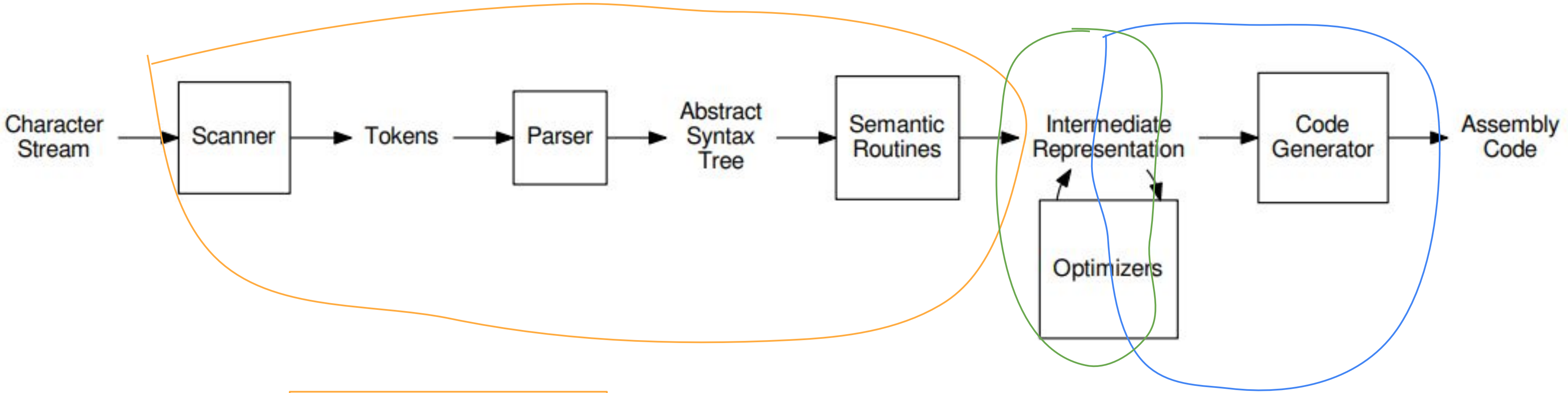
Lec2: scanner (lexical analyzer)

- Recognizing words
- Formal grammars
- Regular expressions
- Finite automata

Content is copied from:

- https://web.stanford.edu/class/cs143/lectures/lecture03.pdf
- https://web.stanford.edu/class/cs143/lectures/lecture04.pdf
- Engineering a Compiler by Cooper and Torczon, 2nd Ed. ch. 1 and sec. 2.1-2.4
- https://www3.nd.edu/~dthain/compilerbook/chapter3.pdf

Review: compiler structure



Front end

Back-end
- ➔ Deals with **target system**

Fig2.2 with some modifications from the book https://www3.nd.edu/~dthain/compilerbook/chapter2.pdf

Scanner (lexical analyzer)

What do we want to do? Example:

```
if (i == j)
   Z = 0;
else
   Z = 1;
```

The input is just a string of characters:

➔ **\tif (i == j)\n\t\tz = 0;\n\telse\n\t\tz = 1;**

➔ **\tif (i == j)\n\t\tz = 0;\n\telse\n\t\tz = 1;**

Goal:

Partition this input string into substrings (tokens)

https://web.stanford.edu/class/cs143/lectures/lecture03.pdf

What is a token?

A syntactic **category**

– In English: noun, verb, adjective, …

Token categories in any Programming Language:

- Keywords
- Identifiers
- Numbers
- Strings
- Comments and whitespace
- …

Scanner identifies **tokens** from the raw text source code of a program.

https://web.stanford.edu/class/cs143/lectures/lecture03.pdf

What are tokens for?

