CS 516: COMPILERS

Lecture 9

Topics

- Lexing
- OCaml Lexer
- Principled Lexing

Materials

- lec09.zip

Announcements

START NOW!!

Announcements

- HW4: LLVM lite
  - **Goal**: "Backend" compiler from LLVMlite —> X86lite
  - **Due**: In 10 days — Thursday, March 2nd at 11:59:59pm
  - **Teams**: Teams of 2. Only one group member needs to submit

START NOW!!

Announcements

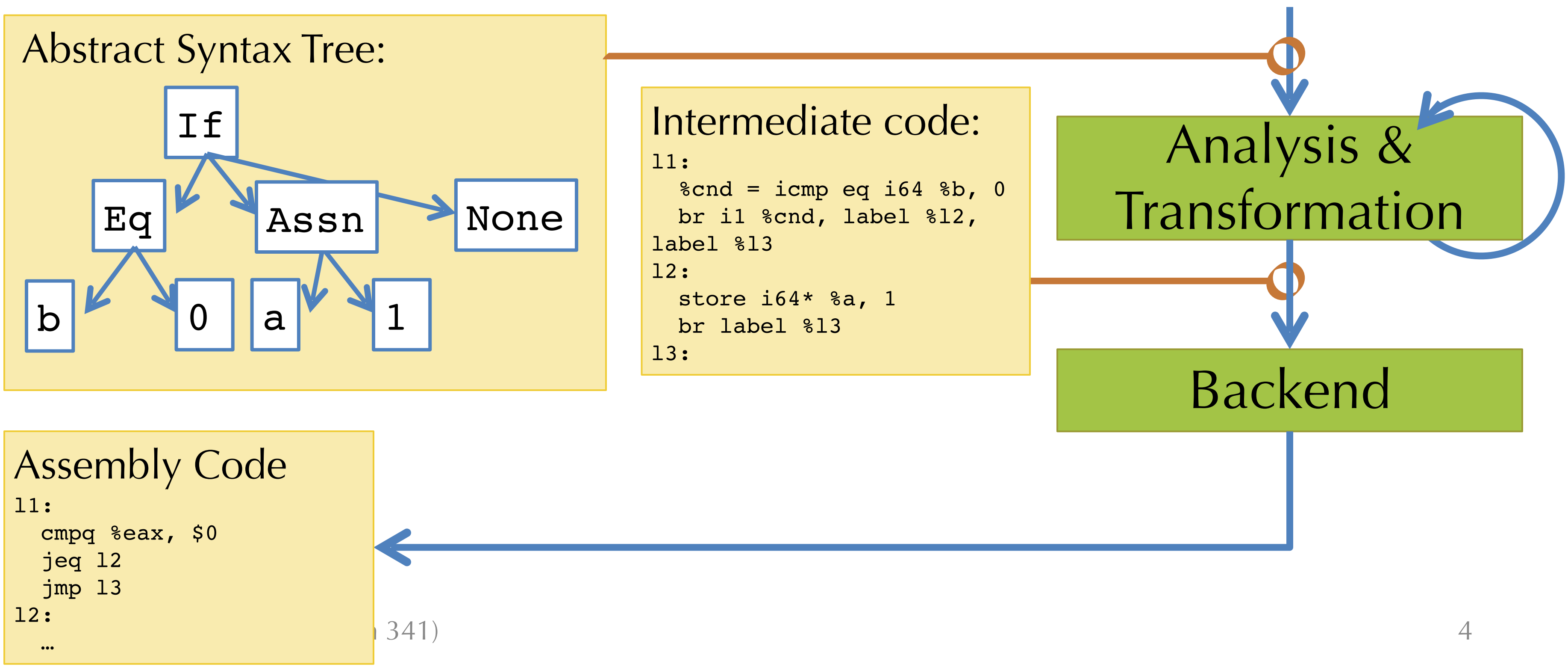
- HW4: LLVM lite
  - **Goal**: "Backend" compiler from LLVMlite —> X86lite
  - **Due**: In 10 days — Thursday, March 2nd at 11:59:59pm
  - **Teams**: Teams of 2. Only one group member needs to submit

