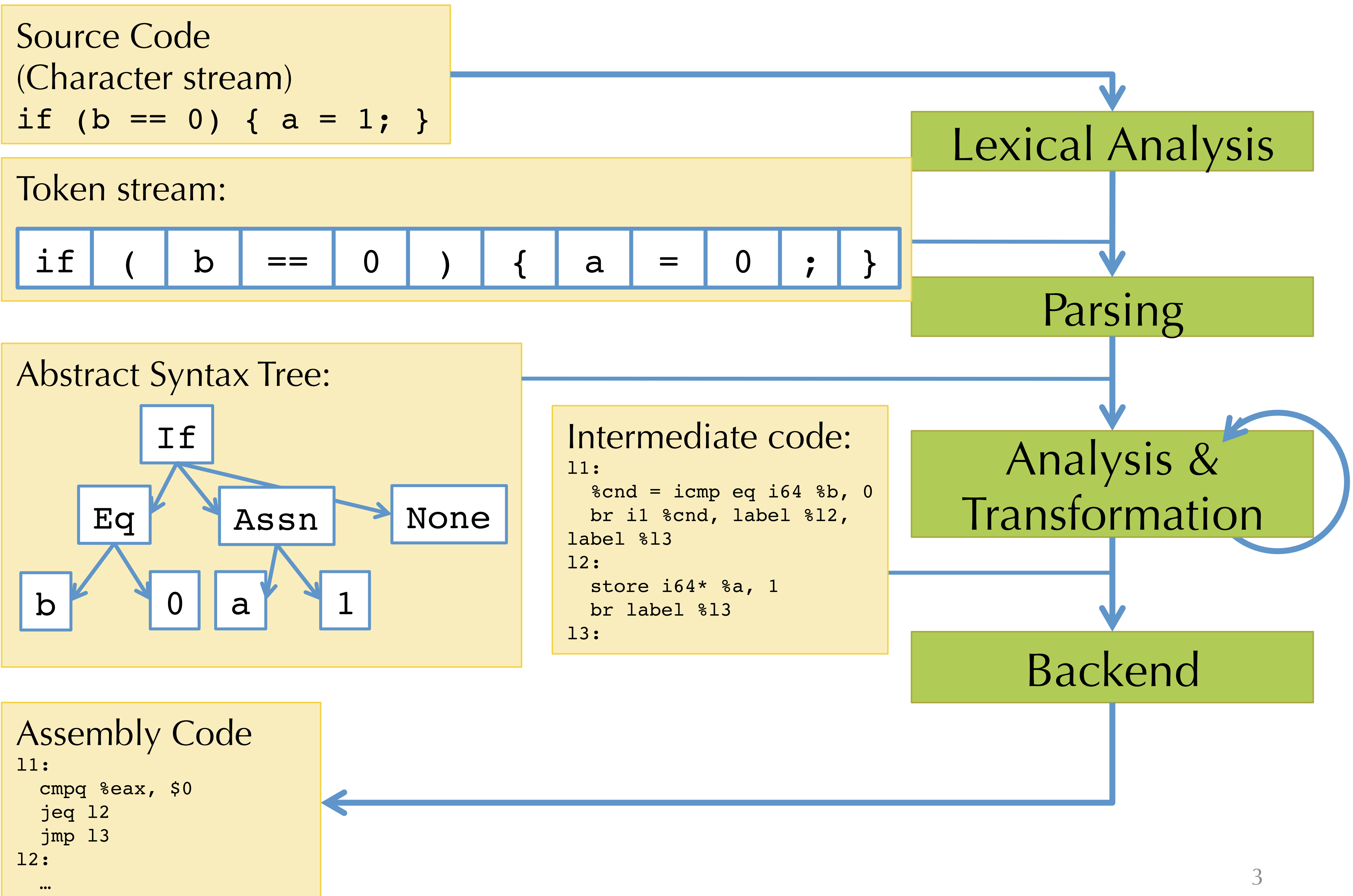
Lecture 9

CIS 341: COMPILERS

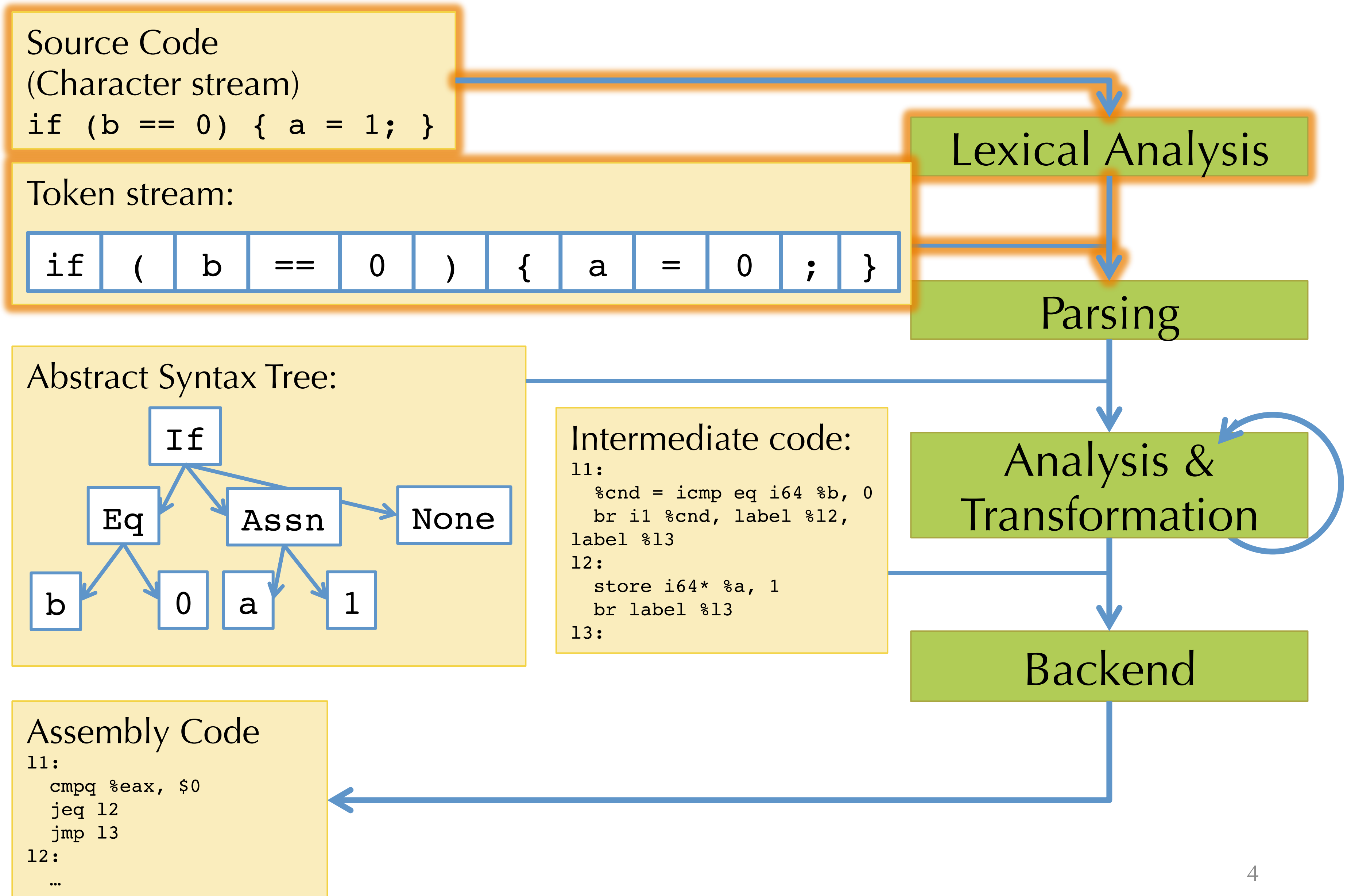
Lexical analysis, tokens, regular expressions, automata

LEXING

Compilation in a Nutshell



Today: Lexing



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: `(* CIS341: Project 1 … *)   /* foo */`

- Often delimited by *whitespace* (' ', \t, etc.)
  - In some languages (e.g. Python or Haskell) whitespace is significant

