

# Lexical Analysis

## COMP3610 – Principles of Programming Languages

Ben Lippmeier

Australian National University  
Semester 2, 2009

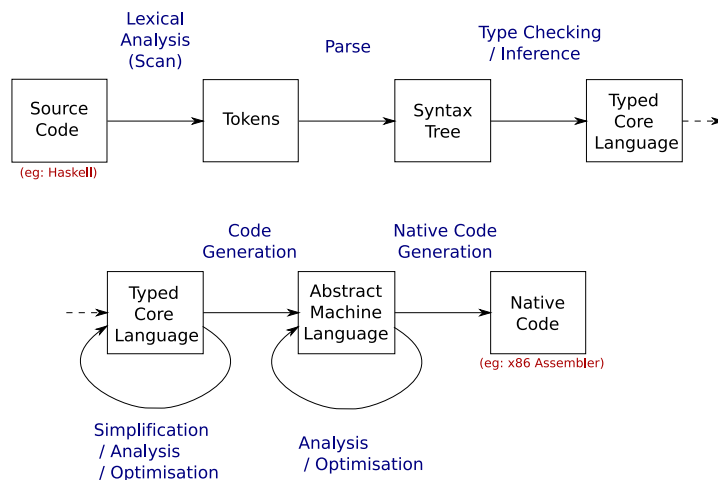
# Scanner Generators

*Flex, Lex, Alex* and other such scanner generators take a specification consisting of a sequence of regular expressions and produce a program that recognises strings that match those regular expressions.

- Flex builds a **finite state automaton (FSA)** corresponding to the regular expressions;
- A specification of the FSA, e.g. a table, is passed to a standard routine (supplied by Flex) that imitates the behaviour of the FSA on the input data.

The following slides look at this process in more detail. . .

# Compiler Structure



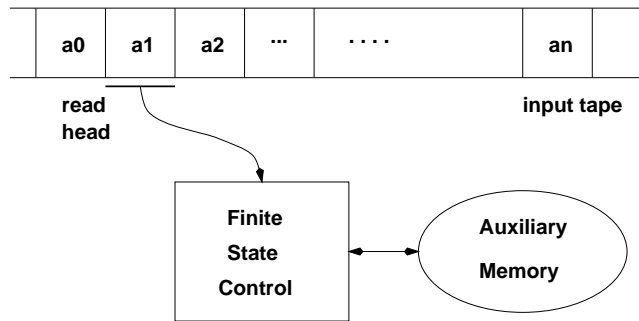
# Languages and Automata

To define a language we can either:

1. Give a set of rules (i.e. a grammar) to produce all the legal strings (sentences) of the language.
2. Provide a machine (i.e. an algorithm) to recognise all the legal strings of the language.

There is a close relationship between the two approaches. Commonly we **define** a language by giving a grammar and then base **recognisers** on the corresponding machine.

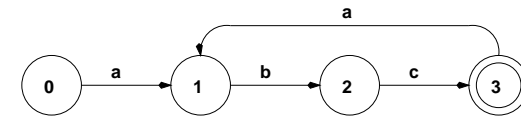
## Basic structure of formal machines



- For the machine to *accept* an input string as a sentence of the language, it must reach a specified goal state, with the input exhausted.

## Finite State Automata recognise Regular Languages

*Example:* A machine to recognise the language  $\{(abc)^n \mid n \geq 1\}$ .  
(i.e. abc, abcabc, abcabcabc, ...)



Note how the states indicate progress in string recognition:

$q_1$  —  $(abc)^i a$  has been recognised ( $i \geq 0$ )

$q_2$  —  $(abc)^i ab$  has been recognised ( $i \geq 0$ )

$q_3$  —  $(abc)^i$  has been recognised ( $i \geq 1$ )

This is the basis for understanding the correspondence between finite automata and regular languages.

- The kind of auxiliary memory in a machine determines the class of languages that the machine can recognise:

Language Class	Memory
regular	none
context-free	stack
context-sensitive	tape (bounded by input length)
unrestricted	unbounded tape

## Theorem

The class of languages recognised by finite automata is *exactly* the class of regular languages.

