Career Services &

Association for
Computing Machinery
udacm.org

present:

Microsoft®

Wednesday, October 1, 6:00-7:30pm
Gore Hall Room 103

Bring your resume for a chance to win great prizes!! FREE FOOD will be provided!

Lexical Analysis: DFA Minimization & Wrap Up

Automating Scanner Construction

PREVIOUSLY

RE→NFA *(Thompson's construction)*

- Build an NFA for each term
- Combine them with ε-moves

NFA →DFA (*subset construction*)

- Build the simulation

TODAY

Some Project Related Material

DFA →Minimal DFA

- Hopcroft's algorithm

The Cycle of Constructions

What we expect of the Scanner

- Report errors for lexicographically malformed inputs
  - → reject illegal characters, or meaningless character sequences
- Return an abstract representation of the code
  - → character sequences (e.g., “if” or “loop”) turned into tokens.
- Resulting sequence of tokens will be used by the parser
- Makes the design of the parser a lot easier.

How to specify a scanner

- A scanner specification (e.g., for JLex), is list of (typically short) regular expressions.
- Each regular expressions has an action associated with it.
- Typically, an action is to return a token.
- On a given input string, the scanner will:
  - → find the longest prefix of the input string, that matches one of the regular expressions.
  - → will execute the action associated with the matching regular expression highest in the list.
- Scanner repeats this procedure for the remaining input.
- If no match can be found at some point, an error is reported.

Example of a Specification

- Consider the following scanner specification.
  1. aaa { return T1 }
  2. a*b { return T2 }
  3. b { return S }
- Given the following input string into the scanner

  aaabbaaa

  the scanner as specified above would output

  T2 T2 T1
- Note that the scanner will report an error for example on the string 'aa'.
- Rule 3 is redundant!

Special Return Tokens

- Sometimes one wants to extract information out of what prefix of the input was matched.
- Example:

  “[a-zA-Z0-9]*”   { return STRING(yytext()) }
- Above RE matches every string that
  - → starts and ends with quotes, and
  - → has any number of alpha-numerical chars between them.
- Associated action returns a string token, which is the exact string that the RE matched.
- Note that yytext() will also include the quotes.
- Note this RE doesn’t handle escaped characters, special characters (e.g., punctuation characters), etc.

Quiz

- Consider the following scanner specification.
  1. bab { return T1 }
  2. ba^+ { return T2 }
  3. b { return S1 }
  4. a { return S2 }
- Given the following input string into the scanner

  baabbabaa

  the scanner as specified above would output

What?

Example of a Specification

- Consider the following scanner specification.
  1. bab { return T1 }
  2. ba^{+} { return T2 }
  3. b { return S1 }
  4. a { return S2 }
- Given the following input string into the scanner

  baabbabaa

  the scanner as specified above would output

  T2 S1 T1 S2 S2

DFA Minimization

Details of the algorithm

- Group states into maximal size sets, *optimistically*
- Iteratively subdivide those sets, as needed
- States that remain grouped together are equivalent

Initial partition, P_0, has two sets: $\{D_F\}$ & $\{D-D_F\}$

$(DFA = (Q,\Sigma,\delta,q_0,F))$

Splitting a set ("partitioning a set by a")

- Assume q_i, & $q_j \in p$, and $\delta(q_i,\underline{a}) = q_x$, & $\delta(q_j,\underline{a}) = q_y$
- If q_x & q_y are not in the same set, then p must be split
  - → q_i has transition on a, q_j does not $\Rightarrow$ a splits p
- One state in the final DFA cannot have two transitions on a

DFA Minimization

The algorithm

$P \leftarrow \{ D_F, \{D-D_F\}\}$

while (P is still changing)

$\quad T \leftarrow \emptyset$

$\quad$ *for each set* $p \in P$

$\qquad T \leftarrow T \cup Split(p)$

$\quad P \leftarrow T$

Split(S)

for each $\alpha \in \Sigma$

$\quad$ *if* α *splits S into* s_1 *and* s_2

$\qquad$ *then return* $\{ s_1, s_2\}$

return S

This is a fixed-point algorithm!

Why does this work?

