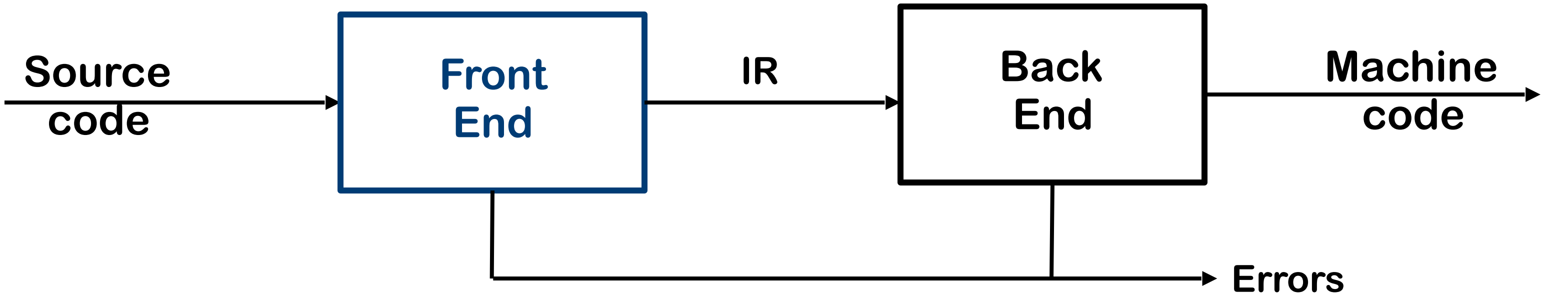
Lexical Analysis

The Front End

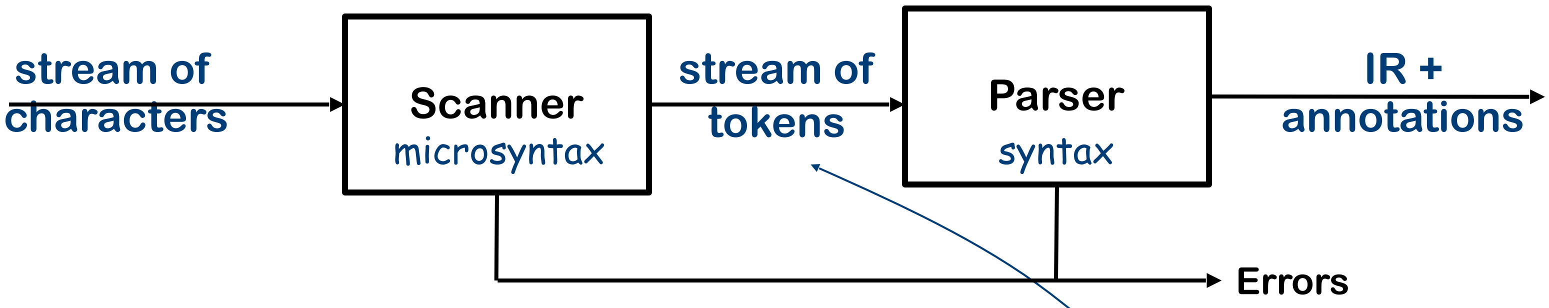


The purpose of the front end is to deal with the input language

- 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 deals with form (syntax) & meaning (semantics)

The Front End



Scanner is only pass that touches every character of the input.

Why separate the scanner and the parser?