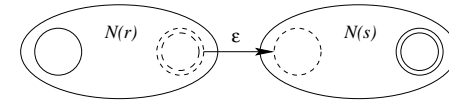
- Regular languages  $\equiv$  those that can be defined using regular expressions.
- Right now, we're only interested in half this result: how to build a FSA from any given regular expression.

## The Language of Regular Expressions

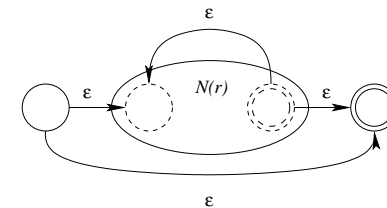
Here is a simple but general language of regular expressions:

- $x$  matches the character 'x'.
- $r^*$  matches zero or more  $r$ 's where  $r$  is a regular expression.
- $r^+$  matches one or more  $r$ 's.
- $r?$  matches zero or one  $r$ .
- $rs$  matches an  $r$  followed by an  $s$  where  $r$  and  $s$  are regular expressions.
- $r|s$  matches either an  $r$  or an  $s$ .
- $(r)$  matches an  $r$ . Parentheses are use to override precedence.

For  $rs$ , construct:



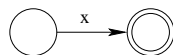
For  $r^*$ , construct:



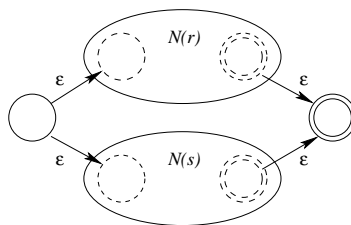
## Regular Expressions to NFAs

NFA = Non-deterministic Finite State Automata. For each regular expression construct, there is a corresponding NFA construct.

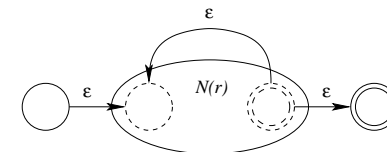
For  $x$ , construct:



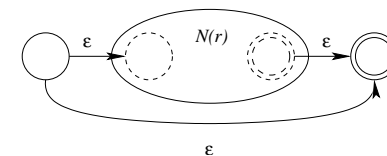
For  $r|s$ , construct:



For  $r^+$ , construct:



For  $r?$ , construct:



## Making it Deterministic

- Unfortunately, the finite state automata thus generated from regular expressions may be *non-deterministic*.
- This is obvious from the common use of *empty transitions*, but in general it is possible to have two different transitions from a state for the same input symbol.
- We can write a language recogniser based on NFA's, but using FSA's is more efficient as we only need to keep track of a single *current state*.
- A lexer generator such as Flex will typically build a NFA from a given regular expression, then convert it to an FSA for efficiency.

The machine on the previous page *can* accept a string if it is legal (that is, some transition sequence exists, ending in the goal state).

For example, there is a transition sequence from

$$(q_0, 011) \vdash (q_S, \_)$$

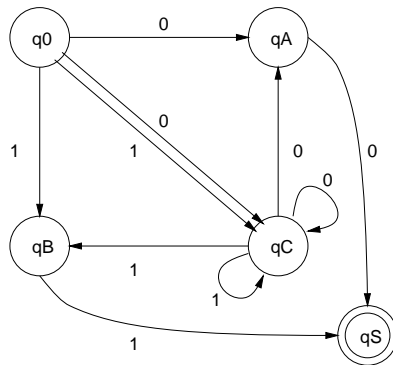
but there are also dead-ends too, for example:

$$(q_0, 011) \vdash (q_A, 11)$$

If we could look at input symbols further ahead we could decide whether the current symbol (0 or 1) was part of the end sequence (00 or 11), but this is not a general solution.

Fortunately there is a fairly simple solution . . .

