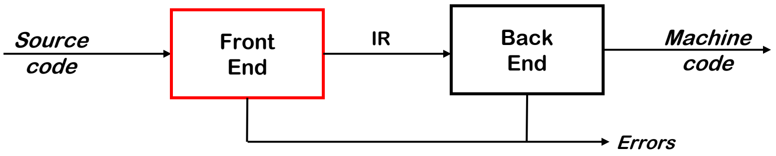
The Front End

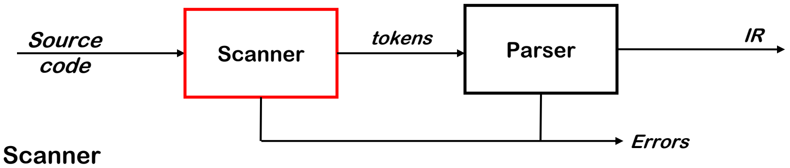


The purpose of the front end is to deal with the input language

- Perform a membership test: code ∈ source language?
- Is the program well-formed (semantically) ?
- Build an IR version of the code for the rest of the compiler

The front end is not monolithic

The Front End



Scanner

- Maps stream of characters into words
    - Basic unit of syntax
    - x = x + y ; *becomes*

      <id,x> <eq,=> <id,x> <pl,+> <id,y> <sc,; >
- Characters that form a word are its *lexeme*
- Its *part of speech* (or *syntactic category*) is called its *token type*
- Scanner discards white space & (often) comments

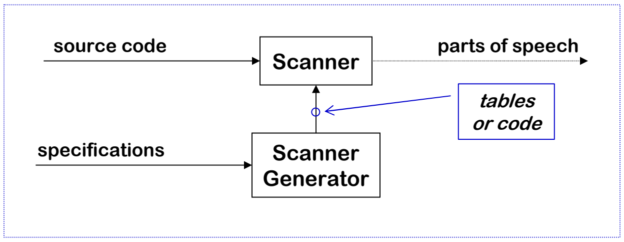
Speed is an issue in scanning

⇒ use a specialized recognizer

The Big Picture

Why study lexical analysis?

- **We want to avoid writing scanners by hand**
- **We want to harness automata theory**



Goals:

- **To simplify specification & implementation of scanners**
- **To understand the underlying techniques and technologies**

Review of Scanners

Lexical Analysis Strategy: Simulation of Finite Automaton

- **States, characters, actions**
- **State transition δ(state,charclass) determines next state**

Next character function

- **Reads next character into buffer**
- **Computes *character class* by fast table lookup**

Transitions from state to state

- **Current state and next character determine (via δ)**
  - **Next state and action to be performed**
  - **Some actions *preload* next character**

Identifiers distinguished from keywords by hashed lookup

- **This differs from EAC advice (discussion later)**
- **Permits translation of identifiers into <type, symbol_index>**
  - **Keywords each get their own type**

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)*

Regular expressions can be used to specify the words to be translated to parts of speech by a lexical analyzer

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

⇒ We study REs and associated theory to automate scanner construction !

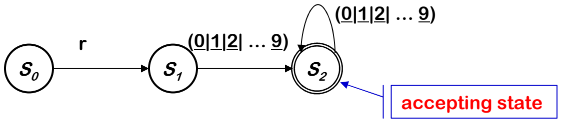
Example *(from Lab 1)*

Consider the problem of recognizing register names

Register → r (0|1|2| … | 9) (0|1|2| … | 9)*

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

RE corresponds to a recognizer (or DFA)



accepting state

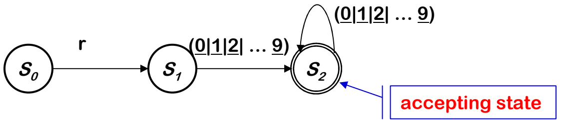
Recognizer for *Register*

Transitions on other inputs go 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)



accepting state

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)

δ / action	r	0,1,2,3, 4,5,6, 7,8,9	*other*
0	*1 / start*	*e / error*	*e / error*
1	*e / error*	*2 / add*	*e / error*
2	*e / error*	*2 / add*	*x / exit*
e	*e / error*	*e / error*	*e / error*