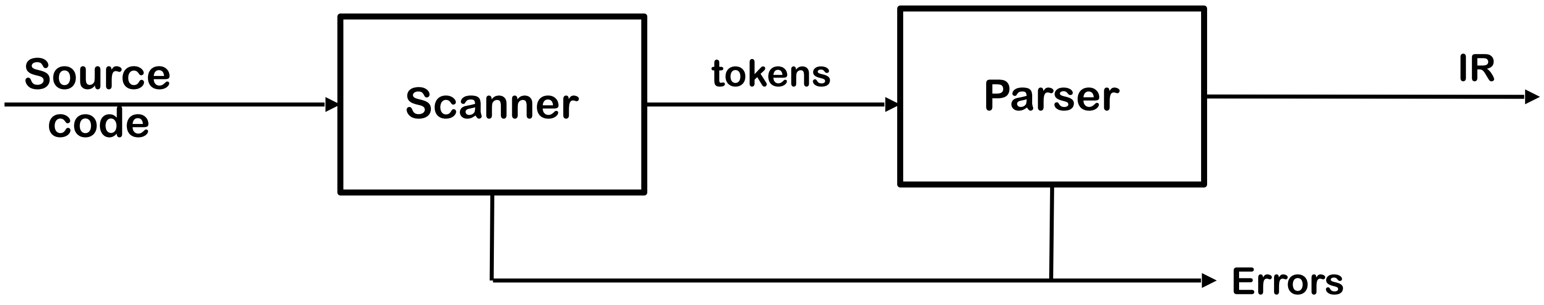
- Scanner classifies words
- Parser constructs grammatical derivations
- Parsing is harder and slower

token is a pair
<part of speech, lexeme>

Separation simplifies the implementation

- Scanners are simple
- Scanner leads to a faster, smaller parser

Our setting: the Front End



Implementation Strategy

	Scanning	Parsing
Specify Syntax	regular expressions	context-free grammars
Implement Recognizer	deterministic finite automaton	push-down automaton
Perform Work	Actions on transitions in automaton	

Lexical Analysis

Relates to the words of the vocabulary of a language, (as opposed to grammar, i.e., correct construction of sentences).

Lexical Analyzer, a.k.a. lexer, scanner or tokenizer, splits the input program, seen as a stream of characters, into a sequence of tokens.

Tokens are the words of the (programming) language, e.g., keywords, numbers, comments, parenthesis, semicolon.

Tokens are classes of concrete input (called lexeme).

Example

```
#include <stdio.h>
    int maximum(int x, int y) {
        // This will compare 2 numbers
        if (x > y)
            return x;
        else {
            return y;
        }
```

Token created

int	Keyword
maximum	Identifier
(	Operator
int	Keyword
x	Identifier
,	Operator
int	Keyword
y	Identifier
(	Operator
{	Operator
if	Keyword

Non-Token

Comment	// This will compare 2 numbers
Pre-processor directive	#include <stdio.h>
Pre-processor directive	#define NUMS 8,9
Macro	NUMS

Lexical analysis

- Lexical analysis is the very **first phase** in the compiler designing, the only one that analyses the entire code

- A **lexeme** is a sequence of characters that are included in the source program according to the matching pattern of a **token**

- Lexical analyzer helps to **identify token** into the symbol table

- A character sequence which is not possible to scan into any valid token is a **lexical error**

Constructing a Lexical Analyser

- By hand - Identify lexemes in input and return tokens
- Automatically - Lexical-Analyser generator: it compiles the patterns

