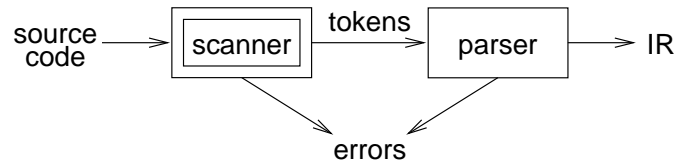


Front end



Scanner:

- maps characters into *tokens* – the basic unit of syntax

`x = x + y;`

becomes

`<id,x> = <id,x> + <id,y>;`

- character string value for a token is a *lexeme*
- typical tokens: `number`, `id`, `+`, `-`, `*`, `/`, `do`, `end`
- eliminates white space (*tabs*, *blanks*, *comments*)
- a key issue is speed
⇒ use specialized recognizer

Specifying patterns

A scanner must recognize various parts of the language's syntax

Some parts are easy:

white space

```

<ws> ::= <ws> ' '
      | <ws> '\t'
      | ' '
      | '\t'
  
```

keywords and operators

specified as literal patterns: `do`, `end`

comments

opening and closing delimiters: `/* ... */`

Specifying patterns

A scanner must recognize various parts of the language's syntax

Other parts are much harder:

identifiers

alphanumeric followed by k alphanumerics ($_$, $\$$, $\&$, ...)

numbers

integers: 0 or digit from 1-9 followed by digits from 0-9

decimals: integer '.' digits from 0-9

reals: (integer or decimal) 'E' (+ or -) digits from 0-9

complex: '(' real ',' real ')'

We need a powerful notation to specify these patterns

Operations on languages

Operation	Definition
union of L and M written $L \cup M$	$L \cup M = \{s \mid s \in L \text{ or } s \in M\}$
concatenation of L and M written LM	$LM = \{st \mid s \in L \text{ and } t \in M\}$
Kleene closure of L written L^*	$L^* = \bigcup_{i=0}^{\infty} L^i$
positive closure of L written L^+	$L^+ = \bigcup_{i=1}^{\infty} L^i$

Regular expressions

Patterns are often specified as *regular languages*
 Notations used to describe a regular language (or a regular set) include both *regular expressions* and *regular grammars*

Regular expressions (over an alphabet Σ):

1. ϵ is a RE denoting the set $\{\epsilon\}$
2. if $a \in \Sigma$, then a is a RE denoting $\{a\}$
3. if r and s are REs, denoting $L(r)$ and $L(s)$, then:
 - (r) is a RE denoting $L(r)$
 - $(r)|(s)$ is a RE denoting $L(r) \cup L(s)$
 - $(r)(s)$ is a RE denoting $L(r)L(s)$
 - $(r)^*$ is a RE denoting $L(r)^*$

If we adopt a *precedence* for operators, the extra parentheses can go away. We assume *closure*, then *concatenation*, then *alternation* as the order of precedence.

Examples

identifiers

$$\begin{aligned} \textit{letter} &\rightarrow (a \mid b \mid c \mid \dots \mid z \mid A \mid B \mid C \mid \dots \mid Z) \\ \textit{digit} &\rightarrow (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9) \\ \textit{id} &\rightarrow \textit{letter} (\textit{letter} \mid \textit{digit})^* \end{aligned}$$

numbers

$$\begin{aligned} \textit{integer} &\rightarrow (+ \mid - \mid \epsilon) (0 \mid (1 \mid 2 \mid 3 \mid \dots \mid 9) \\ &\quad \textit{digit}^*) \\ \textit{decimal} &\rightarrow \textit{integer} . (\textit{digit})^* \\ \textit{real} &\rightarrow (\textit{integer} \mid \textit{decimal}) E (+ \mid -) (\textit{digit})^* \\ \textit{complex} &\rightarrow '(\textit{real} , \textit{real})' \end{aligned}$$

