Welcome to CS 536:
Introduction to Programming Languages and Compilers!

Instructor: Beck Hasti
- hasti@cs.wisc.edu
- Office hours to be determined

TAs
- Jinlang Wang
- Rithik Jain
- Sadman Sakib

Course websites:

`canvas.wisc.edu`

`www.piazza.com/wisc/spring2023/compsci536`

`pages.cs.wisc.edu/~hasti/cs536`

About the course

We will study compilers

We will understand how they work

We will build a full compiler

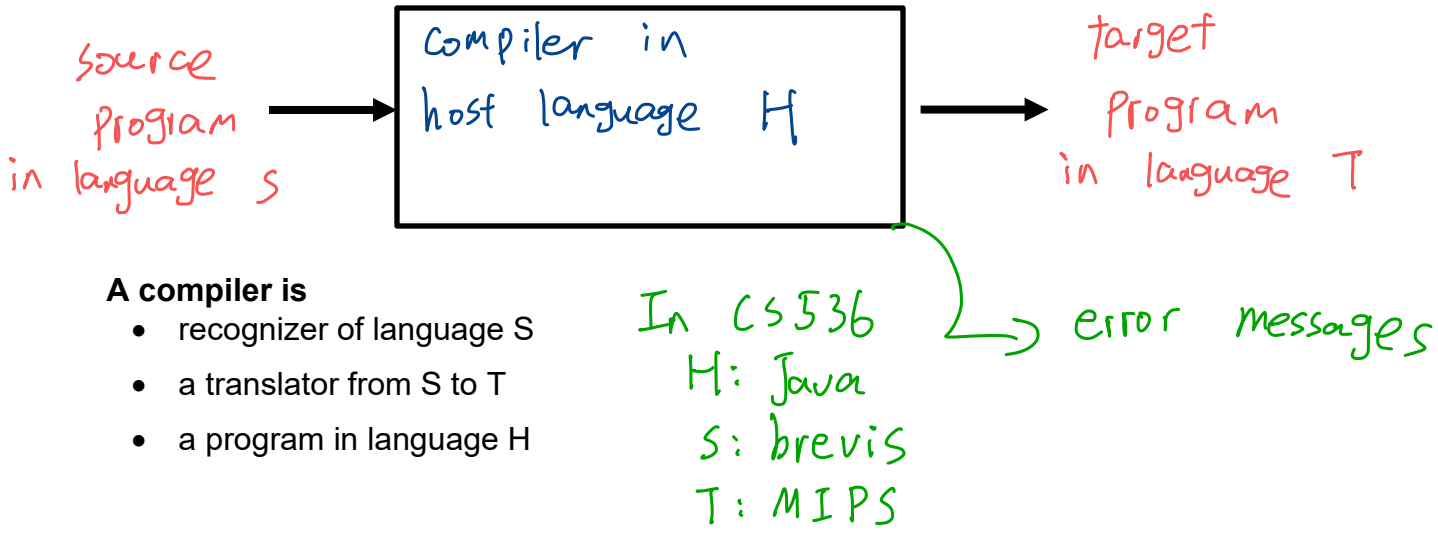
Course mechanics

Exams (60%)
- Midterm 1 (18%): Wednesday, March 1, 7:30 – 9 pm
- Midterm 2 (16%): Wednesday, March 29, 7:30 – 9 pm
- Final (26%): Thursday, May 11, 2:45 – 4:45 pm

Programming Assignments (40%)
- 6 programs: 5% + 7% + 7% + 7% + 7%+ 7%

Homework Assignments
- 7 short homeworks (optional, not graded)

