NTNU | Norwegian University of Science and Technology

Compiler Construction

Lecture 2: Compiler Structure and Lexical Analysis

2020-01-10

Michael Engel

Includes material by Jan Christian Meyer

.org

Theoretical and practical exercises

- TA: Lahiru Rasnayake

- Six problem sets, one every two weeks
- Theoretical questions on scanning, parsing, optimization…
- Practical: build parts of your own small compiler (in C)
  - Get your own software project running

- Solutions need to be handed in on time
  - Rather, an empty solution than a plagiarized one
- Only the final two will be graded
  - 20% of the final grade (80% exam)
- More details next week

Overview

- Overview: definition and tasks of a compiler
- Structure and stages of a typical compiler
- Deterministic finite automata (DFA)
- Lexical analysis (scanning)

Compilers are everywhere

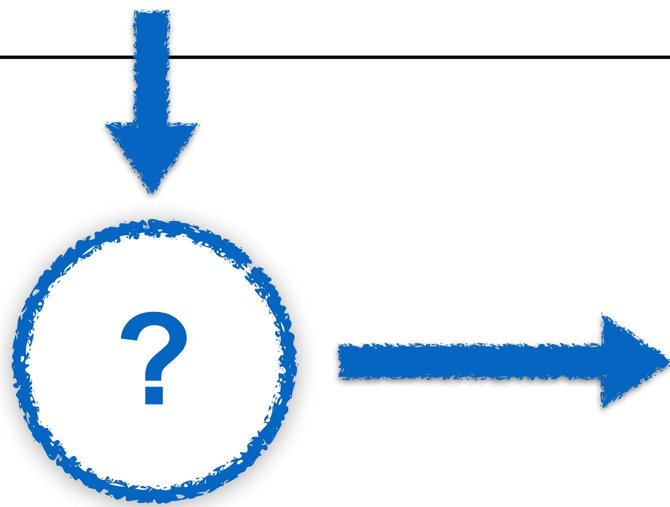
- Original idea: enable programming of computers in *higher-level abstractions* than machine language
  - Zuse's Plankalkül (1940s), FORTAN, LISP, A0 (1950s)
- Today:
  - Many different source languages and target platforms
- Additional uses of compilers:
  - Static analysis and verification
  - Hardware synthesis
  - Source-to-source transformations
  - Just in time (JIT) compilation

What does a compiler do?

- Compiler:
  "Tool that translates software written in one language into another language"
  - must understand both the form, or ***syntax***, and content, or meaning (***semantics***), of the *input language*
  - and understand the rules that govern syntax and meaning in the *output language*
  - needs a scheme for mapping content from the source language to the target language
- Requirements:
  - must preserve the meaning of the program being compiled
  - must improve the input program in some discernible way

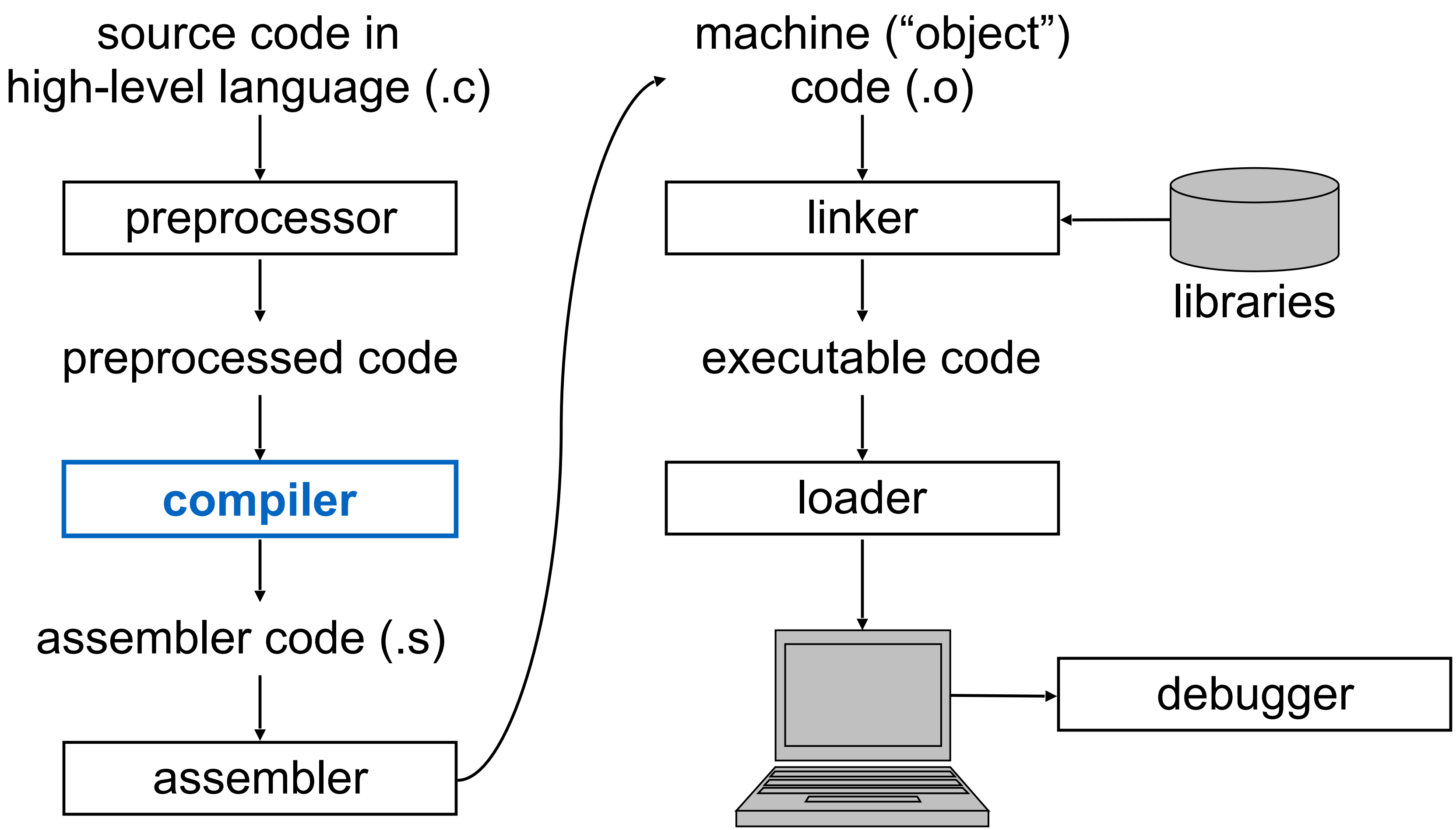
The compilation process black box

```
int factorial(int n)
{
  int fact = 1;
  while (n--)
    fact = fact * n;
  return n;
}
```

