Programming Languages and Compilers (CS 421)

Choonghwan Lee

http://www.cs.illinois.edu/class/cs421/

Based in part on slides by Mattox Beckman, as updated by Vikram Adve, Gul Agha and Elsa Gunter

Meta-discourse

- Language Syntax and Semantics
- Syntax
  - DFSAs and NDFSAs
  - Grammars
- Semantics
  - Natural Semantics
  - Transition Semantics

Language Syntax

- Syntax is the description of which strings of symbols are meaningful expressions in a language
- It takes more than syntax to understand a language; need meaning (semantics) too
- Syntax is the entry point

Syntax of English Language

- Pattern 1

Subject	Verb
David	*sings*
The dog	*barked*
Susan	*yawned*

- Pattern 2

Subject	Verb	Direct Object
David	*sings*	*ballads*
The professor	*wants*	*to retire*
The jury	*found*	*the defendant guilty*

Elements of Syntax

- Character set – previously always ASCII, now often 64 character sets
- Keywords – usually reserved
- Special constants – cannot be assigned to
- Identifiers – can be assigned to
- Operator symbols
- Delimiters (parenthesis, braces, brackets)
- Blanks (aka white space)

Elements of Syntax

- Expressions

  if ... then begin ... ; ... end else begin ... ; ... end
- Type expressions

  $typexpr_1$ -> $typexpr_2$
- Declarations (in functional languages)

  let $pattern_1$ = $expr_1$ in *expr*
- Statements (in imperative languages)

  a = b + c
- Subprograms

  let $pattern_1$ = let rec inner = … in *expr*

Elements of Syntax

- Modules
- Interfaces
- Classes (for object-oriented languages)

Formal Language Descriptions

- Regular expressions, regular grammars, finite state automata

- Context-free grammars, BNF grammars, syntax diagrams

- Whole family more of grammars and automata – covered in automata theory

Grammars

- Grammars are formal descriptions of which strings over a given character set are in a particular language
- Language designers write grammar
- Language implementers use grammar to know what programs to accept
- Language users use grammar to know how to write legitimate programs

Regular Expressions

- Start with a given character set – **a, b, c**...

- Each character is a regular expression
  - It represents the set of one string containing just that character

Regular Expressions

- If **x** and **y** are regular expressions, then **xy** is a regular expression
  - It represents the set of all strings made from first a string described by **x** then a string described by **y**

If x={a,ab} and y={c,d} then xy ={ac,ad,abc,abd}.

- If **x** and **y** are regular expressions, then **x**$\vee$**y** is a regular expression
  - It represents the set of strings described by either **x** or **y**

  If x={a,ab} and y={c,d} then x $\vee$ y={a,ab,c,d}

Regular Expressions

- If **x** is a regular expression, then so is **(x)**
  - It represents the same thing as **x**

- If **x** is a regular expression, then so is **x***
  - It represents strings made from concatenating zero or more strings from **x**

  If x = {a,ab}

  then x* ={“”,a,ab,aa,aab,abab,aaa,aaab,...}

- **ε**
  - It represents {“”}, set containing the empty string

Example Regular Expressions

- **$(0 \vee 1)^*1$**
  - The set of all strings of **0**'s and **1**'s ending in 1, **{1, 01, 11,...}**
- **$a^*b(a^*)$**
  - The set of all strings of a's and b's with exactly one b
- **$((01) \vee (10))^*$**
  - You tell me
- Regular expressions (equivalently, regular grammars) important for lexing, breaking strings into recognized words

Example: Lexing

- Regular expressions good for describing lexemes (words) in a programming language
  - Identifier = $(a \vee b \vee \ldots \vee z \vee A \vee B \vee \ldots \vee Z)\ (a \vee b \vee \ldots \vee z \vee A \vee B \vee \ldots \vee Z \vee 0 \vee 1 \vee \ldots \vee 9)^*$
  - Digit = $(0 \vee 1 \vee \ldots \vee 9)$
  - Number = $0 \vee (1 \vee \ldots \vee 9)(0 \vee \ldots \vee 9)^* \vee \sim (1 \vee \ldots \vee 9)(0 \vee \ldots \vee 9)^*$
  - Keywords: if = if, while = while,...

Implementing Regular Expressions

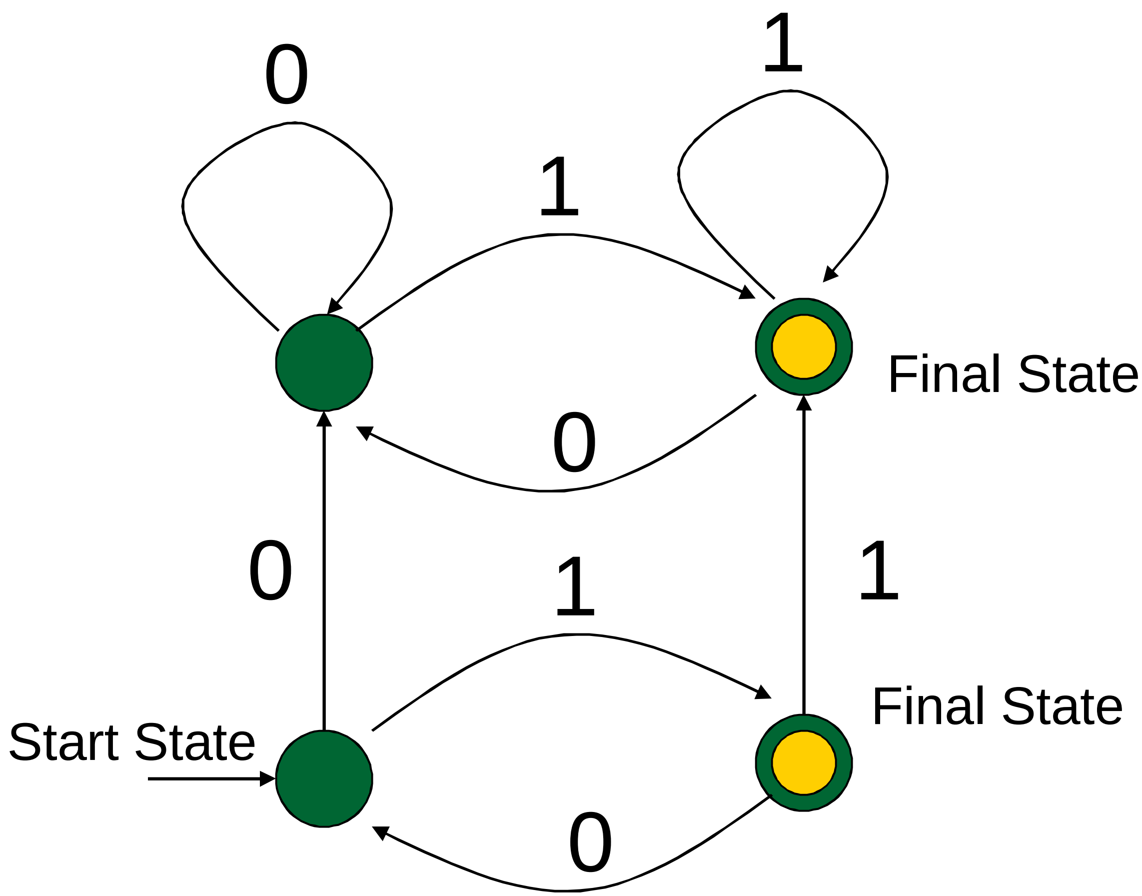
- Regular expressions reasonable way to generate strings in language
- Not so good for recognizing when a string is in language
- Problems with Regular Expressions
  - which option to choose,
  - how many repetitions to make
- Answer: finite state automata