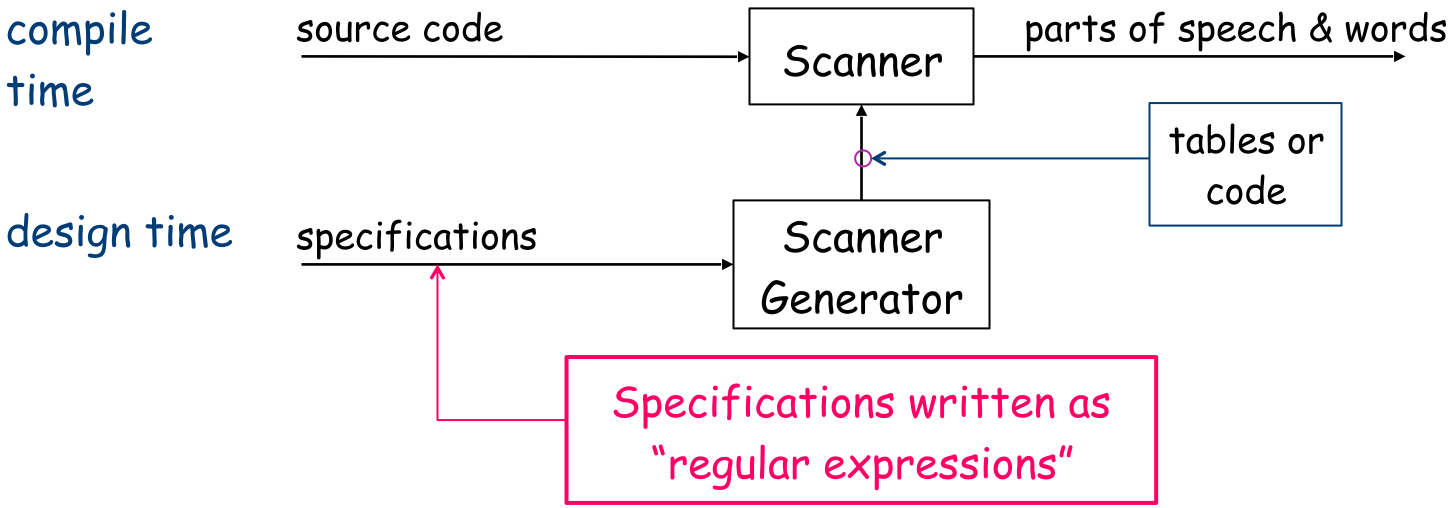
that specify the lexemes into a code (the lexical analyser).

Lexical analysis decides whether the individual tokens are well formed, this can be expressed by a regular language.

Automatic Scanner Construction

Why study automatic scanner construction?

- Avoid writing scanners by hand



In practice, many scanners are hand coded. Even if you build a hand-coded scanner, it is useful to write down the specification and the automaton.

The syntax can be expressed by a regular grammar

The syntax of a programming language can be expressed by a regular grammar

Example

The following grammar generates all the legal identifier

S-> aT|…|zT|AT|…|ZT

T-> ε|0T|…|9T|S

that can be more neatly be expressed using a regular expression!

(a|…|z|A|…|Z) (a|…|z|A|…|Z|0|….|9)*

Examples of Regular Expressions

Identifiers:

Letter → (a|b|c| … |z|A|B|C| … |Z)

Digit → (0|1|2| … |9)

Identifier → Letter (Letter | Digit)*

shorthand for

(a|b|c| … |z|A|B|C| … |Z) ((a|b|c| … |z|A|B|C| … |Z) | (0|1|2| … |9))*

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!

underlining indicates a letter in the input stream

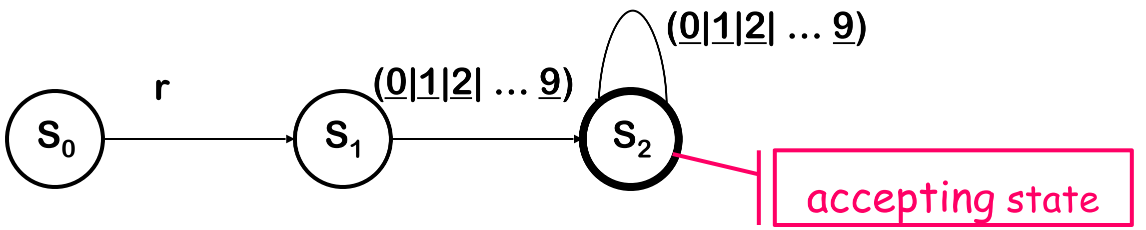
Example

Consider the problem of recognizing ILOC register names

$$\text{Register} \rightarrow \text{r} \ (\underline{0}|\underline{1}|\underline{2}| \ldots | \underline{9}) \ (\underline{0}|\underline{1}|\underline{2}| \ldots | \underline{9})^*$$

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

RE corresponds to a recognizer (or DFA)