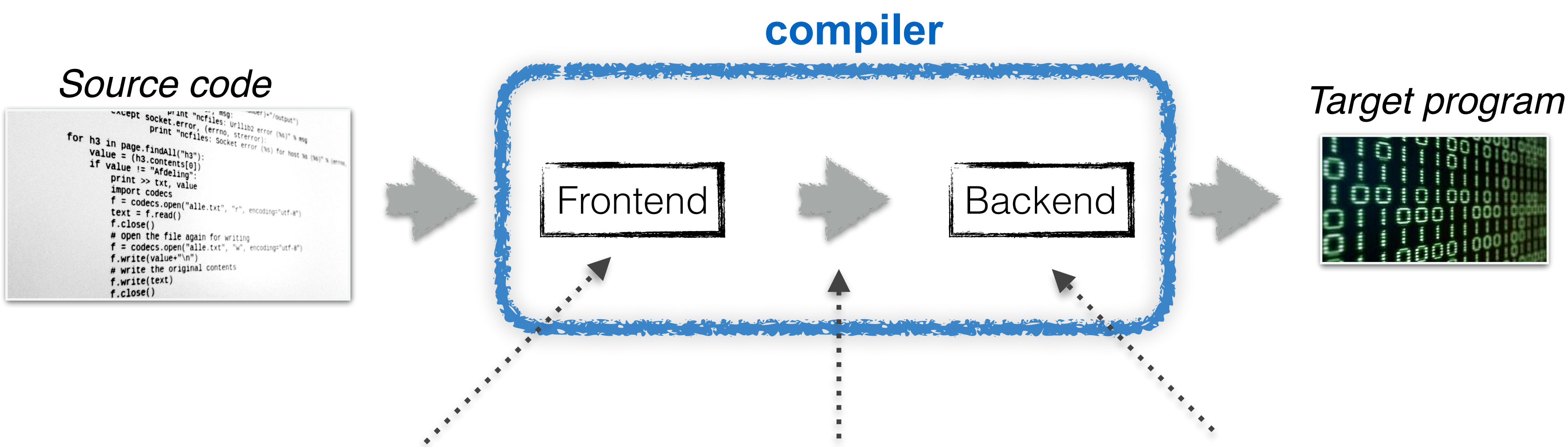



```
. . .
0xE59F1010
0xE59F0008
0xE0815000
0xE59F5008
. . .
```

Compilation process in detail



Structure of a compiler (1)

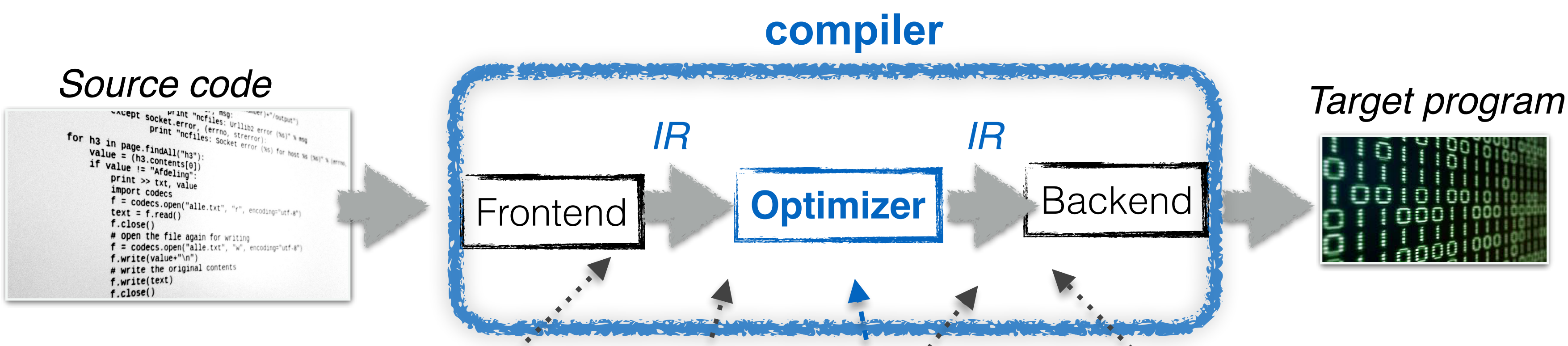


"understand both the form, or ***syntax***, and content, or meaning (***semantics***), of the *input language*"

"scheme for mapping content from the source language to the target language"

"understand the rules that govern syntax and meaning in the output language"

Structure of a compiler (2)