Finite State Automata

- A finite state automata over an alphabet is:
  - a directed graph
  - a finite set of states defined by the nodes
  - edges are labeled with elements of alphabet, or empty string; they define state transition
  - some nodes (or *states*), marked as final
  - one node marked as start state

- Syntax of FSA

Example FSA



Deterministic FSA's

- If FSA has for every state *exactly one* edge for each letter in alphabet then FSA is *deterministic*
  - No edge labeled with ε
- In general FSA in *non-deterministic.*
  - NFSA also allows edges labeled by ε
- Deterministic FSA special kind of non-deterministic FSA

DFSA Language Recognition

- Think of a DFSA as a board game; DFSA is board
- You have string as a deck of cards; one letter on each card
- Start by placing a disc on the start state

DFSA Language Recognition

- Move the disc from one state to next along the edge labeled the same as top card in deck; discard top card
- When you run out of cards,
  - if you are in final state, you win; string is in language
  - if you are not in a final state, you lose; string is not in language

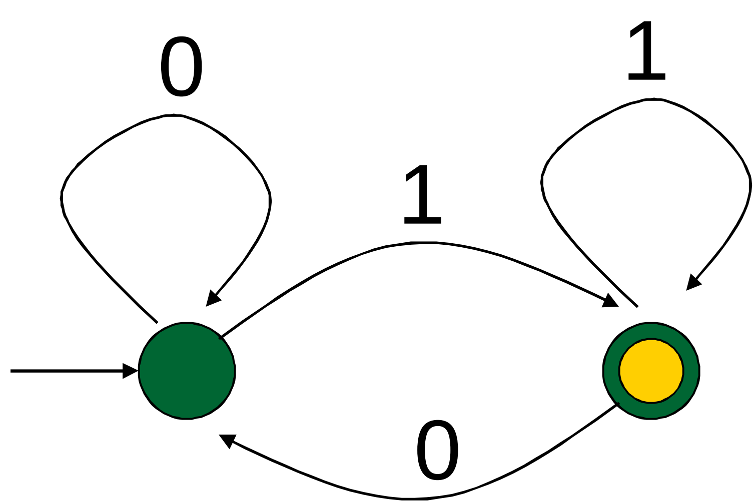
DFSA Language Recognition -Summary

- Given a string over alphabet
- Start at start state
- Move over edge labeled with first letter to new state
- Remove first letter from string
- Repeat until string gone
- If end in final state then string in language