Here is an example of a *NFA*. It was *not* generated using the algorithm on the previous slides. Note the arcs leaving nodes with the same labels:

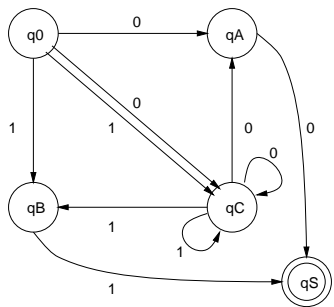


## Theorem

Every non-deterministic finite state automaton (NFA) can be mapped to an equivalent deterministic finite state automaton (DFA).

(“Equivalence” is meant in the sense of the language recognised.)

The basic idea is to replace each set of states *reachable* in a non-deterministic machine by one state.



New states.

$$q_1 = \{q_A, q_C\}$$

$$q_2 = \{q_B, q_C\}$$

$$q_3 = \{q_A, q_C, q_S\}$$

$$q_4 = \{q_B, q_C, q_S\}$$

Note the following correspondences.

$q_1$  — one 0 recognised

$q_2$  — one 1 recognised

$q_3$  — several consecutive 0's recognised

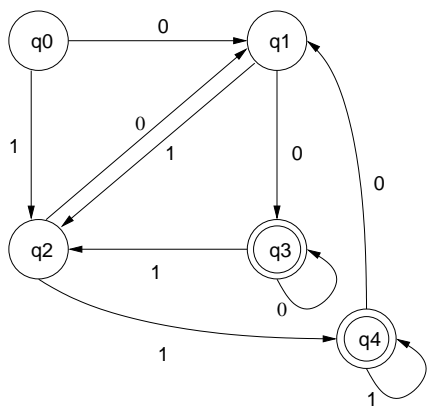
$q_4$  — several consecutive 1's recognised

## From NFA to DFA

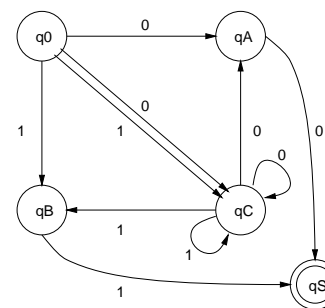
**Basic idea:** the DFA has states corresponding to *sets* of the NFA states that can be reached by scanning the same string, beginning in the start state  $q_0$ .

- What is the *set* of states we can reach from  $q_0$  on input 0?
- What is the *set* of states we can reach from  $q_0$  on input 1?
- What is the set of states we can reach from each state in those sets on input 0?
- etc.

This leads to the following DFA:



Note that there are now 2 goal states, since both  $q_3$  and  $q_4$  include  $q_S$ .



$move(\{q_0\}, 0)$	$= \{q_A, q_C\}$
$move(\{q_0\}, 1)$	$= \{q_B, q_C\}$
$move(\{q_A, q_C\}, 0)$	$= \{q_A, q_C, q_S\}$
$move(\{q_A, q_C\}, 1)$	$= \{q_B, q_C\}$
$move(\{q_B, q_C\}, 0)$	$= \{q_B, q_C, q_S\}$
$move(\{q_B, q_C\}, 1)$	$= \{q_A, q_C\}$
$move(\{q_A, q_C, q_S\}, 0)$	$= \{q_A, q_C, q_S\}$
$move(\{q_A, q_C, q_S\}, 1)$	$= \{q_B, q_C\}$
$move(\{q_B, q_C, q_S\}, 0)$	$= \{q_A, q_C\}$
$move(\{q_B, q_C, q_S\}, 1)$	$= \{q_B, q_C, q_S\}$

## Algorithm to construct a DFA from a NFA

The algorithm consists of

- a process to find sets of states that can be reached by  $\epsilon$ -moves from some given state or states;
- a process to construct the desired subsets of the NFA states, and the corresponding DFA transitions.

The NFA in the worked example above has no  $\epsilon$ -moves but they are important.

In particular, algorithms to construct a finite state automaton from a regular expression generally produce a NFA with many  $\epsilon$ -moves.