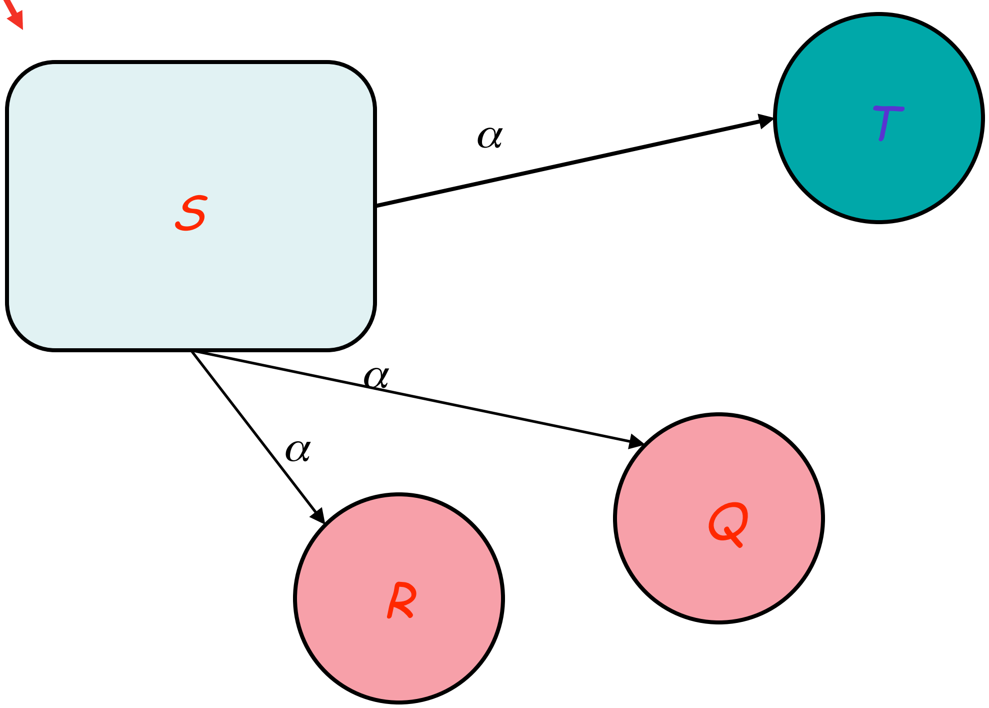
- Partition $P \in 2^D$
- Starts with 2 subsets of D $\{D_F\}$ and $\{D-D_F\}$
- *While* loop takes $P_i \rightarrow P_{i+1}$ by splitting 1 or more sets
- P_{i+1} is at least one step closer to the partition with $|D|$ sets
- Maximum of $|D|$ splits

Note that

- Partitions are <u>never</u> combined
- Initial partition ensures that final states are intact