- Semantics of FSA

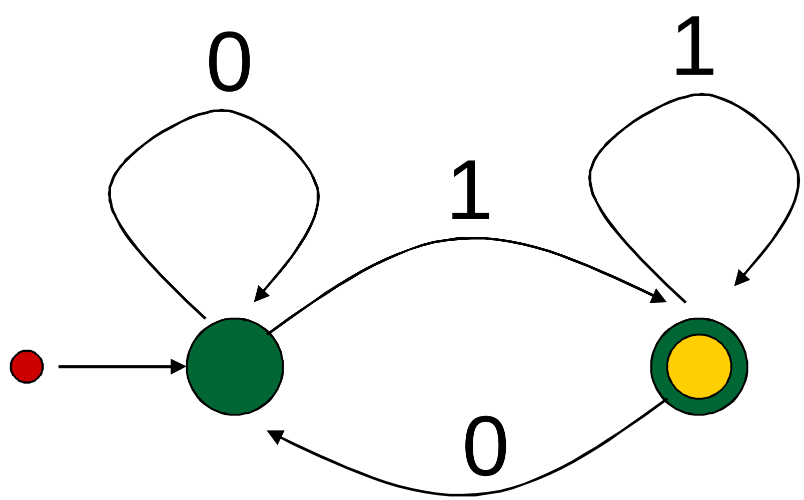
Example DFSA

- Regular expression: $(0 \vee 1)^*\ 1$
- Deterministic FSA



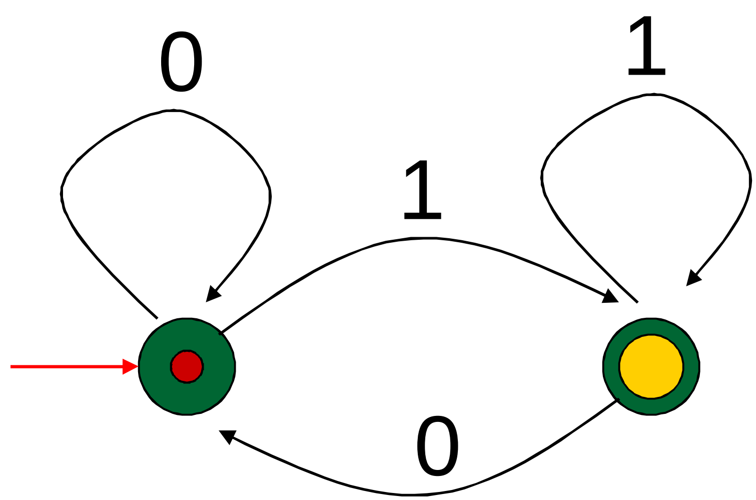
Example DFSA

- Regular expression: $(0 \vee 1)^*\ 1$
- Accepts string 0 1 1 0 1



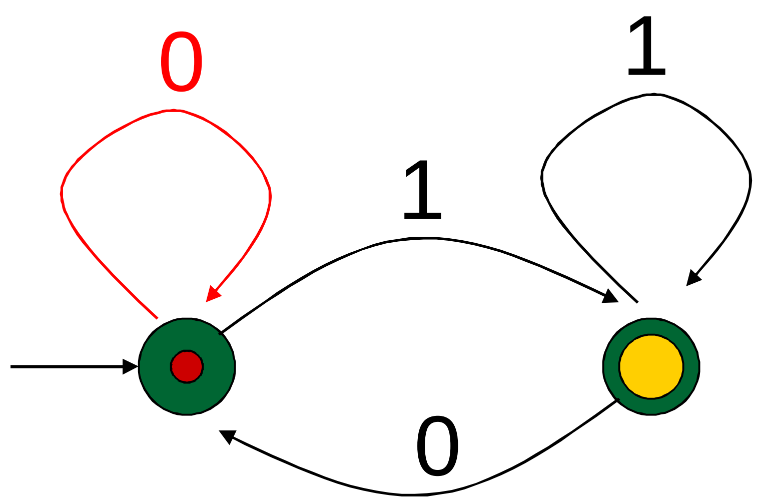
Example DFSA

- Regular expression: $(0 \vee 1)^*\ 1$
- Accepts string 0 1 1 0 1



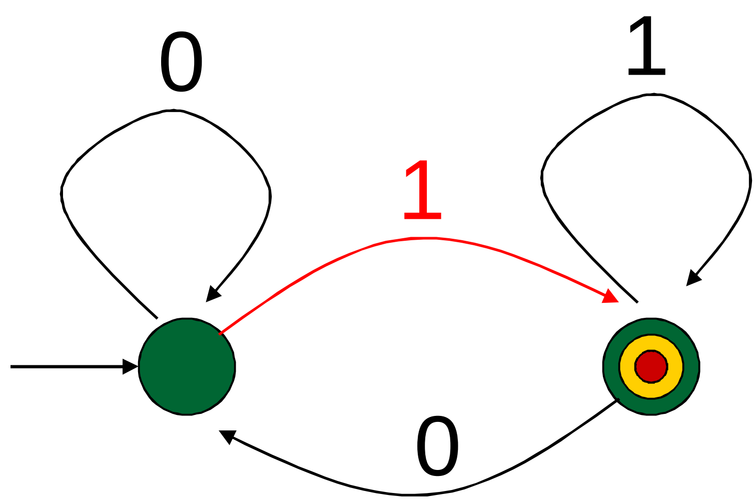
Example DFSA

- Regular expression: $(0 \vee 1)^* \ 1$
- Accepts string ~~0~~ 1 1 0 1



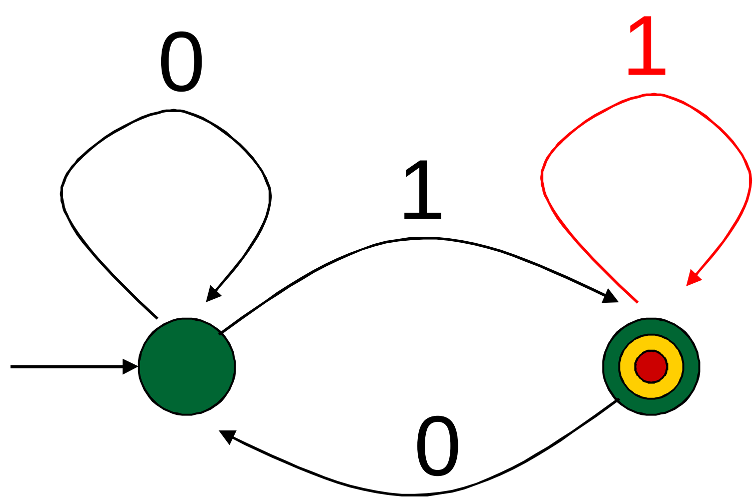
Example DFSA

- Regular expression: $(0 \vee 1)^* \ 1$
- Accepts string $\cancel{0}\ \cancel{1}$ 1 0 1



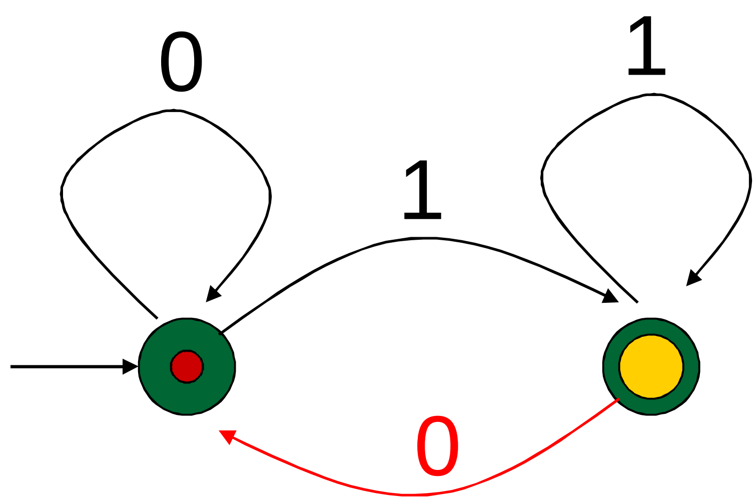
Example DFSA

- Regular expression: $(0 \vee 1)^* \; 1$
- Accepts string ~~0~~ ~~1~~ ~~1~~ 0 1



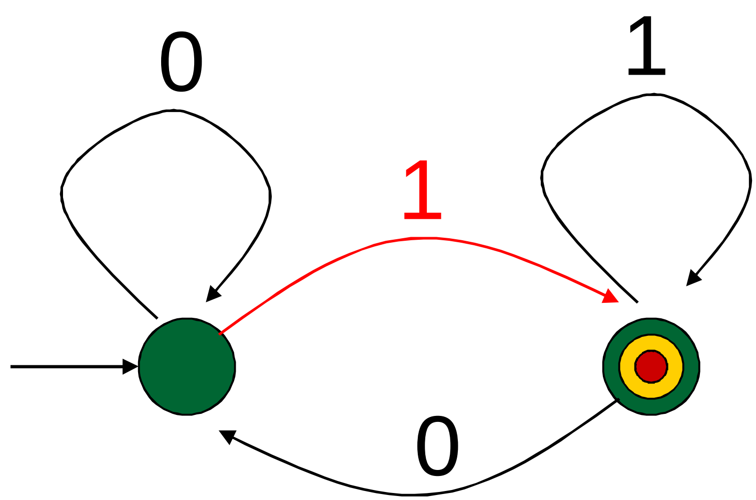
Example DFSA

- Regular expression: (0 ∨ 1)* 1
- Accepts string ~~0~~ ~~1~~ ~~1~~ ~~0~~ 1



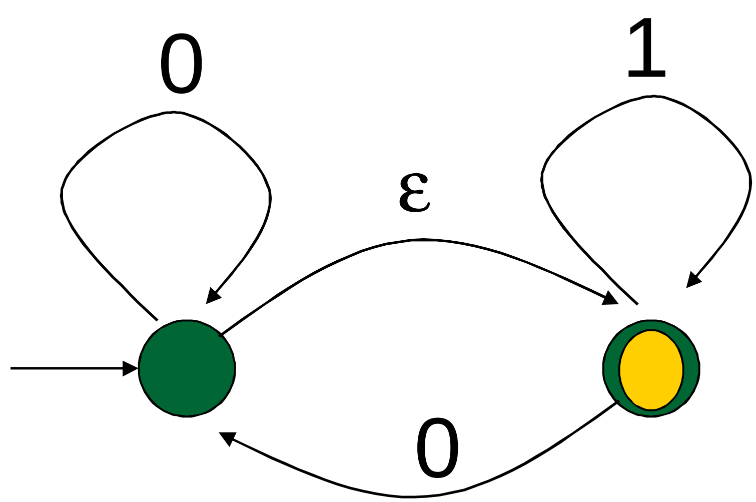
Example DFSA

- Regular expression: $(0 \vee 1)^*\ 1$
- Accepts string ~~0~~ ~~1~~ ~~1~~ ~~0~~ ~~1~~



