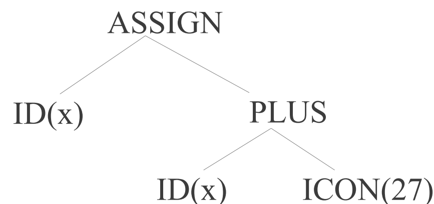
Compilers Course
Lecture 3: Lexical Analysis

Lexer (Scanner):
- Input is a stream (sequence, list) of characters .
- Output is a stream of tokens ("words") with attributes .

Parser:
- Input is the stream of tokens .
- Output is the program's syntax tree .

An example:
... x = x + 27; ...

... ID(x) ASSIGN ID(x) PLUS ICON(27) SEMI ...



What Does a Lexer Do:
- Partition input into substrings ("lexemes") .
- Each substring corresponds to a "word", a blank, or a comment .
- For non-blank substrings the corresponding token is generated:
  a symbolic constant and possibly an attribute ("27" → ICON(27)) .
- Attach input positions to tokens for error messages .

How Does the Lexer work:
- Apply a finite automaton (DFA) to start of the input .
- At accept: a word or blank/comment has been recognized: remove it and generate the token .
- Repeat until end of input is reached .

Why Lexer?
- Separate scanner simplifies both parser and scanner .
- Can use regular expressions to specify the "word" classes of the language:
  - Can automatically generate the DFA, good for the implementor .
  - Declarative and simple specification .

Regular Languages: Repetition :

Alphabet: finite set of symbols. Often denoted by Σ.
Example: Σ = {0, 1} for binary numbers.

String: finite sequence of symbols from a given alphabet. Often denoted by lower-case s, t, or w.
Example: "101".
Operations on strings:

ε	the empty string
$s_1.s_2$	concatenation: "0"."1" = "01" (the dot is often omitted)

$s^i = s^{(i-1)}.s$ if i>0 (repetition)
$s^0 = \varepsilon$

Language: a set (often infinite) of strings from some alphabet. Often denoted by upper-case L or M.
Example: {}, {0}, {10}, {110}
Operations on languages:

$L \cup M = \{s \mid s \in L \vee s \in M\}$	or
$L.M = \{s.t \mid s \in L \wedge t \in M\}$	sequence (dot often omitted)
$L^i = L^{i-1}.L \; if \; i > 0$	bounded repetition
$L^0 = \varepsilon$	
$L^* = \cup L^i$	unbounded repetition
(i>=0)	(Kleene closure)
L+ = L.L*	positive closure

Other set operations are often not interesting when describing programming languages.

Example:
D = {0, 1, ..., 9}
L = {a, b, ..., z, A, B, ..., Z}
M = L.(L ∪ D)* = all identifiers in a typical programming language

Regular expressions: finite specifications of regular languages.

Primitive notation for regular expressions:

r	L(r)
ε	{ε}
a	{a} if a ϵ Σ
r.s	L(r).L(s) (dot often omitted)
r|s	L(r) ∪ L(s)
r*	(L(r))*
(r)	L(r)

Laws:
ε.r = r.ε = r
r|s = s|r
r.(s|t) = (r.s)|(r.t)

Shorthand notation:

r+	r.r*	non-empty sequences
[a-z]	a|b|c|...|z	character sets/classes
[^0-9]	Σ \[0-9] negated character classes	
r?	r|ε optional expressions	

We may name regular expressions and refer to them by name. This can be used to simplify complex expressions. The name may *not* occur in its regular expression, either directly or indirectly!

<u>Examples of token classes in programming languages</u>

Identifiers: i, foo, x27, ab; but not 9x

letter = [a-zA-Z_]
digit = [0-9]
identifier (ID) = letter (letter | digit)*

Token ID with string attribute: "foo" becomes ID("foo")

Integers
decimal = [0-9]+
octal = 0 [0-7]*
hexadecimal = 0x [0-9a-fA-F]+

In some languages any +/- sign is part of the constant, in other languages the sign is a separate unary operator.

Token ICON with integer attribute: "27" becomes ICON(27) , ICON stands for (Integer Constant).

Keywords (Reserved Words) :

- List them, one regexp per word: "if", "while", "return", etc .
- Each keyword has its own unique token, without attribute (the parser *must* be able to identify them directly without also having to check an attribute) .