- *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

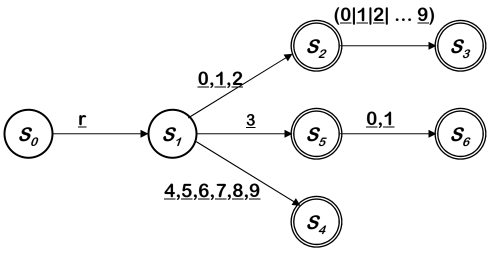
Produces a more complex DFA

- 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)*

state action	r	0,1	2	3	4,5,6 7,8,9	other
0	*1* *start*	*e*	*e*	*e*	*e*	*e*
1	*e*	*2* *add*	*2* *add*	*5* *add*	*4* *add*	*e*
2	*e*	*3* *add*	*3* *add*	*3* *add*	*3* *add*	*x* *exit*
3,4	*e*	*e*	*e*	*e*	*e*	*x* *exit*
5	*e*	*6* *add*	*e*	*e*	*e*	*x* *exit*
6	*e*	*e*	*e*	*e*	*e*	*x* *exit*
e	*e*	*e*	*e*	*e*	*e*	*e*

Automating Scanner Construction

To convert a specification into code:

1 Write down the RE for the input language

2 Build a big NFA

3 Build the DFA that simulates the NFA

4 Systematically shrink the DFA

5 Turn it into code

Scanner generators

- Lex and Flex work along these lines
- Algorithms are well-known and well-understood
- Key issue is interface to parser *(define all parts of speech)*
- You could build one in a weekend!

Automating Scanner Construction

RE→ NFA *(Thompson's construction)*

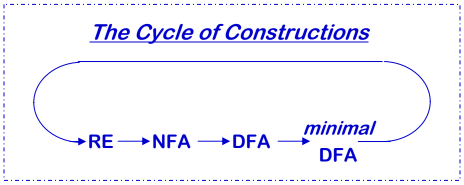
- Build an NFA for each term
- Combine them with ε-moves

NFA → DFA *(subset construction)*

- Build the simulation

DFA → Minimal DFA

- Hopcroft's algorithm



The Cycle of Constructions

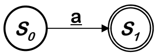
DFA →RE *(Not part of the scanner construction)*

- All pairs, all paths problem
- Take the union of all paths from s_0 to an accepting state
- (Not a particularly obvious algorithm)

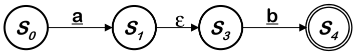
RE →NFA using Thompson's Construction

Key idea

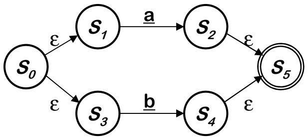
- NFA pattern for each symbol & each operator
- Join them with ε moves in precedence order



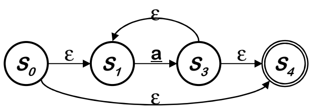
NFA for a



NFA for ab



NFA for a | b



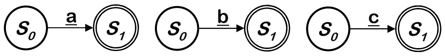
NFA for a*

Ken Thompson, CACM, 1968

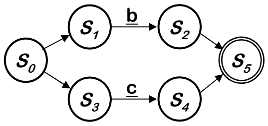
Example of Thompson's Construction

Let's try a (b | c)*

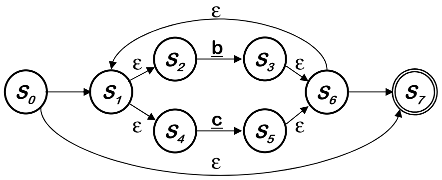
1. a, b, & c



2. b | c

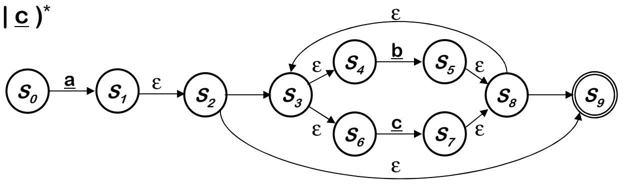


3. (b | c)*

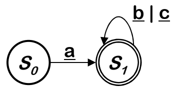


Example of Thompson's Construction *(continued)*

4. a (b | c)*



Of course, a human would design something simpler ...



But, we can automate production of the more complex one ...

NFA →DFA with Subset Construction

Need to build a simulation of the NFA

Two key functions

- *Move*(s_i, a) is set of states reachable from s_i by a
- *ε-closure*(s_i) is set of states reachable from s_i by ε

The algorithm:

- Start state derived from s_0 of the NFA
- Take its ε-closure S_0 = ε-closure(s_0)
- Take the image of S_0 Move(S_0, α) for each $\alpha \in \Sigma$ and take its ε-closure
- Iterate until no more states are added

Sounds more complex than it is…

NFA →DFA with Subset Construction

The algorithm:

```
$s_0 \leftarrow \varepsilon$-closure($q_{0n}$)
while ( S is still changing )
  for each $s_i \in S$
    for each $\alpha \in \Sigma$
      $s_? \leftarrow \varepsilon$-closure(Move($s_i, \alpha$))
      if ( $s_? \notin S$ ) then
        add $s_?$ to S as $s_j$
        $T[s_i, \alpha] \leftarrow s_j$
```

Let's think about why this works

The algorithm halts:

1. *S* contains no duplicates (test before adding)
2. 2^{Qn} is finite
3. while loop adds to *S*, but does not remove from *S (monotone)*

⇒ the loop halts

S contains all the reachable NFA states

It tries each character in each s_i.