- Classify program substrings according to role
- Lexical analysis produces a stream of tokens
  - … which is input to the parser
- Parser relies on token distinctions
  - An identifier is treated differently than a keyword

https://web.stanford.edu/class/cs143/lectures/lecture03.pdf

Designing a Lexical Analyzer (or designing a new language)

Step-1: Define a finite set of tokens

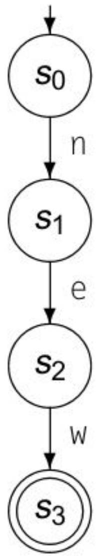
- Tokens describe all items of interest
  - Identifiers,
  - integers,
  - keywords(reserved words)
- Choice of tokens depends on
  - language
  - design of parser

Step-2: Describe which strings belong to each token

https://web.stanford.edu/class/cs143/lectures/lecture03.pdf

Recognizing words: A hand-made scanner

```
c ← NextChar();
if (c = ‘n’)
   then begin;
      c ← NextChar();
      if (c = ‘e’)
         then begin;
            c ← NextChar();
            if (c = ‘w’)
               then report success;
               else try something else;
         end;
         else try something else;
   end;
   else try something else;
```



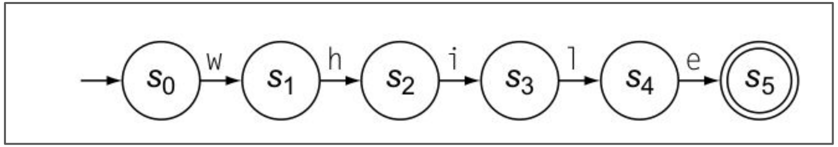
Transition to error states are omitted.

■ **FIGURE 2.1** Code Fragment to Recognize "new".

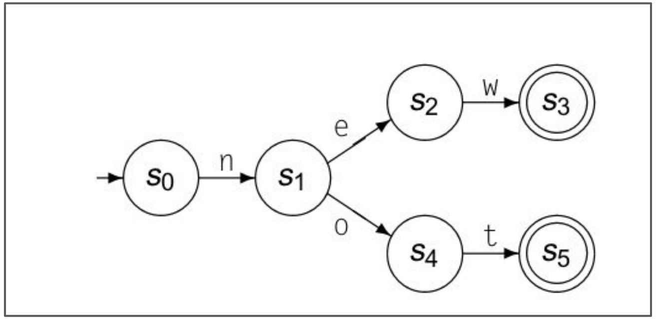
From the book Engineering a Compiler by Cooper and Torczon, 2nd Ed.

Recognizing words: Other examples

while

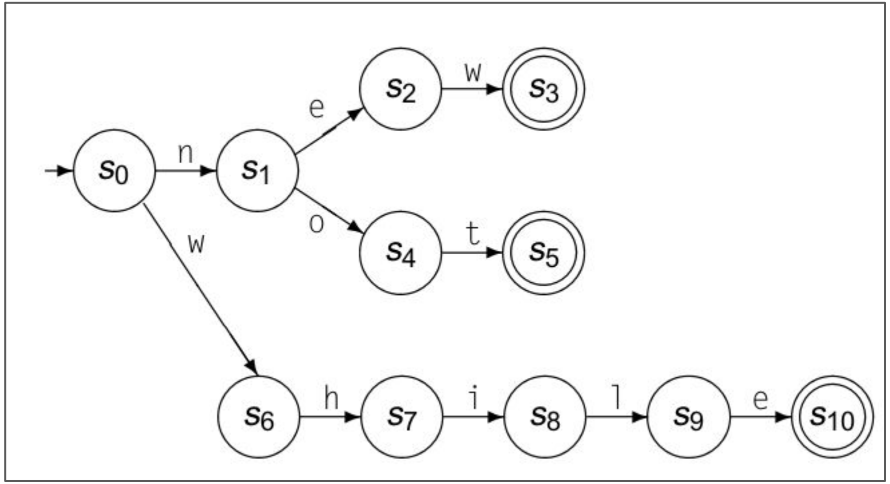


new and **not**



Combined version: **while, new, not**

- **State s0 has transitions for n and w**



From the book Engineering a Compiler by Cooper and Torczon, 2nd Ed.