Symbols: +, -, *, /, , , >=, etc .

- Handled just like keywords .
- Symbols that behave identically in the syntax may be combined, for example: <, <=, >=, and >

may be combined into a RELOP token with an attribute identifying the actual operator .

Floating-point numbers :
Languages vary in whether the decimal part is mandatory or not when an exponent is present .

sign = (+|-)
integer = [0-9]+
exponent = "E" sign? integer
float = sign? integer "." integer exponent?

Fairly strict specification. A more liberal specification could be:
float = sign? (integer ("." integer)? exponent | integer? "." integer exponent?)

which would also accept 1E2 and .27

String literals :
- Start, sequence of elements, end.
- The problem is to express what the elements look like.

string = """ element* """
element = [^"]

Will do for simple strings. In C and ML special characters can be inserted if they are quoted with a "\"-character:
element = [^\\"] | "\"[\\"]

This allows "foobar", "foo\"bar" and "foo\\bar"

Whitespace/blanks:
- List the characters: white = [\t\n\r] .
- When accepting a whitespace regexp, the scanner is told to resume scanning without generating any token .

Comments:
- Specified like strings, but are thrown away like blanks .

C++:
cpluspluscomment = "//" [^\n] \n

C:
ccomment = "/*" stuff* "*/"

The problem is that any "*" within the comment must not be followed by "/", since that would terminate the comment.
stuff = [^*/] | [^*]"/" | "*"[^/]

This works except for "*" just before the terminating "*/", and "/" just after the initial "/*". ("/***/" would go wrong) . We must handle trailing "*"s and leading "/"s specially:

stuff = [^*/] | [^*]"/" | "*"[^/]
ccomment = "/*" "/"* stuff* "*"* "*/" [1]

<u>Ambiguities (BAD)</u> :

- Is "+=" PLUSASSIGN (one token) or PLUS, ASSIGN (two tokens)?
- Is "ifx" an identifier ID("ifx") or the keyword IF followed by the identifier ID("x")?
- Is "if" a keyword or an identifier (IF or ID("if"))?

Solved by meta-rules:

- The character stream is scanned from left to right .
- Longest match: the longest substring that matches any regexp is the one to be accepted; this solves cases 1 and 2 above .
- Priority: the regexps are listed in priority order, and those listed early take priority over those listed later; listing the keywords before the identifier regexp solves case 3 above

<u>Context-sensitive constructs</u> :

Nested comments in SML:

<a piece of code> (* with a comment *)

Now let's comment it out:

(* this code doesn't work
<a piece of code> (* with a comment *)
*)

Problem: $\cup\{x^i.y^i\}$ for i>=0 is not a regular language, regular languages "cannot count".

Solution: Automaton with multiple start states, code to count nesting level, and checks at "*)" to choose next state based on nesting level.
In Fortran whitespace is insignificant:

DO 5 I = 1,25
vs .
DO 5 I = 1.25

The first line is the start of a DO loop statement (due to ","). The second line assigns 1.25 to the variable DO5I.
Problem: the context *after* the string determines how it is to be interpreted.

1Solution from Schreiner & Friedman, "Introduction to Compiler Construction with UNIX", 1985, page 25.

Lexical Analysis: Implementation :

A regular expression r denotes a set of strings: L(r). So s ϵ L(r) determines if a string is a token or not.
Problem: L(r) can be, and often is, infinitely large.

A finite automaton :

- Is of finite size .
- Can answer s ϵ L(r) in |s| time steps (where |s| is the length of s) .

Automata Basics (skip it if they are familiar with it) :

Definition:
A finite automaton (FA) is a 5-tuple M=(Σ, Q, q_0, F, δ) where:
- Σ is a finite set of symbols (the alphabet).
- Q is a finite set of states.
- q_0 ϵ Q is a unique start state.
- F is a subset of Q, the final states.
- δ is a transition function from states and symbols to states.