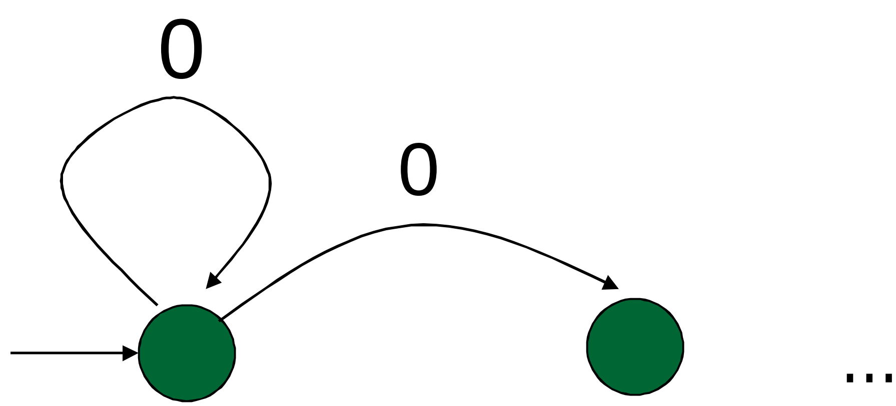
Non-deterministic FSA's

- NFSA generalize DFSA in two ways:
- Include edges labeled by ε
  - Allows process to non-deterministically change state



Non-deterministic FSA's

- Each state can have zero, one or more edges labeled by each letter
  - Given a letter, non-deterministically choose an edge to use

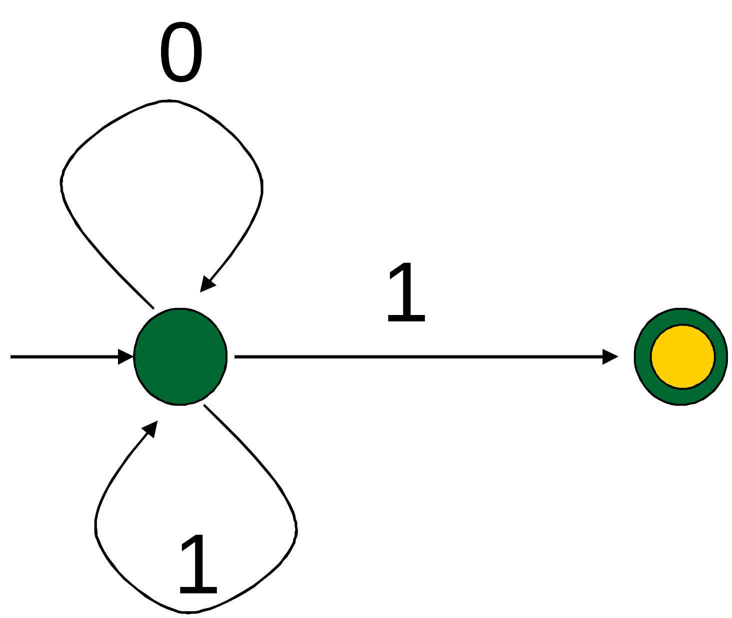


NFSA Language Recognition

- Play the same game as with DFSA
- Free move: move across an edge with empty string label without discarding card
- When you run out of letters, if you are in final state, you win; string is in language
- You can take one or more moves back and try again
- If have tried all possible paths without success, then you lose; string not in language