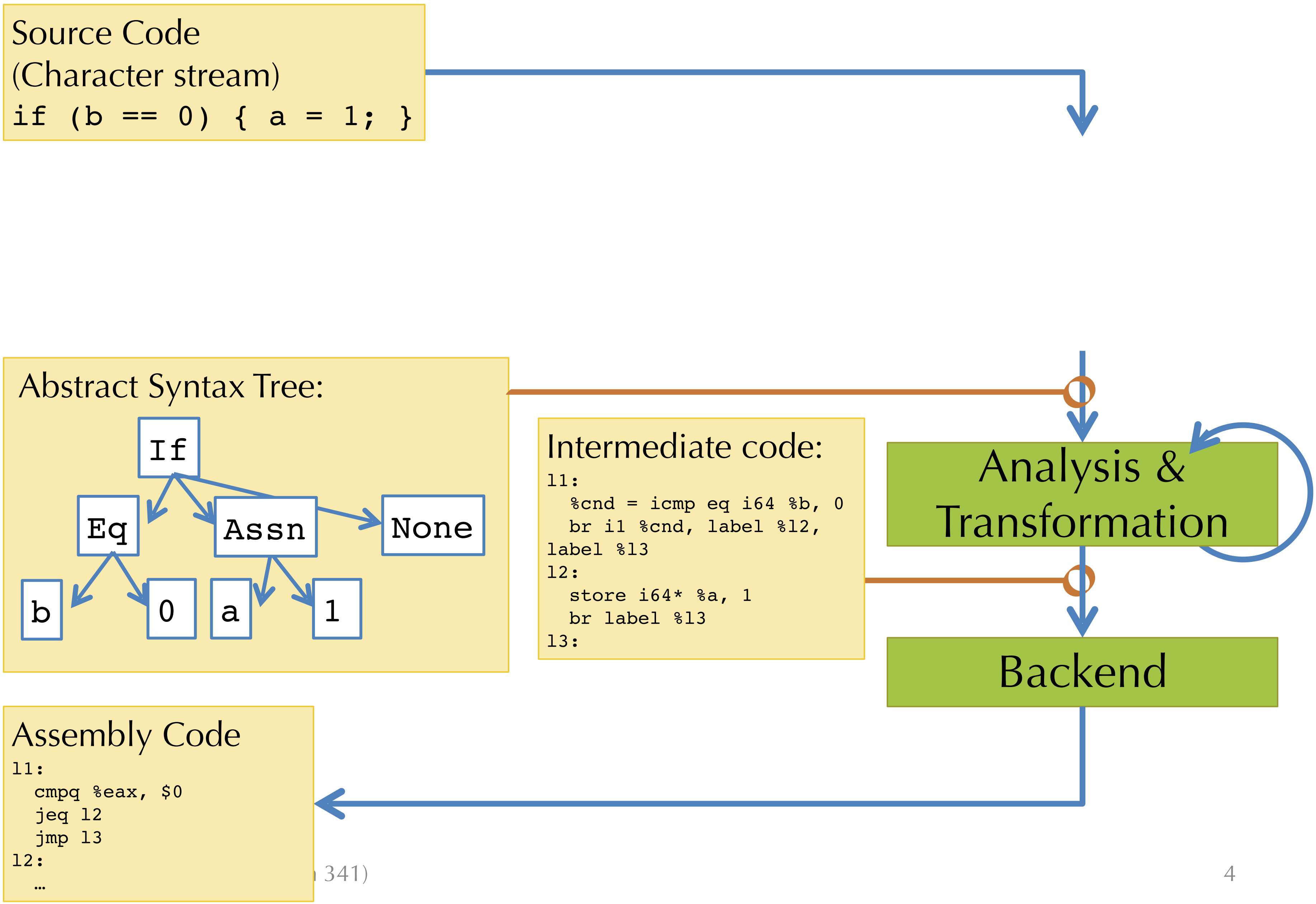
START NOW!!