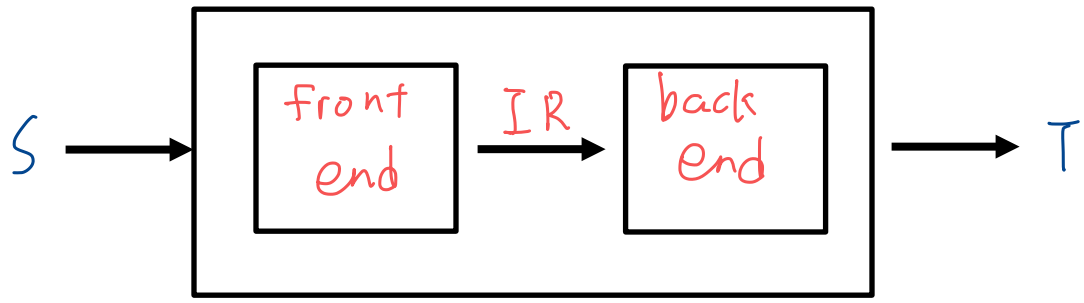
What is a compiler?



A compiler is

- recognizer of language S
- a translator from S to T
- a program in language H

Front end vs back end

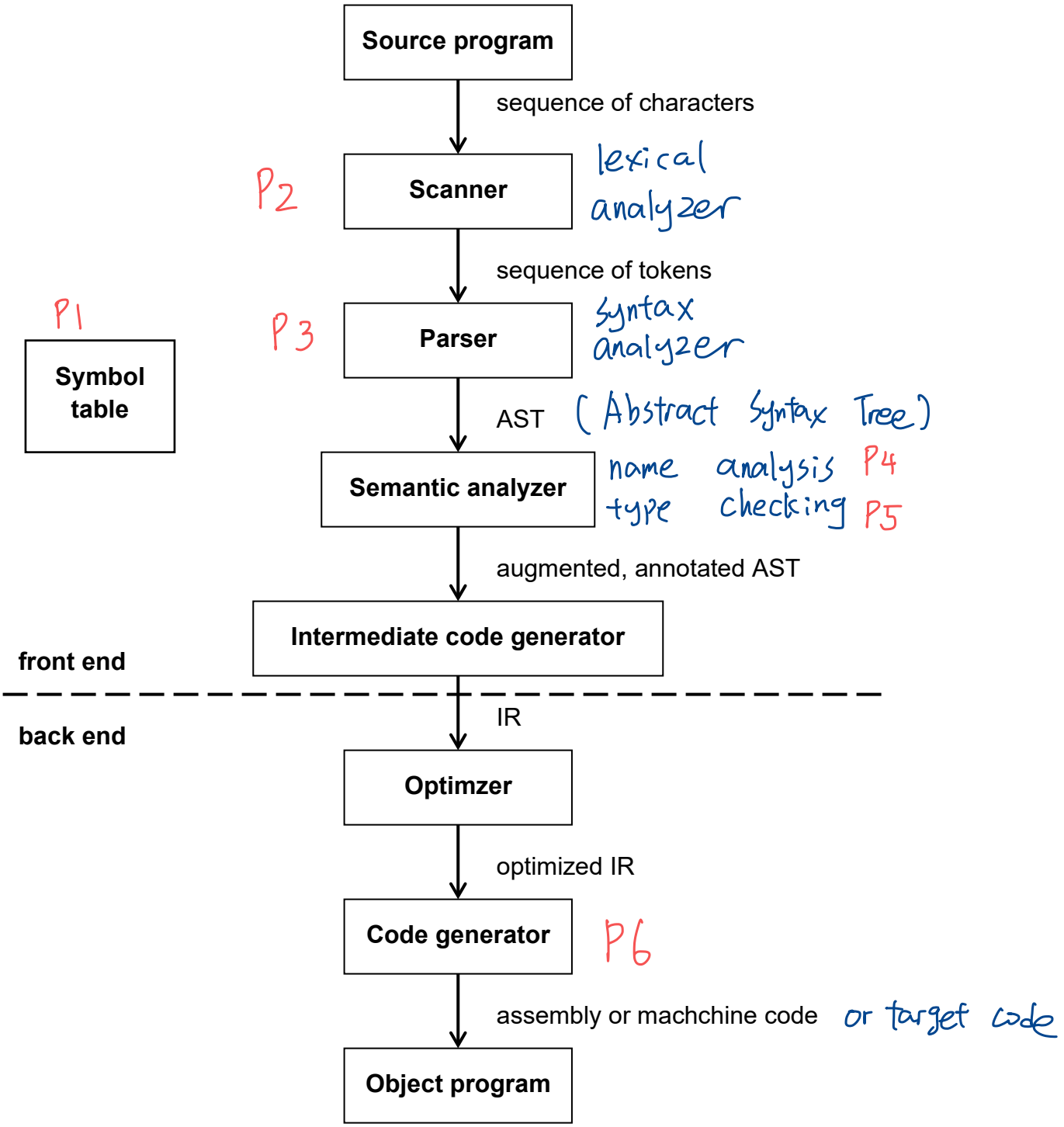


front end = understand source code S; map S to IR

IR = intermediate representation

back end = map IR to T

Overview of typical compiler



Scanner

Input: characters from source program

Output: sequence of tokens
└ possibly with associated info

Actions:

- group characters into lexemes (tokens)
- identify and ignore whitespace, comments, etc.

What errors can it catch?

- bad characters eg ^ in Java
- unterminated strings "Hello
- integer literals that are too large

Parser

Input: sequence of tokens from the scanner

Output: AST (abstract syntax tree)

Actions:

- group tokens into sentences

What errors can it catch?

- syntax errors x=y *=5;
- (possibly) *static semantic* errors use of undeclared variables

Semantic analyzer

Input: AST

Output: annotated AST

Actions: does more static semantic checks

- Name analysis
  process decls & uses of variables
  match uses w/ decls
  enforces scoping rules
  errors - multiply-declared vars, use of undeclared var.
- Type checking
  check types & Augment AST

Intermediate code generator

Input: annotated AST - assumes no syntax / static-semantic errors

Output: intermediate representation (IR)

eg. 3-address code
- instructions have at most 3 operands.
- easy to generate from AST
  ↳ 1 instr. per AST internal node

Example

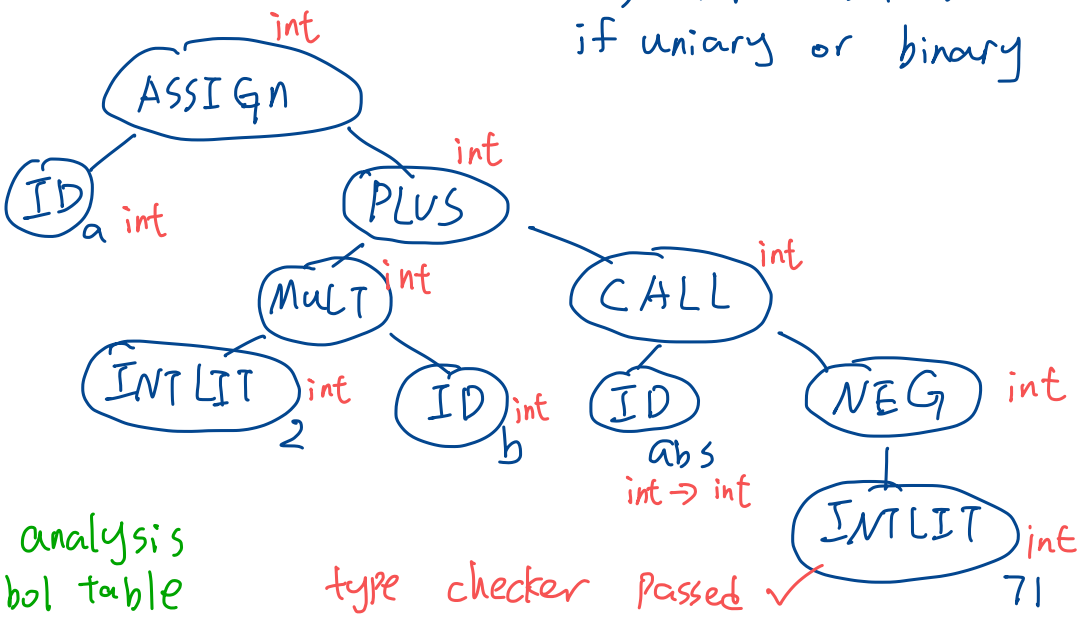
```
a = 2 * b + abs(-71);
```

Scanner produces tokens:

ID(a) ASSIGN INTLIT(2) TIMES ID(b) PLUS ID(abs) LPAREN MINUS INTLIT(71) RPAREN SEMICOLON

Scanner doesn't know if uniary or binary

AST (from parser)



Symbol table Name analysis gives us symbol table

ID	kind	type
a	var	int
b	var	int
abs	func	int → int

type checker passed ✓

3-address code

```
temp1 = 2 * b
temp2 = 0 - 71
move temp2 param1
call abs
move return1 temp3
temp4 = temp1 + temp3
a = temp4
```

} a = temp1 + temp3

Optimizer

Input: IR

Output: optimized IR

Actions: improve code

- make it run faster, make it smaller
- several passes: local and global optimization
- more time spent in compilation; less time in execution

local = look at a few instrs at a time

global = look at entire func or whole prog

Code generator

Input: IR from optimizer

For 536 our IR is an AST

Output: target code

Symbol Table

Compiler keeps track of names in

- semantic analyzer – both name analysis & type checking
- code generation – offsets into stack
- optimizer – Could use to keep track of def-use info

P1 : implement symbol table

Block-structured language eg Java, C, C++, Previs

- nested visibility of names – no access outside of scope of name
- easy to tell which def of a name applies (usually nearest enclosing)
- lifetime of data is bound to scope of identifier that denotes it

Example: (from C)

```
int x, y;

void A() {
   double x, z;
   C(x, y, z);
}

void B(){
   C(x, y, z);
}
```

double int double

int int undeclared

block structure ⇒
- need nesting of sym tables
⇒ List of hash tables

CS 536 Announcements for Monday, January 30, 2023

Course websites:

`pages.cs.wisc.edu/~hasti/cs536`

`www.piazza.com/wisc/spring2023/compsci536`

Programming Assignment 1

- test code due Friday, Feb. 3 by 11:59 pm
- other files due Tuesday, Feb. 7 by 11:59 pm

Last Time

- intro to CS 536
- compiler overview

Today

- start scanning
- finite state machines
  - formalizing finite state machines
  - coding finite state machines
  - deterministic vs non-deterministic FSMs

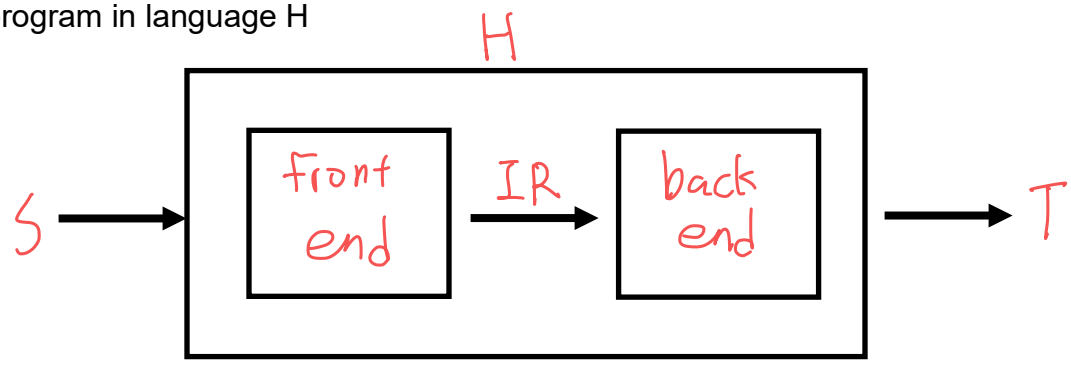
Next Time

- non-deterministic FSMs
- equivalence of NFAs and DFAs
- regular languages
- regular expressions