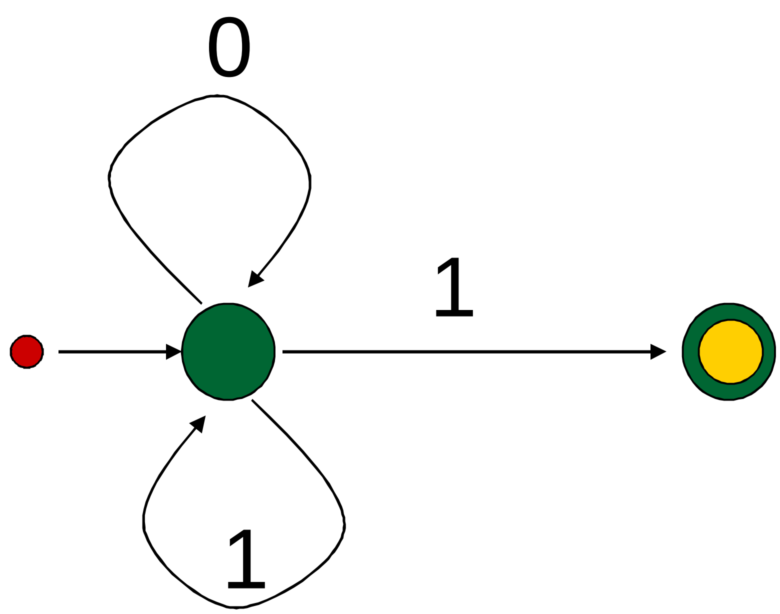
Example NFSA

- Regular expression: $(0 \vee 1)^*\ 1$
- Non-deterministic FSA



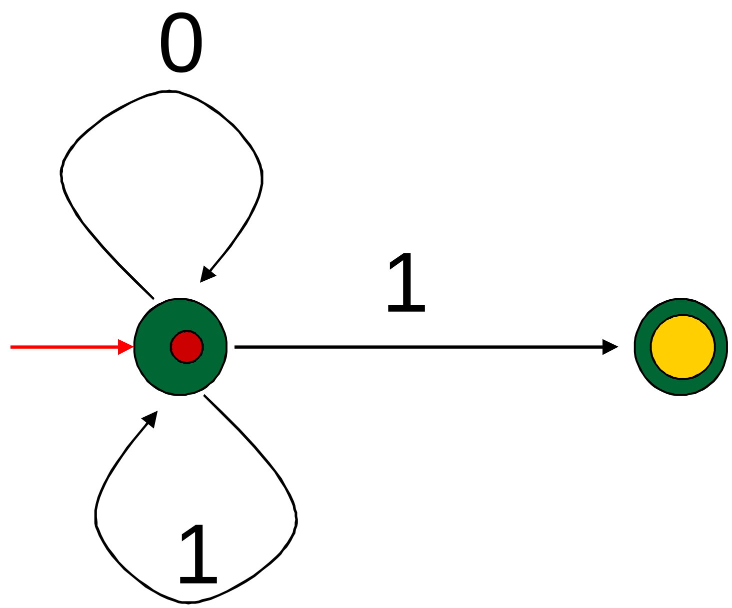
Example NFSA

- Regular expression: $(0 \vee 1)^* \ 1$
- Accepts string 0 1 1 0 1



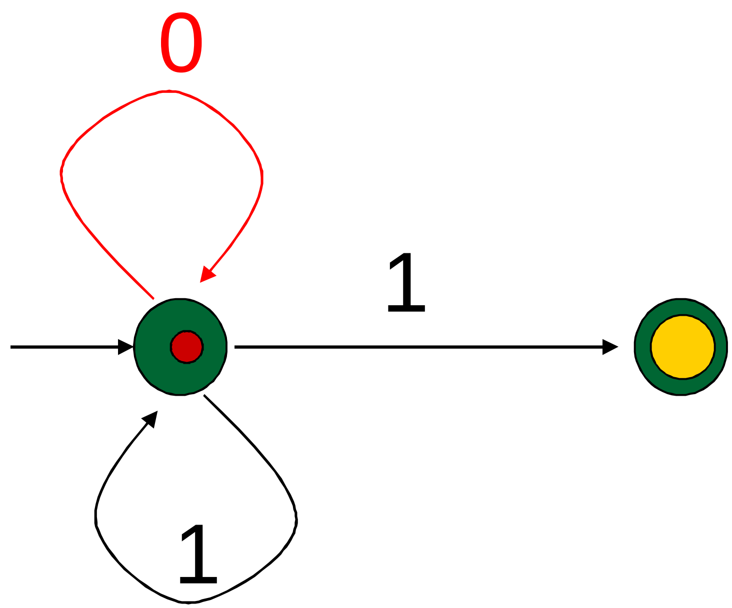
Example NFSA

- Regular expression: $(0 \vee 1)^* \ 1$
- Accepts string   0 1 1 0 1



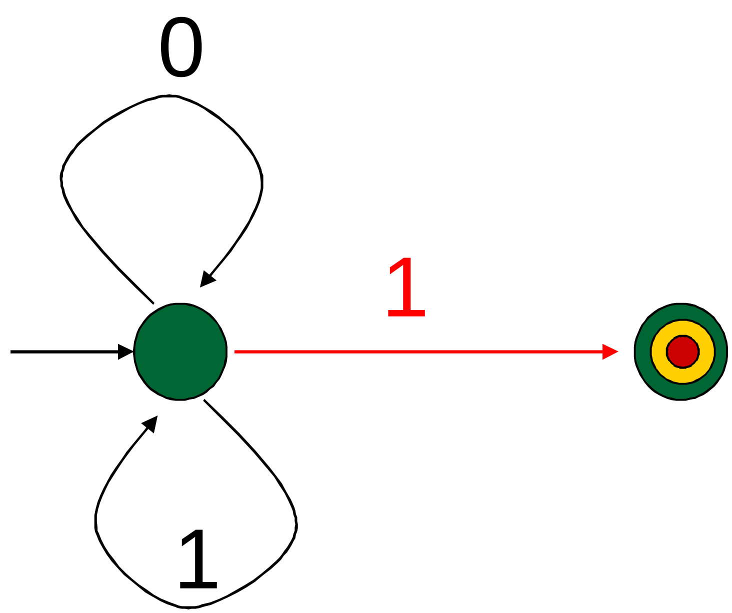
Example NFSA

- Regular expression: (0 ∨ 1)* 1
- Accepts string ~~0~~ 1 1 0 1



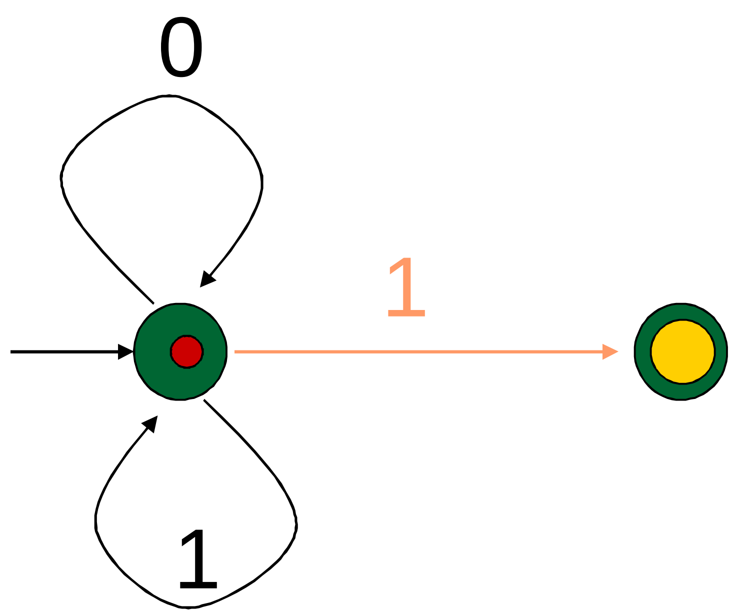
Example NFSA

- Regular expression: $(0 \vee 1)^*\ 1$
- Accepts string ~~0~~ ~~1~~ 1 0 1
- Guess