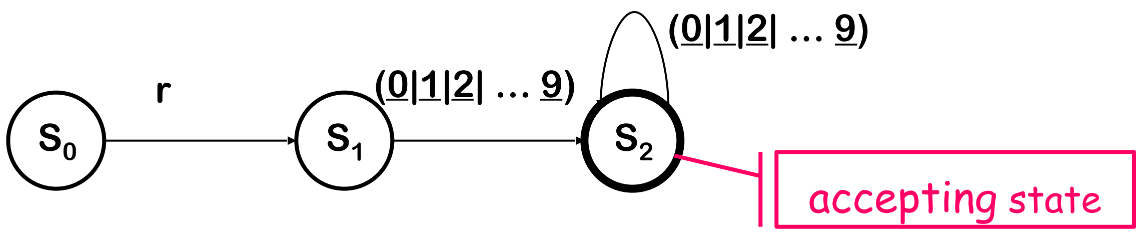
Recognizer for Register

Transitions on other inputs go to an error state, s_e

Example (continued)

DFA operation

- Start in state S_0 & make transitions on each input character

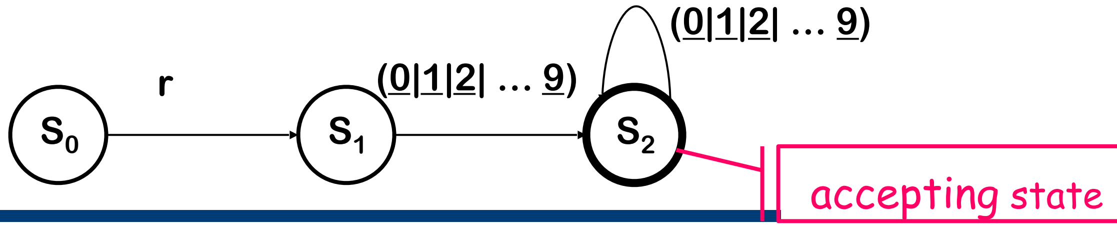


Recognizer for Register

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



To be useful, the recognizer must be converted into code

```
Char ← next character
State ← s0

while (Char ≠ EOF)
    State ← δ(State,Char)
    Char ← next character

if (State is a final state )
    then report success
    else  report failure
```

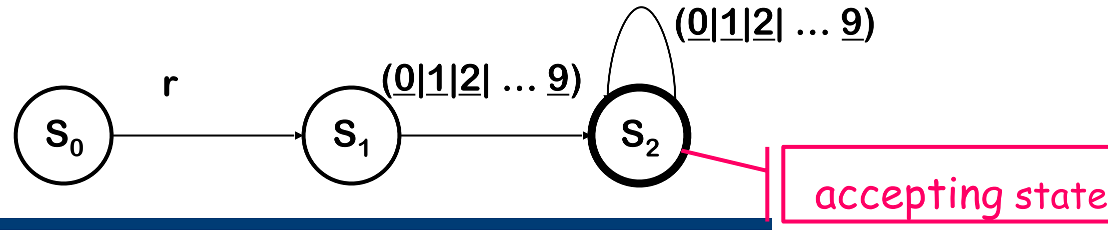
Skeleton recognizer

δ	r	0,1,2,3,4, 5,6,7,8,9	All others
s_0	s_1	s_e	s_e
s_1	s_e	s_2	s_e
s_2	s_e	s_2	s_e
s_e	s_e	s_e	s_e

Table encoding the RE

O(1) cost per character   (or per transition)

Example



We can add "actions" to each transition

```
Char ← next character
State ← s0

while (Char ≠ EOF)
    Next ← δ(State,Char)
    Act  ← α(State,Char)
    perform action Act
    State ← Next
    Char ← next character

if (State is a final state )
    then report success
    else  report failure
```

Skeleton recognizer

δ α	r	0,1,2,3,4, 5,6,7,8,9	All others
s_0	s_1 start	s_e error	s_e error
s_1	s_e error	s_2 add	s_e error
s_2	s_e error	s_2 add	s_e error
s_e	s_e error	s_e error	s_e error