How hard can it be?
handlex.ml

DEMO: HANDLEX

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

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 regular expressions with non-empty intersection
  - 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: $R_1, \ldots, R_n$, one per token.
- Each token has an attached "action" A_i (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 }
```

- Generates scanning code that:
  1. Decides whether the input is of the form $(R_1|\ldots|R_n)*$
  2. Whenever the scanner matches a (longest) token, it runs the associated action

olex.mll

DEMO: OCAMLLEX

Implementation Strategies

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

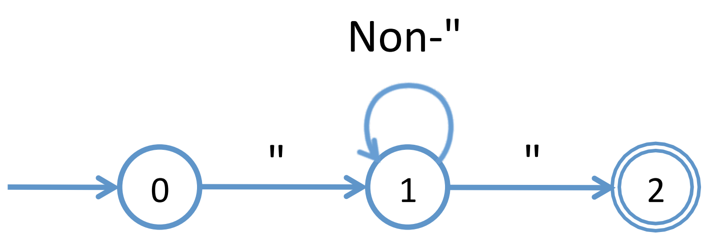
- Other approaches:
  - Brzozowski derivatives
  - Idea: directly manipulate the (abstract syntax of) the regular expression
  - Compute partial "derivatives"
    - Regular expression that is "left-over" after seeing the next character
  - Elegant, purely functional, implementation
  - (very cool!)

Finite Automata

- Consider the regular expression: `‘"’[^’"’]*’"’`
- An automaton (DFA) can be represented as:
  - A transition table:

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

  - A graph:

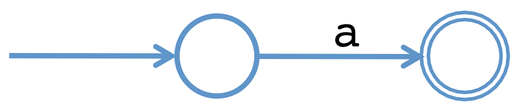


RE to Finite Automaton?

- Can we build a finite automaton for every regular expression?