It builds every possible NFA configuration.

⇒ *S and T form the DFA*

NFA →DFA with Subset Construction

Example of a *fixed-point* computation

- Monotone construction of some finite set
- Halts when it stops adding to the set
- Proofs of halting & correctness are similar
- These computations arise in many contexts

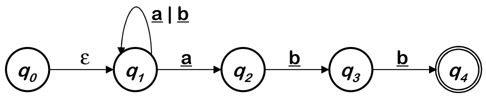
Other fixed-point computations

- Canonical construction of sets of LR(1) items
  - Quite similar to the subset construction
- Classic data-flow analysis (& Gaussian Elimination)
  - Solving sets of simultaneous set equations

We will see many more fixed-point computations

NFA →DFA with Subset Construction

Remember (a | b)* abb ?



Applying the subset construction:

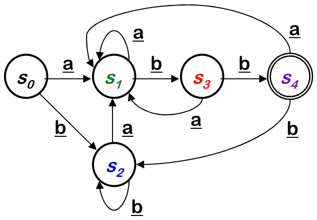
Iter.	State	Contains	ε-closure(move(s_i,a))	ε-closure(move(s_i,b))
0	s_0	q_0, q_1	q_1, q_2	q_1
1	s_1	q_1, q_2	q_1, q_2	q_1, q_3
	s_2	q_1	q_1, q_2	q_1
2	s_3	q_1, q_3	q_1, q_2	q_1, q_4
3	s_4	q_1, q_4	q_1, q_2	q_1

contains q_4 (final state)

Iteration 3 adds nothing to *S*, so the algorithm halts

NFA →DFA with Subset Construction

The DFA for (a | b)* abb



δ	a	b
s_0	s_1	s_2
s_1	s_1	s_3
s_2	s_1	s_2
s_3	s_1	s_4
s_4	s_1	s_2

- Not much bigger than the original
- All transitions are deterministic
- Use same code skeleton as before

Final Task: Generate the Scanner

- How do we specify different token types, etc?
  - One rule per token type
    - RE on right hand side
  - What about ambiguity?
- How do we provide actions in a scanner specified by REs?
  - One simple action: add character to current token
  - More complex actions on token end
    - Part of specification
- How do we generate a scanner for multiple tokens?
  - Combine rules for each token
    - What about ambiguity?
  - How do we identify the tokens (in the input, etc)
    - Legal token if error (or end) transition taken from accepting state
    - Leave error char in input buffer

A Lex Specification, Part I

```
%{
            /* definition of constants BEGIN, END, NAME,
              NUM, STRNG, SPCL, PLUS, MINUS, LT, LE */
%}
/* Regular Definitions */
blank           [ ]
lb              [{]
rb              [}]
comment         {lb}.*{rb} /* . = any but newline */
ws              ({blank}|{comment})+
letter          [A-Za-z]
digit           [0-9]
name            {letter}({letter}|{digit})*
numb            {digit}+
quote           ["]
string          {quote}.*{quote}
%%
```

Lex Specification, Part II

```
/* Translation Rules */
{ws}                    {/* no action and no return */}
begin                   {return(BEGIN)}
end                     {return(END)}
{name}                  {yylval = install_name(); return(NAME);}
{number} {yylval = install_num(); return(NUM);}
{string}    {yylval = install_str(); return(STRNG);}
"+"                     {yylval = PLUS; return(SPCL);}
"-"                     {yylval = MINUS; return(SPCL);}
"<"                     {yylval = LT; return(SPCL);}
"<="                    {yylval = LE; return(SPCL);}
%%
install_name() {
            /* procedure to install the lexeme
              whose first character is pointed to by yytext
              and whose length is yyleng into symbol table
              and return pointer to entry */
}
```

