Midterm Review

EECS 483: COMPILER CONSTRUCTION

Announcements

- Midterm: Tomorrow
  - 7-9pm,
    - If your last name is A-H: DOW 1014
    - Last name J-Z: DOW 1013
  - One-page, letter-sized, double-sided “cheat sheet” of notes permitted
  - See examples of previous exams on the web pages.

- HW4: Compiling Oat v.1
  - Out now

Exam Review

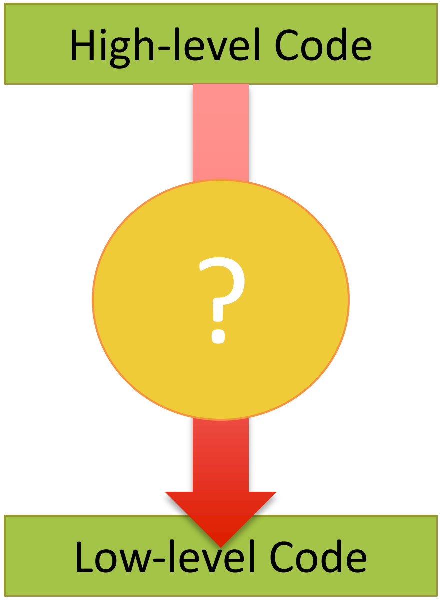
- Today: review of the topics thus far
  - Caveat: everything covered in lecture and homeworks is fair game, even if we don't specifically review it today.
  - I will only use slides from previous lectures to ensure there's no new material covered today.
- Outline
  - Language Semantics
  - Compiler Architecture
  - X86 Assembly code
  - LLVM IR
  - Regular Expressions, Finite Automata, Lexing
  - LL/LR Grammars and Parsing

Lectures 1-3, HW1

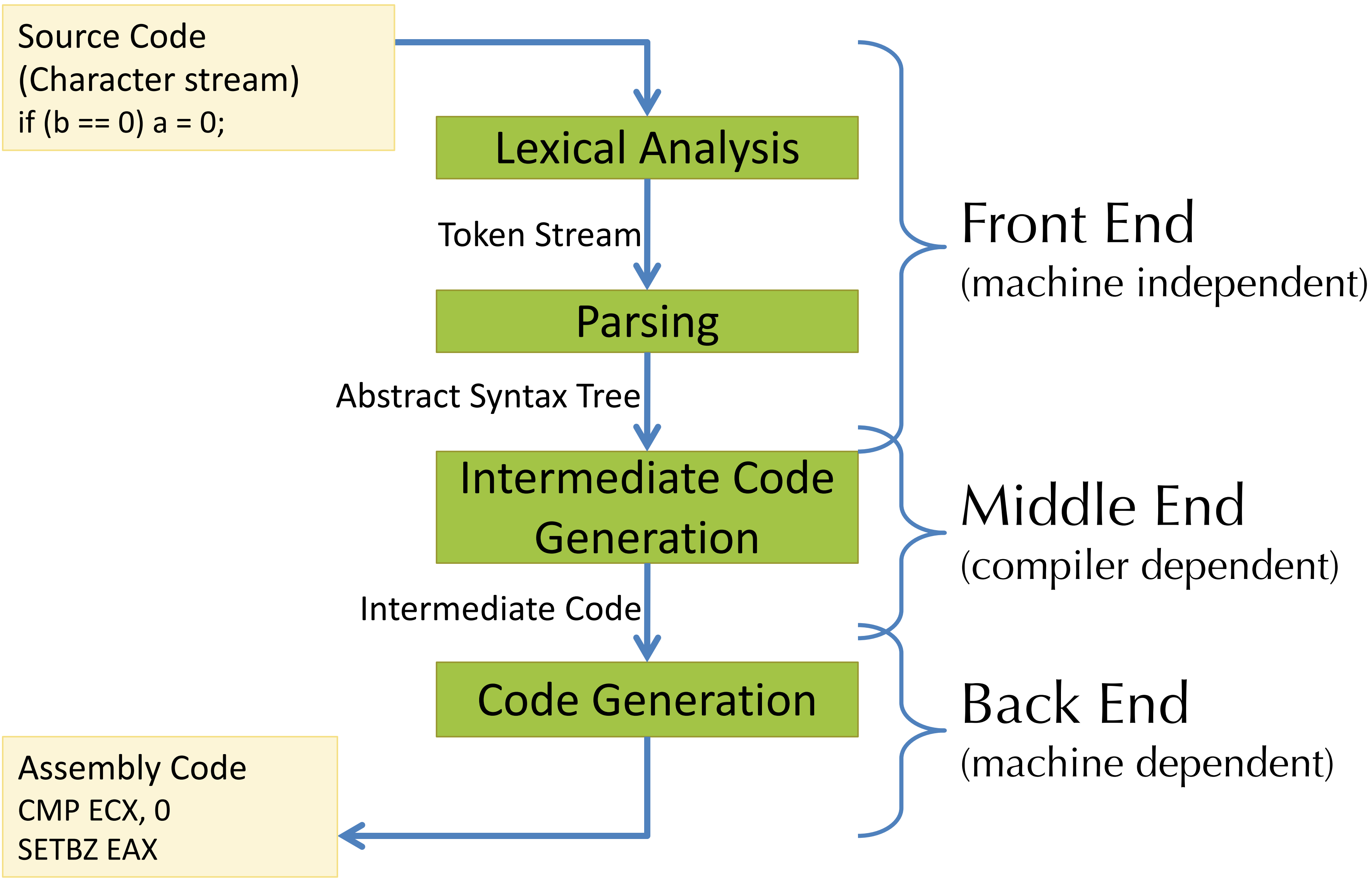
LANGUAGE SEMANTICS, COMPILER ARCHITECTURE

What is a Compiler?

- A compiler is a program that translates from one programming language to another.
- Typically: *high-level source code* to *low-level machine code* (object code)
  - Not always: Source-to-source translators, Java bytecode compiler, GWT Java ⇒ Javascript



(Simplified) Compiler Structure



OCaml Demo

simple.ml
translate.ml

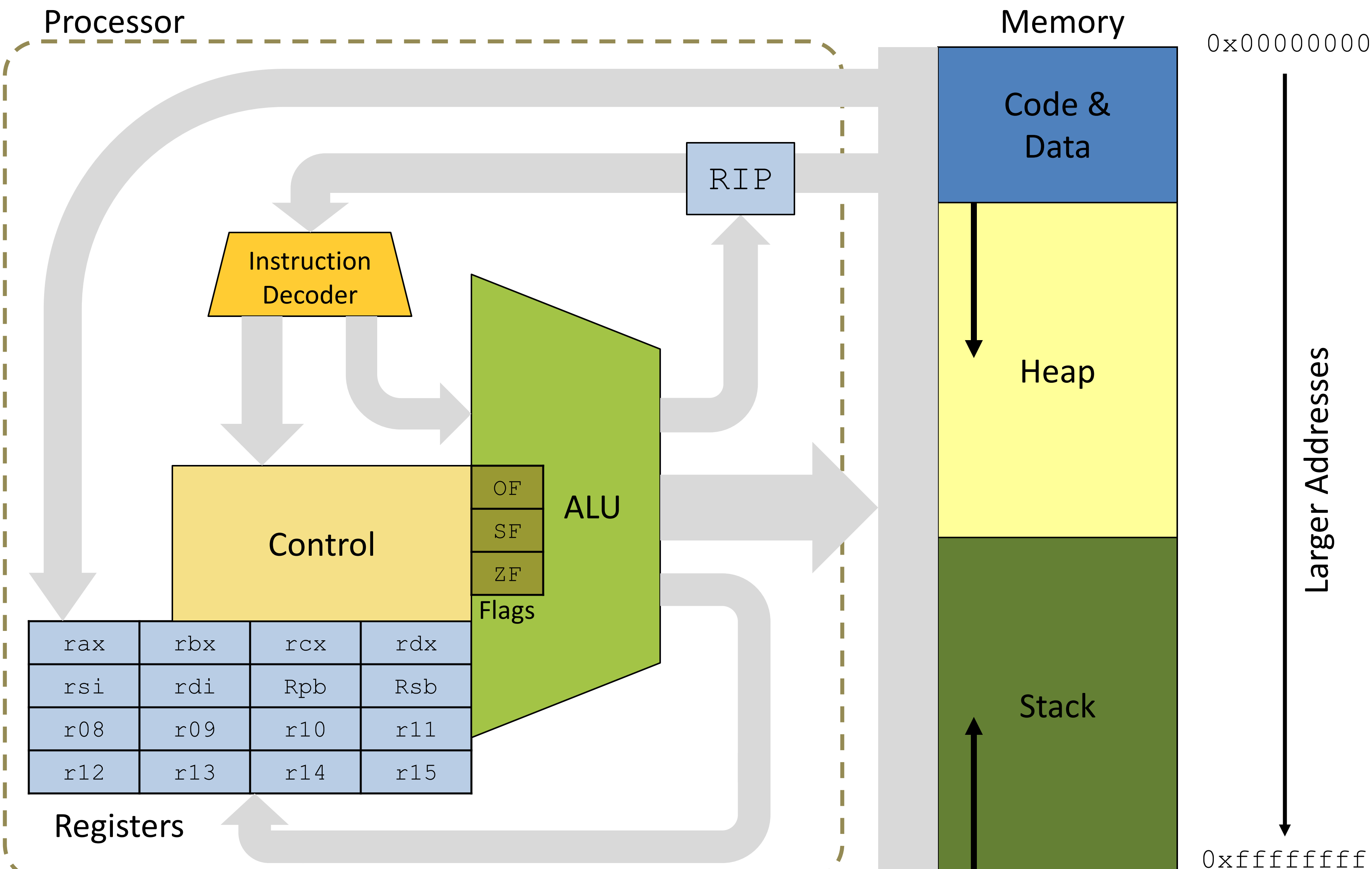
Correctness?

- What does it mean for a compiler to be correct?
- What constitutes the "observable behavior" of a program?
- How do these notions affect what program transformations are allowed?