Key Idea: Splitting S around α

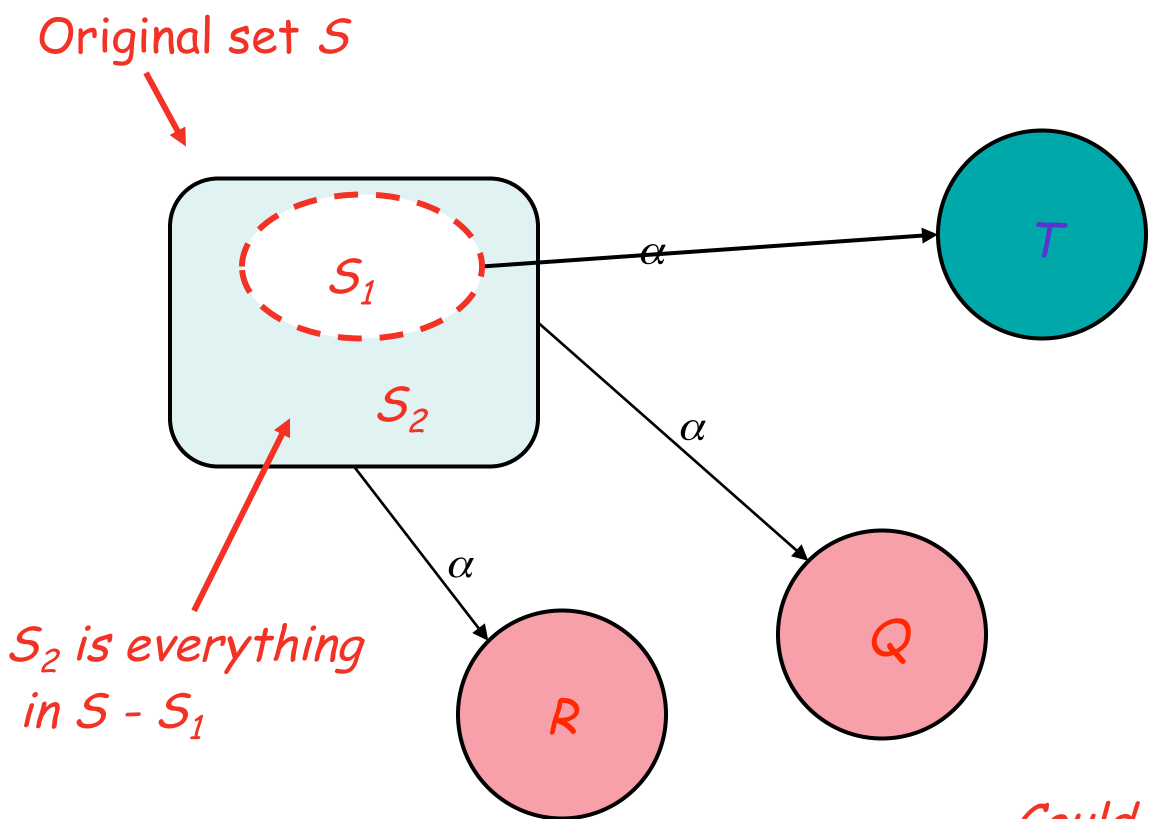
Original set S



S has transitions on α to R, Q, & T

The algorithm partitions S around α

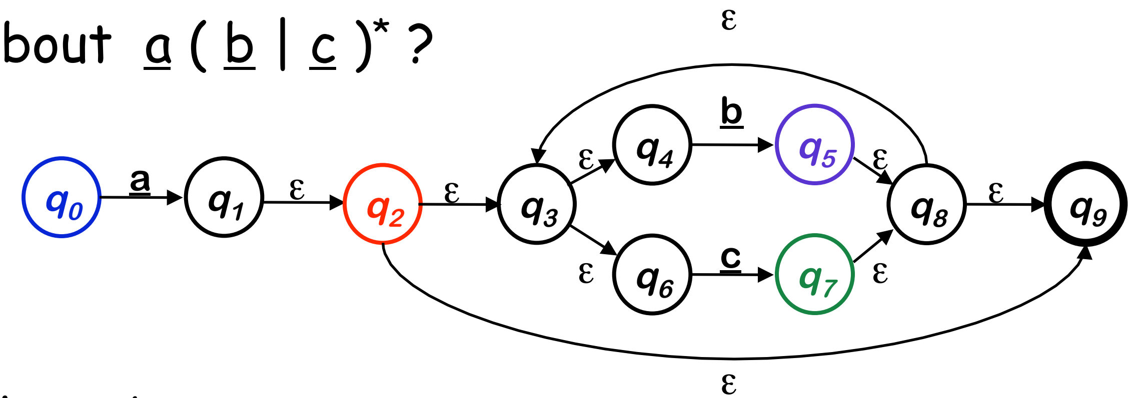
Key Idea: Splitting S around α



Could we split S_2 further?

DFA Minimization

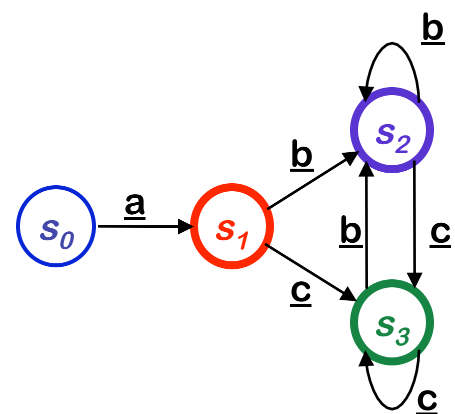
What about a (b | c)* ?