Handmade scanner in C

```
token_t scan_token( FILE *fp ) {
    int c = fgetc(fp);
    if(c=='*') {
        return TOKEN_MULTIPLY;
    } else if(c=='!') {
        char d = fgetc(fp);
        if(d=='=') {
            return TOKEN_NOT_EQUAL;
        } else {
            ungetc(d,fp);
            return TOKEN_NOT;
        }
    } else if(isalpha(c)) {
        do {
           char d = fgetc(fp);
        } while(isalnum(d));
        ungetc(d,fp);
        return TOKEN_IDENTIFIER;
    } else if ( . . . ) {
        . . .
    }
}
```

Figure 3.1: A Simple Hand Made Scanner

Is it as easy as it sounds?

Sort of… if you do not make it hard!

https://web.stanford.edu/class/cs143/lectures/lecture03.pdf

Some history

Lexical Analysis in FORTRAN

FORTRAN rule: Whitespace is insignificant

E.g., **VAR1** is the same as **VA   R1**

- ★ A terrible design!
- ★ Historical footnote: FORTRAN Whitespace rule motivated by inaccuracy of punch card operators

Fortran do-loops

DO 5 I = 1,25

DO 5 I = 1.25

From left-to-right, it cannot tell if **DO5I** or **DO stmt .**

"**Lookahead**" may be required to decide where one token ends and the next token begins

- after **","** is reached, it can be determined.

https://web.stanford.edu/class/cs143/lectures/lecture03.pdf

Lookahead

Even our simple example has lookahead issues

– i vs. if

– = vs. ==

https://web.stanford.edu/class/cs143/lectures/lecture03.pdf

PL/I keywords are not reserved

IF ELSE THEN THEN = ELSE; ELSE ELSE = THEN

PL/I Declarations:

DECLARE (ARG1,. . ., ARGN)

- Cannot tell whether DECLARE is a keyword or array reference until after the **).**
  - Requires arbitrary **lookahead**!

https://web.stanford.edu/class/cs143/lectures/lecture03.pdf

The problems continue today

Lexical Analysis in C++

C++ template syntax:

`Foo<Bar>`

C++ stream syntax:

`cin >> var;`

But there is a conflict with nested templates:

`Foo<Bar<Bazz>>`

https://web.stanford.edu/class/cs143/lectures/lecture03.pdf

Goal

The goal of lexical analysis is to

- Partition the input string into lexemes
- Identify the token of each lexeme

★ Left-to-right scan => lookahead sometimes required

We need

– A way to describe the lexemes of each token

– A way to resolve ambiguities

- Is **if** two variables **i** and **f**?
- Is **==** two equal signs **= =**?

Regular Languages

There are several formalisms for specifying tokens

• Regular languages are the most popular

– Simple and useful theory

– Easy to understand

– Efficient implementations

Languages

Definition:

Alphabet(Σ) is a set of characters (symbols).

String (s): a finite, possibly empty sequence of symbols from an alphabet

Language (L(s)) over **Σ** is a set of strings of characters drawn from **Σ**.

Alphabet = English characters

Language = English sentences

★ Not every string of English characters is an English sentence

Alphabet = ASCII

➔ Note: ASCII character set is different from English character set

Language = C programs

Regular expressions

Need some notation for specifying which sets we want

The standard notation for **regular languages** is **regular expressions**.

- **s** is a string
- **L(s)** is the "language of **s.**"

A regular expression s is a string which denotes **L(s)**, a set of strings drawn from an alphabet **Σ**.

Regular expression base case

- **s** is a string
- **L(s)** is the “language of **s.**”

A regular expression s is a string which denotes **L(s)**, a set of strings drawn from an alphabet **Σ**.

Single character

- If **a** $\in$ **Σ** then,
  - a is a regular expression
  - and **L(a) = {a}**

ε is a regular expression

- **L(ε)** contains only the empty string, {“”}

Regular expression built up rules

Union (alternation)

s|t is a RE such that

L(s|t) = L(s) ∪ L(t).

Different notation

A+ B = {s| s∈ A or s∈ B}

Concatenation

st is a RE such that

- L(st) contains all strings formed by the concatenation of a string in L(s) followed by a string in L(t).