Lectures 3-5, HW2

X86 AND ASSEMBLY CODE PROGRAMMING

X86 Schematic



X86lite Memory Model

- The X86lite memory consists of 2^{64} bytes numbered `0x00000000` through `0xffffffff`.
- X86lite treats the memory as consisting of 64-bit (8-byte) quadwords.
- Therefore: legal X86lite memory addresses consist of 64-bit, quadword-aligned pointers.
  - All memory addresses are evenly divisible by 8

- `leaq` Ind, DEST   DEST ← addr(Ind)   loads a pointer into DEST

- By convention, there is a stack that grows from high addresses to low addresses
- The register `rsp` points to the top of the stack
  - `pushq` SRC   `rsp` ← `rps` - 8; Mem[`rsp`] ← SRC
  - `popq` DEST   DEST ← Mem[`rsp`]; `rsp` ← `rsp` + 8

Code Blocks & Labels

- X86 assembly code is organized into *labeled blocks*:

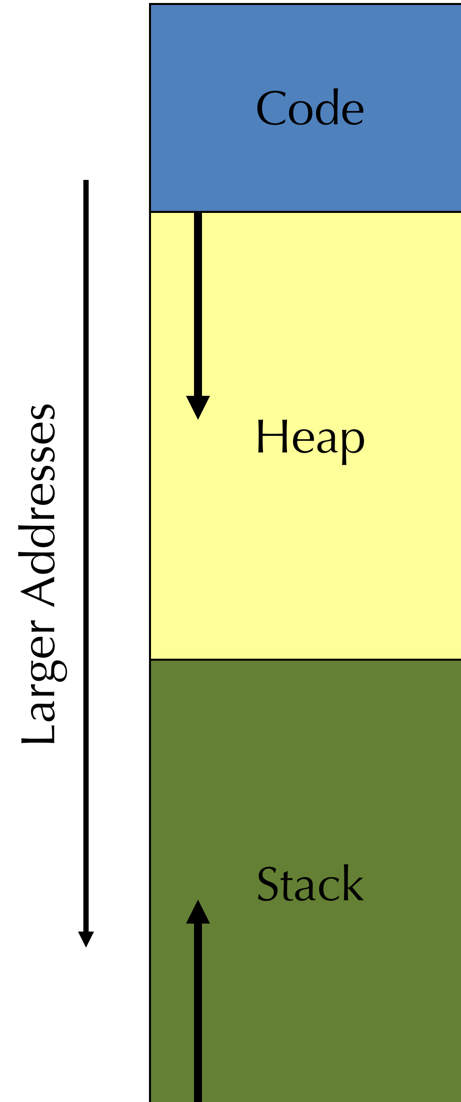
```
label1:
        <instruction>
        <instruction>
        …
        <instruction>

label2:
        <instruction>
        <instruction>
        …
        <instruction>
```

- Labels indicate code locations that can be jump targets (either through conditional branch instructions or function calls).
- Labels are translated away by the linker and loader – instructions live in the heap in the "code segment"
- An X86 program begins executing at a designated code label (usually "main").

3 parts of the C memory model

- The code & data (or "text") segment
  - contains compiled code, constant strings, etc.
- The Heap
  - Stores dynamically allocated objects
  - Allocated via "malloc"
  - Deallocated via "free"
  - C runtime system
- The Stack
  - Stores local variables
  - Stores the return address of a function

- In practice, most languages use this model.



Local/Temporary Variable Storage

- Need space to store:
  - Global variables
  - Values passed as arguments to procedures
  - Local variables (either defined in the source program or introduced by the compiler)

- Processors provide two options
  - Registers: fast, small size (64 bits), very limited number
  - Memory: slow, very large amount of space (2 GB)
    - caching important

- In practice on X86:
  - Registers are limited (and have restrictions)
  - Divide memory into regions including the *stack* and the *heap*

Calling Conventions

- Specify the locations (*e.g.*, register or stack) of arguments passed to a function and returned by the function

```
int64_t g(int64_t a, int64_t b) {
        return a + b;
}

int64_t f(int64_t x) {
   int64_t ans = g(3,4) + x;
   return ans;
}
```

f is the *caller*

g is the *callee*

- Designate registers either:
  - Caller Save – *e.g.*, freely usable by the called code
  - Callee Save – *e.g.*, must be restored by the called code
- Define the protocol for deallocating stack-allocated arguments
  - Caller cleans up
  - Callee cleans up (makes variable arguments harder)

X86-64 SYSTEM V AMD 64 ABI

- More modern variant of C calling conventions
  - used on Linux, Solaris, BSD, OS X

- Callee save: %rbp, %rbx, %r12-%r15
- Caller save: all others

- Parameters 1 .. 6 go in: %rdi, %rsi, %rdx, %rcx, %r8, %r9
- Parameters 7+ go on the stack (in right-to-left order)
  - so: for n > 6, the n^{th} argument is located at (((n-7)+2)*8)(%rbp)
  - e.g.: argument 7 is at 16(%rbp) and argument 8 is at 24(%rbp)

- Return value: in %rax

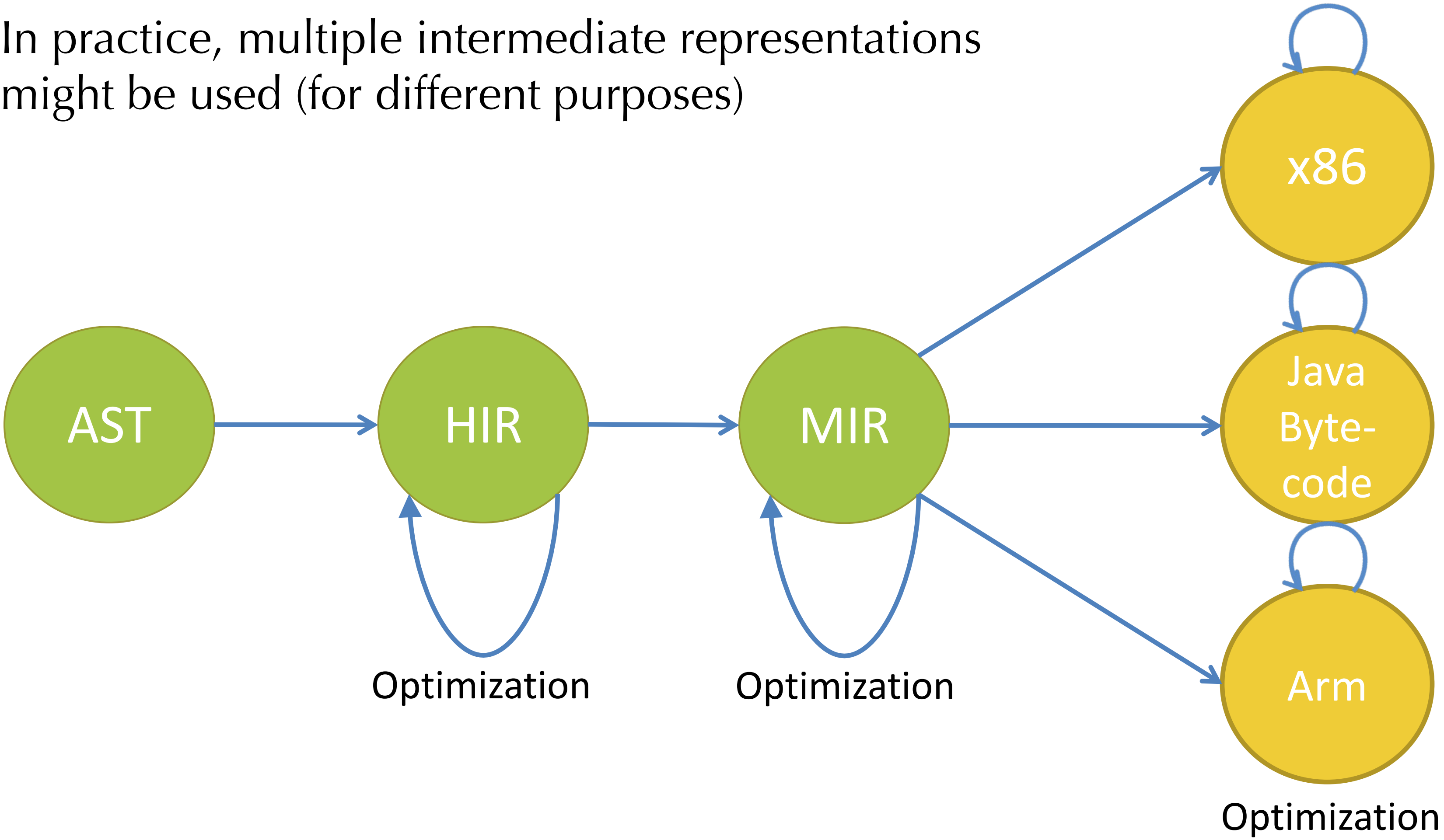
- 128 byte "red zone" – scratch pad for the callee's data
  - typical of C compilers, not required
  - can be optimized away