## Subset Construction Algorithm

Let  $move(T, a)$  be the set of states to which there is a transition on input  $a$  from one of the states in the set  $T$ .

An NFA  $N$  with start state  $q_0$  can be in any of the states in  $T = \epsilon\text{-closure}(q_0)$  before scanning any input. If the input is  $a$  then  $N$  can move to any of  $\epsilon\text{-closure}(move(T, a))$ . And so on ...

The subset construction algorithm starts with  $\epsilon\text{-closure}(q_0)$  as a  $D$ -state and generates other  $D$ -states by the process above. Once the  $D$ -states have been considered, they are *marked*.

### $\epsilon$ -closure algorithm

For some subset  $T$  of the states of an NFA, the  $\epsilon$ -closure of  $T$  is the set of states of the NFA that can be reached from one of the states in  $T$  by  $\epsilon$ -moves alone.

**Input:** a set  $T$  of states of an NFA.

**Output:** the  $\epsilon$ -closure of  $T$  (a set).

```
push all states in  $T$  onto  $Stack$ ;
```

```
 $\epsilon\text{-closure}(T) = T$ ;
```

```
while  $Stack$  not empty do
```

```
     $t = \text{pop}(Stack)$ ;
```

```
    for each  $u$  with an empty edge from  $t$  do
```

```
        add  $u$  to  $\epsilon\text{-closure}(T)$ ;
```

```
        push  $u$  onto  $Stack$ ;
```

**Input:** NFA with an unmarked  $D$ -state =  $\epsilon\text{-closure}(q_0)$ .

**Output:** DFA that recognises the same language as the NFA.

```
while there is an unmarked state  $T$  in  $Dstates$  do
    mark  $T$ ;
    for each alphabet symbol  $a$  do
         $U = \epsilon\text{-closure}(move(T, a))$ ;
        if  $U$  not in  $Dstates$  then
            add  $U$  to  $Dstates$  as unmarked;
         $Dtran[T, a] = U$ ;
```

Now  $Dtran$  is the transition table for  $D$ .

The start state for  $D$  is  $\epsilon\text{-closure}(q_0)$ .

The accept states for  $D$  are all those that contain an accept state of  $N$ .

If the NFA has  $n$  states then the constructed DFA has at most  $2^n - 1$  states, but usually far fewer.

### Algorithm trace:

$e\text{-closure}(0) = \{ 0, 1, 2, 4, 7 \} = A$

$T_A^a = \{ 3, 8 \}$   
 $e\text{-closure}(T_A^a) = \{ 1, 2, 3, 4, 6, 7, 8 \} = B$   
 $Dtran[A, a] = B$

$T_A^b = \{ 5 \}$   
 $e\text{-closure}(T_A^b) = \{ 1, 2, 4, 5, 6, 7 \} = C$   
 $Dtran[A, b] = C$

$T_B^a = \{ 3, 8 \}$   
 $e\text{-closure}(T_B^a) = B$   
 $Dtran[B, a] = B$

$T_B^b = \{ 5, 9 \}$   
 $e\text{-closure}(T_B^b) = \{ 1, 2, 4, 5, 6, 7, 9 \} = D$   
 $Dtran[B, b] = D$