Different notation

AB = {ab | a∈ A and b∈ B}

Closure(Iteration)

The Kleene closure of a set A, denoted A^*

$$A^* = \bigcup_{i \geq 0} A^i = A^0 \cup A^1 \cup A^2 \cup A^3 \cup A^4 \cup \cdots.$$

Examples of closures

Closure(Iteration)

The Kleene closure of a set A, denoted A*

$$A^* = \bigcup_{i \geq 0} A^i = A^0 \cup A^1 \cup A^2 \cup A^3 \cup A^4 \cup \cdots.$$

$$A^0 = \{\epsilon\}$$

$$A^1 = \{A\}$$

{"ab","c"}* =

{ ε, "ab", "c", "abab", "abc", "cab", "cc", "ababab", "ababc", "abcab", "abcc", "cabab", "cabc", "ccab", "ccc", ...}.

{"a", "b", "c"}* =

{ ε, "a", "b", "c", "aa", "ab", "ac", "ba", "bb", "bc", "ca", "cb", "cc", "aaa", "aab", ...}.

$\varnothing^*$ = {ε}.

examples

<table>
<tr><th>Regular Expression s</th><th>Language L(s)</th></tr>
<tr><td>hello</td><td>{hello}</td></tr>
<tr><td>d(o|i)g</td><td>{dog, dig}</td></tr>
<tr><td>moo*</td><td>{mo, moo, mooo, ...}</td></tr>
<tr><td>(moo)*</td><td>{ǫ,moo,moomoo,moomoomoo,... }</td></tr>
<tr><td>a(b|a)*a</td><td>{aa,aaa,aba,aaaa,aaba,abaa,... }</td></tr>
</table>

Abbreviations and some properties

$s? = (s|\boldsymbol{\varepsilon})$

$s+ = ss^*$

$[a\text{-}z] = (a|b|...|z)$

[^x] is Σ - x

- Union: $A + B \equiv A \mid B$
- Option: $A + \varepsilon \equiv A?$
- Range: ‘a’+’b’+…+’z’ ≡ [a-z]
- Excluded range: complement of [a-z] ≡ [^a-z]

Associativity:	`a|(b|c) = (a|b)|c`
Commutativity:	`a|b = b|a`
Distribution:	`a(b|c) = ab|ac`
Idempotency:	`a** = a*`

https://web.stanford.edu/class/cs143/lectures/lecture03.pdf

exercises

[0-9]+(.[0-9]+)?

Which one matches?

- **123**
  - matches
- **3.14**
  - matches
- **.15**
  - Do not match
- **30.**
  - Do not match

[A-Z]+([A-Z]|[0-9])*

Which one matches?

- **PRINT**
  - matches
- **MODE5**
  - matches
- **hello**
  - Do not match
- **4YOU**
  - Do not match

<[^>]*>

Which one matches?

- **<tricky part>**
  - matches
- **<<<look left>**
  - matches
- **<this is an <illegal> comment>**
  - Does not match

examples

Regular Expression s	Language L(s)
[abc]+	
[abc]*	
[0-9]+	
[1-9][0-9]*	
[a-zA-Z][a-zA-Z0-9_]*	

Regular expression is used to describe grammars

Keywords

“else” or “if” or “begin” or …

‘else’ + ‘if’ + ‘begin’ + . . .

Abbreviation: ‘else’ = ‘e’ ‘l’ ‘s’ ‘e’

Integers: an empty string of digits

```
digit = '0' + '1' + '2' + '3' + '4' + '5' + '6' '7'
+ '8'+ '9'
```

```
integer = digit digit*
```

Abbreviation: $A^+ = AA^*$

Abbreviation: [0-2] = '0' + '1' + '2'

https://web.stanford.edu/class/cs143/lectures/lecture03.pdf

Identifier

Identifier: strings of letters or digits, starting with a letter

letter = ‘A’ + . . . + ‘Z’ + ‘a’ + . . . + ‘z’

identifier = letter (letter + digit)*

Is **(letter* + digit*)** = **(letter + digit)*** ?

https://web.stanford.edu/class/cs143/lectures/lecture03.pdf

Whitespace

Whitespace: a **non-empty sequence of blanks**, **newlines**, and **tabs**

$$(\text{' '} + \text{'\textbackslash n'} + \text{'\textbackslash t'})^+$$

https://web.stanford.edu/class/cs143/lectures/lecture03.pdf

Numeric constants

digit ::= [0-9]

digits ::= digit+

number ::= digits (. digits)?