Lectures 6-9, HW3

INTERMEDIATE REPRESENTATIONS, LLVM

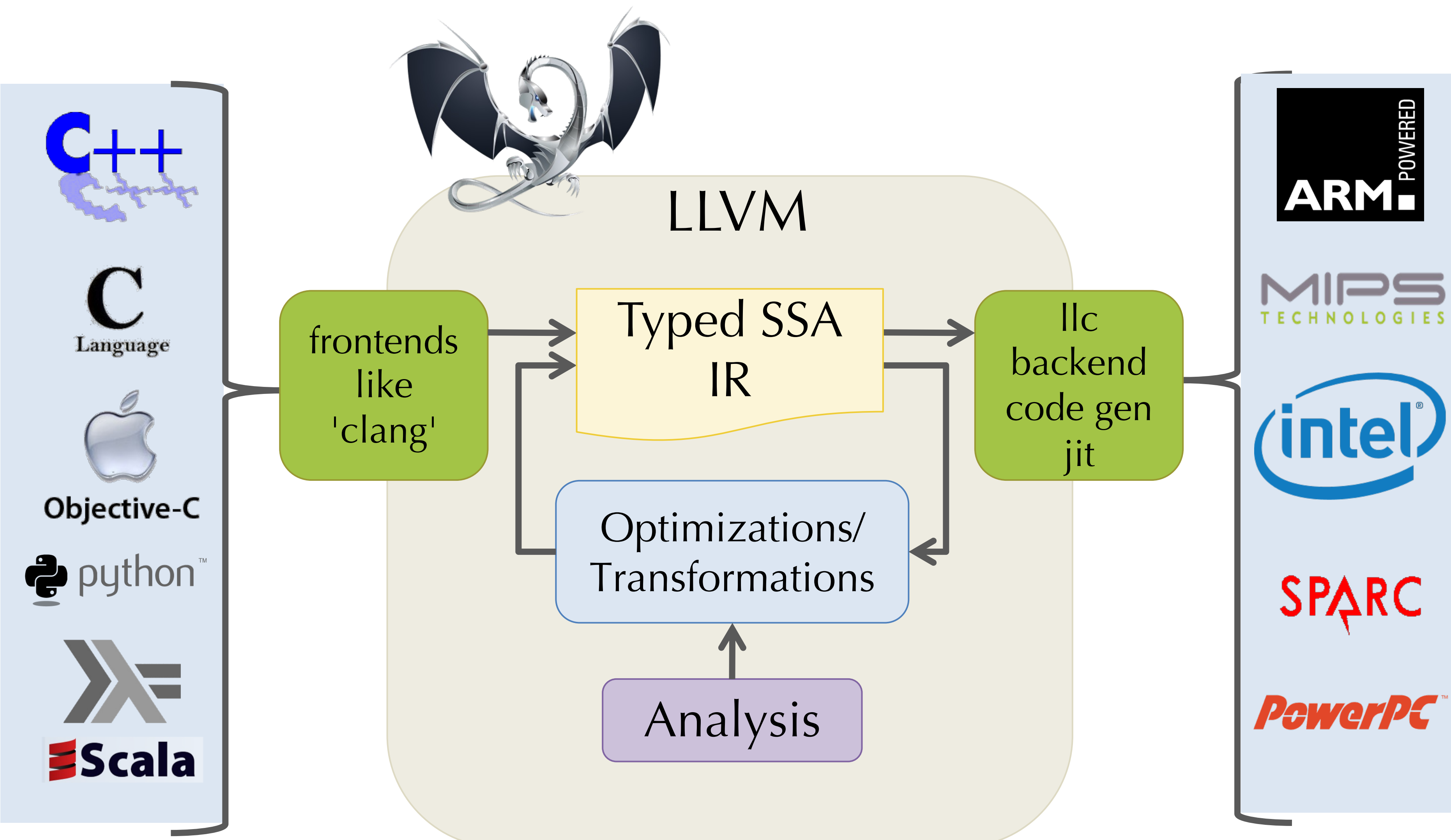
Multiple IR's

- Goal: get program closer to machine code without losing the information needed to do analysis and optimizations
- In practice, multiple intermediate representations might be used (for different purposes)



LLVM Compiler Infrastructure

[Lattner et al.]

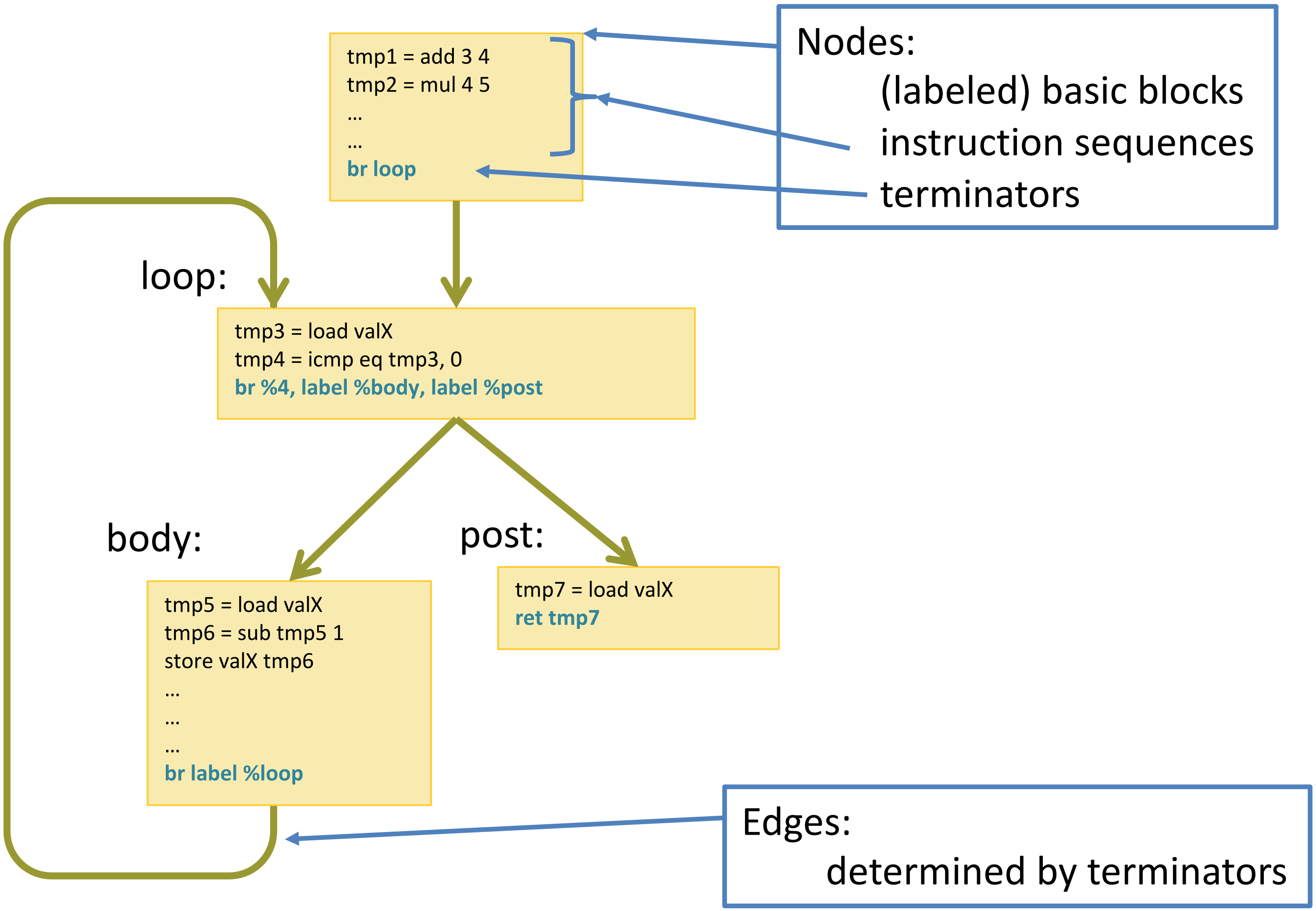


Basic Blocks

- A sequence of instructions that is always executed starting at the first instruction and always exits at the last instruction.
  - Starts with a label that names the *entry point* of the basic block.
  - Ends with a control-flow instruction (e.g., branch or return) the “link”
  - Contains no other control-flow instructions
  - Contains no interior label used as a jump target

- Basic blocks can be arranged into a *control-flow graph*
  - Nodes are basic blocks
  - There is a directed edge from node A to node B if the control flow instruction at the end of basic block A might jump to the label of basic block B.

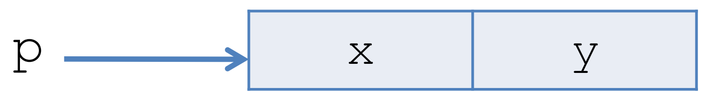
Control-flow Graphs



Representing Structs

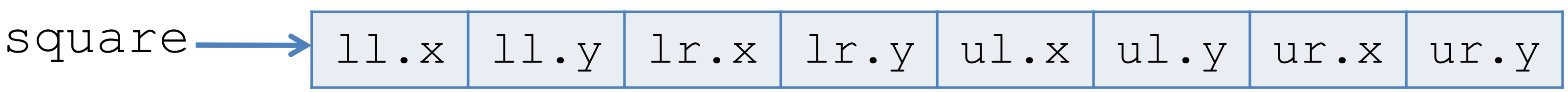
```
struct Point { int x; int y;};
```

- Store the data using two contiguous words of memory.
- Represent a `Point` value `p` as the address of the first word.




```
struct Rect  { struct Point ll, lr, ul, ur };
```

- Store the data using 8 contiguous words of memory.



- Compiler needs to know the *size* of the struct at compile time to allocate the needed storage space.
- Compiler needs to know the *shape* of the struct at compile time to index into the structure.

Multi-Dimensional Arrays

- In C, **`int M[4][3]`** yields an array with 4 rows and 3 columns.
- Laid out in *row-major* order:

`M[0][0]`	`M[0][1]`	`M[0][2]`	`M[1][0]`	`M[1][1]`	`M[1][2]`	`M[2][0]`	…

- M[i][j] compiles to?
- In Fortran, arrays are laid out in *column major order*.

`M[0][0]`	`M[1][0]`	`M[2][0]`	`M[3][0]`	`M[0][1]`	`M[1][1]`	`M[2][1]`	…

- In ML and Java, there are no multi-dimensional arrays:
  - (int array) array is represented as an array of pointers to arrays of ints.
- Why is knowing these memory layout strategies important?

Switch Compilation

- Consider the C statement:

```
switch (e) {
    case sun: s1; break;
    case mon: s2; break;
    …
    case sat: s3; break;
}
```

- How to compile this?
  - What happens if some of the break statements are omitted? (Control falls through to the next branch.)