NFA_ε:
- δ: state x (symbol or ε) → set of states.
- Non-deterministic

NFA:
- δ: state x symbol → set of states.
- Non-deterministic

DFA:
δ: state x symbol → set of a single state:
- Deterministic.
- In theory a DFA should be total: delta should be defined for all state/symbol pairs and return a single state for each such pair.
- In practice we allow DFAs to be partial: if delta is defined for a state/symbol pair then it returns a single state, otherwise it returns an error marker.
- A partial DFA is equivalent to a total DFA where the error marker is an additional non-final error state, and transitions on the error state and any symbol return to the error state.

A configuration is a 2-tuple <state, string>.
The delta function defines a relation on configurations, ⇒, as follows:
<q, aw> ⇒ <q', w>
    if q' ϵ δ(q, a)
    one transition step, consuming one input symbol
<q, w> ⇒ <q', w>

if $<q, \varepsilon> \in Dom(\delta)$ and $q' \in delta(q, \varepsilon)$
one transition step, no input consumed

The $\Rightarrow$ relation is then extended to $\Rightarrow^*$:

$<q, w> \Rightarrow^* <q, w>$

$<q, w> \Rightarrow^* <q'', w''>$ if $<q, w> \Rightarrow^* <q', w'>$ and $<q', w'> \Rightarrow <q'', w''>$

That is, we can apply $\Rightarrow$ zero or more times.

Finally, the language of the automaton, L(M) is defined as:

$L(M) = \{ w \mid <q_0,w> \Rightarrow^* <q', \varepsilon> \text{ and } q' \in F \}$

That is, starting from q0 there exists a sequence of transition steps that consume the entire string and end in a final state.

Example:

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

$Q = \{ q_0, q_1 \}$

$F = \{ q_1 \}$

$\delta[q_0, a] = q_1$

$\delta[q_0, b] = q_0$

$\delta[q_1, a] = q_1$

$\delta[q_1, b] = q_0$

is a DFA accepting strings of a:s and b:s that end with an a.

$<q_0, aba> \Rightarrow <q_1, ba> \Rightarrow <q_0, a> \Rightarrow <q_1, \varepsilon>$

Implementation of a Lexer

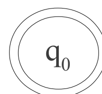
1. Define regular expressions for tokens and blanks: r = r1 | r2 | ... | rn
2. Convert r to a DFA:
   1. Convert r to an NFA_ε.
   2. Convert the NFA_ε to a DFA.
   3. [Optional] Minimize the DFA.
3. Implement the DFA in program code.
4. Add code for input management, actions to run when tokens are recognized, conflict resolution (longest match, priority), and error handling.

Converting a Regular Expression r to an NFA_ε

For complex regular expressions composed of smaller ones, we translate the components first, and then combine their automata to form a larger automaton for the original regular expression.

$r = \varepsilon$

M: state q_0
q_0 is both start and final state



r = "a"

M: states q_0, f_0

q_0 is start state, f_0 final state

connect q_0 and f_0 with a transition on "a":

$\delta(q_0, a) = f_0$

r = $r_1.r_2$

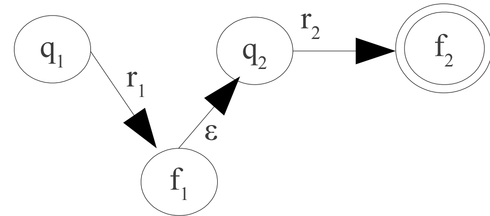
translate r_1 to M_1 with start state q_1 and final state f_1

translate r_2 to M_2 with start state q_2 and final state f_2

connect f_1 to q_2 with an ε transition

the result M has q_1 as start state and f_2 as final state

(f_1 is no longer a final state)



r = r_1 | r_2

translate r_1 to M_1 with start state q_1 and final state f_1

translate r_2 to M_2 with start state q_2 and final state f_2

add new states q_0 and f_0

connect q_0 to q_1 with an ε transition

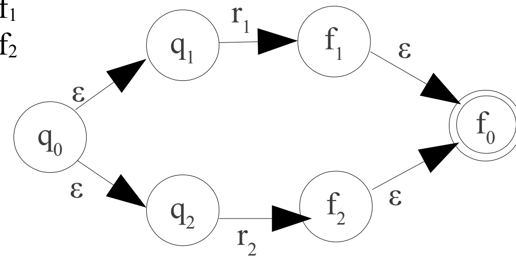
connect q_0 to q_2 with an ε transition

connect f_1 to f_0 with an ε transition

connect f_2 to f_0 with an ε transition

the result M has q_0 as start state and f_0 as final state

(f_1 and f_2 are no longer final states)