([eE] (+ | -)? digits) ?

Phone Numbers

Consider (650)-723-3232

$\sum$ = digit + {-, (,),}

exchange = digit^3

phone = digit^4

area = digit^3

phone_number = '(' area ')-' exchange '-'phone

https://web.stanford.edu/class/cs143/lectures/lecture03.pdf

Recognizing REs

Given string s and rexp R,

is $s \in L(R)$?

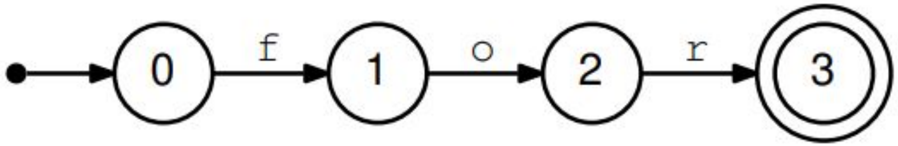
Finite automata can be used to recognize strings generated by regular expressions

Can build by hand or automatically

– Reasonably straightforward, and can be done systematically

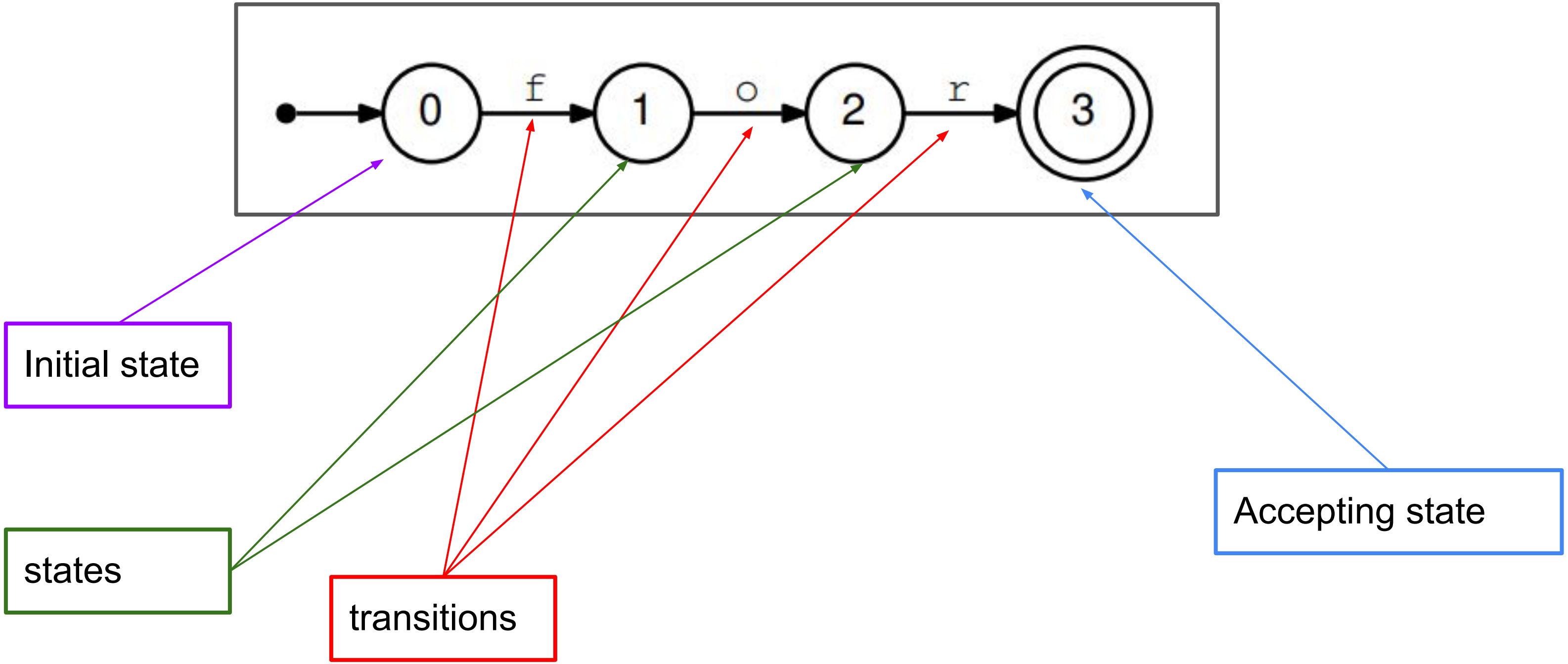
– Tools like **Lex, Flex, JFlex** etc this automatically, given a set of REs.