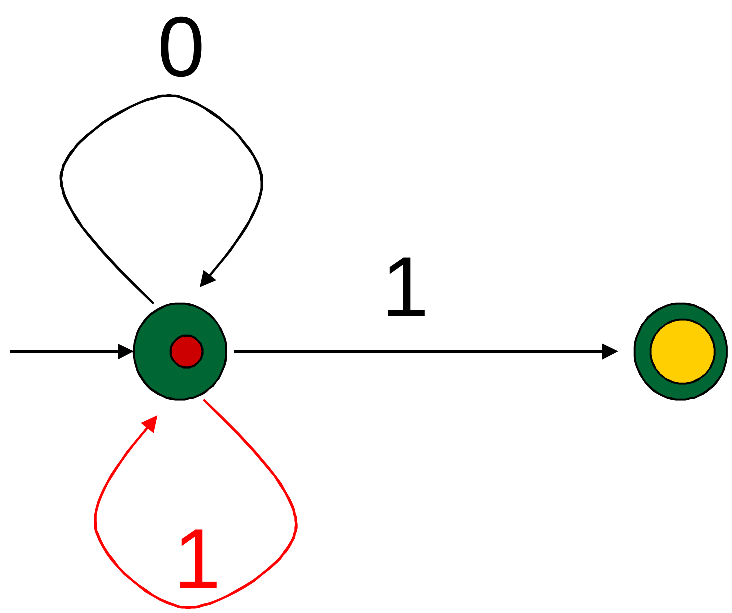
Example NFSA

- Regular expression: $(0 \vee 1)^* \ 1$
- Accepts string 0 1 1 0 1
- Backtrack



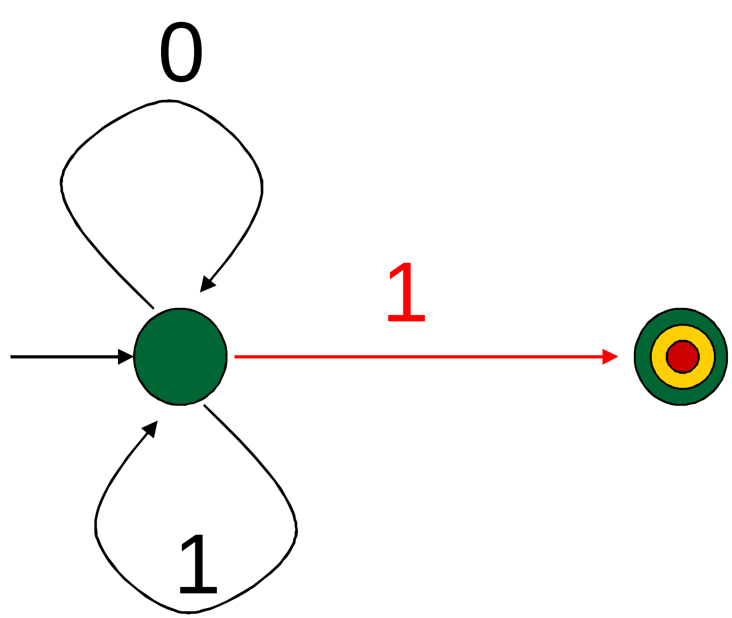
Example NFSA

- Regular expression: $(0 \vee 1)^* \ 1$
- Accepts string ~~0~~ ~~1~~ 1 0 1
- Guess again



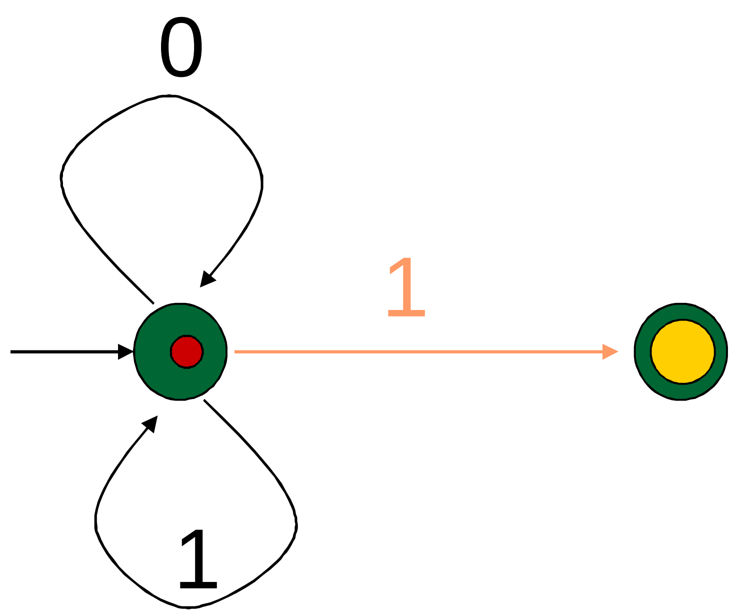
Example NFSA

- Regular expression: $(0 \vee 1)^* \, 1$
- Accepts string ~~0~~ ~~1~~ ~~1~~ 0 1
- Guess



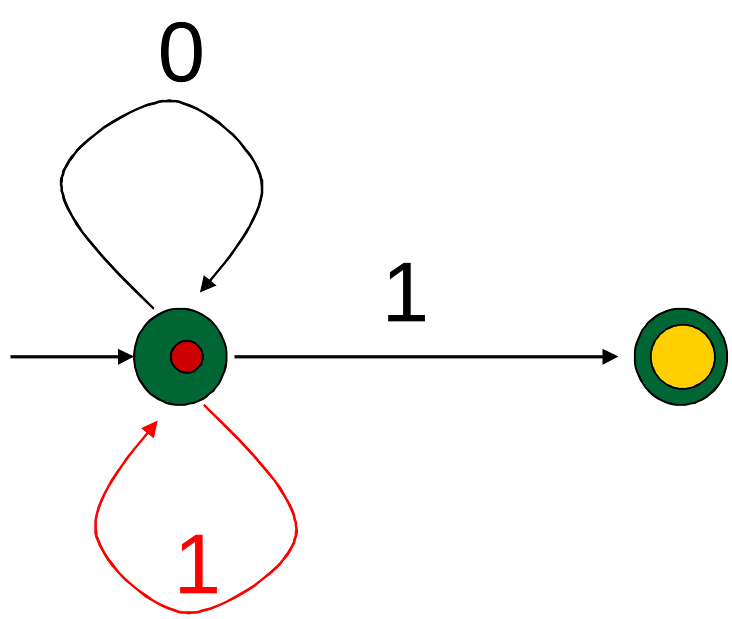
Example NFSA

- Regular expression: $(0 \vee 1)^*\ 1$
- Accepts string ~~0~~ ~~1~~ 1 0 1
- Backtrack



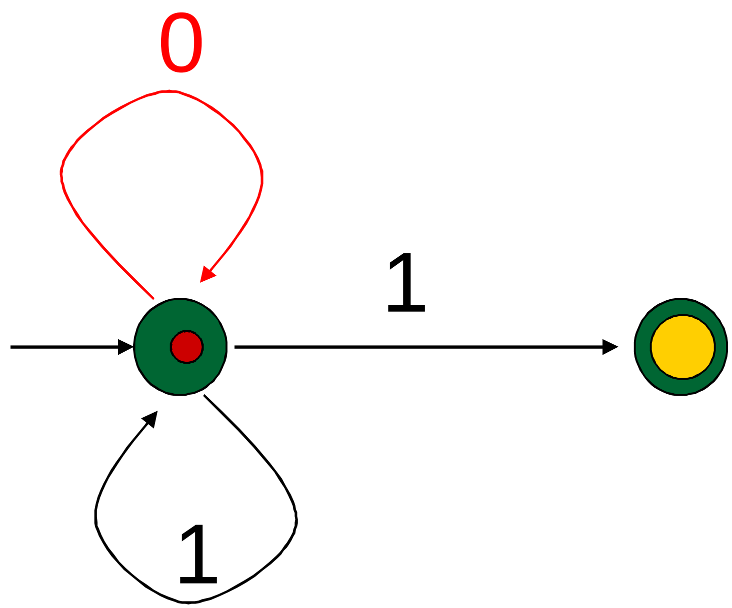
Example NFSA

- Regular expression: $(0 \vee 1)^{*}\ 1$
- Accepts string ~~0~~ ~~1~~ ~~1~~ 0 1
- Guess again