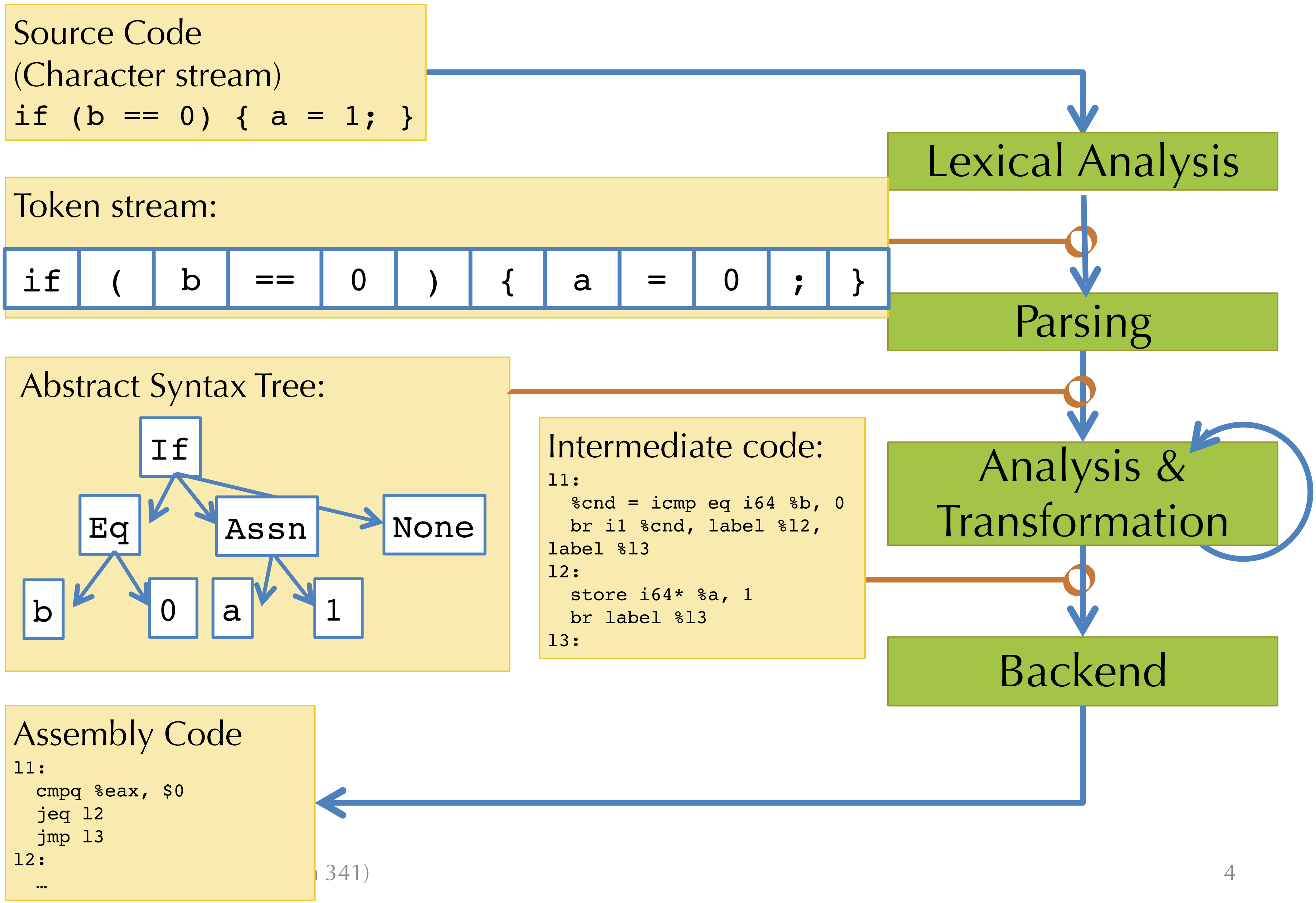
- Next week (Feb 28): Quiz
  - Covers lec05-lec08 (IR, LLVM, Structured Data)
  - Roughly 30 minutes long, closed book.

Lexical analysis, tokens, regular expressions, automata

LEXING


Abstract Syntax Tree:
If
Eq
Assn
None
b
0
a
1
Intermediate code:
l1:
%cnd = icmp eq i64 %b, 0
br i1 %cnd, label %l2,
label %l3
l2:
store i64* %a, 1
br label %l3
l3:
Analysis &
Transformation
Backend
Assembly Code
l1:
cmpq %eax, $0
jeq l2
jmp l3
l2:
…


Source Code
(Character stream)
if (b == 0) { a = 1; }
Abstract Syntax Tree:
If
Eq
Assn
None
b
0
a
1
Intermediate code:
l1:
%cnd = icmp eq i64 %b, 0
br i1 %cnd, label %l2,
label %l3
l2:
store i64* %a, 1
br label %l3
l3:
Analysis &
Transformation
Backend
Assembly Code
l1:
cmpq %eax, $0
jeq l2
jmp l3
l2:
…


Source Code
(Character stream)
if (b == 0) { a = 1; }
Lexical Analysis
Token stream:
if (b == 0) { a = 0 ; }
Parsing
Abstract Syntax Tree:
If
Eq
Assn
None
b
0
a
1
Analysis &
Transformation
Intermediate code:
l1:
%cnd = icmp eq i64 %b, 0
br i1 %cnd, label %l2, label %l3
l2:
store i64* %a, 1
br label %l3
l3:
Backend
Assembly Code
l1:
cmpq %eax, $0
jeq l2
jmp l3
l2:
…