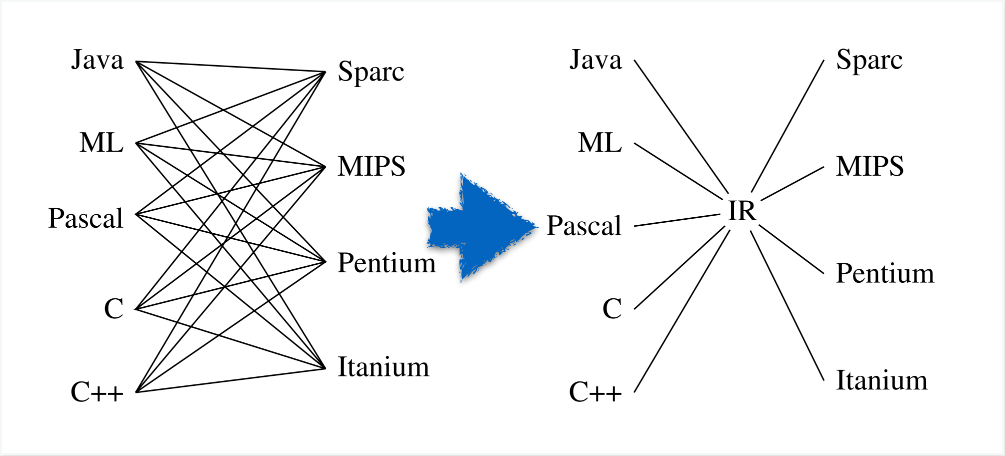
"understand both the form, or ***syntax***, and content, or meaning (***semantics***), of the *input language*"

"understand the rules that govern syntax and meaning in the output language"

"scheme for mapping content from the source language to the target language"

"must improve the input program in some discernible way"

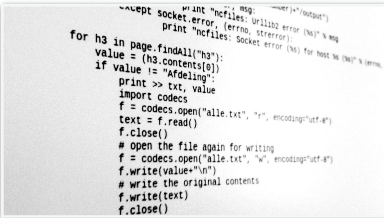
Intermediate representation (IR)



- Early compilers directly generated machine code
- *n* source languages, *m* targets: *n x m compilers required!*
- Idea: use a common description format: "*Intermediate Representation*" (IR)
  – Transform source to IR *(front end)* and IR to target code *(back end)*: only *n + m compilers required* now
- Additional advantages of using intermediate representations:
  – Easy to change source or target language
  – Easier optimizations: developed only for the intermediate representation
  – Intermediate representation can be directly interpreted

Stages of a compiler (1)

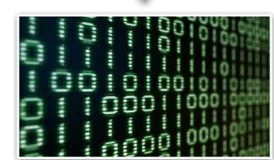
Source code



character stream

Lexical analysis → Syntax analysis → Semantic analysis → Code optimization → Code generation

token sequence



machine-level program

Lexical analysis (scanning):

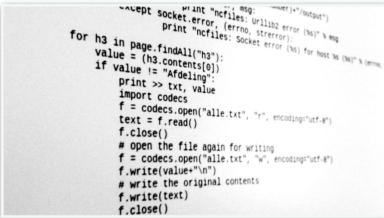
- Split source code into *lexical units*
- Recognize *tokens* (using regular expressions/automata)
- Token: character sequence relevant to source language grammar

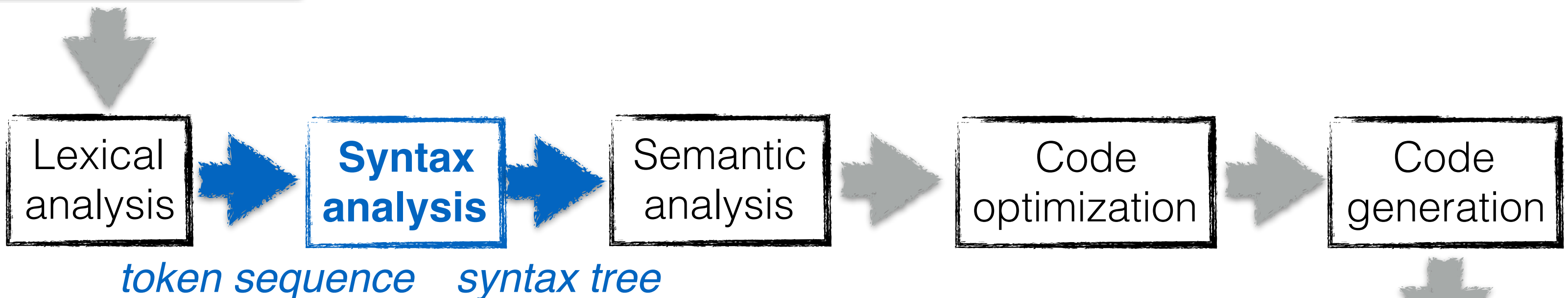
x = y + 42 → id(x) op(=) id(y) op(+) number(42)

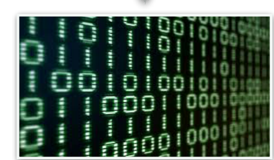
character stream → *token sequence*

Stages of a compiler (2)

Source code





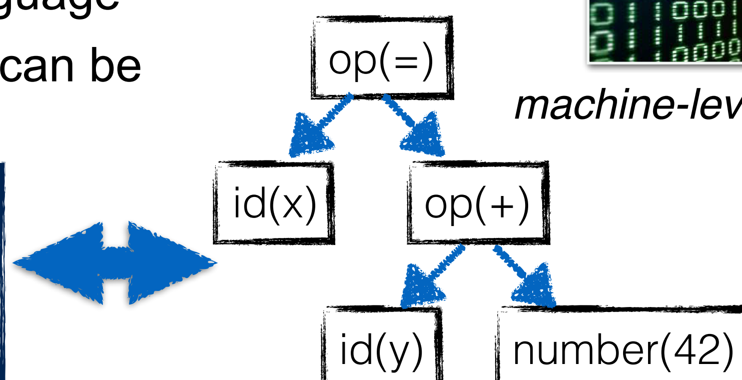


machine-level program

Syntax analysis (parsing)