GEP Example*

```
struct RT {
     int A;
     int B[10][20];
     int C;
}
struct ST {
     struct RT X;
     int Y;
     struct RT Z;
}
int *foo(struct ST *s) {
  return &s[1].Z.B[5][13];
}
```

1. `%s` is a pointer to an (array of) `%ST` structs, suppose the pointer value is ADDR
2. Compute the index of the 1st element by adding `size_ty(%ST)`.
3. Compute the index of the `Z` field by adding `size_ty(%RT) + size_ty(i32)` to skip past `X` and `Y`.
4. Compute the index of the `B` field by adding `size_ty(i32)` to skip past `A`.
5. Index into the 2d array.

```
%RT = type { i32, [10 x [20 x i32]], i32 }
%ST = type { %RT, i32, %RT }
define i32* @foo(%ST* %s) {
entry:
    %arrayidx = getelementptr %ST* %s, i32 1, i32 2, i32 1, i32 5, i32 13
    ret i32* %arrayidx
}
```

Final answer: ADDR + `size_ty(%ST) + size_ty(%RT) + size_ty(i32)`
`+ size_ty(i32) + 5*20*size_ty(i32) + 13*size_ty(i32)`

*adapted from the LLVM documentaion: see http://llvm.org/docs/LangRef.html#getelementptr-instruction

Lectures 9-10

LEXING, REGULAR EXPRESSIONS, AUTOMATA

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: `(* 483: Project 1 … *)  /* foo */`

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

Regular Expressions

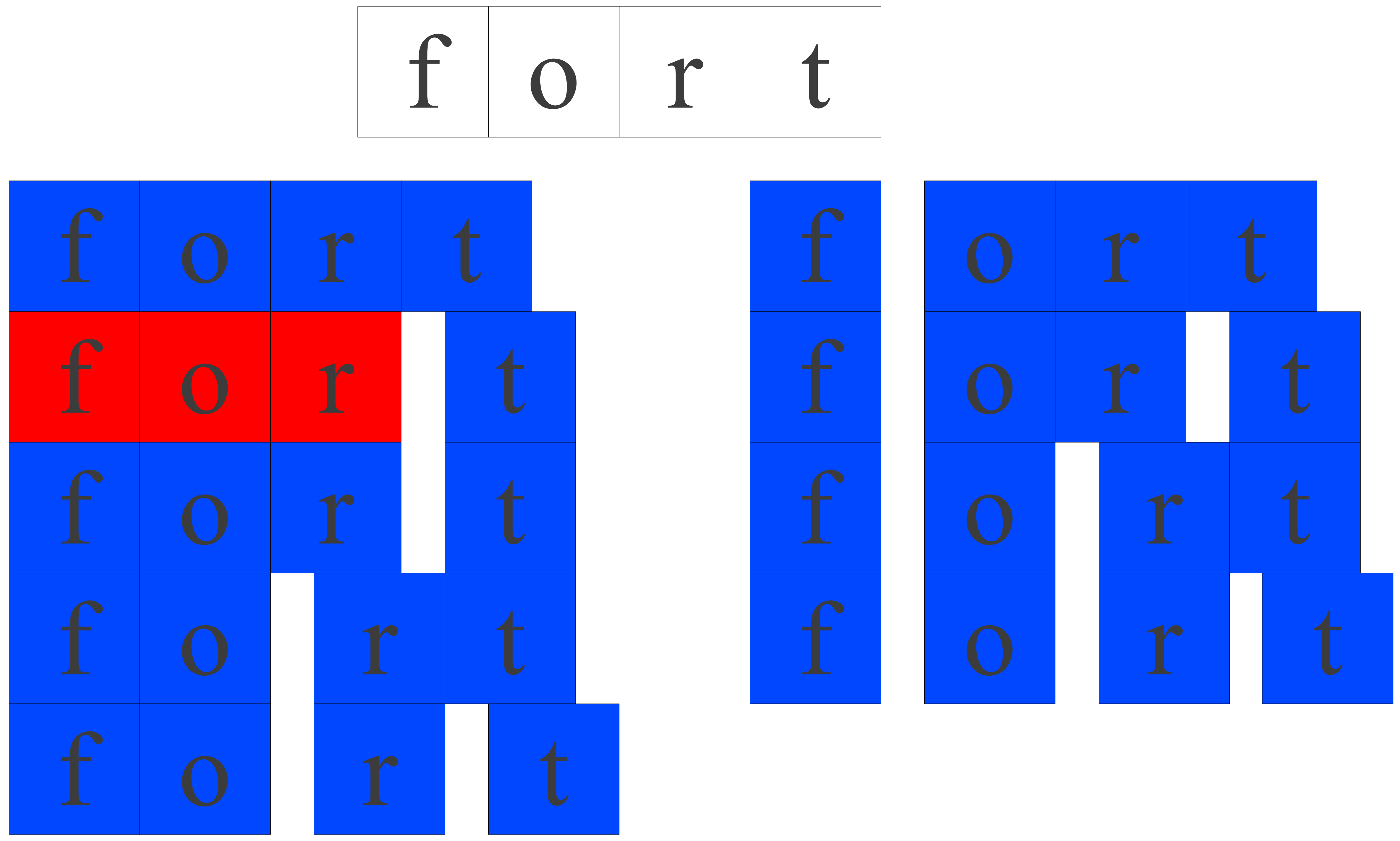
- Regular expressions precisely describe sets of strings.
- A regular expression R has one of the following forms:
  - ε — Epsilon stands for the empty string
  - `'a'` — An ordinary character stands for itself
  - $R_1 \mid R_2$ — Alternatives, stands for choice of R_1 or R_2
  - R_1R_2 — Concatenation,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

How to Match?

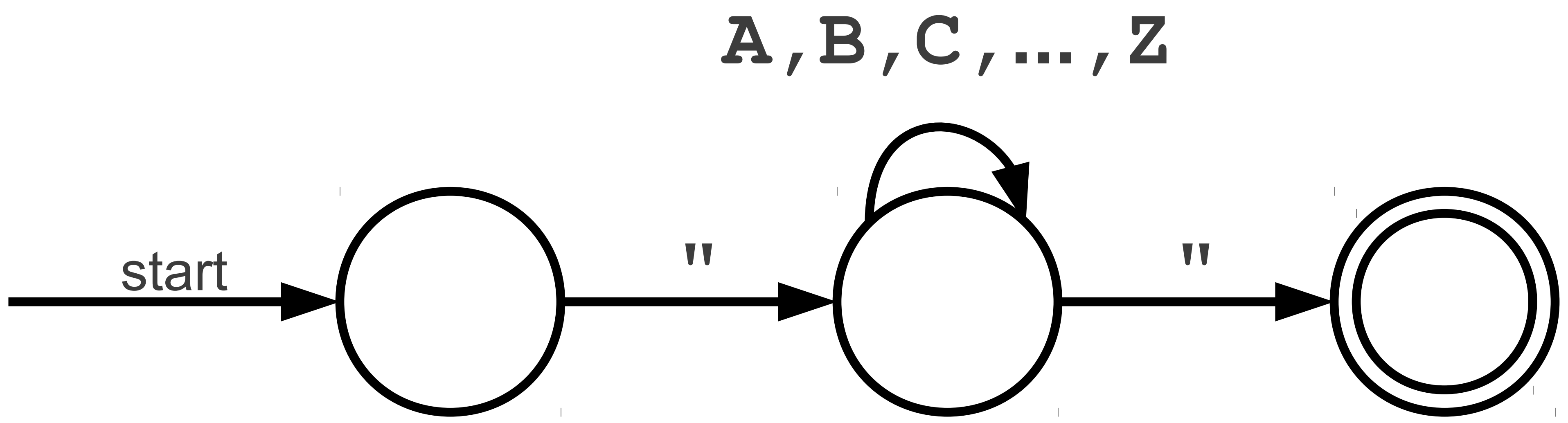
- Consider the input string: `ifx = 0`
  - Could lex as: `if` | `x` | `=` | `0` or as: `ifx` | `=` | `0`
- Regular expressions alone are ambiguous, need a rule for choosing between the options above
- Most languages choose "longest match"
  - So the 2^{nd} option above will be picked
  - Note that only 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

Lexing Ambiguities

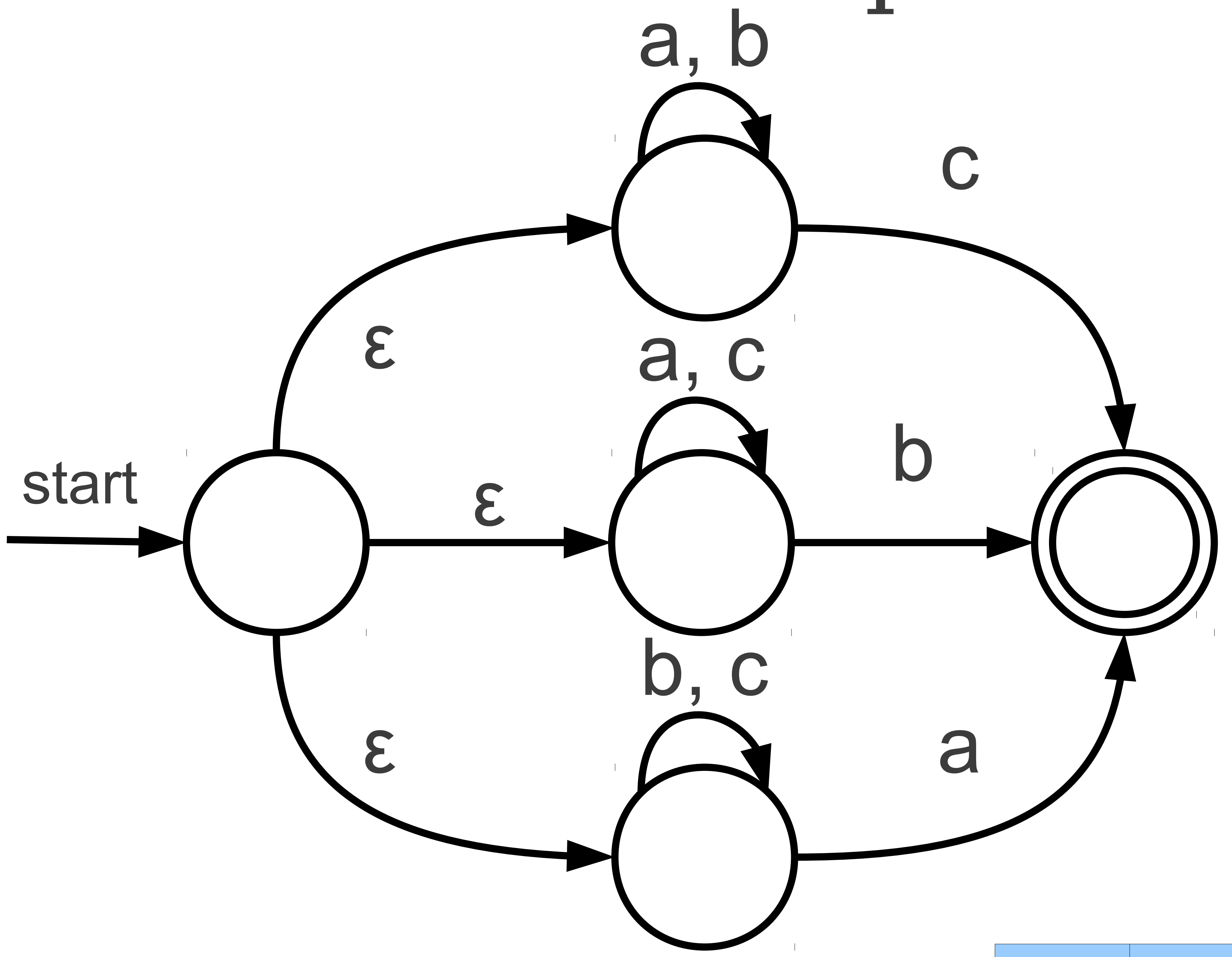
```
T_For           for
T_Identifier    [A-Za-z_][A-Za-z0-9_]*
```