Today: Lexing

Source Code
(Character stream)

```
if (b == 0) { a = 1; }
```

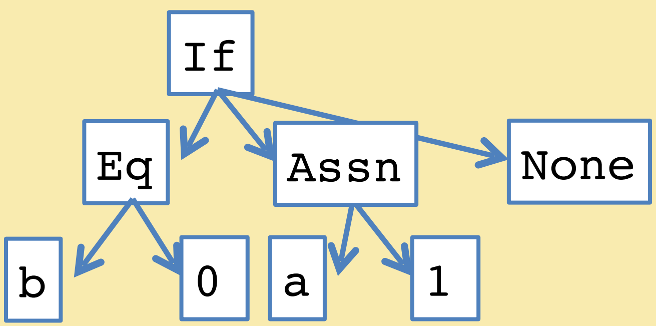
Lexical Analysis

Token stream:

if	(	b	==	0	)	{	a	=	0	;	}

Parsing

Abstract Syntax Tree:



Analysis & Transformation

Intermediate code:

```
l1:
  %cnd = icmp eq i64 %b, 0
  br i1 %cnd, label %12,
label %13
l2:
  store i64* %a, 1
  br label %13
l3:
```

Backend

Assembly Code

```
l1:
  cmpq %eax, $0
  jeq l2
  jmp l3
l2:
  …
```

First Step: Lexical Analysis

`if`	`(`	`b`	`==`	`0`	`)`	`{`	`a`	`=`	`0`	`;`	`}`

First Step: Lexical Analysis

- Change the *character stream* "`if (b == 0) a = 0;`" into *tokens*:

`if`	`(`	`b`	`==`	`0`	`)`	`{`	`a`	`=`	`0`	`;`	`}`

```
IF; LPAREN; Ident("b"); EQEQ; Int(0); RPAREN; LBRACE;
Ident("a"); EQ; Int(0); SEMI; RBRACE
```

First Step: Lexical Analysis

- Change the *character stream* "`if (b == 0) a = 0;`" into *tokens*:

`if`	`(`	`b`	`==`	`0`	`)`	`{`	`a`	`=`	`0`	`;`	`}`

```
IF; LPAREN; Ident("b"); EQEQ; Int(0); RPAREN; LBRACE;
Ident("a"); EQ; Int(0); SEMI; RBRACE
```

- Token: data type that represents indivisible "chunks" of text:
  - Identifiers: `a  y11  elsex  _100`
  - Keywords: `if  else  while`
  - Integers: `2  200  -500  5L`
  - Floating point: `2.0  .02  1e5`
  - Symbols: `+ * `` ` `` { } ( ) ++ << >> >>>`
  - Strings: `"x"  "He said, \"Are you?\""`
  - Comments: `(* CS516: HW 1 … *)  /* foo */  // bar`

First Step: Lexical Analysis

- Change the *character stream* “`if (b == 0) a = 0;`” into *tokens*:

`if`	`(`	`b`	`==`	`0`	`)`	`{`	`a`	`=`	`0`	`;`	`}`

```
IF; LPAREN; Ident(“b”); EQEQ; Int(0); RPAREN; LBRACE;
Ident(“a”); EQ; Int(0); SEMI; RBRACE
```

- Token: data type that represents indivisible “chunks” of text:
  - Identifiers: `a   y11  elsex   _100`
  - Keywords: `if   else   while`
  - Integers: `2   200   -500   5L`
  - Floating point: `2.0   .02   1e5`
  - Symbols: `+   *   ``   {   }   (   )   ++   <<   >>   >>>`
  - Strings: `“x”   “He said, \”Are you?\””`
  - Comments: `(* CS516: HW 1 … *)   /* foo */   // bar`

- Often delimited by *whitespace* (‘ ‘, \t, etc.)
  - In some languages (e.g. Python or Haskell) whitespace is significant

How hard can it be?
handlex0.ml and handlex.ml

DEMO: HANDLEX

Lexing By Hand

- How hard can it be?
  - Tedious and painful!

Lexing By Hand

- How hard can it be?
  - Tedious and painful!

- Problems:
  - Precisely define tokens
  - Matching tokens simultaneously
  - Reading too much input (need look ahead)
  - Error handling
  - Hard to compose/interleave tokenizer code
  - Hard to maintain

PRINCIPLED SOLUTION TO LEXING

Regular Expressions

Regular Expressions

- Regular expressions precisely describe sets of strings.