Recall

A compiler is

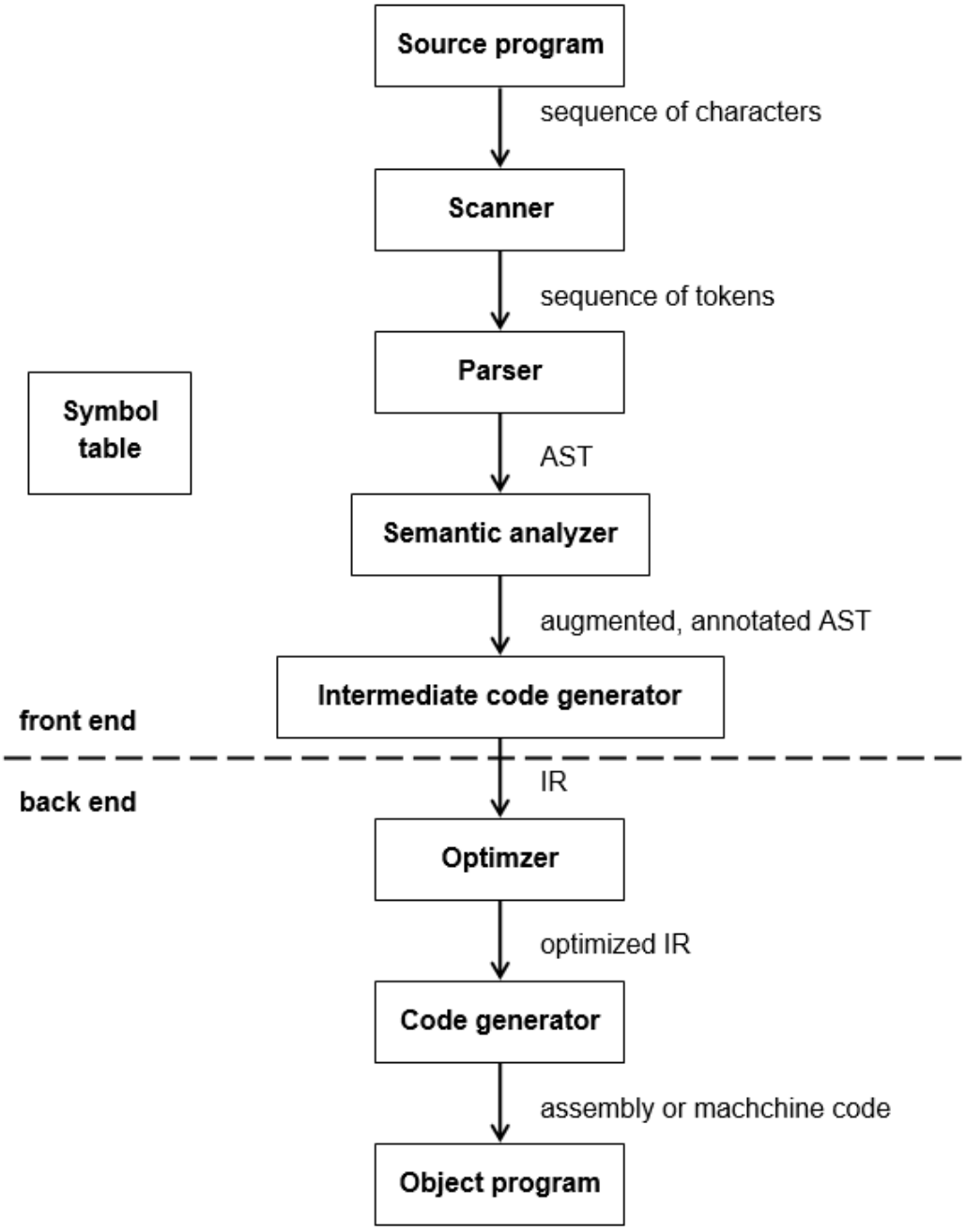
- recognizer of language S
- a translator from S to T
- a program in language H