– Same techniques used in **grep**, **sed**, and other packages/tools



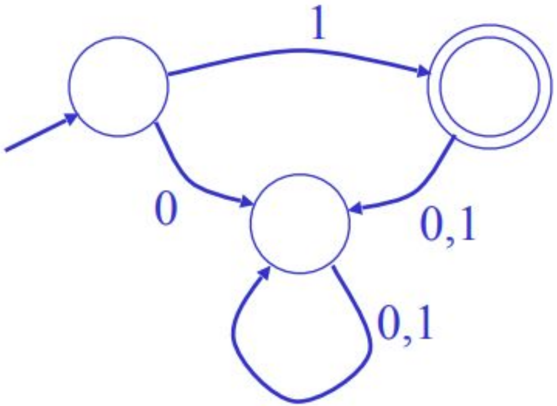
Finite Automata state graphs

A finite automaton (FA) is an abstract machine that can be used to represent certain forms of computation.



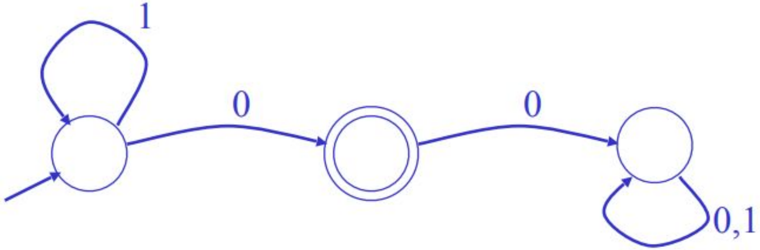
A simple example

A finite automaton that accepts only “1”



A finite automaton accepting any number of 1’s followed by a single 0

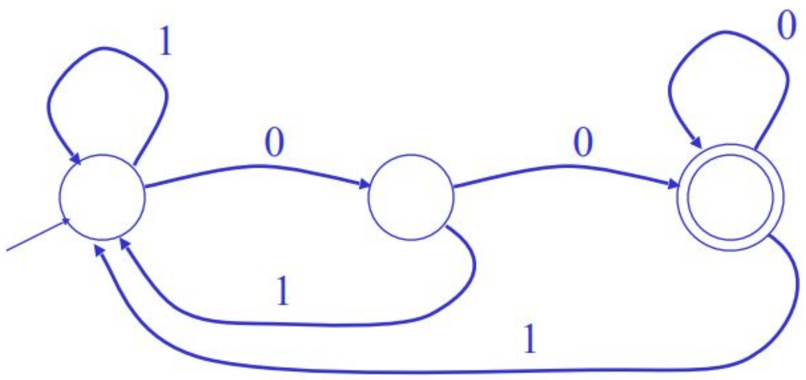
Alphabet: {0,1}



https://web.stanford.edu/class/cs143/lectures/lecture03.pdf

Alphabet {0,1}

What language does this recognize?



https://web.stanford.edu/class/cs143/lectures/lecture03.pdf

Deterministic finite automata(DFA)

A DFA is a special case of an FA

- where **every state has no more than one outgoing edge for a given symbol**.

A DFA has no ambiguity:

for every combination of state and input symbol, there is exactly one choice of what to do next.

https://www3.nd.edu/~dthain/compilerbook/chapter3.pdf

Deterministic finite automata(DFA)

One integer (c) is needed to keep track of the current state.