Regular Expressions

- Regular expressions precisely describe sets of strings.
- A regular expression R has one of the following forms:
  - ε — Epsilon stands for the empty string
  - `'a'` — An ordinary character stands for itself
  - $R_1 \mid R_2$ — Alternatives, stands for choice of R_1 or R_2
  - R_1R_2 — Concatenation, stands for R_1 followed by R_2
  - `R*` — Kleene star, stands for *zero or more* repetitions of `R`

Regular Expressions

- Regular expressions precisely describe sets of strings.
- A regular expression R has one of the following forms:
  - ε — Epsilon stands for the empty string
  - `'a'` — An ordinary character stands for itself
  - $R_1 \mid R_2$ — Alternatives, stands for choice of R_1 or R_2
  - R_1R_2 — Concatenation, stands for R_1 followed by R_2
  - `R*` — Kleene star, stands for *zero or more* repetitions of `R`
- *Useful extensions:*
  - `"foo"` — Strings, equivalent to `'f''o''o'`
  - `R+` — One or more repetitions of `R`, equivalent to `RR*`
  - `R?` — Zero or one occurrences of `R`, equivalent to `(ε|R)`
  - `['a'-'z']` — One of `a` or `b` or `c` or … `z`, equivalent to `(a|b|…|z)`
  - `[^'0'-'9']` — Any character except `0` through `9`
  - `R as x` — Name the string matched by `R` as `x`

Example Regular Expressions

- Recognize the keyword "if": `"if"`
- Recognize a digit: `['0'-'9']`
- Recognize an integer literal: `'-'?['0'-'9']+`
- Recognize an identifier:
  `(['a'-'z']|['A'-'Z'])(['0'-'9']|'_'|['a'-'z']|['A'-'Z'])*`

- In practice, it's useful to be able to *name* regular expressions:

```
let lowercase = ['a'-'z']
let uppercase = ['A'-'Z']
let character = uppercase | lowercase
```

How to Match?

- Consider the input string: `ifx = 0`
  - Could lex as: | `if` | `x` | `=` | `0` | or as: | `ifx` | `=` | `0` |

- Regular expressions alone are ambiguous, need a rule for choosing between the options above
- Most languages choose "longest match"
  - So the 2nd option above will be picked
  - Note that only the first option is "correct" for parsing purposes

- Conflicts: arise due to two tokens whose regular expressions have a shared prefix
  - Ties broken by giving some matches higher priority
  - Example: keywords have priority over identifiers
  - Usually specified by order the rules appear in the lex input file

Lexer Generators

- Reads a list of regular expressions: `R1,…,Rn` , one per token.
- Each token has an attached "action" `Ai` (just a piece of code to run when the regular expression is matched):

```
rule token = parse
| '-'?digit+                 { Int (Int32.of_string (lexeme lexbuf)) }
| '+'                                  { PLUS }
| 'if'                                 { IF }
| character (digit|character|'_')*     { Ident (lexeme lexbuf) }
| whitespace+                          { token lexbuf }
```

token
regular expressions

actions

- Generates scanning code that:
  1. Decides whether the input is of the form `(R1|…|Rn)*`
  2. Whenever the scanner matches a (longest) token, it runs the associated action

lexlex.mll

DEMO: OCAMLLEX

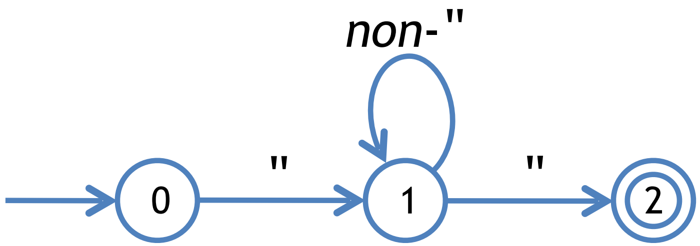
Implementation Strategies

- Most Tools: lex, ocamllex, flex, etc.:
  - Table-based
  - Deterministic Finite Automata (DFA)
  - Goal: Efficient, compact representation, high performance

- There are other approaches (eg based on Brzozowski derivatives)

Finite Automata

- Consider the regular expression: '"'[^'"']*'"'
- An automaton (DFA) can be represented as:
  - A graph:



  - A transition table:

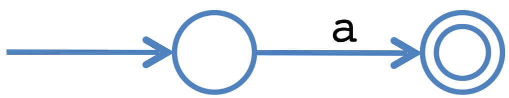
	"	Non-"
0	1	ERROR
1	2	1
2	ERROR	ERROR

RE to Finite Automaton?