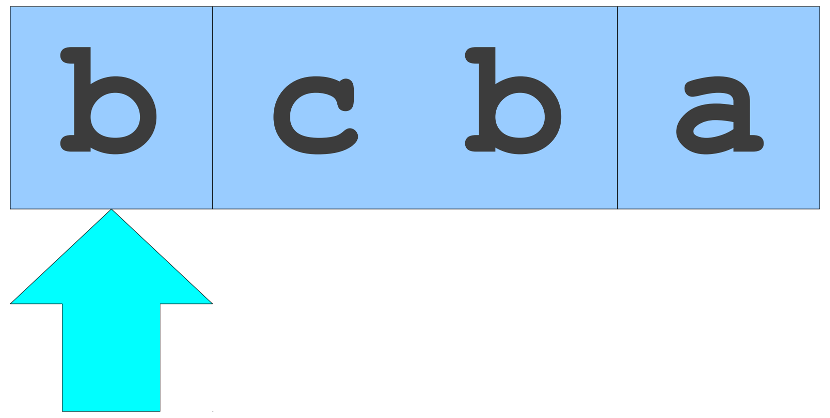


A Simple Automaton



An Even More Complex Automaton





Automating Lexical Analyzer (scanner) Construction

RE→ NFA *(Thompson's construction)*

- Build an NFA for each term
- Combine them with ε-moves

NFA → DFA *(subset construction)*

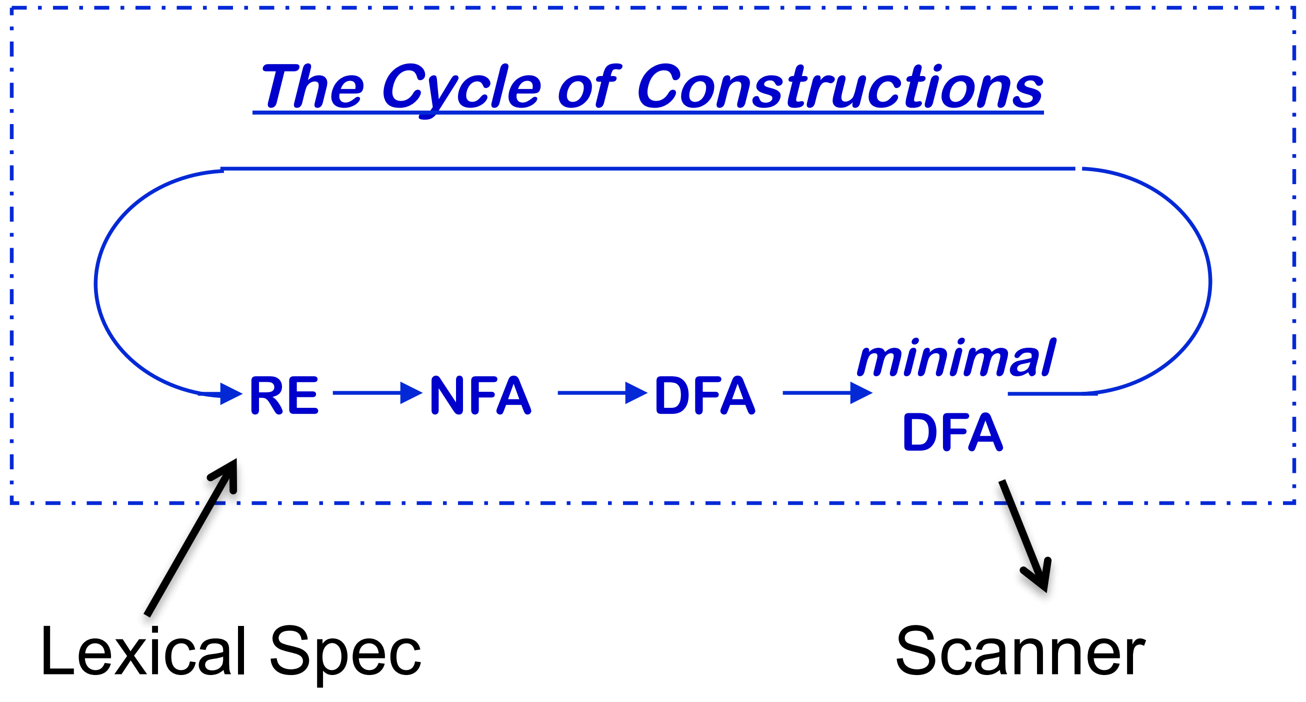
- Build the simulation

DFA → Minimal DFA

- Hopcroft's algorithm

DFA →RE *(Not part of the scanner construction)*

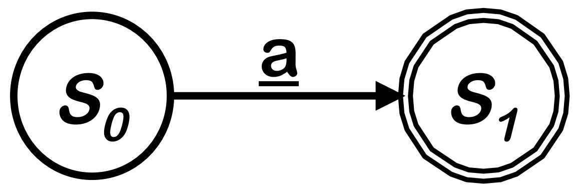
- All pairs, all paths problem
- Take the union of all paths from s_0 to an accepting state



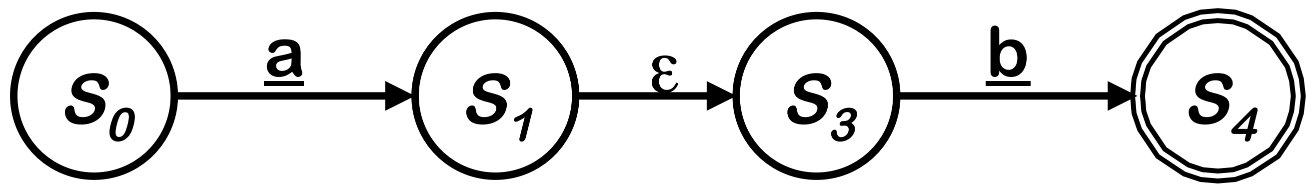
RE →NFA using Thompson’s Construction

Key idea

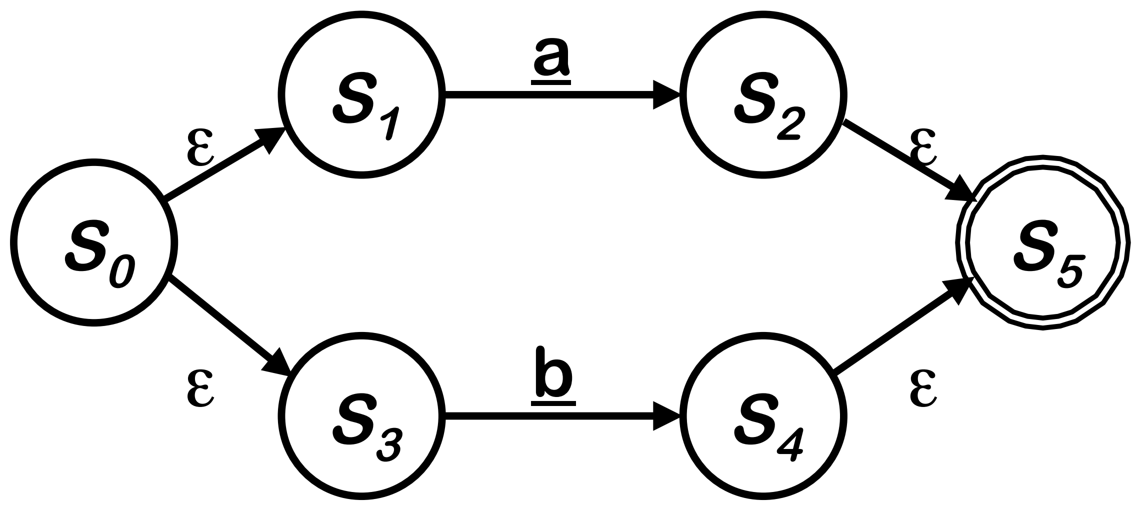
- NFA pattern for each symbol & each operator
- Join them with ε moves in precedence order