$T_C^a = \{ 3, 8 \}$   
 $e\text{-closure}(T_C^a) = B$   
 $Dtran[C, a] = B$

$T_C^b = \{ 5 \}$   
 $e\text{-closure}(T_C^b) = C$   
 $Dtran[C, b] = C$

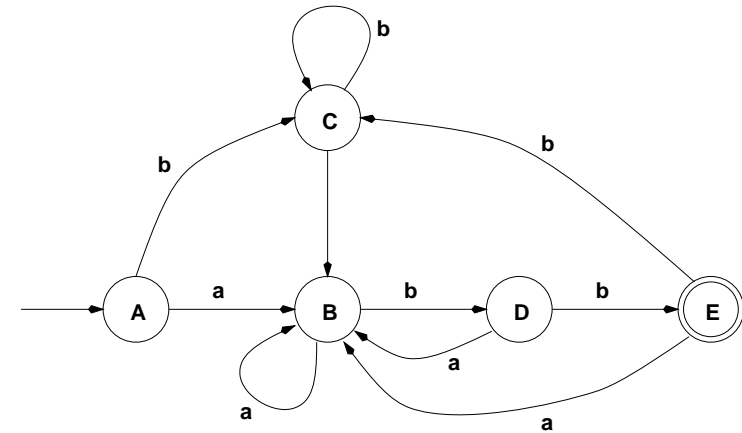
$T_D^a = \{ 3, 8 \}$   
 $e\text{-closure}(T_D^a) = B$   
 $Dtran[D, a] = B$

$T_D^b = \{ 5, 10 \}$   
 $e\text{-closure}(T_D^b) = \{ 1, 2, 4, 5, 6, 7, 10 \} = E$   
 $Dtran[D, b] = E$

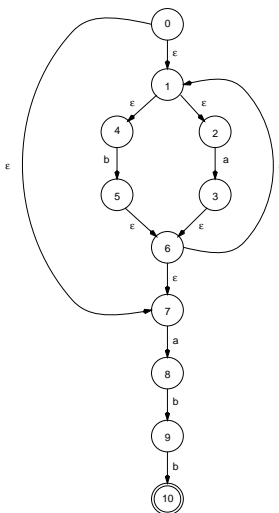
$T_E^a = \{ 3, 8 \}$   
 $e\text{-closure}(T_E^a) = B$   
 $Dtran[E, a] = B$

$T_E^b = \{ 5 \}$   
 $\text{Modula2-style } e\text{-closure}(T_E^b) = C$   
 $Dtran[E, b] = C$

### Derived DFA for $(a|b)^*abb$



### Example: NFA for $(a|b)^*abb$



### Lexical Analysis using Finite Automata

Since the lexical analysis phase of a compiler deals with a regular language fragment, we should expect that DFAs may provide a “table-driven” approach for a lexical analyser generator such as Flex).

Lexer generators such as Flex take as input a specification of the tokens, using regular expressions.

This can be automatically converted into a transition table for a DFA and the lexer proceeds by simulating a DFA for that table.

The following slide is a lexer for the tokens +, \*, :=, numerals, identifiers, (, ). White space and comments (\* ... \*) are skipped.

(It’s only an example — I’m not suggesting that it would be generated by Flex.)

	cr	+	*	:	=	0	...	9	A	...	Z	(	)	
q <sub>0</sub>	0	0	1	2	3	6	7	...	7	9	...	9	11	15
q <sub>1</sub>	return plussymbol													
q <sub>2</sub>	return multsymbol													
q <sub>3</sub>	4	4	4	4	4	5	4	...	4	4	...	4	4	4
q <sub>4</sub>	return colonsymbol													
q <sub>5</sub>	return assignsymbol													
q <sub>6</sub>	return equalsymbol													
q <sub>7</sub>	8	8	8	8	8	8	7	...	7	8	...	8	8	8
q <sub>8</sub>	return numeral													
q <sub>9</sub>	10	10	10	10	10	10	9	...	9	9	...	9	10	10
q <sub>10</sub>	return identifier													
q <sub>11</sub>	12	12	12	13	12	12	12	...	12	12	...	12	12	12
q <sub>12</sub>	return lparen													
q <sub>13</sub>	13	13	13	14	13	13	13	...	13	13	...	13	13	13
q <sub>14</sub>	13	13	13	14	13	13	13	...	13	13	...	13	13	0
q <sub>15</sub>	return rparen													

## Exercises

- Apply the NFA  $\rightarrow$  DFA algorithm to the regular expression given back on slide 14, that is,  $(0|1)^*(00|11)$
- Express the following regular expression as a NFA:  
 $(0|1)^* 0 (0|1)^n$ , for some small value of n.
- Try and write down a DFA for the above expression, why is this difficult?