Automating Scanner Construction

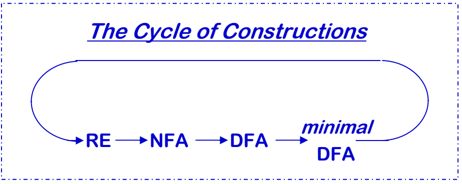
RE→NFA ***(Thompson's construction)***

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

NFA →DFA ***(subset construction)***

- **Build the simulation**

DFA →Minimal DFA


The Cycle of Constructions

- **Hopcroft's algorithm**

DFA →RE (not really part of scanner construction)

- **All pairs, all paths problem**
- **Union together paths from s_0 to a final state**

DFA Minimization

The Big Picture

- Discover sets of equivalent states
- Represent each such set with just one state

Two states are equivalent if and only if:

- The set of paths leading to them are equivalent
- $\forall \alpha \in \Sigma$, transitions on α lead to equivalent states **(DFA)**
- α -transitions to distinct sets $\Rightarrow$ states must be in distinct sets

A partition *P* of *S*

- Each $s \in S$ is in exactly one set $p_i \in P$
- The algorithm iteratively partitions the DFA's states

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: *{F}* & *{Q-F}* $(D = (Q,\Sigma,\delta,q_0,F))$

Splitting a set ("partitioning a set by <u>a</u>")

- Assume q_a, & $q_b \in s$, and
- $\delta(q_a,\underline{a}) = q_x$, & $\delta(q_b,\underline{a}) = q_y$
- If q_x & q_y are not in the same set, then *s* must be split
- One state in the final DFA cannot have two transitions on <u>a</u>

DFA Minimization

The algorithm

```
$P \leftarrow \{ F, \{Q-F\}\}$
while ( $P$ is still changing)
  $T \leftarrow \{ \}$
  for each set $s \in P$
    for each $\alpha \in \Sigma$
      partition $s$ by $\alpha$
        into $s_1, s_2, \ldots, s_k$
      $T \leftarrow T \cup s_1, s_2, \ldots, s_k$
  if $T \neq P$ then
    $P \leftarrow T$
```

This is a fixed-point algorithm!

Why does this work?

- Partition $P \in 2^Q$
- Start off with 2 subsets of Q
  {F} and {Q-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 $|Q|$ sets
- Maximum of $|Q|$ splits

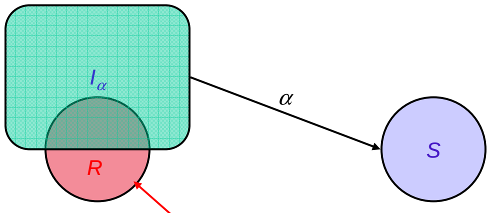
Note that

- Partitions are never combined
- Initial partition ensures that final states are intact

Hopcroft's Algorithm

```
$W \leftarrow \{F, Q-F\}$;  $P \leftarrow \{F, Q-F\}$;  // W is the worklist, P the current partition
while ( $W$ is not empty ) do begin
    select and remove $S$ from $W$;  // $S$ is a set of states
    for each $\alpha$ in $\Sigma$ do begin
        let $I_\alpha \leftarrow \delta_\alpha^{-1}( S )$;
        for each $R$ in $P$ such that $R \cap I_\alpha$ is not empty
            and $R$ is not contained in $I_\alpha$ do begin
            partition $R$ into $R_1$ and $R_2$ such that $R_1 \leftarrow R \cap I_\chi$; $R_2 \leftarrow R - R_1$;
            replace $R$ in P with $R_1$ and $R_2$ ;
            if $R \in W$ then replace $R$ with $R_1$ in $W$ and add $R_2$ to $W$;
            else if $|| R_1 || \leq || R_2 ||$
                then add add $R_1$ to $W$;
                else add $R_2$ to $W$;
        end
    end
end
```

Key Idea



This part must have an α-transition to some other state!

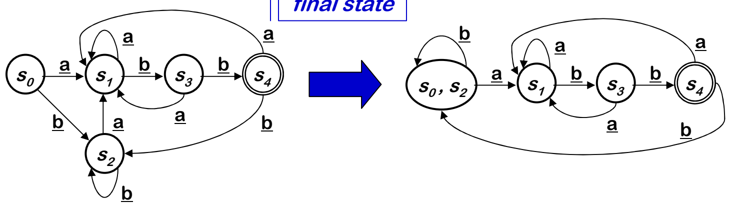
DFA Minimization

Enough theory, does this stuff work?