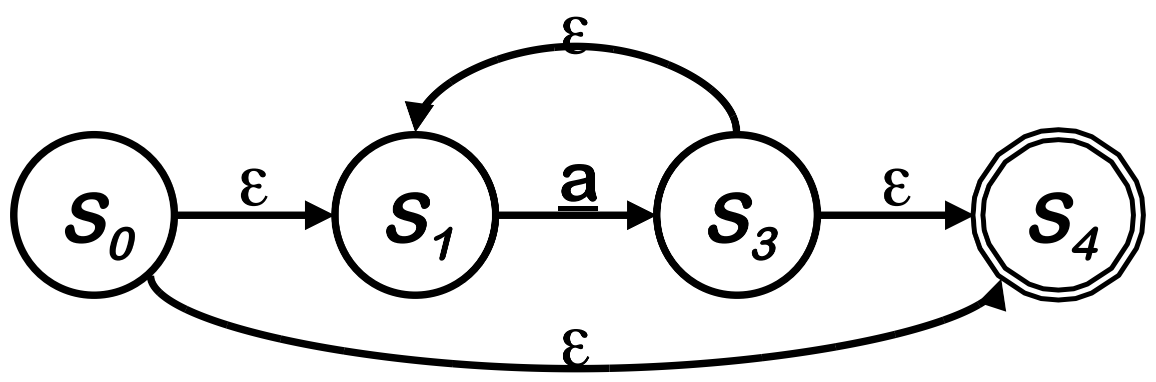
NFA for a



NFA for ab



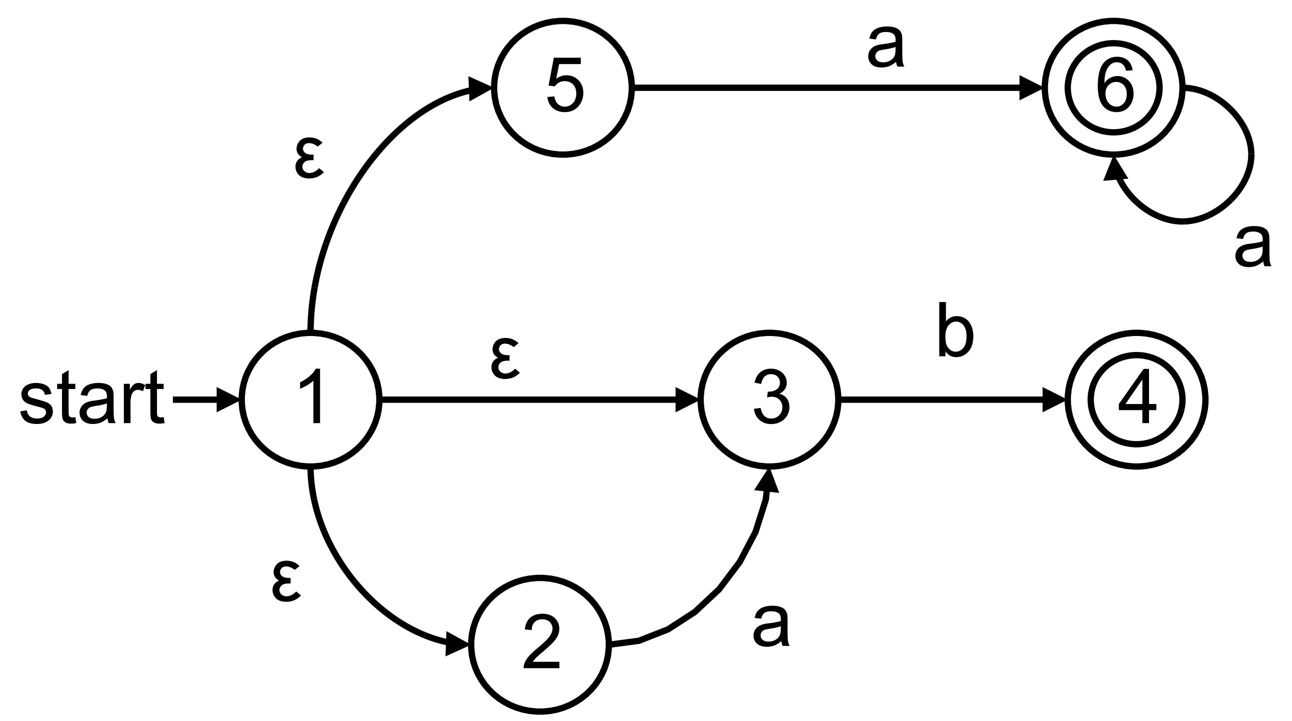
NFA for a | b



NFA for a^*

Ken Thompson, CACM, 1968

NFA to DFA - Example



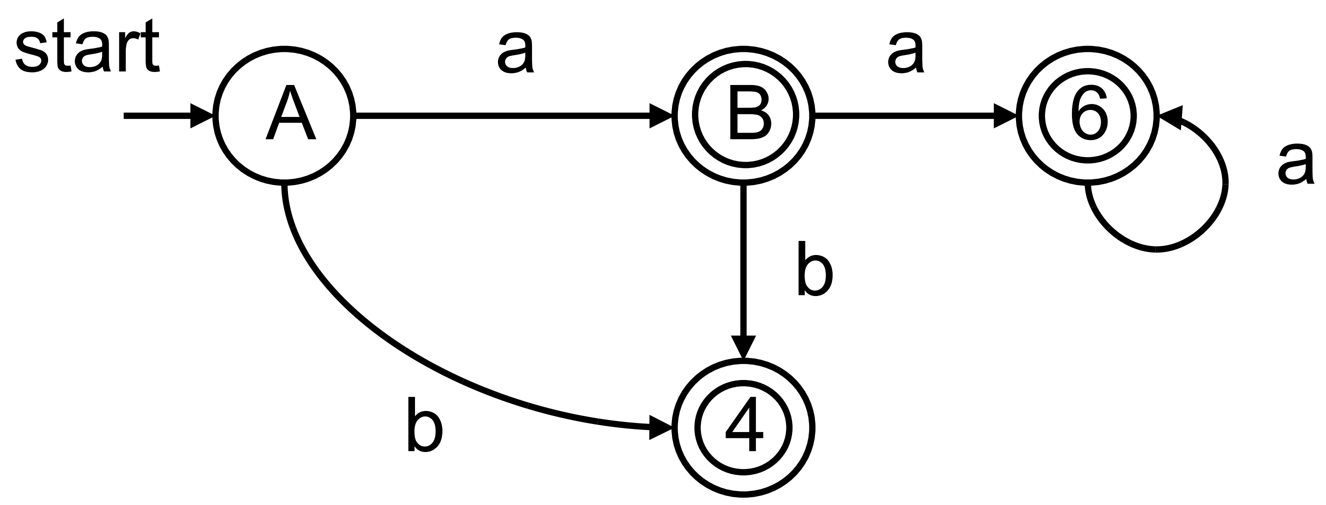
ε-closure(1) = {1, 2, 3, 5}

Create a new state A = {1, 2, 3, 5}

move(A, a) = {3, 6} + ε-closure(3,6) = {3,6}

Create B = {3,6}

move(A, b) = {4} + ε-closure(4) = {4}



move(B, a) = {6} + ε-closure(6) = {6}

move(B, b) = {4} + ε-closure(4) = {4}

move(6, a) = {6} + ε-closure(6) = {6}

move(6, b) → ERROR

move(4, a|b) → ERROR

Lectures 11-13

PARSING, LL/LR GRAMMARS

Derivations in CFGs

- Example: derive (1 + 2 + (3 + 4)) + 5
- $\underline{\mathbf{S}} \longmapsto \underline{\mathbf{E}} + S$

$\longmapsto (\underline{\mathbf{S}}) + S$

$\longmapsto (\underline{\mathbf{E}} + S) + S$

$\longmapsto (1 + \underline{\mathbf{S}}) + S$

$\longmapsto (1 + \underline{\mathbf{E}} + S) + S$

$\longmapsto (1 + 2 + \underline{\mathbf{S}}) + S$

$\longmapsto (1 + 2 + \underline{\mathbf{E}}) + S$

$\longmapsto (1 + 2 + (\underline{\mathbf{S}})) + S$

$\longmapsto (1 + 2 + (\underline{\mathbf{E}} + S)) + S$

$\longmapsto (1 + 2 + (3 + \underline{\mathbf{S}})) + S$

$\longmapsto (1 + 2 + (3 + \underline{\mathbf{E}})) + S$

$\longmapsto (1 + 2 + (3 + 4)) + \underline{\mathbf{S}}$

$\longmapsto (1 + 2 + (3 + 4)) + \underline{\mathbf{E}}$

$\longmapsto (1 + 2 + (3 + 4)) + 5$

$S \longmapsto E + S \mid E$

$E \longmapsto \text{number} \mid (S)$

For arbitrary strings α, β, γ and production rule $A \longmapsto \beta$
a single step of the derivation is:

$$\alpha A \gamma \longmapsto \alpha \beta \gamma$$

(*substitute* β for an occurrence of A)

In general, there are many possible derivations for a given string

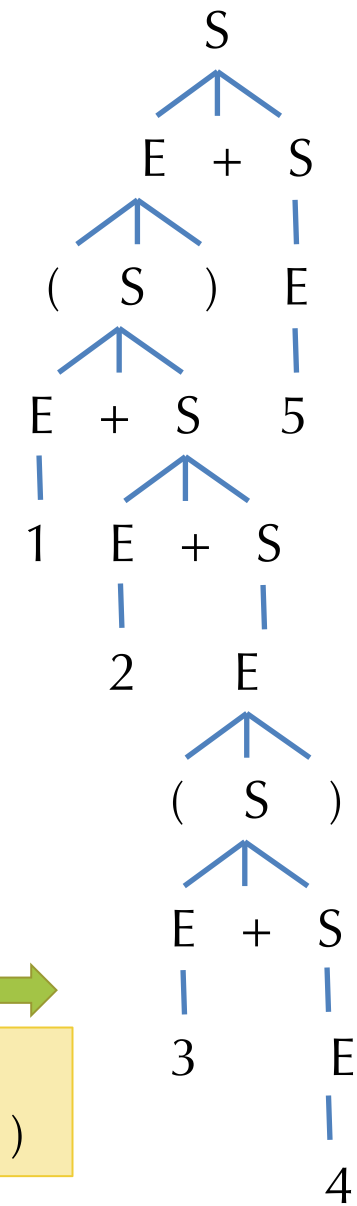
Note: Underline indicates symbol being expanded.

From Derivations to Parse Trees

- Tree representation of the derivation
- Leaves of the tree are terminals
  - In-order traversal yields the input sequence of tokens
- Internal nodes: nonterminals
- No information about the *order* of the derivation steps

- (1 + 2 + (3 + 4)) + 5

$S \longmapsto E + S \mid E$
$E \longmapsto \text{number} \mid (S)$

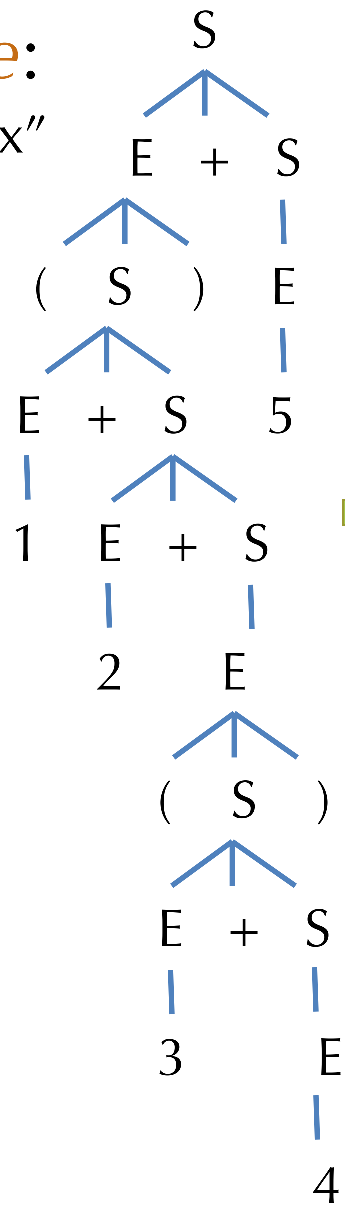


Parse Tree

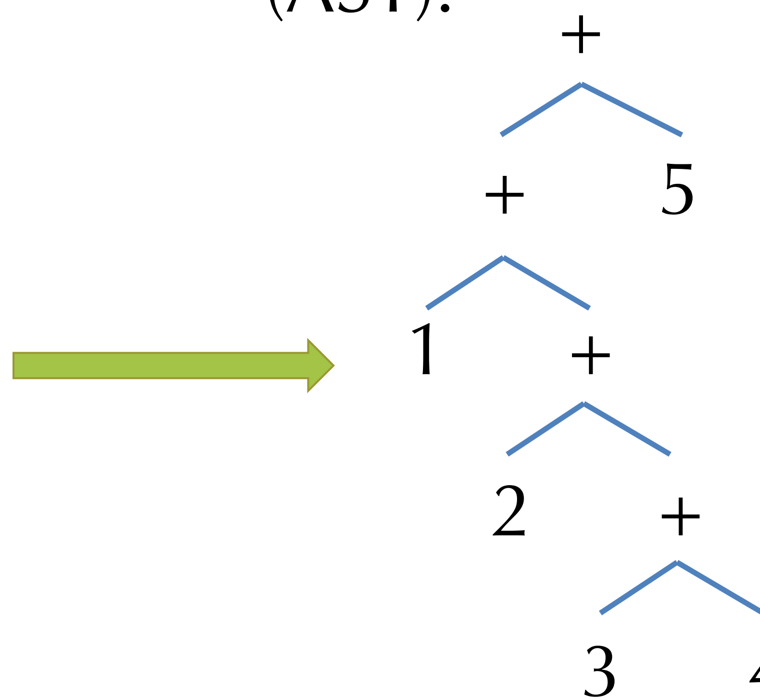
From Parse Trees to Abstract Syntax

- *Parse tree*:

"concrete syntax"



- *Abstract syntax tree* (AST):

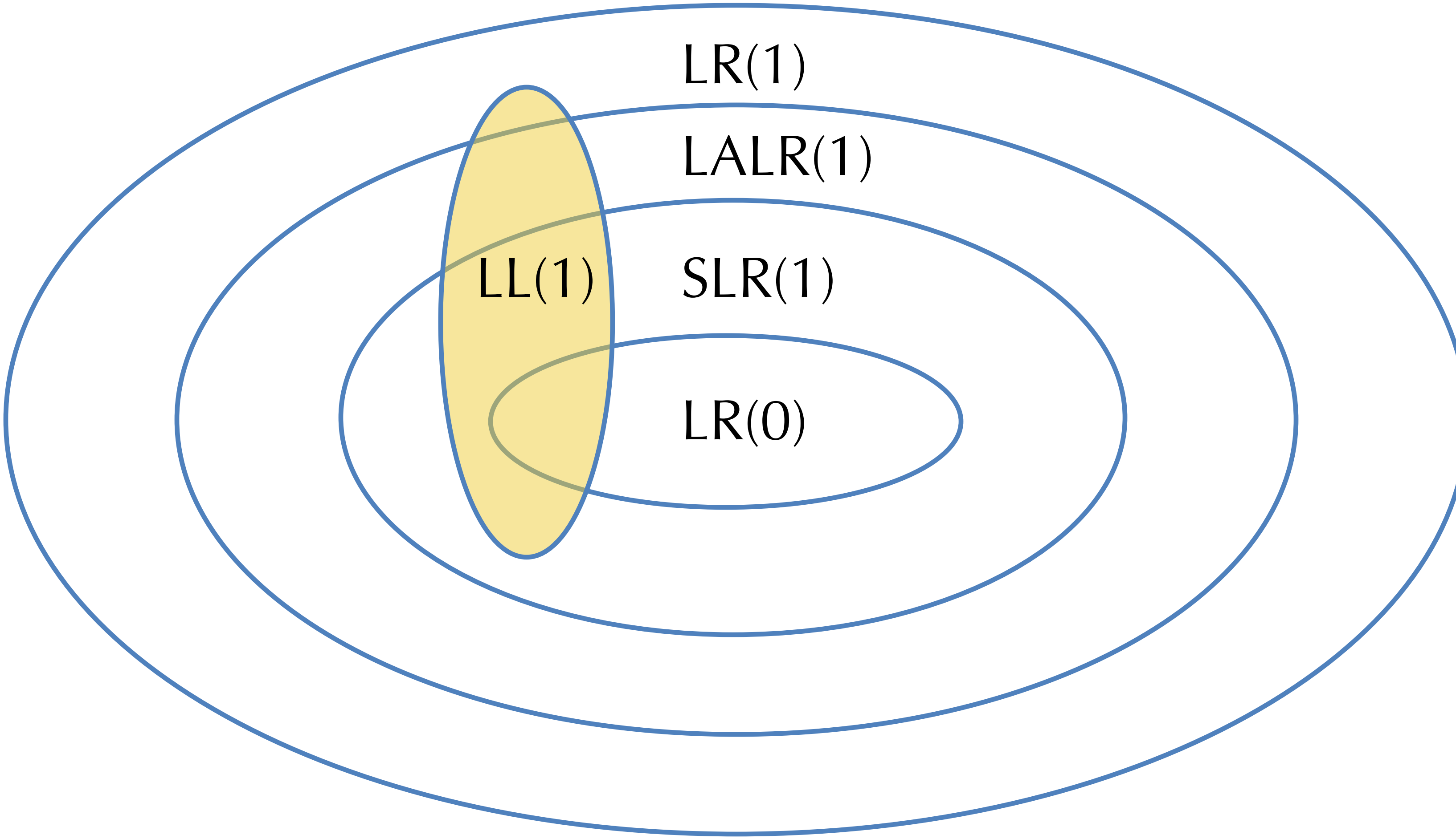


- Hides, or *abstracts*, unneeded information.

Context Free Grammars: Summary

- Context-free grammars allow concise specifications of programming languages.
  - An unambiguous CFG specifies how to parse: convert a token stream to a (parse tree)
  - Ambiguity can (often) be removed by encoding precedence and associativity in the grammar.
- Even with an unambiguous CFG, there may be more than one derivation
  - Though all derivations correspond to the same abstract syntax tree.
- Still to come: finding a derivation
  - But first: menhir

Classification of Grammars

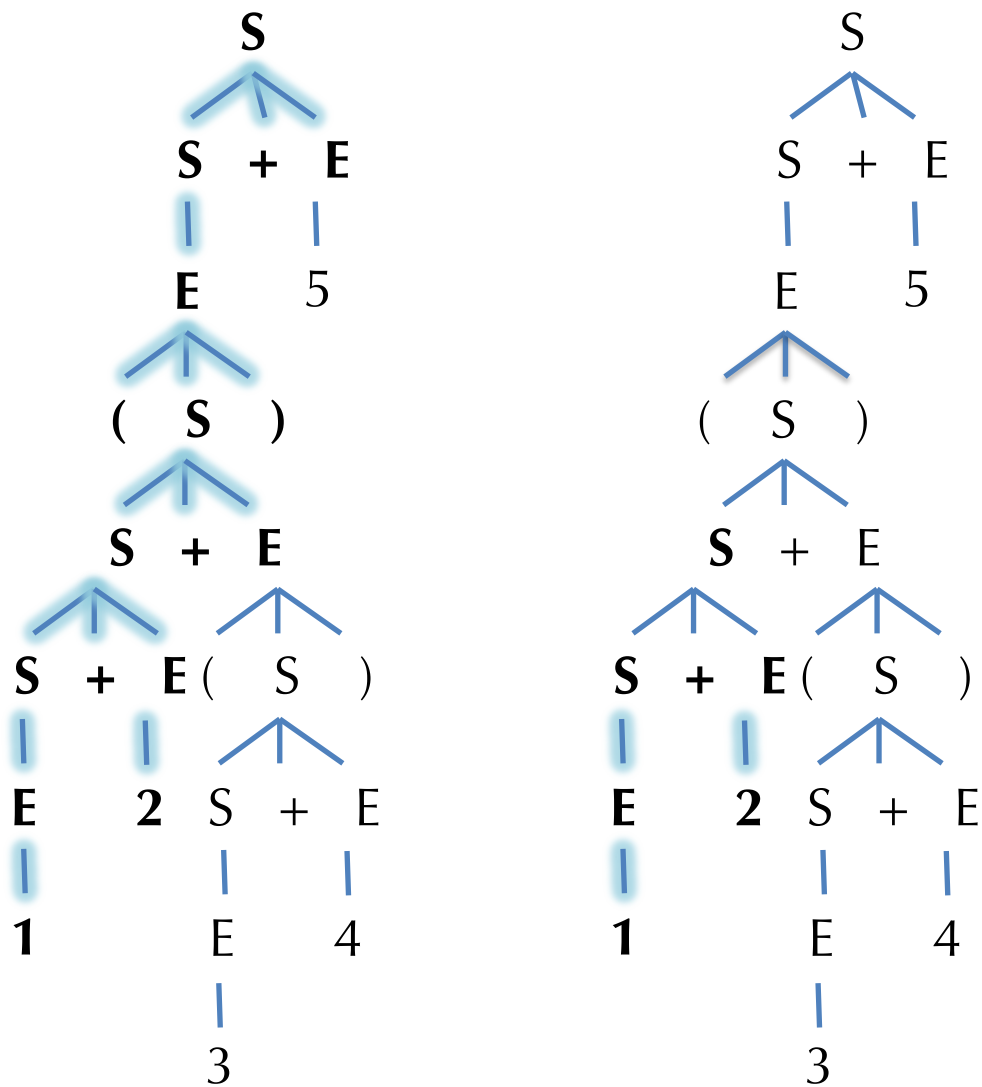


Top-down vs. Bottom up

- Consider the left-recursive grammar:

$S \longmapsto S + E \mid E$
$E \longmapsto \text{number} \mid (S)$

- (1 + 2 + (3 + 4)) + 5
- We want to parse by doing a linear scan, left-to-right
- Top-down: construct a partial tree from the root
- Bottom-up: construct partial tree from the leaves



Top-down

Bottom-up

Grammar is the problem

- Not all grammars can be parsed "top-down" with only a single lookahead symbol.
- *Top-down*: starting from the start symbol (root of the parse tree) and going down

- LL(1) means
  - **L**eft-to-right scanning
  - **L**eft-most derivation,
  - **1** lookahead symbol

- This language isn't "LL(1)"
- Is it LL(k) for some k?

$$S \longmapsto E + S \mid E$$
$$E \longmapsto \text{number} \mid (S)$$

- What can we do?