r = r_1*

translate r_1 to M_1 with start state q_1 and final state f_1

add new states q_0 and f_0

connect q_0 to f_0 with an epsilon transition

(the case when the empty string is accepted)

connect q_0 to q_1 and f_1 to f_0 with epsilon transitions

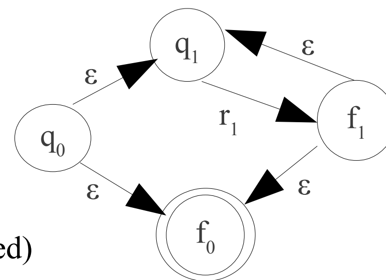
(the case when a single instance of r_1 is accepted)

connect f_1 to q_1 with an epsilon transition

(the case when more than one instance of r_1 is accepted)

the result M has q_0 as start state and f_0 as final state

(f_1 is longer a final state)

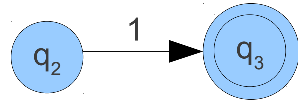


NOTE: Appel's Tiger Book uses a slightly different presentation where each intermediate "automaton" has a start *edge* but no start *state*. This results in fewer states and ε transitions in the NFA_ε, but is more problematic from a presentation point of view.

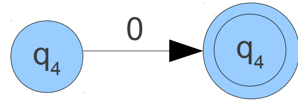
Example: r = (1|0)*1

Translate inner-most components first, then move up and connect things.

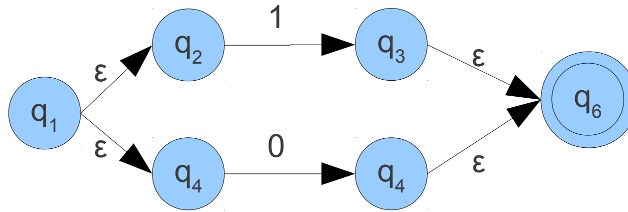
The first 1:



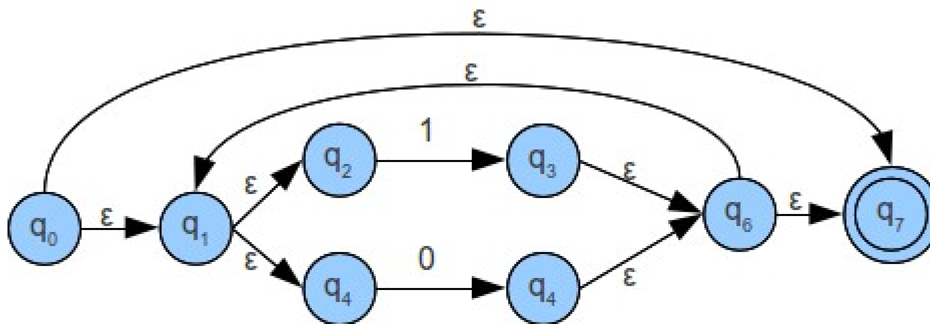
The first 0:



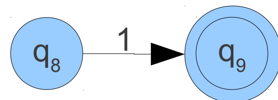
(1|0):



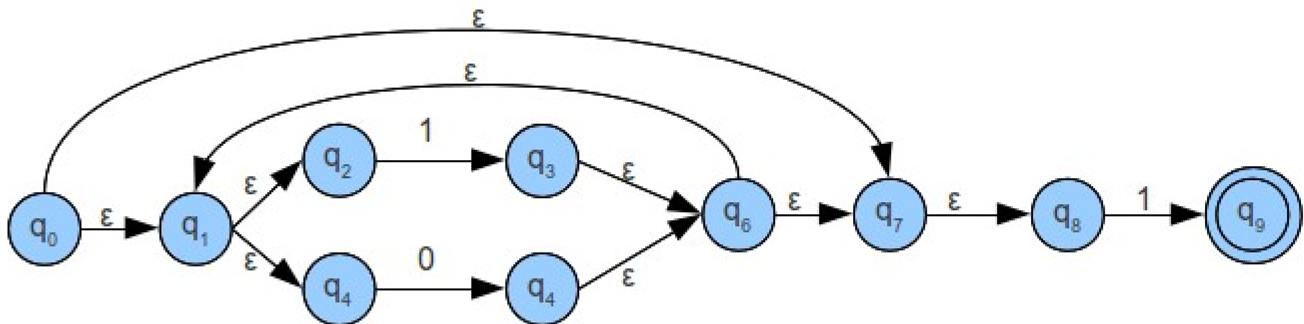
(1|0)*:



The last 1:



Connect (1|0)* with 1:



q_0 is the start state, q_9 is the final state

Converting an $NFA_ε$ to a DFA

The idea is to create a DFA that simulates the $NFA_ε$. A state in the DFA will be a *set* of states in the $NFA_ε$. The transition function in the DFA will follow all possible transitions in the $NFA_ε$, *and* all possible ε-transitions.

ε-closure(S)

Given an initial set of states S the automaton can be in, ε-closure(S) gives the larger set of states T the automaton also can be following only ε-transitions.

Definition:
ε-closure(S) = T, where
1. S and T are sets of states.
2. T is a superset of S.
3. If q ϵ T and <q, ε> → q', then q' ϵ T
4. T is the smallest set that satisfies 2) and 3)

Constructing the DFA

Let q_i and δ belong to the given $NFA_ε$, and q'_i and δ' belong to the DFA being constructed.

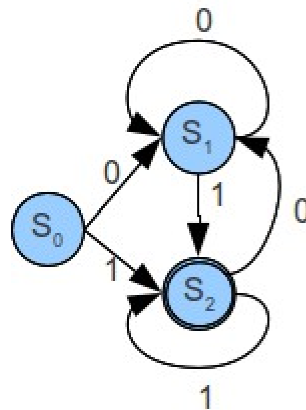
1. The start state, q'_0 is given by: q'_0 = ε-closure($\{q_0\}$).
2. If $\{q_1, ..., q_n\}$ is a state in the DFA and a is a symbol, then $δ'(\{q_1,...,q_n\}, a)$ = ε-closure(∪(1<=i<=n)$δ(q_i, a)$) Given a set of possible states, we first follow all possible transitions on a, and then follow all possible ε-transitions. **Step 2 is repeated until no new states in the DFA are created.**
3. A state $q' = \{q_1,...,q_n\}$ in the DFA is a final state if any qi is a final state of the NFAε.

Example: convert the (1|0)*1 automaton to a DFA.

S_0 = ε-closure($\{q_0\}$) = $\{q_0,q_1,q_2,q_4,q_7,q_8\}$

$\delta'(S_0,0)$ = ε-closure(delta($\{q_0,q_1,q_2,q_4,q_7,q_8\}$,0))
= ε-closure($\{q_5\}$)
= $\{q_5,q_6,q_1,q_2,q_4,q_7,q_8\}$
= $\{q_1,q_2,q_4,q_5,q_6,q_7,q_8\} = S_1$
$\delta'(S_0,1)$ = ε-closure($\delta(\{q_0,q_1,q_2,q_4,q_7,q_8\},1)$)
= ε-closure($\{q_3,q_9\}$)
= $\{q_3,q_9,q_6,q_1,q_2,q_4,q_7,q_8\}$
= $\{q_1,q_2,q_3,q_4,q_6,q_7,q_8,q_9\} = S_2$
$\delta'(S_1,0)$ = ε-closure($\delta(\{q_1,q_2,q_4,q_5,q_6,q_7,q_8\},0)$)
= ε-closure($\{q_5\}$) = S_1
$\delta'(S_1,1)$ = ε-closure($\delta(\{q_1,q_2,q_4,q_5,q_6,q_7,q_8\},1)$)
= ε-closure($\{q_3,q_9\}$) = S_2
$\delta'(S_2,0)$ = ε-closure($\delta(\{q_1,q_2,q_3,q_4,q_6,q_7,q_8,q_9\},0)$)
= ε-closure($\{q_5\}$) = S_1
$\delta'(S_2,1)$ = ε-closure($\delta(\{q_1,q_2,q_3,q_4,q_6,q_7,q_8,q_9\},1)$)
= ε-closure($\{q_3,q_9\}$) = S_2

q_9 is final in the NFA_ε, so S_2 is final in the DFA



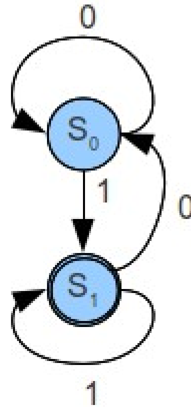
DFA Minimization (sketchy)

Two states q_1 and q_2 are equivalent (==) if:

- Both are final or both are non-final.
- $\delta(q_1,a)$ == delta(q_2,a) for all symbols a.