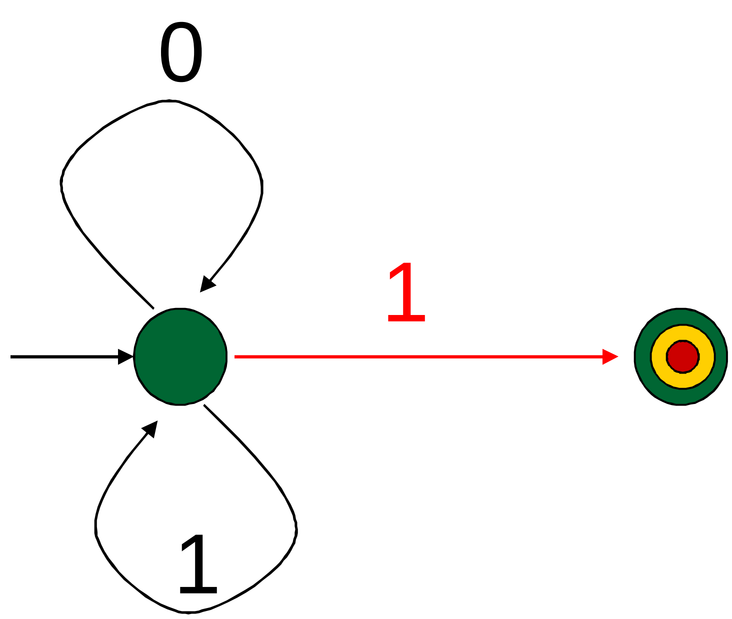
Example NFSA

- Regular expression: $(0 \vee 1)^*\ 1$
- Accepts string ~~0~~ ~~1~~ ~~1~~ ~~0~~ 1



Example NFSA

- Regular expression: $(0 \vee 1)^* \ 1$
- Accepts string $\cancel{0}\ \cancel{1}\ \cancel{1}\ \cancel{0}\ \cancel{1}$
- Guess (Hurray!!)



Rule Based Execution

- Search
- When stuck backtrack to last point with choices remaining
- Executing the NFSA in last example was example of rule based execution
- FSA's are rule-based programs; transitions between states (labeled edges) are rules; set of all FSA's is programming language

Rule Based Execution

- Search
- When stuck backtrack to last point with choices remaining

- FSA’s are rule-based programs; transitions between states (labeled edges) are rules; set of all FSA’s is programming language

Where We Are Going

- We want to turn strings (code) into computer instructions
- Done in phases
- Turn strings into abstract syntax trees (parse)
- Translate abstract syntax trees into executable instructions (interpret or compile

Lexing and Parsing

- Converting strings to abstract syntax trees done in two phases
  - **Lexing:** Converting string (or streams of characters) into lists (or streams) of tokens (the “words” of the language)
  - **Parsing:** Convert a list of tokens into an abstract syntax tree

Lexing

- Different syntactic categories of “words”: tokens

Example:

- Convert sequence of characters into sequence of strings, integers, and floating point numbers.
- "asd 123 jkl 3.14" will become:

  [String "asd"; Int 123; String "jkl"; Float 3.14]

Lexing

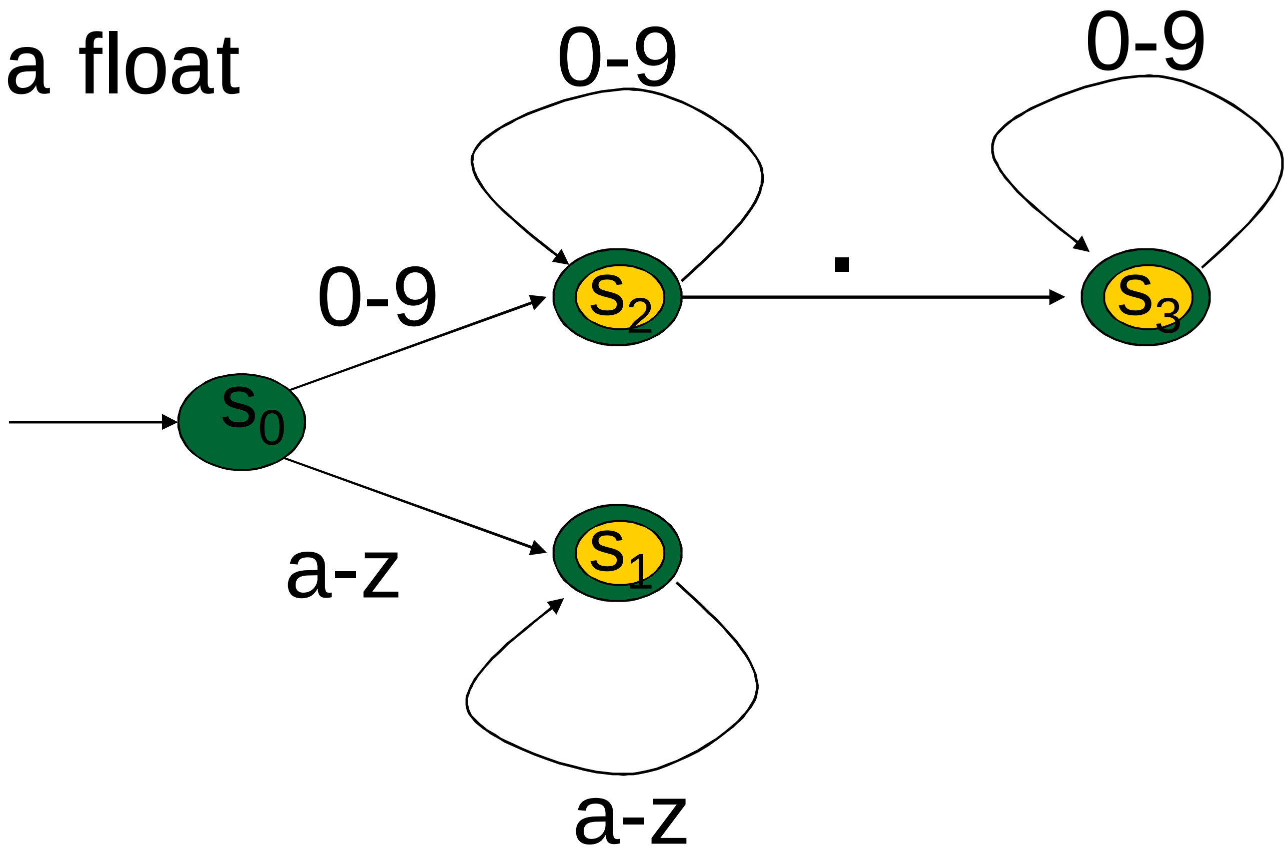
- Each category described by regular expression (with extended syntax)
- Words recognized by (encoding of) corresponding finite state automaton
- Problem: we want to pull words out of a string; not just recognize a single word

Lexing

- Modify behavior of DFA
- When we encounter a character in a state for which there is no transaction
  - Stop processing the string
  - If in an accepting state, return the token that corresponds to the state, and the remainder of the string
  - If not, fail
- Add recursive layer to get sequence

Example

- s_1 return a string
- s_2 return an integer
- s_3 return a float



Lex, ocamllex

- Could write the reg exp, then translate to DFA by hand
  - A lot of work
- Better: Write program to take reg exp as input and automatically generates automata
- Lex is such a program
- ocamllex version for ocaml

How to do it

- To use regular expressions to parse our input we need:
  - Some way to identify the input string — call it a lexing buffer
  - Set of regular expressions,
  - Corresponding set of actions to take when they are matched.