The transitions between states are represented by a matrix M

- **M[s, i]** encodes the next state, given the current state **s** and input symbol **i**.

- **Error:** (If the transition is not allowed, we mark it with E to indicate an error.)

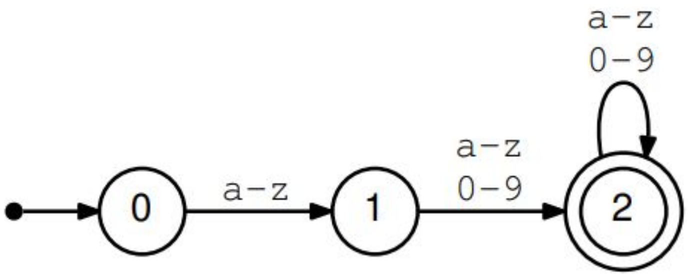
Acceptance:

For each symbol,

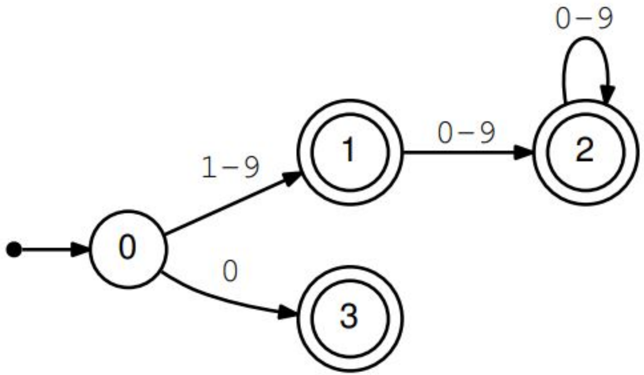
- We compute **c = M[s, i]** until all the input is consumed, or an error state is reached.