Equivalent states can be merged without changing the language accepted by the automaton.

Example: S_0 and S_1 are both non-final, and they have identical transitions, hence S_0 == S_1, and the final DFA for (1|0)*1 is:



Recognizing Longest Prefixes:

Using a DFA makes it easy to check if some string s ϵ L(r).

However, in reality we must:

- Partition s into words $s_1.s_2.s_3...s_n$
- At each point recognize the longest prefix s' of s where s' ϵ L(r).
- Generate tokens.
- Restart where previous word ended.
- Handle lookahead:
  r1 = aa
  r2 = aabb
  input = aabc...
  The lexer will scan aab, but the word to recognize is aa.

Solution:

- When reaching a final state, remember the input position and state, but continue to read input and run the automaton.
- At end-of-file or no-transition error, reset input to the remembered position, generate token based on remembered state, and then restart the lexer.

Pseudo-code

States: small integers 0..n
Symbols: small integers 0..m
δ: matrix [0..n][0..m] containing state numbers or "undefined"
final: vector [0..n] containing booleans
We also need to know which regular expression a given final state corresponds to.
input: string of characters, accessed as an array [0..]
pos: index into input, initially 0

1: q_a := undefined; pos_a := undefined; q := q_0; pos_0 := pos
2: while pos < strlen(input):
    q := δ[q][input[pos]]
    if q == undefined then break while loop and go to 3
    pos := pos+1
    if final[q] then q_a := q, pos_a := pos
3: if q_a == undefined then report error
4: generate token based on q_a and input[pos_0,..,pos_{a-1}]
5: pos := pos_a, goto 1

Implementation Tricks/Notes

Keywords look like identifiers but need not be encoded in the automaton. Instead have the code that creates ID tokens from identifier words check if the word actually is a keyword, and if it is, create the corresponding keyword token instead. The check can use a small binary search table or a hash table. This reduces the number of states (rows).

Assume the digits 0..9 have identical transitions in all rows. Then compress those columns to a single one, indexed by character class "digit". Finally add a mapping from characters to character classes that maps 0..9 to "digit". Both digits and letters can often be compressed using this technique, especially if keywords are handled outside of the automaton. This reduces the number of symbols (columns).

Equivalent rows in the transition table can be shared by making it a vector [0..n] of pointers to vectors [0..m] of states.

Rows can be viewed as lists of <symbol,state> pairs: equivalent parts of two or more rows can be pulled out and be represented as a list. Then a row can point to a list starting with the <symbol,state> pairs unique for that row, followed by a tail pointing to the shared list of transitions. This is a common compression technique, but it slows down table inspection.

The lexer is typically implemented as a procedure called from the parser: each time the lexer is called, it restarts from its previous position, applies the automaton, and returns a single token. The lexer thus needs private state to remember its previous position. This can be implemented by global variables, or by passing a reference to a state record as a parameter to the lexer.

Multiple Start States

Some constructs, like nested comments in SML, are difficult or impossible to handle as pure regular expressions.

Many lexer generators allow you to specify multiple start states, and to restrict regular expressions to specific start states.

Finally, most lexer generators allow you to execute arbitrary code after recognizing specific regular expressions, and to conditionally either resume scanning in a given start state, or to return a token.

Example: handling nested comments in SML

```
(* we have two start states, INITIAL and COMMENT *)
%S COMMENT

<INITIAL>"(*" ⇒ (YYBEGIN COMMENT; level := 1; continue())
...
<COMMENT>"(*" ⇒ (level := level + 1; continue())
<COMMENT>"*)" ⇒ (level := level - 1;
            if level == 0 YYBEGIN INITIAL;
                continue())
<COMMENT>. ⇒ (continue())
```