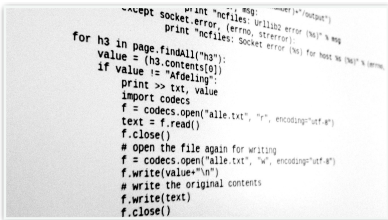
- Uses *grammar* of the source language
- Decides if input *token sequence* can be derived from the grammar

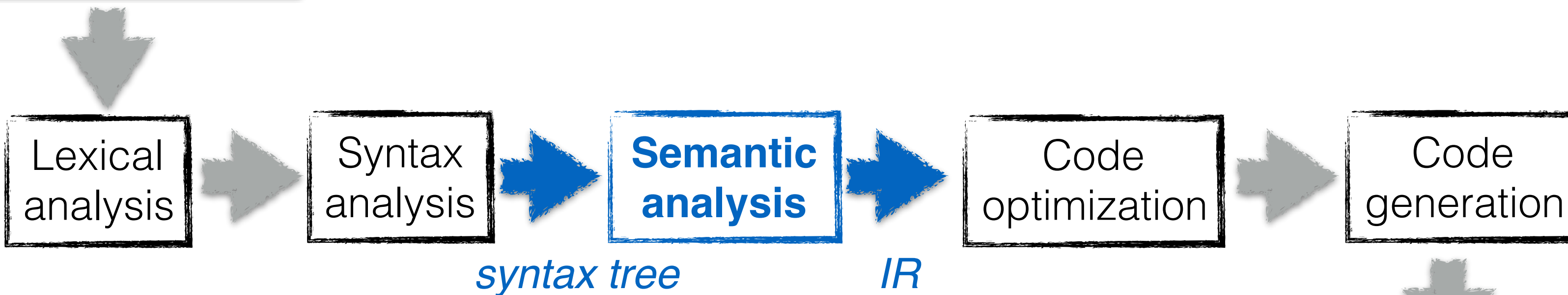
```
expression → term { (+|-) term }
term →       factor { (*|/) factor }
factor →     '(' expression ')'
             | id | number
```

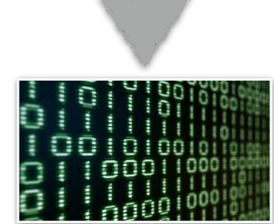


Stages of a compiler (3)

Source code







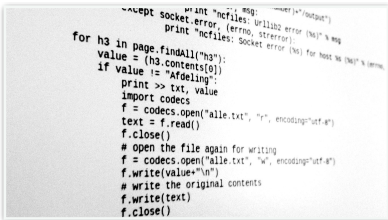
machine-level program

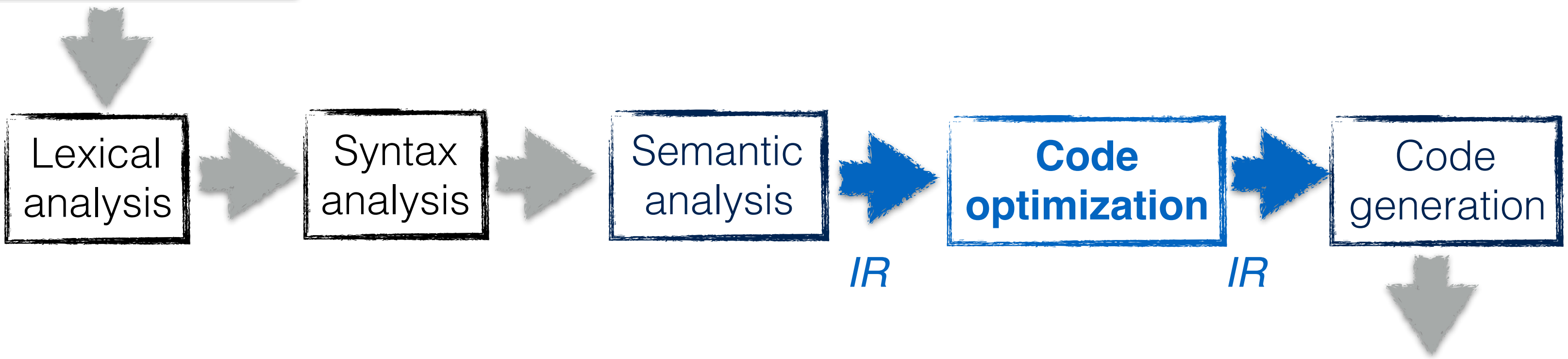
Semantic analysis

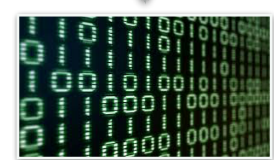
- *Name analysis* (check def. & scope of symbols)
- *Type analysis* (check correct type of expressions)
- Creation of *symbol tables* (map identifiers to their types and positions in the source code)

Stages of a compiler (5)