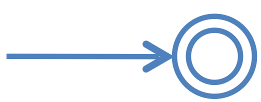
- Can we build a finite automaton for every regular expression?
  - Yes! Recall CS 334 for the complete theory…

- Strategy: consider every possible regular expression (by induction on the structure of the regular expressions):

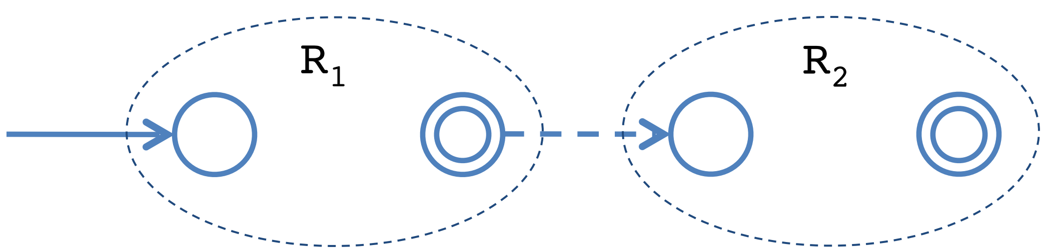
`'a'`



ε



R_1R_2

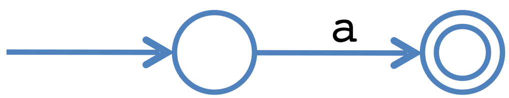


RE to Finite Automaton?

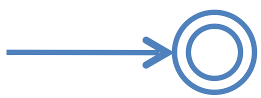
- Can we build a finite automaton for every regular expression?
  - Yes! Recall CS 334 for the complete theory…

- Strategy: consider every possible regular expression (by induction on the structure of the regular expressions):

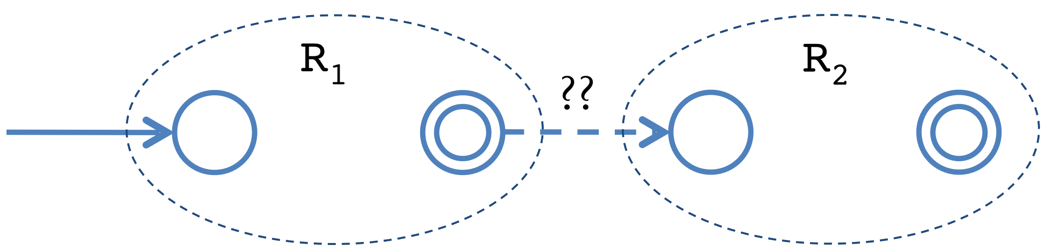
`'a'`



ε



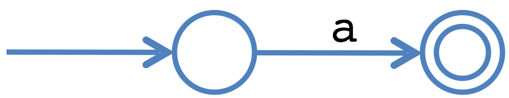
R_1R_2



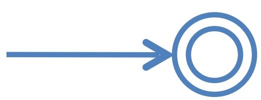
RE to Finite Automaton?

- Can we build a finite automaton for every regular expression?
  - Yes! Recall CS 334 for the complete theory…
- Strategy: consider every possible regular expression (by induction on the structure of the regular expressions):

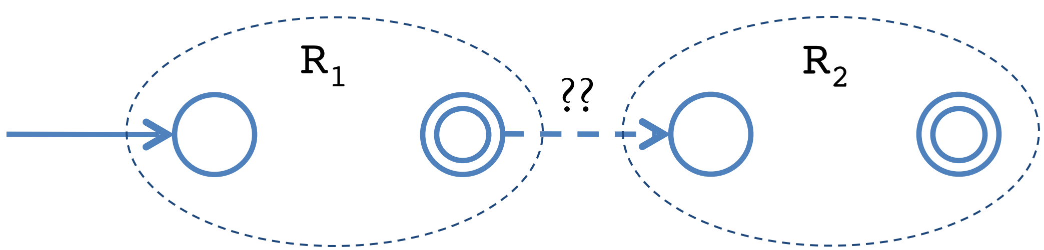
'a'



ε



R_1R_2

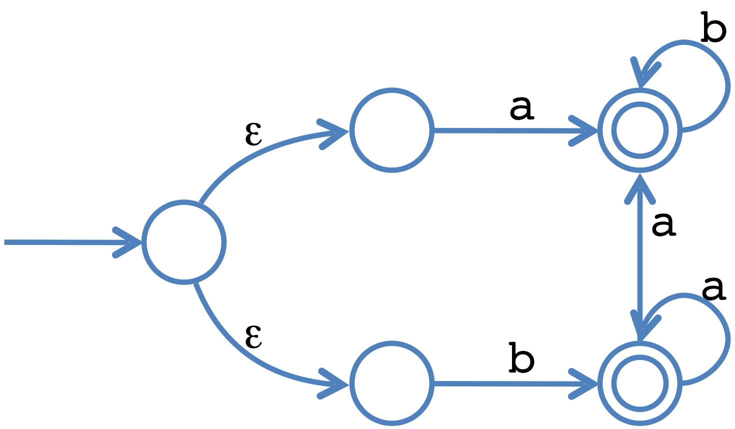


What about?