https://www3.nd.edu/~dthain/compilerbook/chapter3.pdf

[a-z][a-z0-9]+



([1-9][0-9]*)|0

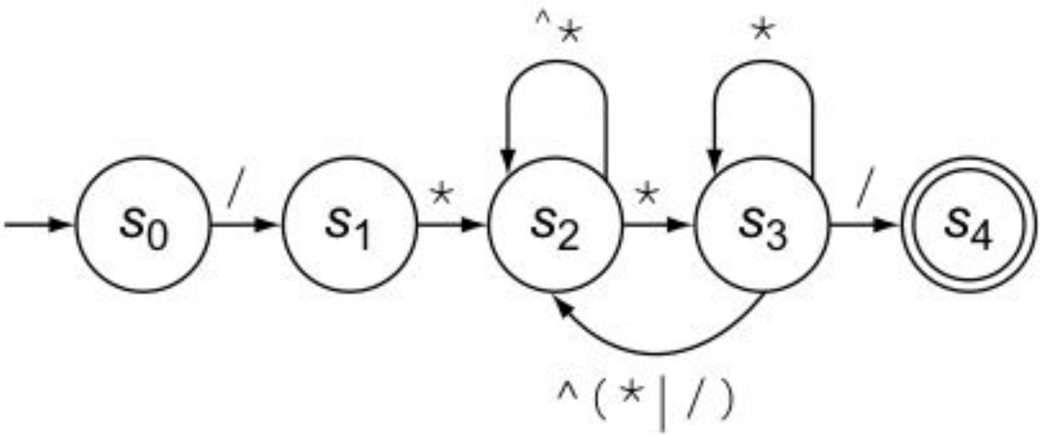


https://www3.nd.edu/~dthain/compilerbook/chapter3.pdf

RE for comments in C++, Java

/* (^*)* */ without *

/* (^* | * + ^/)* */

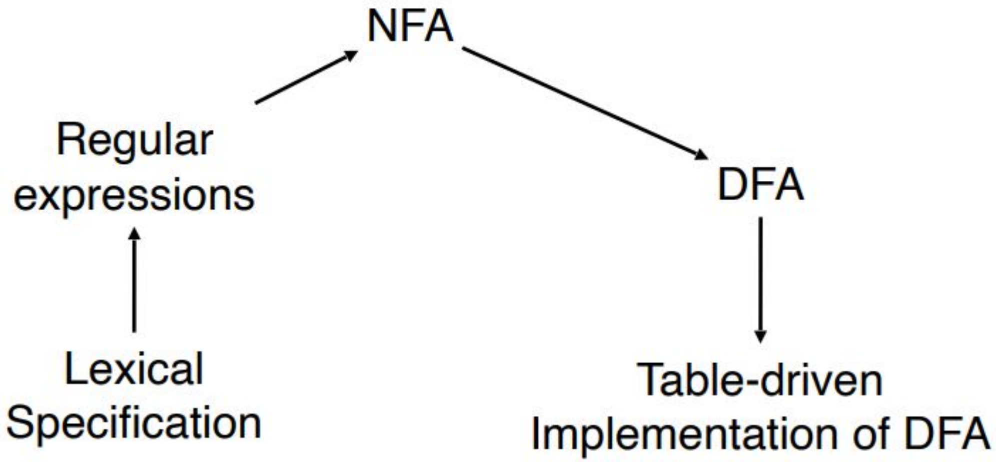


A more correct written form with escape char\ and without nested comments:

```
/\* ([^*]|\*+[^*/])*\*+/
```

Convert Regular Expressions to Finite Automata

High level sketch



https://web.stanford.edu/class/cs143/lectures/lecture04.pdf

Regular Expressions in Lexical Specification

a specification for the predicate

$$s \in L(R)$$

- **yes/no answer is not enough**!
- Instead: **partition the input into tokens**

- We will adapt regular expressions to this goal

https://web.stanford.edu/class/cs143/lectures/lecture04.pdf

Lexical Specification → Regex in five steps

1. Write a regex for each token

- `Number = digit +`
- `Keyword = 'if' + 'else' + …`
- `Identifier = letter (letter + digit)*`
- `OpenPar = '('`
- …

2. Construct R, matching all lexemes for all tokens

```
R     = Keyword + Identifier + Number + …
      = R1 + R2 + …
```

`(This step is done automatically by tools like flex)`

https://web.stanford.edu/class/cs143/lectures/lecture04.pdf

3. Let input be x1 … xn

for 1 ≤ i ≤ n check

x1…xi ∈ L(R)

4. **If success,**

then we know that

x1…xi ∈ L(Rj) for some j

5. Remove x1…xi from input and go to (3)

https://web.stanford.edu/class/cs143/lectures/lecture04.pdf

Ambiguity

There are ambiguities in the algorithm

- How much input is used?

What if **x1…xi ∈ L(R) and** also **x1…xk ∈ L(R)**

- Rule: Pick longest possible string in L(R)
  - Pick **k if k > i**
  - The "maximal munch"

https://web.stanford.edu/class/cs143/lectures/lecture04.pdf

Ambiguity

Which token is used?

What if **x1…xi** $\in$ **L(Rj)** **and** **x1…xi** $\in$ **L(Rk)**

- Rule: use rule listed first
  - Pick **j** if **j < k**
  - E.g., treat "if" as a keyword, not an identifier

https://web.stanford.edu/class/cs143/lectures/lecture04.pdf

Error handling

What if **No rule matches a prefix** of input?

- **Problem:** Can't just get stuck …
- **Solution:**
  - Write a rule matching all "bad" strings
  - Put it last (lowest priority)

https://web.stanford.edu/class/cs143/lectures/lecture04.pdf

Error handling

Regular expressions provide a concise notation for string patterns

- Use in lexical analysis requires small extensions
  - To resolve ambiguities
  - To handle errors
- Good algorithms
  - Require only single pass over the input
  - Few operations per character (table lookup)

https://web.stanford.edu/class/cs143/lectures/lecture04.pdf

Finite automata again

Regular expressions = specification

Finite automata = implementation

Deterministic Finite Automata (DFA)

– Exactly one transition per input per state

– No ε-moves

Nondeterministic Finite Automata (NFA)

– Can have zero, one, or multiple transitions for one input in a given state

– Can have ε-moves

https://web.stanford.edu/class/cs143/lectures/lecture04.pdf

HW1

Posted!