Source code







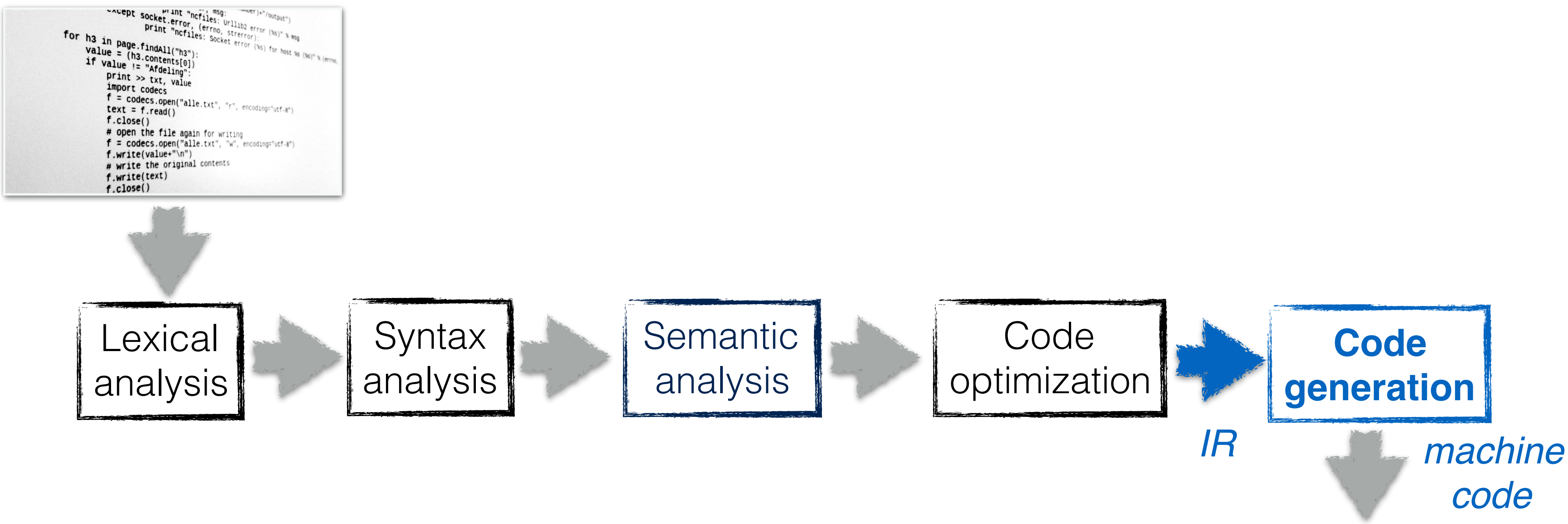
machine-level program

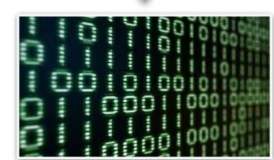
Code optimization

- Analyzes & applies patterns of redundancy
  - e.g., store of a variable followed by a load of it
- Often, different stages/levels of optimization with different intermediate representations are applied

Stages of a compiler (4)

Source code





machine-level program

Code generation

- Determines and outputs equivalent machine instructions for components of the IR *(instruction selection)*
- Determines correct instruction order with respect to pipeline constraints, exploitation of instruction-level parallelism *(instruction scheduling)*
- Assigns variables to registers *(register allocation)* and memory locations

Lexical analysis (scanning)

Lexical analysis

- The compiler input is simply a stream (sequence) of bytes:

    72, 101, 108, 108, 111, 32, 119, 111, 114, 108, 100, ...

- By convention, these are mapped to letters, digits, etc.:

    ‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘ ‘, ‘w’,’o’,’r’,’l’,’d’, ...

ASCII encoding

- Other mappings (encodings) exist
    - e.g. Unicode UTF-8, EBCDIC

- On this level, the input program is just a lot of bytes without any structure

Lexical analysis (scanning)

- Naive approach to scanning:
  Read letters one by one, e.g., for a key word “while”:

  w (119), **h** (104), **i** (105), **l** (108), **e** (10)

- Writing a compiler that has to detect this pattern every time the programmer wants to start a loop is inconvenient:

  - A programmer might choose to call a variable 'whilf':

    w (119), **h** (104), **i** (105), **l** (108), *(looking good so far…)*
    f (10) *(oh no, start from scratch, that’s not a loop)*

Identifying syntactical units

Lexical analysis

- Better approach:
  Group letters into meaningful units and operate on those:

‘i’, ‘f’, ‘(‘, ‘w’,’h’, ‘i’, ‘l’, ‘f’, ‘=’, ‘=’, ‘2’, ‘)’, ‘{‘, ‘x’, ‘=’, ‘5’, ‘;’, ‘}’

if (whilf == 2) { x = 5; }

- Here, we use color coding to identify the various units:

keywords and punctuation
delimiters of groups
variables
operators
numbers

Deriving code structure

Lexical analysis

- What use is the coloring of our units?

  We've already seen this one:

  if (whilf == 2) { x = 5; }

  How would we color that line?

  while (a < 42) { a += 2; }

  Using the same coloring roles, we get:

  while (a < 42) { a += 2; }

keywords and punctuation
delimiters of groups
variables
operators
numbers

- These two statements have completely different meanings **but share the same (syntactic) structure** (here: sequence of colors)
  - We'll talk about structure later
  - Today, we will look at ***lexical analysis***

Useful definitions

- ***Lexeme***
  - Lexemes are units of lexical analysis, words
  - They're like entries in the dictionary, "house", "walk", "smooth"
- ***Token***
  - Tokens are units of syntactic analysis
  - They are like units of a sentence, "noun","verb","adjective"
- ***Semantic***
  - The meaning of something (there is no sensible unit)
  - Similar to explanations in the dictionary:
    - house: "a building which someone inhabits"
    - walk: "the act of putting one foot in front of the other"
    - smooth: "the property of a surface which offers little resistance

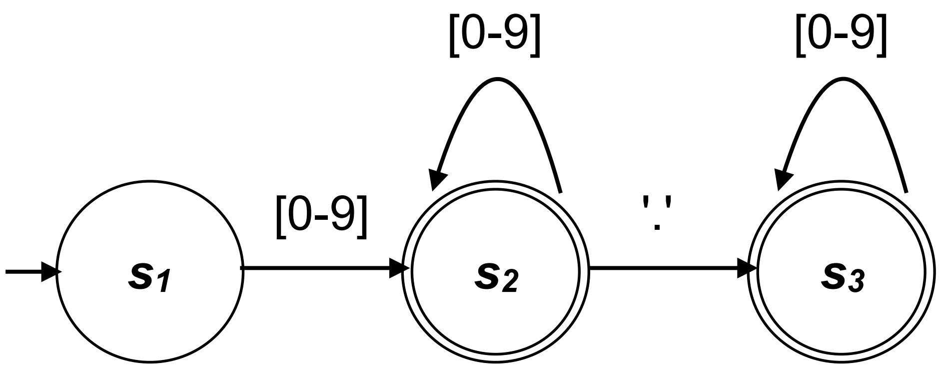
Classes of lexemes

- Lexemes with a ***fixed meaning***
  - keywords or reserved words
  - “if", “while”, “for”, “==“, …
  - Most languages forbid the use of these as identifiers (variable/ function/… names)
    - Source is easier to parse, less ambiguous code