First, the subset construction:

		ε-closure(Delta(s,*))		
	NFA states	a	b	c
s_0	q_0	q_1, q_2, q_3, q_4, q_6, q_9	*none*	*none*
s_1	q_1, q_2, q_3, q_4, q_6, q_9	*none*	q_5, q_8, q_9, q_3, q_4, q_6	q_7, q_8, q_9, q_3, q_4, q_6
s_2	q_5, q_8, q_9, q_3, q_4, q_6	*none*	s_2	s_3
s_3	q_7, q_8, q_9, q_3, q_4, q_6	*none*	s_2	s_3

Final states

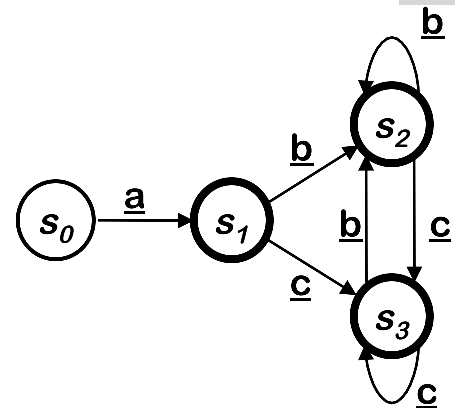


DFA Minimization

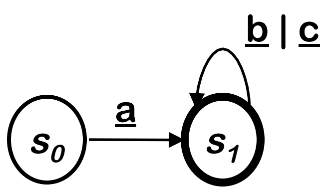
Then, apply the minimization algorithm

		Split on		
	Current Partition	a	b	c
P_0	$\{s_1, s_2, s_3\}$ $\{s_0\}$	*none*	*none*	*none*

final states



To produce the minimal DFA



In lecture 4, we observed that a human would design a simpler automaton than Thompson's construction & the subset construction did.

Minimizing that DFA produces the one that a human would design!

Abbreviated Register Specification

Start with a regular expression

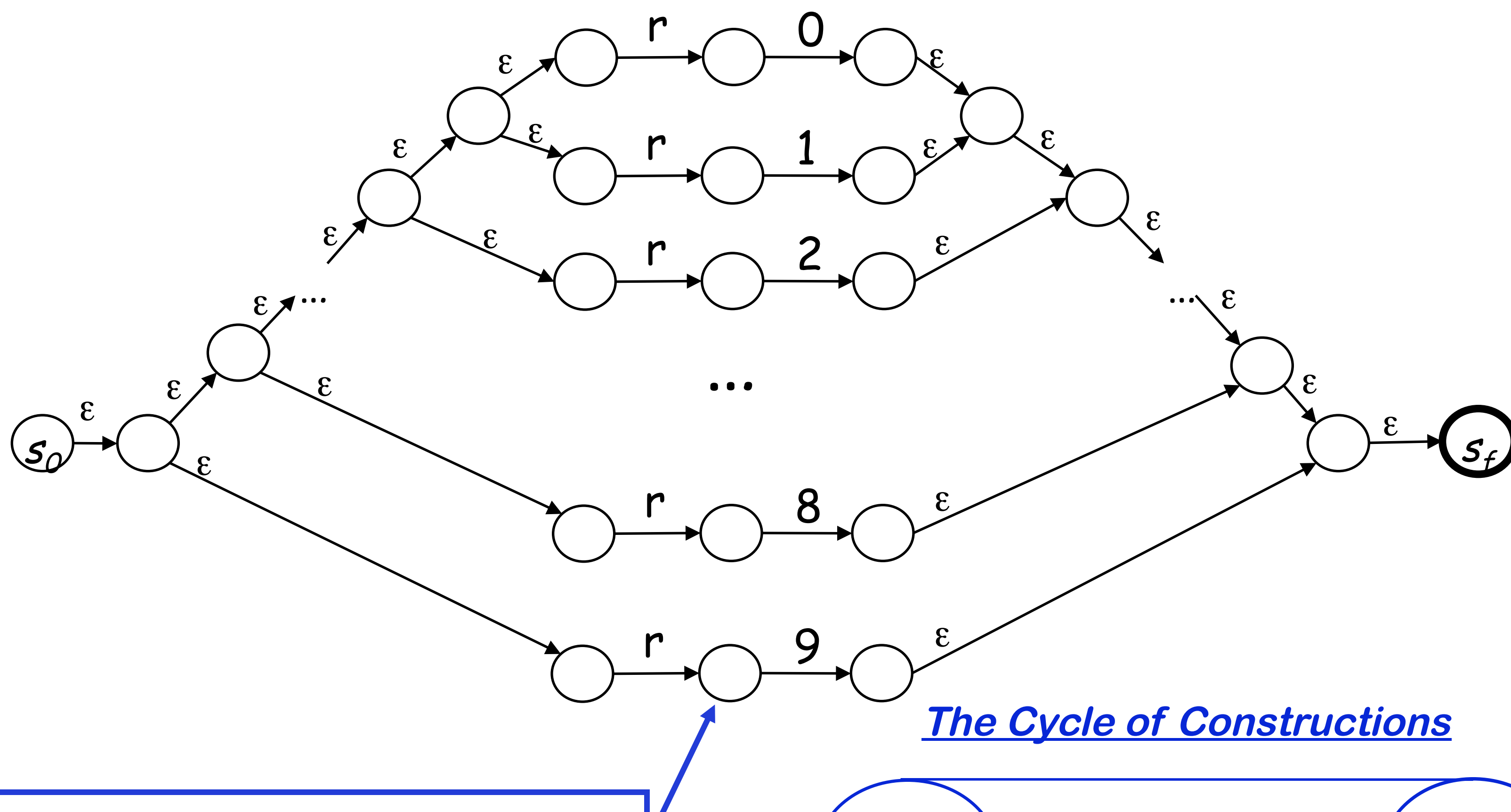
r0 | r1 | r2 | r3 | r4 | r5 | r6 | r7 | r8 | r9