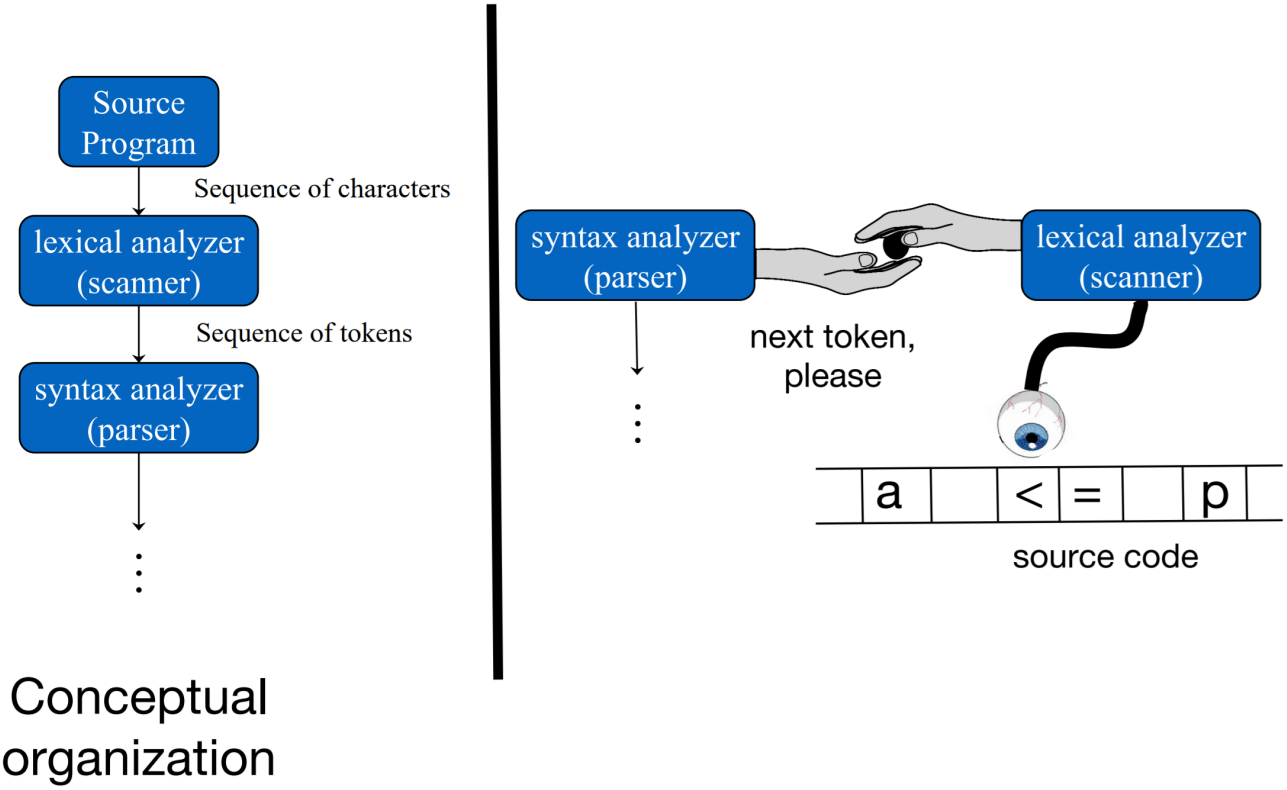
front end = understand source code S; map S to IR
IR = intermediate representation
back end = map IR to T

Why do we need a compiler?

- processors can execute only binaries (machine-code/assembly programs)
- writing assembly programs will make you lose your mind
- allows you to write programs in nice(ish) high-level languages like C; compile to binaries



Special linkage between scanner and parser (in most compilers)



Conceptual organization

Scanning

Scanner translates sequence of chars into sequence of tokens

Each time scanner is called it should:

- find longest sequence of chars corresponding to a token
- return that token

avg += 7 INTLIT(7)
ID(avg) PLUSASG

Scanner generator

- **Inputs**:
  - one regular expression for each token
  - one regular expression for each item to ignore (comments, whitespace, etc.)
- **Output**: scanner program

To understand how a scanner generator works, we need to understand FSMs

FA Finite-state machines FSM
(aka finite automata, finite-state automata)

- **Inputs**: string (sequence of characters) – finite length
- **Output**: accept / reject – is string in the language L