- ***Classes with countably infinite instances***
  - e.g. 1, 2, 3, … 65535, …
  - All of these are specific cases of the class “integer number"

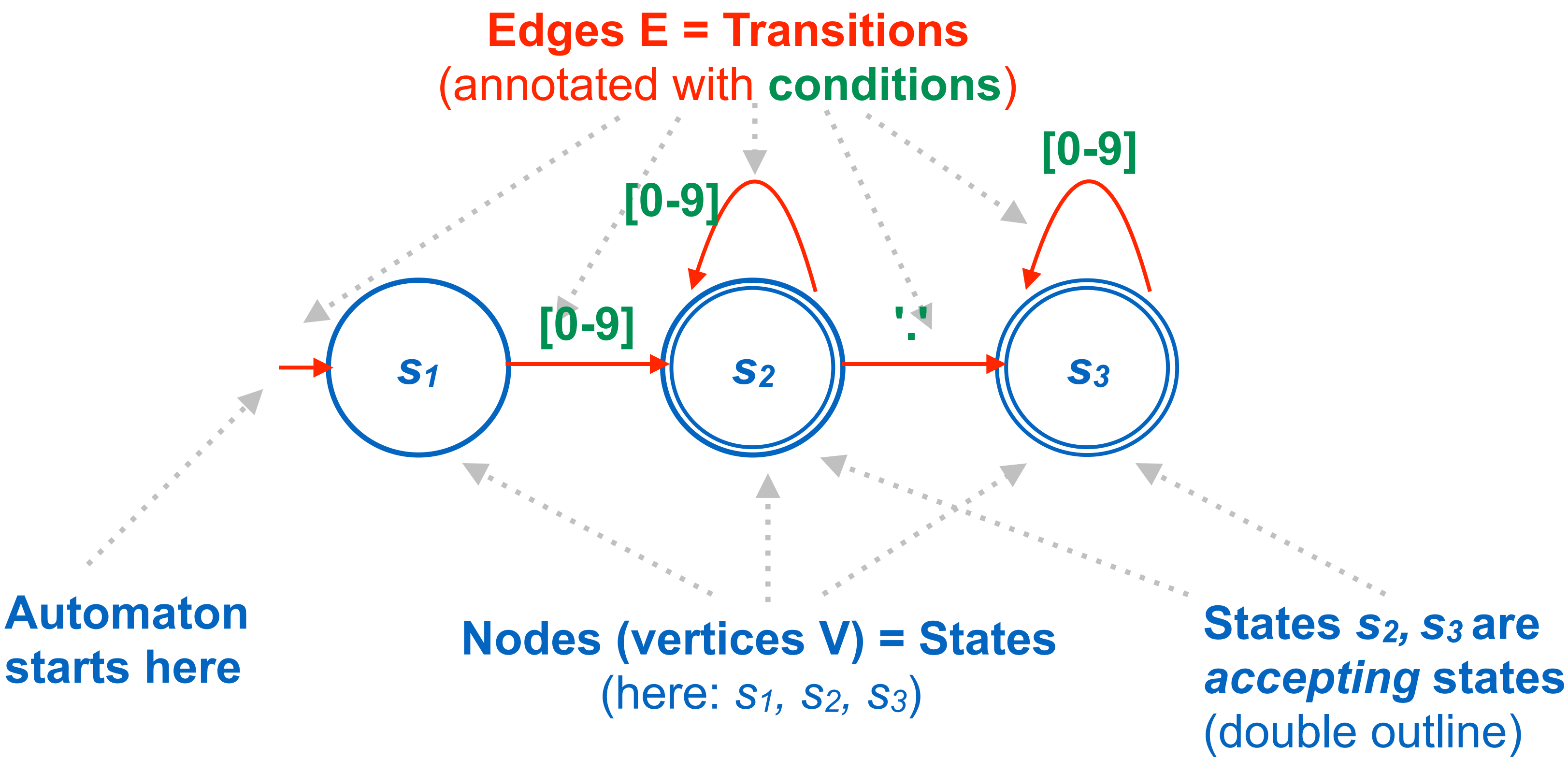
Finite automata

- Required:
  Mechanism to identify ***classes of words*** (not just single words)
  - Example: mechanism to recognize real numbers
- Informal description:
  "A real number starts with one or more digits optionally followed by a decimal point followed by zero or more digits"
- Formal approach: ***Deterministic Finite Automaton*** (DFA)
  - example given as a directed graph here (easy to follow)



DFA structure

DFAs are often represented as ***directed graph*** *G = (V, E)*



DFA formal definition

Formal definition: DFA = 5-tuple $(Q, \Sigma, \delta, q_0, F)$

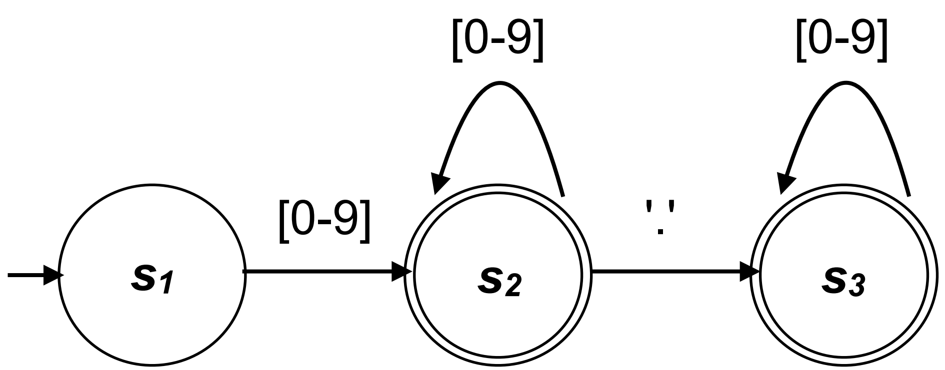
Q is a finite set called the ***states***,