The Cycle of Constructions

RE → NFA → DFA → minimal DFA

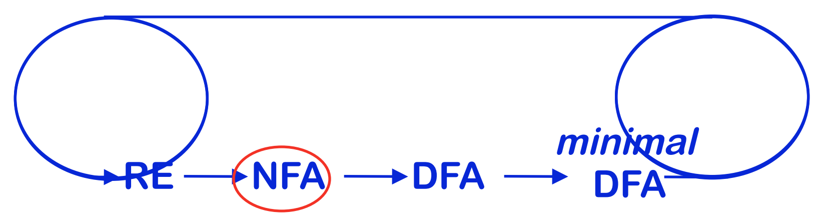
Abbreviated Register Specification

Thompson's construction produces



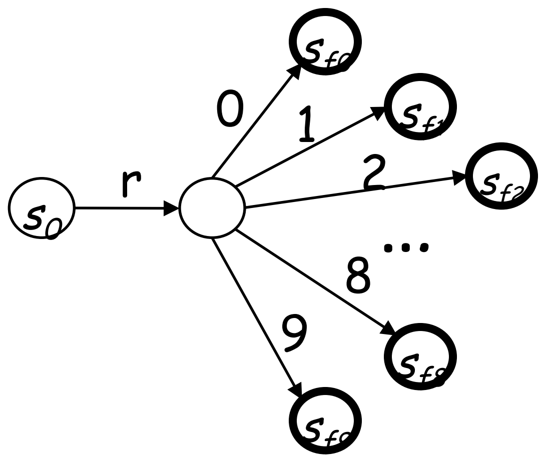
To make it fit, we've eliminated the ε -transition between "r" and "0...9".