$R_1 | R_2$

Nondeterministic Finite Automata

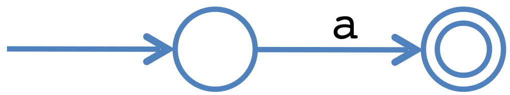
- A finite set of states, a start state, and accepting state(s)
- Transition arrows connecting states
  - Labeled by input symbols
  - Or ε (which does not consume input)
- *Nondeterministic*: two arrows leaving the same state may have the same label



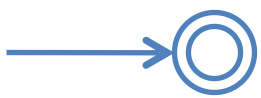
RE to NFA?

- Converting regular expressions to NFAs is easy.
- Assume each NFA has one start state, unique accept state

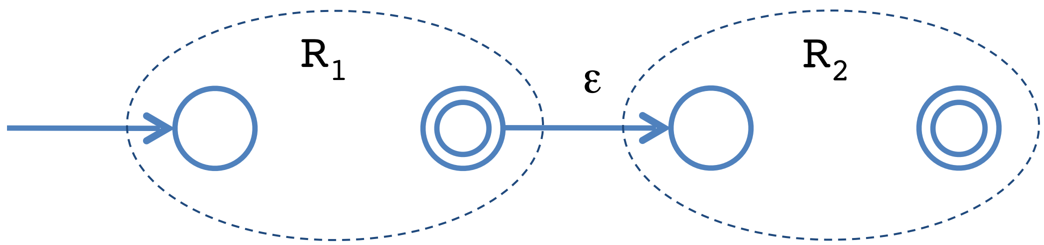
'a'