Σ is a finite set called the ***alphabet***,

$\boldsymbol{\delta: Q \times \Sigma \rightarrow Q}$ is the ***transition function***,

$\boldsymbol{q_0} \in \boldsymbol{Q}$ is the ***start state***, and

$\boldsymbol{F} \subseteq \boldsymbol{Q}$ is the set of ***accepting states***



$\boldsymbol{Q} = \{s_1, s_2, s_3\}$

$\boldsymbol{\Sigma} = \{0,1,2,3,4,5,6,7,8,9,.\}$

$\boldsymbol{q_0} = s_1$

$\boldsymbol{F} = \{s_2, s_3\}$

δ = ???

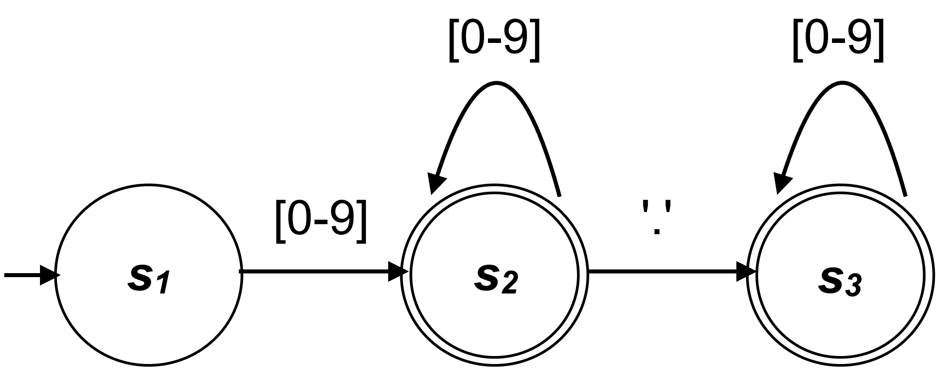
Transition function of a DFA

Give the subsequent state for each state and each possible input, commonly as a table:

input character

current state

δ	0	1	2	3	4	5	6	7	8	9	.
s_1	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	
s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_3
s_3	s_3	s_3	s_3	s_3	s_3	s_3	s_3	s_3	s_3	s_3	



$Q = \{s_1, s_2, s_3\}$

$\Sigma = \{0,1,2,3,4,5,6,7,8,9,.\}$

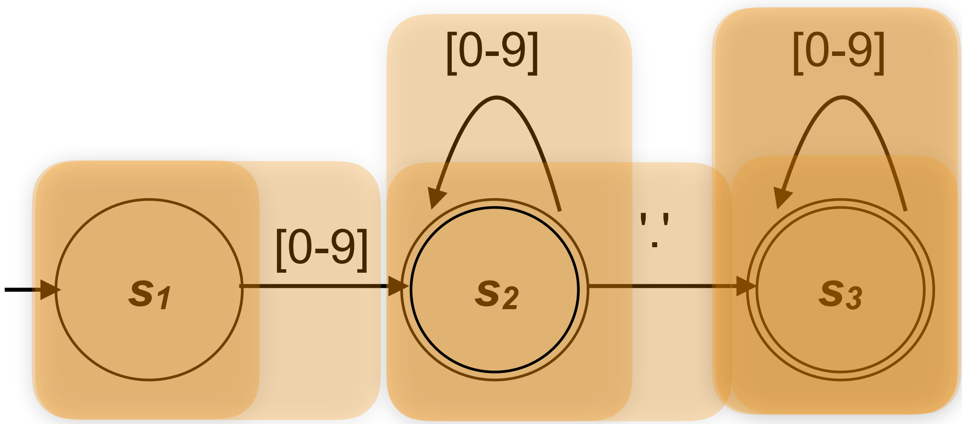
$q_0 = s_1$

$F = \{s_2, s_3\}$

$\delta = ???$

Example DFA transition

δ	0	1	2	3	4	5	6	7	8	9	.
s_1	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	
s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_3
s_3	s_3	s_3	s_3	s_3	s_3	s_3	s_3	s_3	s_3	s_3	



Input character sequence:

4 2 . 2 3

Start: in state s_1

Read 1st char: '4' → change to s_2

Read 2nd char: '2' → stay in s_2

Read 3rd char: '.' → change to s_3

Read 4th char: '2' → stay in s_3

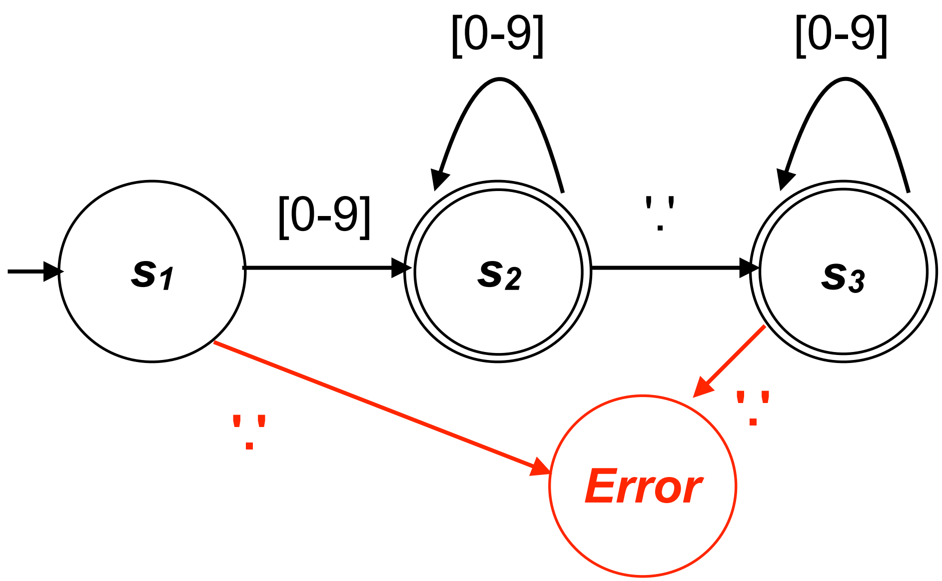
Read 5th char: '3' → stay in s_3

End of sequence in accepting state ✔

Error handling

- What happens when a character '.' is read in state s_1 or s_3?

δ	0	1	2	3	4	5	6	7	8	9	.
s_1	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	???
s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_3
s_3	s_3	s_3	s_3	s_3	s_3	s_3	s_3	s_3	s_3	s_3	???



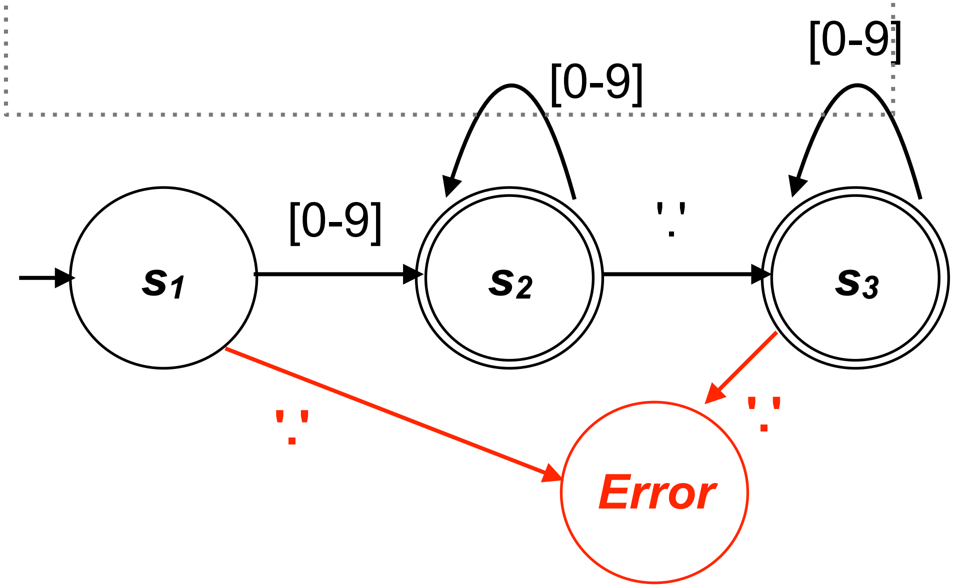
The error state is often omitted in DFA descriptions.

Implied: all non indicated characters → error

Implementing a DFA in C the hard way

```c
enum {error = 0, success};

int scan_real_number(void) {
  char c;
  enum states = {s1, s2, s3};
  enum states cur = s1;
  while (1) {
    c = getchar(); // get next char
    if (c==EOF) break; // end?
    switch(cur) {
      case s1:
        if (c>='0' && c<='9')
          cur = s2;
        else return error;
        break;
      case s2:
        if (c>='0' && c<='9')
          cur = s2;
        else if (c=='.')
          cur = s3;
        else return error;
        break;
      case s3:
        if (c>='0' && c<='9')
          cur = s3;
        else return error;
        break;
    }  // switch
  }  // while
  // check for accepting state
  if (cur != s2 && cur != s3) return error;
  else return success;
}
```



Implementing a table-driven DFA in C

```c
enum {error = 0, success};
enum states {s1, s2, s3, er};
enum states cur = s1;
char alphabet[] = { '0', '1', '2', '3', '4',
                    '5', '6', '7', '8', '9', '.' };

// next state for each char in alphabet (columns)
struct scanner {
  enum states next[sizeof(alphabet)];
};

// rows of the transition table
struct scanner delta[sizeof(enum states)] = {
//  0    1    2    3    4    5    6    7    8    9    .
  {s2, s2, s2, s2, s2, s2, s2, s2, s2, s2, er}, // s1
  {s2, s2, s2, s2, s2, s2, s2, s2, s2, s2, s3}, // s2
  {s3, s3, s3, s3, s3, s3, s3, s3, s3, s3, er}, // s3
  {er, er, er, er, er, er, er, er, er, er, er}, // er
};
```

```c
int scan_real_number(void) {
  char c;
  while (1) {
    c = getchar(); // get next char
    if (c==EOF) break; // end?
    cur = delta[cur].next[lookup(c)];
  } // while
  // check for accepting state
  if (cur!=s2 && cur!=s3)
    return error;
  else return success;
}
```

What is the task of the function call lookup(c) here and how would you implement it?

Beware: there's a subtle but potentially dangerous bug in the code! Can you find it?

δ	0	1	2	3	4	5	6	7	8	9	.
s_1	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	er
s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_3
s_3	s_3	s_3	s_3	s_3	s_3	s_3	s_3	s_3	s_3	s_3	er

Scanner generators

- Programming a word-class recognizer (lexical analyzer, or scanner) with ad-hoc logic is complicated and error-prone
- Writing one using tables is a bit easier, but it requires punching in a bunch of boring table entries to represent specific DFAs

- Can we ***generate*** code for a scanner automatically from a simple description?
  - Specify word classes as ***regular expressions***
  - Let a program write a large table of states that includes all of the expressions
  - More on this next week!