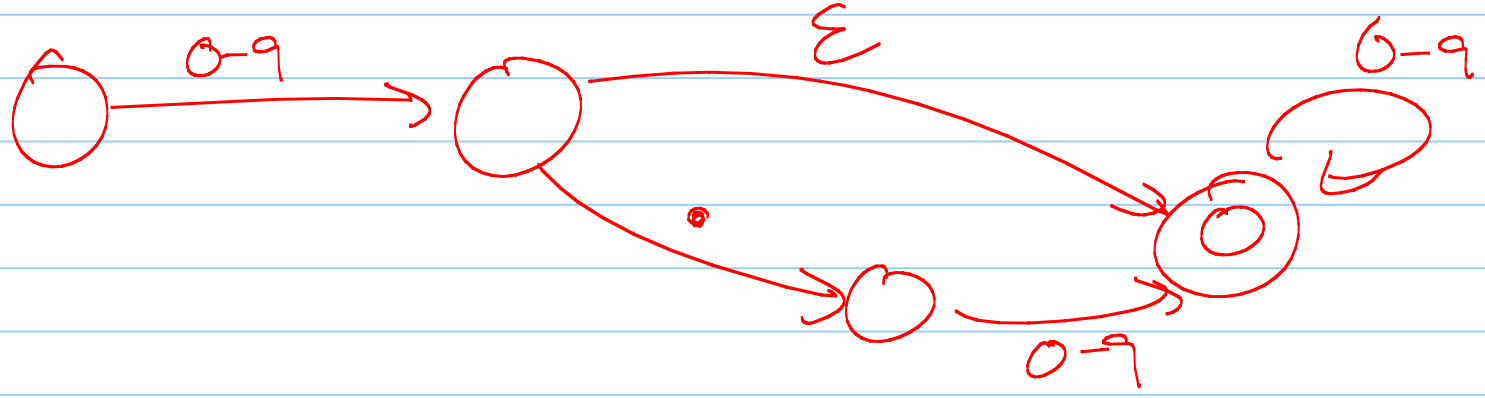
$100 \in L$

Ex: Some things are easier with NFA!

unsigned_number → unsigned_int (ε | . unsigned_int)

unsigned_int → [0-9]

NFA:



Essentially, we can think of an NFA as modeling a parallel set of possibilities (or a tree of them).

Thm: Every NFA has an equivalent DFA.

So: Both recognize reg. languages!

Limitations of Regular Expressions

Certain languages are not regular.

Ex: $\{w \mid w$ has an equal number of 0's and 1's$\}$

Somehow, this needs a type of memory, which regular expressions do not have.

$0^n 1^n$

Why do we care?

Need to "nest" expressions.

Ex: expr → id | number | -expr | (expr) | expr op expr

op → + | - | / | *

Regular expressions can't quite do this.

(This will come up more in parsing — next week or later)

Scanners: do this in code

Find the syntax (not semantics) of code.

Output tokens.

A few types:

- Ad-hoc
- Finite automata
  - nested case statements
  - table + driver

Ad-hoc: case based code

```
if current ∈ { "(", ")", "+", "-", "*" }
      return that symbol
if current = ":"
      read next
       if it is =, announce "assign"
            else announce error
if   current = "/"
      read next
       if it is "*" or "/"
            read until "*/" or "newline" (resp.)
       else return divide
 etc.
```

Ad-hoc approach

Advantage:
code is fast & compact

Disadvantage:
very ad-hoc!
- hard to debug
- no explicit representation

DFA approach:

Given a regular expression, convert to a DFA.

We'll walk through this next week - in Ch 2 of the book.

However →

Scanning Programs

In reality, this DFA is often done automatically.

Specify the rules of regular language, & the program does this for you.

Many such examples:

Lex (flex), Jlex / Jflex, Quex, Ragel, ...

Next time:

Lex / flex : C-style driver

Look for HW on regular expressions, NFA / DFA, & context free languages

Next programming assignment will use flex.