Table encoding RE

Typical action is to capture the lexeme

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

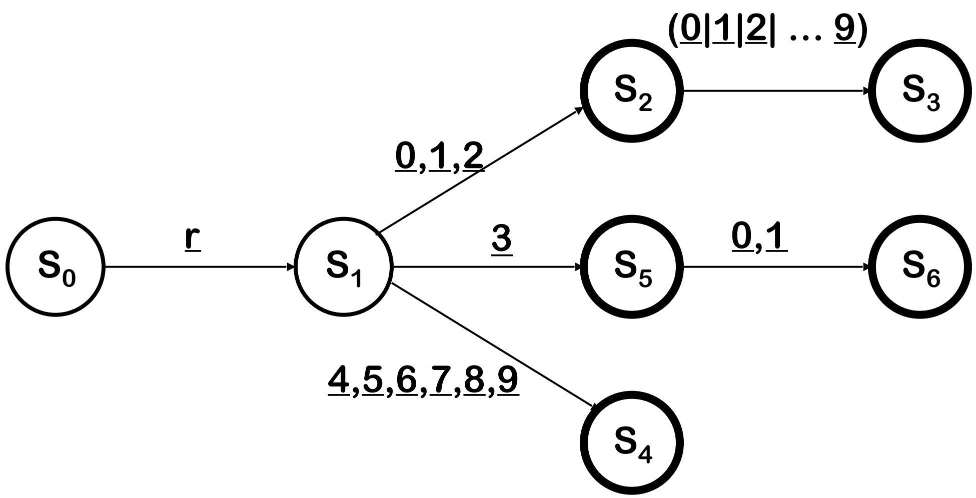
- DFA has more states
- DFA has same cost per transition
- DFA has same basic implementation

> More states implies a larger table. The larger table might have mattered when computers had 128 KB or 640 KB of RAM. Today, when a cell phone has megabytes and a laptop has gigabytes, the concern seems outdated.

Tighter register specification (continued)

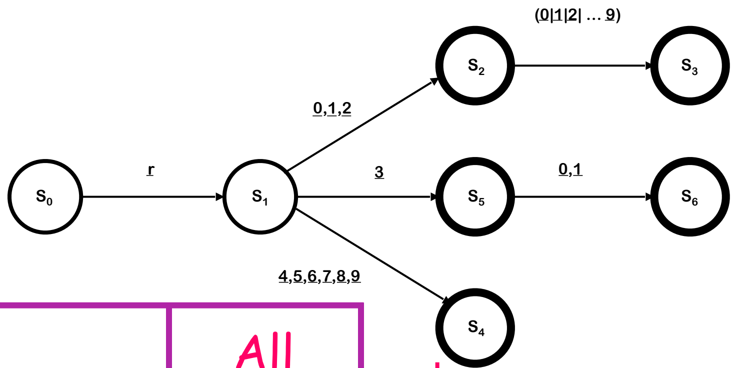
The DFA for

Register → r ((0|1|2) (Digit | ε) | (4|5|6|7|8|9) | (3|30|31))



- Accepts a more constrained set of register names
- Same set of actions, more states

Tighter register specification

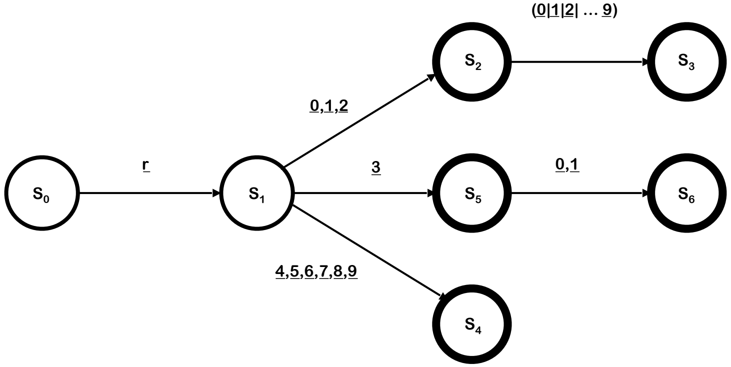


δ	r	0,1	2	3	4-9	All others
s_0	s_1	s_e	s_e	s_e	s_e	s_e
s_1	s_e	s_2	s_2	s_5	s_4	s_e
s_2	s_e	s_3	s_3	s_3	s_3	s_e
s_3	s_e	s_e	s_e	s_e	s_e	s_e
s_4	s_e	s_e	s_e	s_e	s_e	s_e
s_5	s_e	s_6	s_e	s_e	s_e	s_e
s_6	s_e	s_e	s_e	s_e	s_e	s_e
s_e	s_e	s_e	s_e	s_e	s_e	s_e