  – Yes! Recall CIS 262 for the complete theory…

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

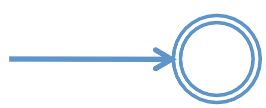
'a'



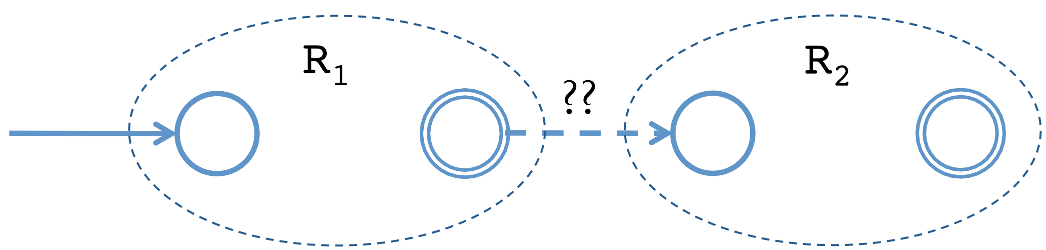
What about?

$R_1 | R_2$

ε

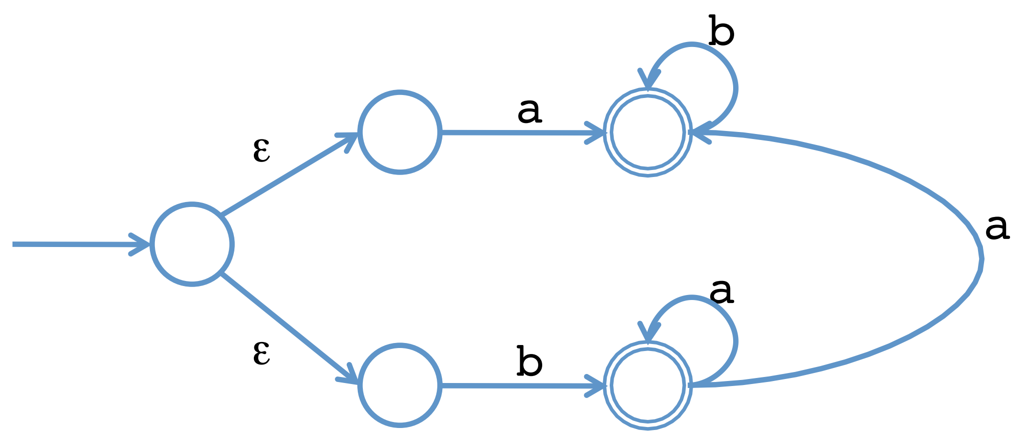


R_1R_2



Nondeterministic Finite Automata

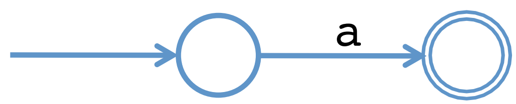
- A finite set of states, a start state, and accepting state(s)
- Transition arrows connecting states
  - Labeled by input symbols
  - Or ε (which does not consume input)
- *Nondeterministic*: two arrows leaving the same state may have the same label



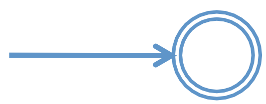
RE to NFA?

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

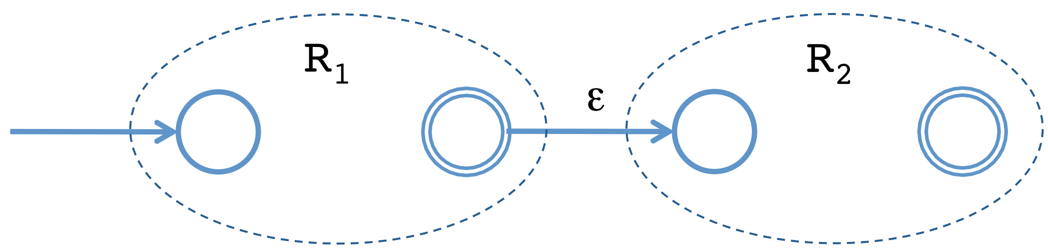
'a'



ε



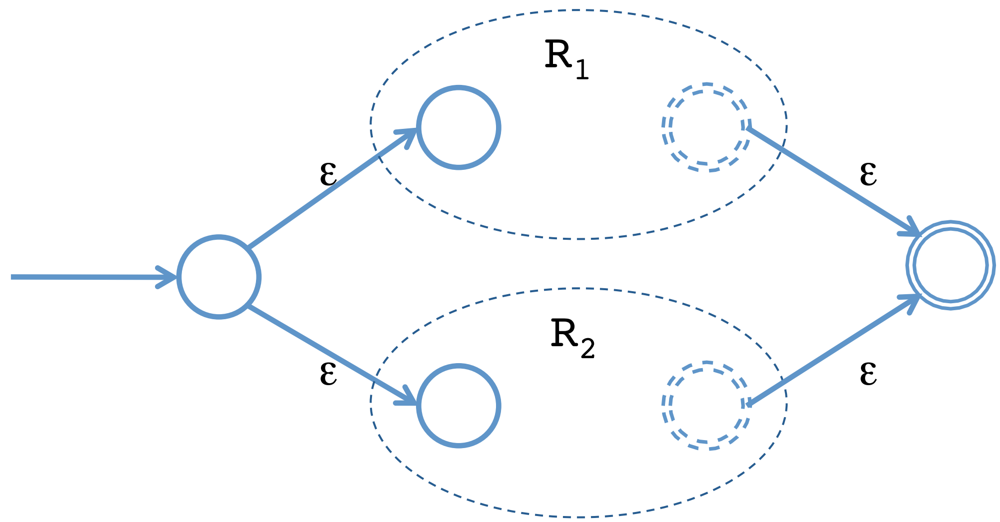
R_1R_2



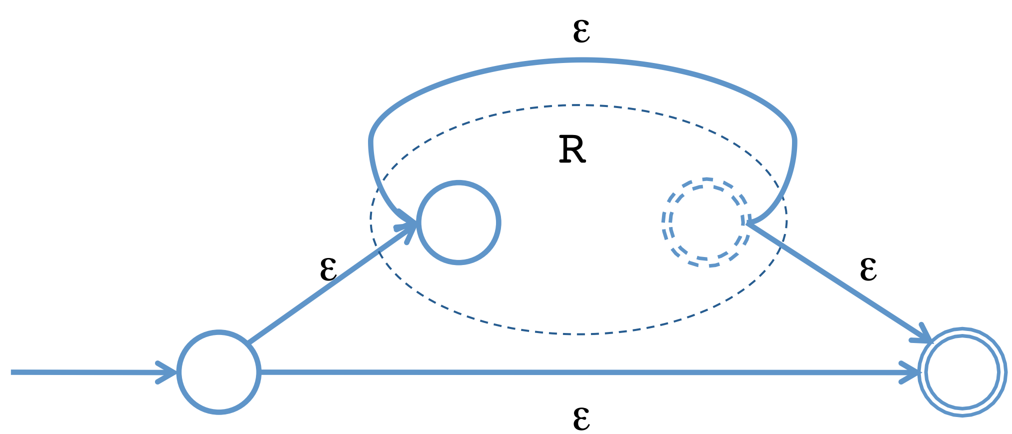
RE to NFA (cont'd)

- Sums and Kleene star are easy with NFAs

$R_1 \mid R_2$



R*



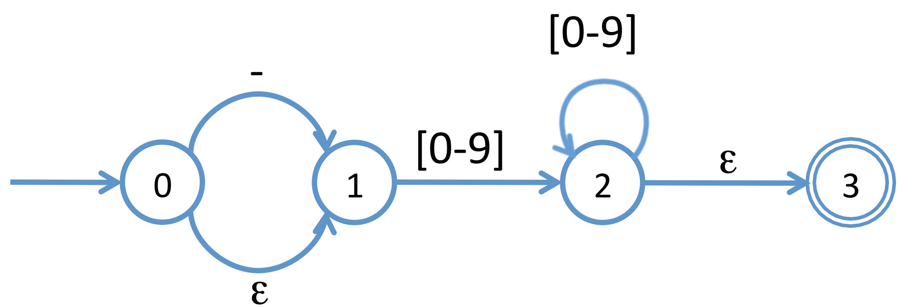
DFA versus NFA

- 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

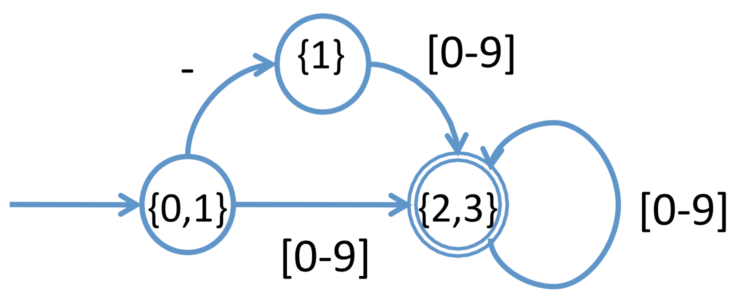
- NFA:
  - Automaton potentially has a choice at every step
  - Automaton accepts an input string if there *exists* a way to reach an accepting state
  - Less obvious how to implement efficiently

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 R_i and it's action A_i