ε



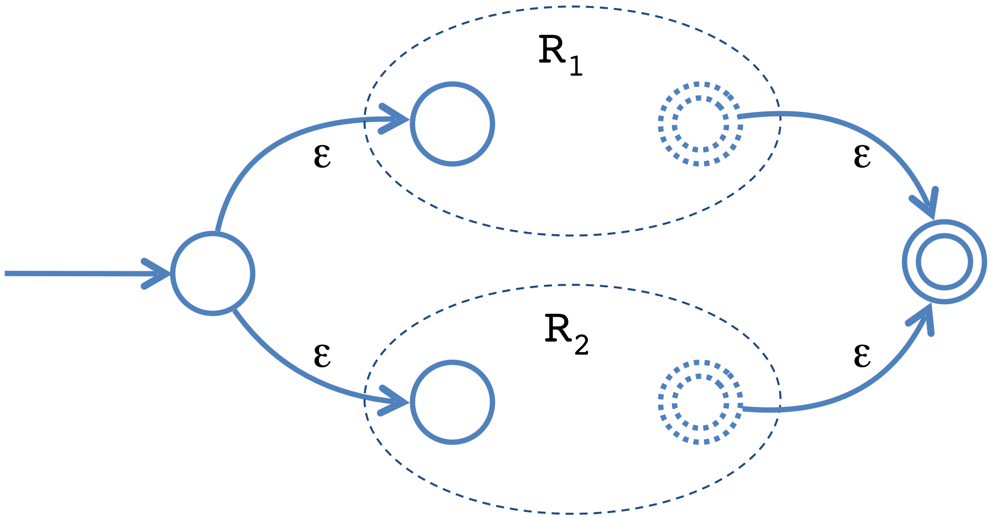
R_1R_2



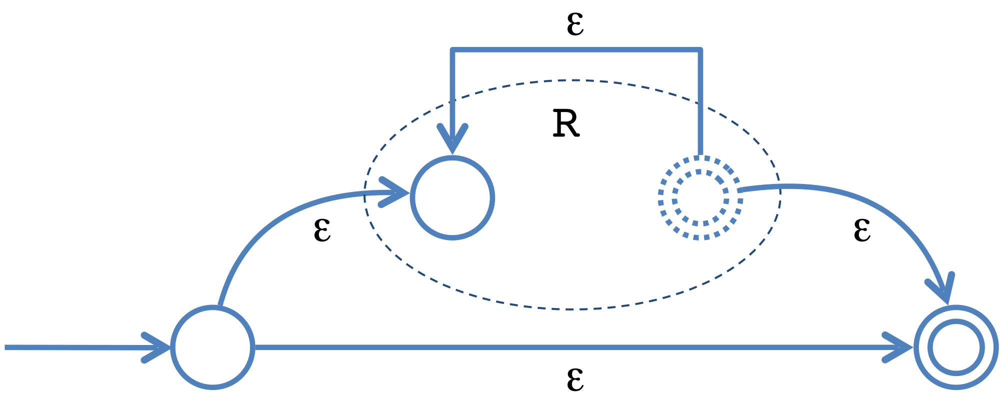
RE to NFA (cont'd)

- Sums and Kleene star are easy with NFAs

$R_1 | R_2$



R^*



DFA versus NFA

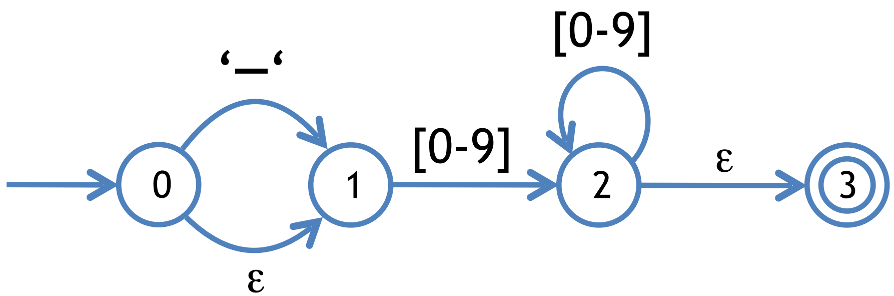
- DFA:
  - Action of the automaton for each input is fully determined
  - Automaton accepts if the input is consumed upon reaching an accepting state
  - Obvious table-based implementation

- NFA:
  - Automaton potentially has a choice at every step
  - Automaton accepts an input string if there ***exists*** a way to reach an accepting state
  - Less obvious how to implement efficiently

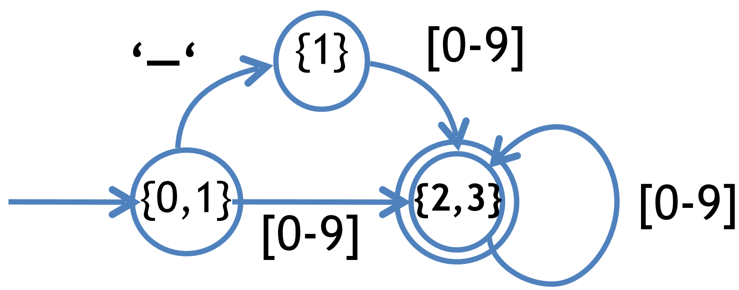
NFA to DFA conversion (Intuition)

- Idea: Run all possible executions of the NFA "in parallel"
- Keep track of a set of possible states: "finite fingers"
- Consider: `-?[0-9]+`

- NFA representation:



- DFA representation:



Summary of Lexer Generator Behavior

- Take each regular expression $\mathtt{R_i}$ and it's action $\mathtt{A_i}$
- Compute the **NFA** formed by $\mathtt{(R_1 \mid R_2 \mid \ldots \mid R_n)}$
  - Remember the actions associated with the accepting states of the $\mathtt{R_i}$
- Compute the **DFA** for this big NFA
  - There may be multiple accept states (why?)
  - A single accept state may correspond to one or more actions (why?)
- Compute the minimal equivalent DFA
  - There is a standard algorithm due to Myhill & Nerode
- Produce the transition table
- Implement longest match:
  - Start from initial state
  - Follow transitions, remember last accept state entered (if any)
  - Accept input until no transition is possible (i.e. next state is "ERROR")
  - Perform the highest-priority action associated with the last accept state; if no accept state there is a lexing error

Lexer Generators in Practice

- Many existing implementations: lex, Flex, Jlex, ocamllex, …
  - For example ocamllex program
    - see lexlex.mll, olex.mll, piglatin.mll on course website
- Error reporting:
  - Associate line number/character position with tokens
  - Use a rule to recognize '\n' and increment the line number
  - The lexer generator itself usually provides character position info.
- Sometimes useful to treat comments specially
  - Nested comments: keep track of nesting depth

- Lexer generators are usually designed to work closely with parser generators…

lexlex.mll, olex.mll, piglatin.mll

DEMO: OCAMLLEX