Numbers can get much more complicated

Most programming language tokens can be described with REs

We can use REs to build scanners automatically

Examples

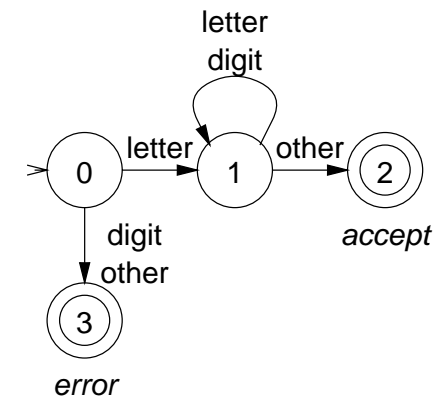
Let $\Sigma = \{a, b\}$

1. $a|b$ denotes $\{a, b\}$
2. $(a|b)(a|b)$ denotes $\{aa, ab, ba, bb\}$
i.e., $(a|b)(a|b) = aa|ab|ba|bb$
3. a^* denotes $\{\epsilon, a, aa, aaa, \dots\}$
4. $(a|b)^*$ denotes the set of all strings of a 's and b 's
(including ϵ)
i.e., $(a|b)^* = (a^*b^*)^*$
5. $a|a^*b$ denotes $\{a, b, ab, aab, aaab, aaaab, \dots\}$

Recognizers

From a regular expression we can construct a *deterministic finite automaton* (DFA)

Recognizer for identifier :



Identifier:

$$letter \rightarrow (a | b | c | \dots | z | A | B | C | \dots | Z)$$

$$digit \rightarrow (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)$$

$$id \rightarrow letter (letter | digit)^*$$

Code for the recognizer

```

char = next_char();
state = 0; /* code for state 0 */
done = false;
token_value = "" /* empty string */
while (not done)
{
    class = char_class[char];
    state = next_state[class,state];
    switch(state)
    {
        case 1: /* building an id */
            token_value = token_value + char;
            char = next_char();
            break;
        case 2: /* accept state */
            token_type = identifier;
            done = true;
            break;
        case 3: /* error */
            token_type = error;
            done = true;
            break;
    }
}
return token_type;

```

Tables for the recognizer

Two tables control the recognizer

char_class:		<i>a - z</i>		<i>A - Z</i>		<i>0 - 9</i>		oth
value		letter		letter		digit		oth

next_state:	class		0		1		2		3
letter			1		1		-		-
digit			3		1		-		-
other			3		2		-		-

To change languages, we can just change tables

Automatic construction

Scanner generators automatically construct code from regular expression-like descriptions

- construct a dfa
- use state minimization techniques
- emit code for the scanner
(table driven or direct code)

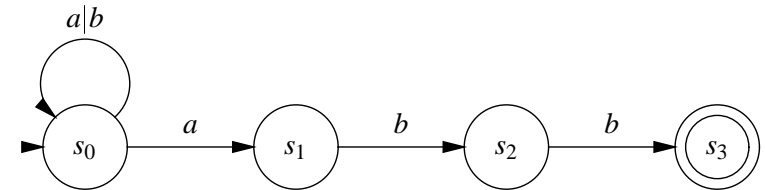
A key issue in automation is an interface to the parser

JavaCC:

- emits Java code for scanner
- provides macro definitions for each token
(used in the parser)

From regular expressions to finite automata

What about the RE $(a|b)^*abb$?



State s_0 has multiple transitions on a !
 \Rightarrow *nondeterministic finite automaton*

	a	b
s_0	$\{s_0, s_1\}$	$\{s_0\}$
s_1	—	$\{s_2\}$
s_2	—	$\{s_3\}$

Finite automata

A *non-deterministic finite automaton* (NFA) consists of:

1. a set of *states* $S = \{s_0, \dots, s_n\}$
2. a set of input symbols Σ (the alphabet)
3. a transition function *move* mapping state-symbol pairs to sets of states
4. a distinguished *start state* s_0
5. a set of distinguished *accepting* or *final* states F

A *Deterministic Finite Automaton* (DFA) is a special case of an NFA:

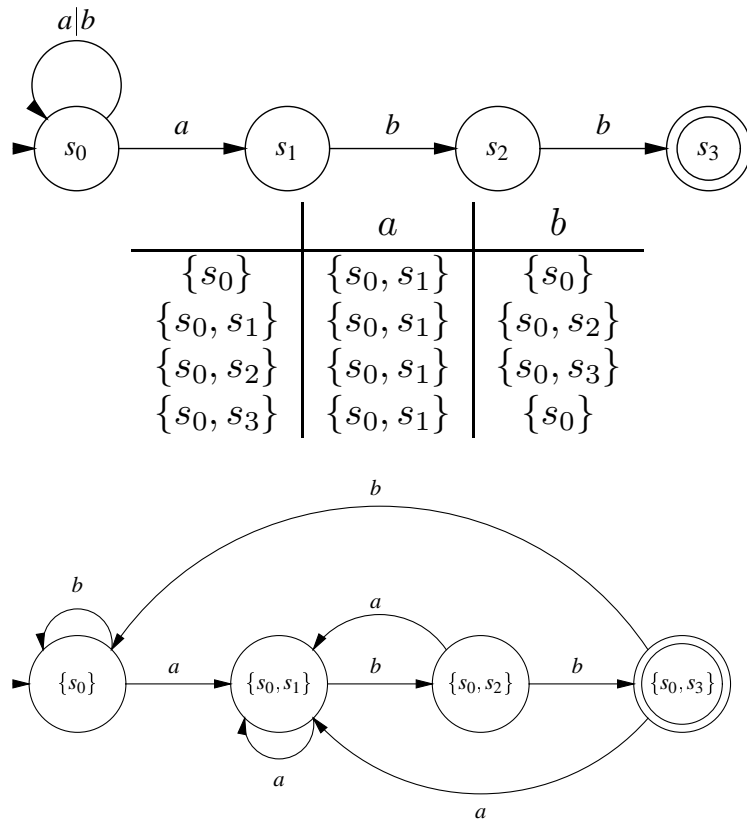
1. no state has a ϵ -transition, and
2. for each state s and input symbol a , there is at most one edge labelled a leaving s .

A DFA accepts x iff. there exists a *unique* path through the transition graph from the s_0 to an accepting state such that the labels along the edges spell x .

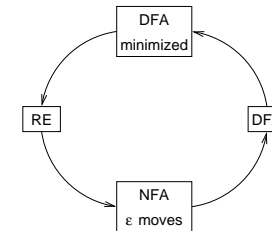
DFAs and NFAs are equivalent

1. DFAs are clearly a subset of NFAs
2. Any NFA can be converted into a DFA, by simulating sets of simultaneous states:
 - each DFA state corresponds to a set of NFA states
 - possible exponential blowup

NFA to DFA using the subset construction: example 1



Constructing a DFA from a regular expression



RE \rightarrow NFA with ϵ moves

build NFA for each term
connect them with ϵ moves

NFA with ϵ moves to DFA

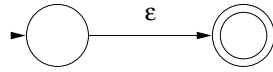
construct the simulation
the “subset” construction

DFA \rightarrow minimized DFA

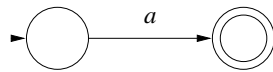
merge compatible states

RE to NFA

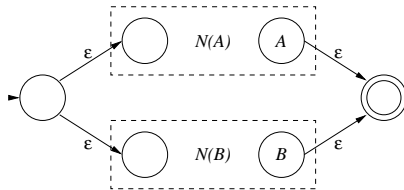
$N(\epsilon)$



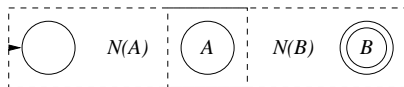
$N(a)$



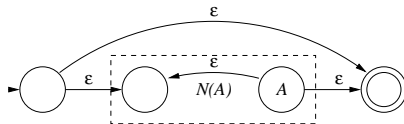
$N(A|B)$



$N(AB)$



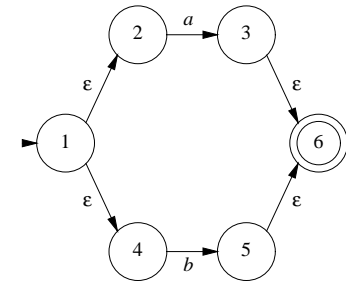
$N(A^*)$



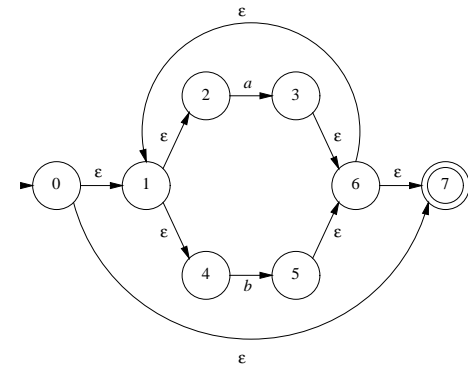
RE to NFA: example

$(a|b)^*abb$

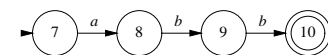
$a|b$



$(a|b)^*$



abb



NFA to DFA: the subset construction

Input: NFA N
 Output: A DFA D with states $Dstates$ and transitions $Dtrans$ such that $L(D) = L(N)$
 Method: Let s be a state in N and T be a set of states, and using the following operations:

Operation	Definition
$\epsilon\text{-closure}(s)$	set of NFA states reachable from NFA state s on ϵ -transitions alone
$\epsilon\text{-closure}(T)$	set of NFA states reachable from some NFA state s in T on ϵ -transitions alone
$\text{move}(T, a)$	set of NFA states to which there is a transition on input symbol a from some NFA state s in T

NFA to DFA: the subset construction

```

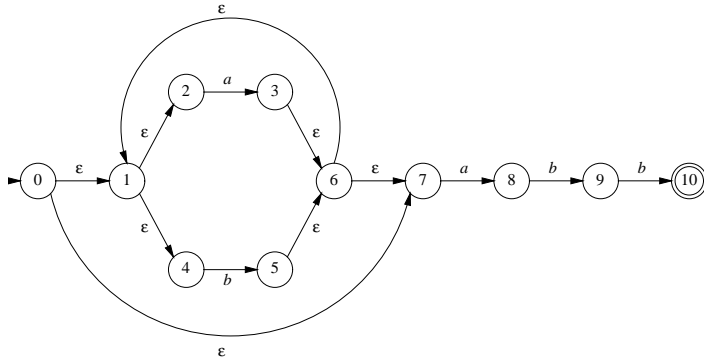
add state  $T = \epsilon\text{-closure}(s_0)$  unmarked to  $Dstates$ 
while  $\exists$  unmarked state  $T$  in  $Dstates$ 
    mark  $T$ 
    for each input symbol  $a$ 
         $U = \epsilon\text{-closure}(\text{move}(T, a))$ 
        if  $U \notin Dstates$  then add  $U$  to  $Dstates$  unmarked
         $Dtrans[T, a] = U$ 
    endfor
endwhile

```

$\epsilon\text{-closure}(s_0)$ is the start state of D

A state of D is accepting if it contains at least one accepting state in N

NFA to DFA using subset construction: example 2



$$A = \{0, 1, 2, 4, 7\}$$

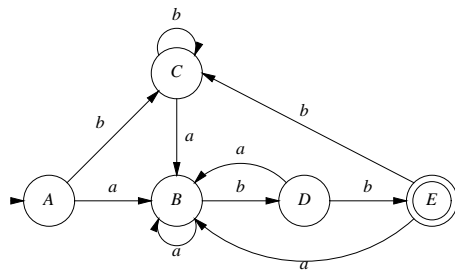
$$B = \{1, 2, 3, 4, 6, 7, 8\}$$

$$C = \{1, 2, 4, 5, 6, 7\}$$

$$D = \{1, 2, 4, 5, 6, 7, 9\}$$

$$E = \{1, 2, 4, 5, 6, 7, 10\}$$

	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C



Limits of regular languages

Not all languages are regular

One cannot construct DFAs to recognize these languages:

- $L = \{p^k q^k\}$
- $L = \{w c w^r \mid w \in \Sigma^*\}$

Note: neither of these is a regular expression!
(DFAs cannot count!)

But, this is a little subtle. One can construct DFAs for:

- alternating 0's and 1's
 $(\epsilon|1)(01)^*(\epsilon|0)$
- sets of pairs of 0's and 1's
 $(01|10)^+$

So what is hard?

Language features that can cause problems:

reserved words

PL/I had no reserved words

```
if then then then = else; else else = then;
```

significant blanks

FORTRAN and Algol68 ignore blanks

```
do 10 i = 1,25
```

```
do 10 i = 1.25
```

string constants

special characters in strings

```
newline, tab, quote, comment delimiter
```

finite closures

some languages limit identifier lengths

adds states to count length

FORTRAN 66 → 6 characters

These can be swept under the rug in the language design