How to do it

- The lexer will take the regular expressions and generate a state machine.
- The state machine will take our lexing buffer and apply the transitions...
- If we reach an accept state from which we can go no further, the machine will perform the appropriate action.

Mechanics

- Put table of reg exp and corresponding actions (written in ocaml) into a file <*filename*>.mll
- Call

  ocamllex <*filename*>.mll

- Produces Ocaml code for a lexical analyzer in file <*filename*>.ml

Sample Input

```
rule main = parse
 ['0'-'9']+ { print_string "Int\n"}
 | ['0'-'9']+'.'['0'-'9']+ { print_string "Float\n"}
 | ['a'-'z']+ { print_string "String\n"}
 | _ { main lexbuf }
 {
 let newlexbuf = (Lexing.from_channel stdin) in
 print_string "Ready to lex.\n";
 main newlexbuf
}
```

General Input

{ *header* }

let *ident* = *regexp* …

rule *entrypoint* [*arg1*… *argn*] = parse

        regexp { *action* }

    | …

    | *regexp* { *action* }

and *entrypoint* [*arg1*… *argn*] = parse …and

…

{ *trailer* }

Ocamllex Input

- *header* and *trailer* contain arbitrary ocaml code put at top an bottom of *<filename>*.ml

- let *ident = regexp* ... Introduces *ident* for use in later regular expressions

Ocamllex Input

- *<filename>*.ml contains one lexing function per *entrypoint*
  - Name of function is name given for *entrypoint*
  - Each entry point becomes an Ocaml function that takes *n*+1 arguments, the extra implicit last argument being of type Lexing.lexbuf
- *arg1*... a*rgn* are for use in *action*

Ocamllex Regular Expression

- Single quoted characters for letters: 'a'
- _: (underscore) matches any letter
- Eof: special "end_of_file" marker
- Concatenation same as usual
- "*string*": concatenation of sequence of characters
- $e_1 \mid e_2$: choice - what was $e_1 \vee e_2$

Ocamllex Regular Expression

- [c_1 - c_2]: choice of any character between first and second inclusive, as determined by character codes
- [^c_1 - c_2]: choice of any character NOT in set
- *e**: same as before
- *e+*: same as *e e**
- *e?*: option - was $e_1 \vee \varepsilon$

Ocamllex Regular Expression

- *e_1 # e_2*: the characters in e_1 but not in e_2; e_1 and e_2 must describe just sets of characters
- *ident*: abbreviation for earlier reg exp in let *ident* = *regexp*
- e_1 as *id*: binds the result of e_1 to *id* to be used in the associated *action*

Ocamllex Manual

- More details can be found at

http://caml.inria.fr/pub/docs/manual-ocaml/manual026.html

Example : test.mll

```
{ type result = Int of int | Float of float |
  String of string }
let digit = ['0'-'9']
let digits = digit +
let lower_case = ['a'-'z']
let upper_case = ['A'-'Z']
let letter = upper_case | lower_case
let letters = letter +
```

Example : test.mll

```
rule main = parse
   (digits)'.'digits as f  { Float (float_of_string f) }
 | digits as n              { Int (int_of_string n) }
 | letters as s             { String s}
 | _ { main lexbuf }
 { let newlexbuf = (Lexing.from_channel stdin) in
 print_string "Ready to lex.";
 print_newline ();
 main newlexbuf  }
```

Example

```
# #use "test.ml";;
...
val main : Lexing.lexbuf -> result = <fun>
val __ocaml_lex_main_rec : Lexing.lexbuf -> int ->
  result = <fun>
Ready to lex.
hi there 234 5.2
- : result = String "hi"
```

What happened to the rest?!?

Example

```
# let b = Lexing.from_channel stdin;;
# main b;;
hi 673 there
- : result = String "hi"
# main b;;
- : result = Int 673
# main b;;
- : result = String "there"
```

Problem

- How to get lexer to look at more than the first token at one time?
- Answer: *action* has to tell it to -- recursive calls
- Side Benefit: can add "state" into lexing
- Note: already used this with the _ case

Example

```
rule main = parse
  (digits) '.' digits as f { Float
  (float_of_string f) :: main lexbuf}
 | digits as n          { Int (int_of_string n) ::
  main lexbuf }
 | letters as s         { String s :: main
  lexbuf}
 | eof                   { [] }
 | _                     { main lexbuf }
```

Example Results

Ready to lex.

hi there 234 5.2

- : result list = [String "hi"; String "there"; Int 234; Float 5.2]

#

Used Ctrl-d to send the end-of-file signal

Dealing with comments

First Attempt

```
let open_comment = "(*"
let close_comment = "*)"
rule main = parse
  (digits) '.' digits as f { Float (float_of_string
  f) :: main lexbuf}
 | digits as n          { Int (int_of_string n) ::
  main lexbuf }
 | letters as s         { String s :: main lexbuf}
```

Dealing with comments

```
 | open_comment        { comment  lexbuf}
 | eof                 { [] }
 | _ { main lexbuf }
and comment = parse
   close_comment       { main lexbuf }
 | _                   { comment lexbuf }
```

Dealing with nested comments

```
rule main = parse ...
 | open_comment          { comment 1 lexbuf}
 | eof                   { [] }
 | _ { main lexbuf }
and comment depth = parse
   open_comment          { comment (depth+1) lexbuf
   }
 | close_comment         { if depth = 1
                            then main lexbuf
                           else comment (depth - 1) lexbuf }
 | _                     { comment depth lexbuf }
```

Dealing with nested comments

rule main = parse

(digits) '.' digits as f { Float (float_of_string f) :: main lexbuf}

| digits as n { Int (int_of_string n) :: main lexbuf }

| letters as s { String s :: main lexbuf}

| open_comment { (comment 1 lexbuf}

| eof { [] }

| _ { main lexbuf }

Dealing with nested comments

```
and comment depth = parse
  open_comment        { comment (depth+1) lexbuf
  }
 | close_comment      { if depth = 1
                        then main lexbuf
                        else comment (depth - 1) lexbuf }
 | _                  { comment depth lexbuf }
```