This table runs in the same skeleton recognizer

Table encoding RE for the tighter register specification

Tighter register specification



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	e exit
3,4,6	e	e	e	e	e	e exit
5	e	6 add	e	e	e	e 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

1 Build a ε-NFA collecting all the NFA for the RE

2 Build a NFA corresponding to the ε-NFA

3 Build the DFA that simulates the NFA

4 Systematically minimise 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
- You could build one in a weekend!

The overall construction: RE-> ε-NFA->NFA->DFA->minimized DFA

How we transform a DFA into code? 3 different approaches

- Table driven scanners
- Direct code scanners
- Hand-coded scanners

all will simulate the DFA!

The actions in common

- They repeatedly read the next character in the input and simulate the corresponding DFA transition

- This process stops when the DFA recognises a word: there are not outgoing transition from the state s with the input character

- If s is an accepting state the scanner recognises the word and its syntactic category

- If s is a nonaccepting state the scanner must determine whether or not it passes a final state at some point, if yes the scanner should roll back its internal state and its input stream and report success

The differences

- Table driven scanners

- Direct code scanners

- Hand-coded scanners

All constant cost per character (with different constants) +
the cost of rollback

Differs from the way they implement the transition table
and simulate the operations of the DFA

Table-Driven Scanners

- Table(s) + Skeleton Scanner
  - So far, we have used a simplified skeleton

```
state ← s0 ;
while (state ≠ serror ) do
   char ← NextChar( )            // read next character
   state ← δ(state,char);        // take the transition
```

- In practice, the skeleton is more complex
  - Character classification for table compression
  - Building the lexeme
  - Recognizing subexpressions
    - → Practice is to combine all the REs into one DFA
    - → Must recognize individual words without hitting EOF

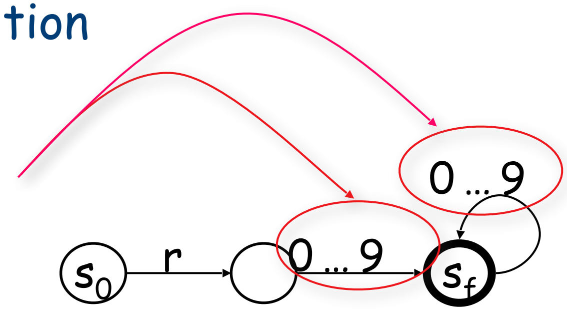


Table-Driven Scanners

Character Classification

- Group together characters by their actions in the DFA
  - Combine identical columns in the transition table, δ
  - Indexing δ by class shrinks the table

```
state ← s0 ;
while (state ≠ serror) do
   char ← NextChar( )          // read next character
   cat ← CharCat(char)         // classify character
   state ← δ(state,cat)        // take the transition
```

r	0, 1, 2, ..., 9	EOF	**Other**
Register	*Digit*	*Other*	*Other*

The Classifier Table, CharCat

	Register	***Digit***	***Other***
s_0	s_1	s_e	s_e
s_1	s_e	s_2	s_e
s_2	s_e	s_2	s_e
s_e	s_e	s_e	s_e

The Transition Table, δ

Table-Driven Scanners

Building the Lexeme

- Scanner produces syntactic category (part of speech)
  - Most applications want the lexeme (word), too

```
state ← s0
lexeme ← empty string
while (state ≠ s_error) do
    char ← NextChar( )          // read next character
    lexeme ← lexeme + char      // concatenate onto lexeme
    cat ← CharCat(char)         // classify character
    state ← δ(state,cat)        // take the transition
```