Making a grammar LL(1)

- *Problem:* We can't decide which S production to apply until we see the symbol after the first expression.
- *Solution:* "Left-factor" the grammar. There is a common S prefix for each choice, so add a new non-terminal S′ at the decision point:

$$
\begin{aligned}
S &\longmapsto E + S \mid E \\
E &\longmapsto \text{number} \mid (S)
\end{aligned}
\quad\Longrightarrow\quad
\begin{aligned}
S &\longmapsto ES' \\
S' &\longmapsto \varepsilon \\
S' &\longmapsto + S \\
E &\longmapsto \text{number} \mid (S)
\end{aligned}
$$

- Also need to eliminate left-recursion somehow. Why?
- Consider:

$$
\begin{aligned}
S &\longmapsto S + E \mid E \\
E &\longmapsto \text{number} \mid (S)
\end{aligned}
$$

Bottom-up Parsing (LR Parsers)

- LR(k) parser:
  - Left-to-right scanning
  - Rightmost derivation
  - k lookahead symbols

- LR grammars are more expressive than LL
  - Can handle left-recursive (and right recursive) grammars; virtually all programming languages
  - Easier to express programming language syntax (no left factoring)

- Technique: "Shift-Reduce" parsers
  - Work bottom up instead of top down
  - Construct right-most derivation of a program in the grammar
  - Used by many parser generators (e.g. yacc, ocamlyacc, lalrpop, etc.)
  - Better error detection/recovery

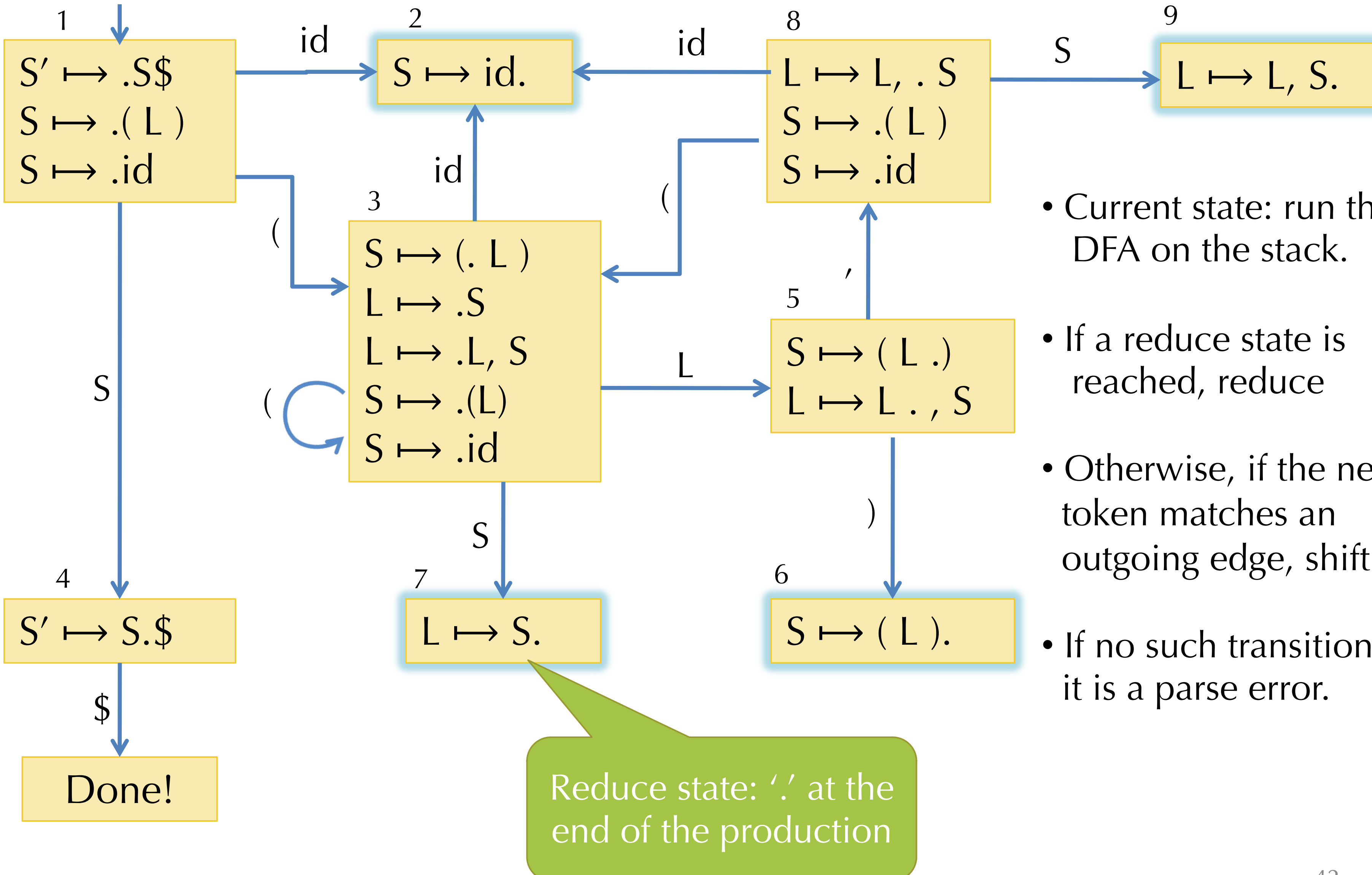
Shift/Reduce Parsing

$S \longmapsto S + E \mid E$
$E \longmapsto \text{number} \mid (S)$

- Parser state:
  - Stack of terminals and nonterminals.
  - Unconsumed input is a string of terminals
  - Current derivation step is stack + input
- Parsing is a sequence of *shift* and *reduce* operations:
- Shift: move look-ahead token to the stack
- Reduce: Replace symbols γ at top of stack with nonterminal X such that X $\longmapsto \gamma$ is a production. (pop γ, push X)

Stack	Input	Action
	(1 + 2 + (3 + 4)) + 5	shift (
(	1 + 2 + (3 + 4)) + 5	shift 1
(1	+ 2 + (3 + 4)) + 5	reduce: E $\longmapsto$ number
(E	+ 2 + (3 + 4)) + 5	reduce: S $\longmapsto$ E
(S	+ 2 + (3 + 4)) + 5	shift +
(S +	2 + (3 + 4)) + 5	shift 2
(S + 2	+ (3 + 4)) + 5	reduce: E $\longmapsto$ number
(S + E	+ (3 + 4)) + 5	reduce: S $\longmapsto$ S + E
(S	+ (3 + 4)) + 5	shift +

Full DFA for the Example



- Current state: run the DFA on the stack.
- If a reduce state is reached, reduce
- Otherwise, if the next token matches an outgoing edge, shift.
- If no such transition, it is a parse error.

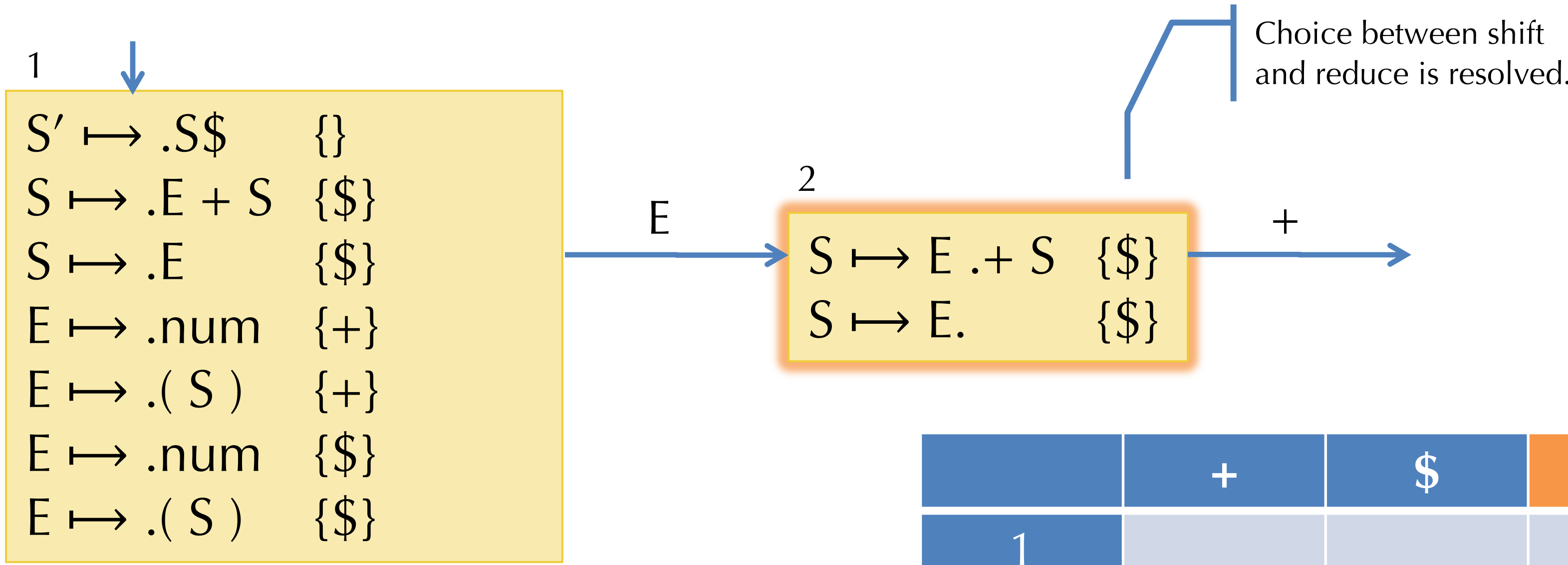
LR(0) Limitations

- An LR(0) machine only works if states with reduce actions have a *single* reduce action.
  - In such states, the machine *always* reduces (ignoring lookahead)
- With more complex grammars, the DFA construction will yield states with shift/reduce and reduce/reduce conflicts:

OK	shift/reduce	reduce/reduce
$S \longmapsto (\ L\ ).$	$S \longmapsto (\ L\ ).$ $L \longmapsto .L\ ,\ S$	$S \longmapsto L\ ,S.$ $S \longmapsto ,S.$

- Such conflicts can often be resolved by using a look-ahead symbol: SLR(1) or LR(1)

Using the DFA



	+	$	E
1			g2
2	s3	$S \mapsto E$	

Fragment of the Action & Goto tables

- The behavior is determined if:
  - There is no overlap among the look-ahead sets for each reduce item, and
  - None of the look-ahead symbols appear to the right of a '.'

Exam Review

- Outline
  - Language Semantics
  - Compiler Architecture
  - X86 Assembly code
  - LLVM IR
  - Regular Expressions, Finite Automata, Lexing
  - LL/LR Grammars and Parsing
- Now: questions?
- I will start today's office hours early 3:30-5pm.