- Compute the NFA formed by $(R_1 \mid R_2 \mid \ldots \mid R_n)$
  - Remember the actions associated with the accepting states of the 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

CORRECTNESS?

Correct Execution?

- What does it mean for an Imp program to be executed correctly?

- Even at the interpreter level we could show *equivalence* between the small-step and the large-step operational semantics:

$$\text{cmd} \,/\, \text{st} \longmapsto^{*} \texttt{SKIP} \,/\, \text{st}'$$

iff

$$\text{cmd} \,/\, \text{st} \Downarrow \text{st}'$$

Compiler Correctness?

- We have to relate the source and target language semantics across the compilation function $\mathfrak{C}$[-] : source → target.

$$\text{cmd} / \text{st} \;\; {}_S\!\longmapsto^{*} \;\; \texttt{SKIP} / \text{st}'$$

$$\text{iff}$$

$$\mathfrak{C}[\text{cmd}] / \mathfrak{C}[\text{st}] \;\; {}_T\!\longmapsto^{*} \;\; \mathfrak{C}[\text{st}']$$

- Is this enough?
- What if cmd goes into an infinite loop?

Comparing Behaviors

- Consider two programs P and P′ possibly in different languages.
  – e.g. P is an LLVMlite program, P′ is its compilation to x86
- The semantics of the languages associate to each program a set of observable behaviors:

$$\mathfrak{B}(P) \text{ and } \mathfrak{B}(P')$$

- Note: $|\mathfrak{B}(P)| = 1$ if P is deterministic, > 1 otherwise

What is Observable?

- For Imp-like languages:

```
observable behavior ::=
      | terminates(st)        (i.e. observe the final state)
      | diverges
      | goeswrong
```

- For pure functional languages:

```
observable behavior ::=
      | terminates(v)         (i.e. observe the final value)
      | diverges
      | goeswrong
```

What about I/O?

- Add a *trace* of input-output events performed:

	t	::=	[]	|	e :: t	(finite traces)
coind.	T	::=	[]	|	e :: T	(finite and infinite traces)

observable behavior ::=
  | terminates(t, st) (end in state st after trace t)
  | diverges(T) (loop, producing trace T)
  | goeswrong(t)

Examples

- P1:
  `print(1);` / st $\Rightarrow$ terminates(out(1)::[],st)

- P2:
  `print(1); print(2);` / st
  $\Rightarrow$ terminates(out(1)::out(2)::[],st)

- P3:
  `WHILE true DO print(1) END` / st
  $\Rightarrow$ diverges(out(1)::out(1)::…)

- So $\mathfrak{B}(\mathrm{P1}) \neq \mathfrak{B}(\mathrm{P2}) \neq \mathfrak{B}(\mathrm{P3})$

Bisimulation

- Two programs P1 and P2 are bisimilar whenever:

$$\mathfrak{B}(P1) = \mathfrak{B}(P2)$$

- The two programs are completely indistinguishable.

- But… this is often too strong in practice.

Compilation Reduces Nondeterminism

- Some languages (like C) have underspecified behaviors:
  - Example: order of evaluation of expressions   f() + g()

- Concurrent programs often permit nondetermism
  - Classic optimizations can reduce this nondterminism
  - Example:

    a := x + 1; b := x + 1 || x := x+1

    vs.

    a := x + 1; b := a || x := x+1

Backward Simulation

- Program P2 can exhibit fewer behaviors than P1:

$$\mathfrak{B}(P1) \;\supseteq\; \mathfrak{B}(P2)$$

- All of the behaviors of P2 are permitted by P1, though some of them may have been eliminated.
- Also called *refinement*.

What about goeswrong?

- Compilers often translate away bad behaviors.

x := 1/y ; x := 42 (divide by 0 error)	vs.	x := 42 (always terminates)

- Justifications:
  - Compiled program does not "go wrong" because the program type checks or is otherwise formally verified
  - Or just "garbage in/garbage out"

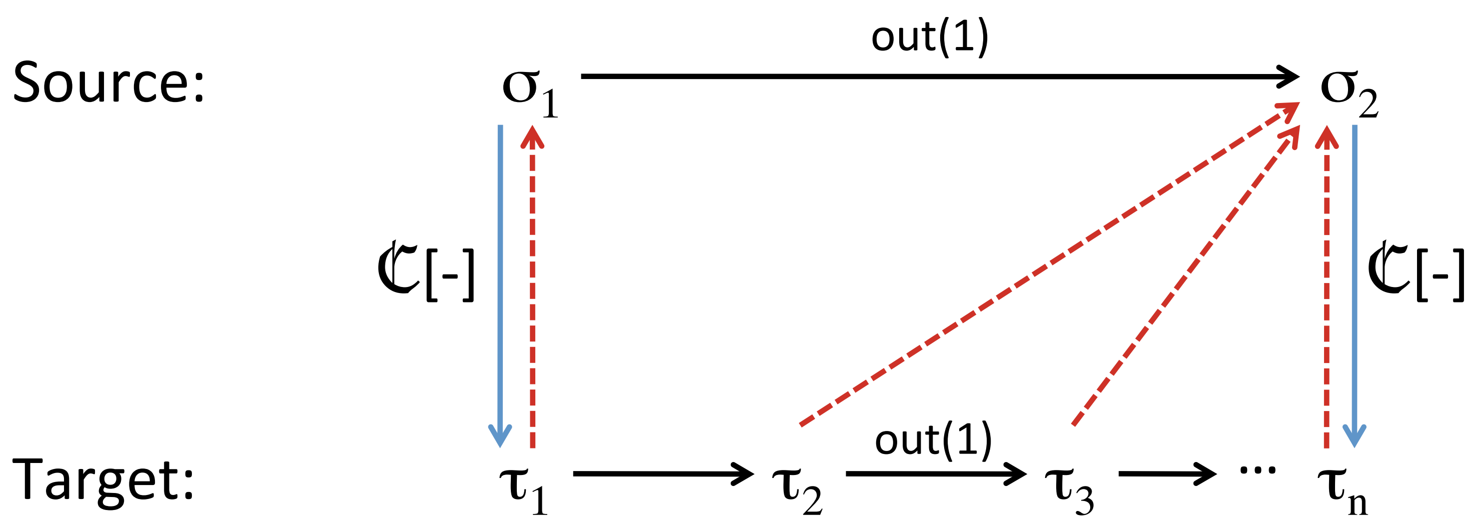
Safe Backwards Simulation

- Only require the compiled program's behaviors to agree if the source program could not go wrong:

$$\text{goeswrong}(t) \notin \mathfrak{B}(P1) \quad \Rightarrow \quad \mathfrak{B}(P1) \supseteq \mathfrak{B}(P2)$$

- Idea: let S be the functional specification of the program: A set of behaviors not containing goeswrong(t).
  - A program P satsifies the spec if $\mathfrak{B}(P) \subseteq S$
- Lemma: If P2 is a safe backwards simulation of P1 and P1 satisfies the spec, then P2 does too.

Building Backward Simulations



Idea: The event trace along a (target) sequence of steps originating from a compiled program must correspond to some source sequence.

Tricky parts:

- Must consider all possible target steps
- If the compiler uses many target steps for once source step, we have invent some way of relating the intermediate states to the source.
- the compilation function goes the wrong way to help!