- **Recall our example: $(\underline{a} \mid \underline{b})^{*}\ \underline{abb}$**

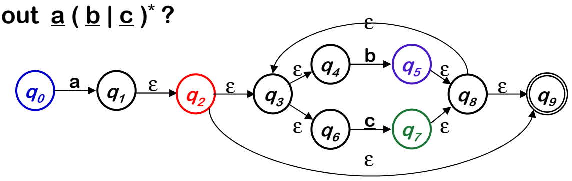
	Current Partition	*W*	*s*	*Split on a*	*Split on b*
P_0	$\{s_4\}\ \{s_0, s_1, s_2, s_3\}$	$\{s_4\}\ \{s_0, s_1, s_2, s_3\}$	$\{s_4\}$	none	$\{s_0, s_1, s_2\}\ \{s_3\}$
P_1	$\{s_4\}\{s_3\}\{s_0, s_1, s_2\}$	$\{s_0, s_1, s_2\}\ \{s_3\}$	$\{s_3\}$	none	$\{s_0, s_2\}\{s_1\}$
P_2	$\{s_4\}\{s_3\}\{s_1\}\{s_0, s_2\}$	$\{s_0, s_2\}\{s_1\}$	$\{s_1\}$	none	none

final state



DFA Minimization

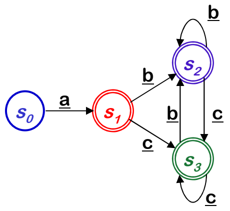
What about $\underline{a}\,(\,\underline{b}\mid\underline{c}\,)^*$?



First, the subset construction:

		ε-closure(move(s,*))		
	NFA states	$\underline{a}$	$\underline{b}$	$\underline{c}$
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

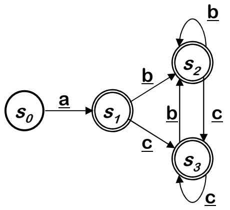


DFA Minimization

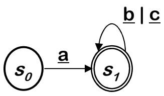
Then, apply the minimization algorithm

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

final states



To produce the minimal DFA



In lecture 5, we observed that a human would design a simpler automaton than Thompson's construction did.

The algorithms produce that same DFA!

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 → [a-zA-Z] ([a-zA-z] | [0-9])*
Op → + | - | * | /
Expr → (*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 ⊂ CFL's ⊂ CSL's

You cannot construct DFA's to recognize these languages

- $L = \{ p^k q^k \}$ *(parenthesis languages)*
- $L = \{ wcw^r \mid w \in \Sigma^* \}$

Neither of these is 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 and even number of 0's and 1's
  See Homework 1!

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)**
- Significant blanks **(Fortran & Algol68)**
  do 10 i = 1,25
  do 10 i = 1.25
- String constants with special characters **(C, others)**
  newline, tab, quote, comment delimiters, …
- Finite closures
  - Limited identifier length
  - Adds states to count length

What can be so hard? *(Fortran 66/77)*

```
        INTEGERFUNCTIONA
        PARAMETER(A=6,B=2)
        IMPLICIT CHARACTER*(A-B)(A-B)
        INTEGER FORMAT(10), IF(10), DO9E1
100     FORMAT(4H)=(3)
200     FORMAT(4    )=(3)
        DO9E1=1
        DO9E1=1,2
  9     IF(X)=1
        IF(X)H=1
        IF(X)300,200
300     CONTINUE
        END
C       THIS IS A “COMMENT CARD”
     $  FILE(1)
        END
```

How does a compiler do this?

- First pass finds & inserts blanks
- Can add extra words or tags to create a scanable language
- Second pass is normal scanner

Example due to Dr. F.K. Zadeck

Building Faster Scanners from the DFA

Table-driven recognizers waste a lot of 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

```
char ← next character;
state ← $s_0$;
call action(state,char);
while (char ≠ eof)
  state ← $\delta$(state,char);
  call action(state,char);
  char ← next character;

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

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

Building Faster Scanners from the DFA

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

```
    goto $s_0$;
$s_0$: word ← Ø;
    char ← next character;
    if (char = 'r')
      then goto $s_1$;
      else goto $s_e$;
$s_1$: word ← word + char;
    char ← next character;
    if ('0' ≤ char ≤ '9')
      then goto $s_2$;
      else goto $s_e$;
s2: word ← word + char;
    char ← next character;
    if ('0' ≤ char ≤ '9')
      then goto $s_2$;
      else if (char = eof)
        then report acceptance;
        else goto $s_e$;
$s_e$: print error message;
    return failure;
```

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

Building Faster Scanners

Hashing keywords versus encoding them directly

- **Some compilers recognize keywords as identifiers and check them in a hash table** *(some well-known compilers do this!)*
- **Encoding it 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