The Cycle of Constructions

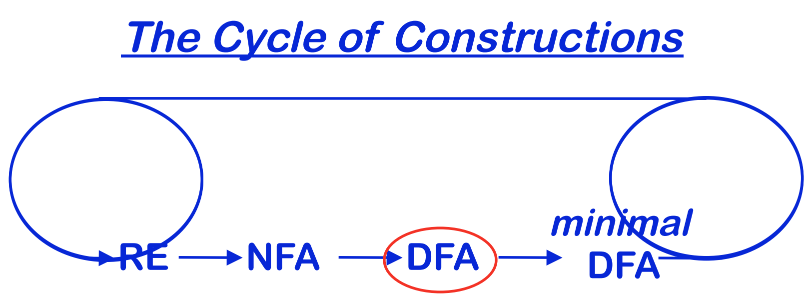


Abbreviated Register Specification

The subset construction builds

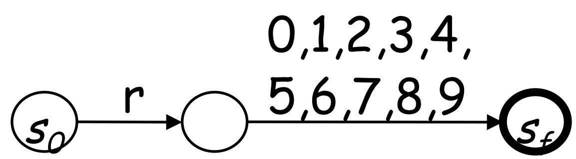


This is a DFA, but it has a lot of states …

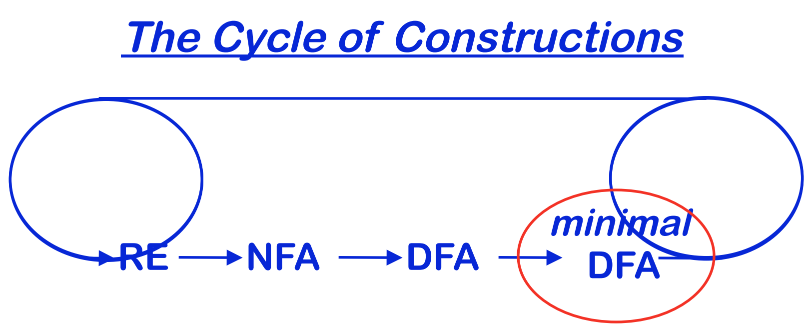


Abbreviated Register Specification

The DFA minimization algorithm builds



This looks like what a skilled compiler writer would do!



Limits of Regular Languages

Advantages of Regular Expressions

- Simple & powerful notation for specifying patterns
- Automatic construction of fast recognizers
- Many kinds of syntax can be specified with REs

Example — an expression grammar

$Term \rightarrow$ [a-zA-Z] ([a-zA-z] | [$\underline{0}$-$\underline{9}$])*

$Op \rightarrow \underline{+} \mid \underline{-} \mid \underline{*} \mid \underline{/}$

$Expr \rightarrow (\ Term\ Op\ )^*\ Term$

Of course, this would generate a DFA …

If REs are so useful …

Why not use them for everything?

Limits of Regular Languages

Not all languages are regular

RL's $\subset$ CFL's $\subset$ CSL's

You cannot construct DFA's to recognize these languages

- $L = \{ p^k q^k \}$ *(parenthesis languages)*

This is not a regular language *(nor an RE)*

But, this is a little subtle. You can construct DFA's for

- Strings with alternating 0's and 1's

  $(\varepsilon \mid 1) (01)^* (\varepsilon \mid 0)$

- Strings with an even number of 0's and 1's

RE's can count bounded sets and bounded differences

What can be so hard?

Poor language design can complicate scanning

- Reserved words are important

  if then then then = else; else else = then (PL/I)

- Insignificant blanks (Fortran & Algol68)

  do 10 i = 1,25

  do 10 i = 1.25

- String constants with special characters (C, C++, Java, …)

  newline, tab, quote, comment delimiters, …

- Finite closures (Fortran 66 & Basic)
  - → Limited identifier length
  - → Adds states to count length

Building Faster Scanners from the DFA

Table-driven recognizers waste effort

- Read (& classify) the next character
- Find the next state
- Assign to the state variable
- Trip through case logic in *action()*
- Branch back to the top

We can do better

- Encode state & actions in the code
- Do transition tests locally
- Generate ugly, spaghetti-like code
- Takes (many) fewer operations per input character

```
char ← next character;
state ← s0 ;
call action(state,char);
while (char ≠ eof)
  state ← δ(state,char);
  call action(state,char);
  char ← next character;

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

Building Faster Scanners from the DFA

A direct-coded recognizer for r *Digit Digit**

- Many fewer operations per character
- Almost no memory operations
- Even faster with careful use of fall-through cases

```
     goto s0;
s0: word ← Ø;
     char ← next character;
     if (char = 'r')
        then goto s1;
        else goto se;
s1: word ← word + char;
     char ← next character;
     if ('0' ≤ char ≤ '9')
        then goto s2;
        else goto se;
```

```
s2: word ← word + char;
     char ← next character;
     if ('0' ≤ char ≤ '9')
        then goto s2;
        else if (char = eof)
            then report success;
            else goto se;
se: print error message;
     return failure;
```

Building Faster Scanners

Hashing keywords versus encoding them directly

- Some *(well-known)* compilers recognize keywords as identifiers and check them in a hash table
- Encoding keywords in the DFA is a better idea
  - → O(1) cost per transition
  - → Avoids hash lookup on each identifier

It is hard to beat a well-implemented DFA scanner

Building Scanners

The point

- All this technology lets us automate scanner construction
- Implementer writes down the regular expressions
- Scanner generator builds NFA, DFA, minimal DFA, and then writes out the (table-driven or direct-coded) code
- This reliably produces fast, robust scanners

For most modern language features, this works

- You should think twice before introducing a feature that defeats a DFA-based scanner
- The ones we've seen (e.g., insignificant blanks, non-reserved keywords) have not proven particularly useful or long lasting