- This problem is trivial
  - Save the characters

Table-Driven Scanners

Choosing a Category from an Ambiguous RE

- We want one DFA, so we combine all the REs into one
  - Some strings may fit RE for more than 1 syntactic category
    - → Keywords versus general identifiers
    - → Would like to encode them into the RE & recognize them
  - Scanner must choose a category for ambiguous final states
    - → Classic answer: specify priority by order of REs (return 1st)

Identifiers:

Letter → (a|b|c| … |z|A|B|C| … |Z)

Digit → (0|1|2| … |9)

Identifier → Letter (Letter | Digit)*

Keywords:

key → if |….   ife

A table driven scanner for register names

initialization

```
NextWord()
   state ← s0 ;
   lexeme ← " ";
   clear stack;
   push(bad);
```

scanning loop

```
   while (state≠se) do
      NextChar(char);
      lexeme ← lexeme + char;
      if state ∈ SA     final states
         then clear stack;
      push(state);
      cat ← CharCat[char];
      state ← δ[state,cat];
   end;
```

roll-back

```
   while(state ∉ SA and
         state≠bad) do
      state ← pop();
      truncate lexeme;
      RollBack();
   end;
```

final-section

```
   if state ∈ SA
      then return Type[state];
      else return invalid;
```

r	0, 1, 2, ..., 9	EOF	**Other**
Register	*Digit*	*Other*	*Other*

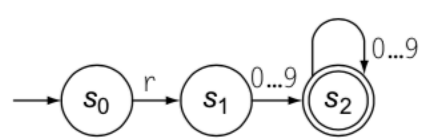
The Classifier Table, CharCat

	Register	***Digit***	***Other***
s_0	s_1	s_e	s_e
s_1	s_e	s_2	s_e
s_2	s_e	s_2	s_e
s_e	s_e	s_e	s_e

The Transition Table, δ

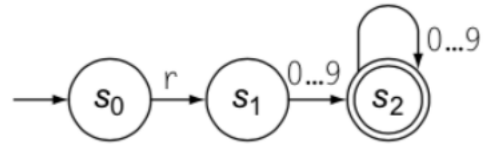
s_0	s_1	s_2	s_e
invalid	*invalid*	*register*	*invalid*

The Token Type Table, Type



The Underlying DFA

Direct-Coded scanners



The Underlying DFA

For each character, the table driven scanner performs two table lookups, one in CharCat and the other in δ

```
s_init : lexeme ← " ";
         clear stack;
         push(bad);
         goto s0;

s0 :     NextChar(char);
         lexeme ← lexeme + char;
         if state ∈ S_A
             then clear stack;
         push(state);
         if (char='r')
             then goto s1;
             else goto s_out;

s1 :     NextChar(char);
         lexeme ← lexeme + char;
         if state ∈ S_A
             then clear stack;
         push(state);
         if ('0'≤char≤'9')
             then goto s2;
             else goto s_out;

s2 :     NextChar(char);
         lexeme ← lexeme + char;
         if state ∈ S_A
             then clear stack;
         push(state);
         if '0'≤char≤'9'
             then goto s2;
             else goto s_out

s_out :  while (state ∉ S_A and
                state ≠ bad) do
             state ← pop();
             truncate lexeme;
             RollBack();
         end;
```

If end of state test is complex (e.g., many cases), scanner generator should consider other schemes

- Table lookup (with classification?)
- Binary search

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

Of course, not everything fits into a regular language …

What About Hand-Coded Scanners?

Many (most?) modern compilers use hand-coded scanners

- Starting from a DFA simplifies design & understanding
- Avoiding straight-jacket of a tool allows flexibility
  - Computing the value of an integer
    - → In LEX or FLEX, many folks use sscanf()
    - → Can use old assembly trick and compute value as it appears
  - Combine similar states
- Scanners are fun to write
  - Compact, comprehensible, easy to debug