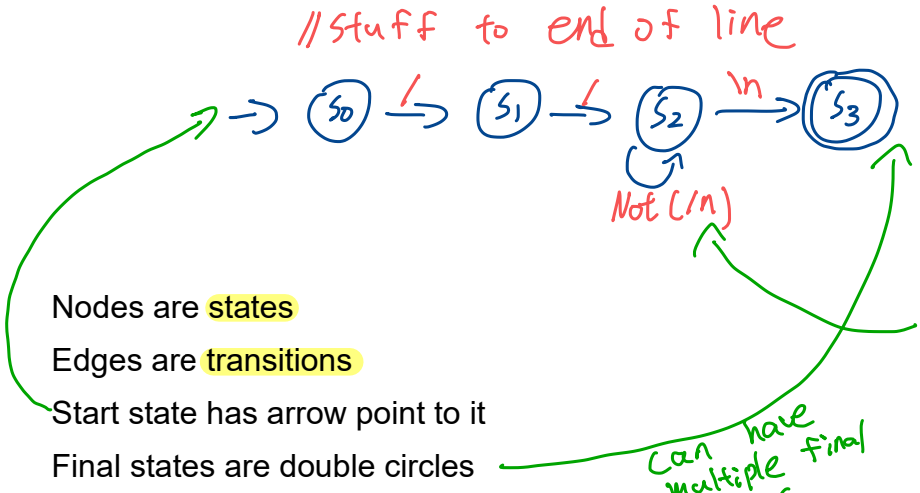
Language defined by an FSM = the set of strings accepted by the FSM

compiler recognize legal program in source lang. S

FSM recognize legal strings in some lang. L

Example 1:

Language: single-line comments starting with // (in Java / C++)



Nodes are states

Edges are transitions

Start state has arrow point to it

Final states are double circles

can have multiple final states

short hand for many transitions. (1 for every char that is not \n)

Consider

// red \n ✓

// green ✗

// \n ✓

// blue EOF ✗ stuck

// cyan \n teal ✗ stuck

How a finite state machine works

```
curr_state = start_state
let in_ch= current input character
repeat
    if there is edge out of curr_state with
            label in_ch into next_state
        curr_state = next_state
        in_ch = next char of input
    otherwise
        stuck // error condition
until stuck or input string is consumed
if entire string is consumed and
     curr_state is a final state
    accept string
otherwise
    reject string
```

- follow transition

Formalizing finite-state machines

alphabet (Σ) = finite, non-empty set of elements called **symbols**

string over Σ = finite sequence of symbols from Σ

language over Σ = set of strings over Σ

finite state machine $M = (Q, \Sigma, \delta, q, F)$ where

- Q = set of states - Finite
- Σ = alphabet - Finite
- δ = state transition function $Q \times \Sigma \to Q$ given (state, symbol), return state
- q = start state - only 1, $q \in Q$
- F = set of accepting (or final) states $F \subseteq Q$

L(M) = the language of FSM M = set of all strings M accepts - can be infinite

finite automata M **accepts** $x = x_1x_2x_3...x_n$ iff

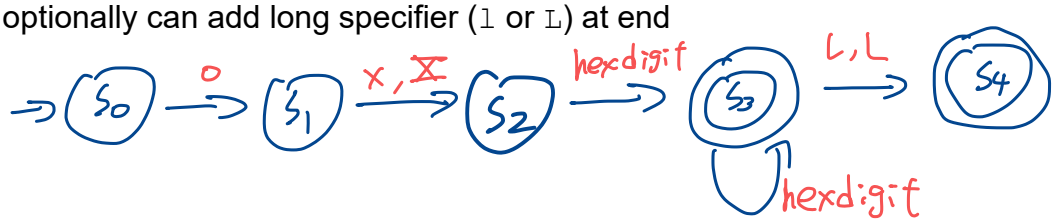
$$\delta(\delta(\delta(\ldots\ \delta(\delta(\delta(s_0, x_1), x_2), x_3), \ldots\ x_{n-2}), x_{n-1}), x_n) \in F$$

end in final state

Example 2: hexadecimal integer literals in Java

Hexadecimal integer literals in Java:

- must start 0x or 0X ← number 0 (not letter capital-O)
- followed by at least one hexadecimal digit (hexdigit)
  - hexdigit = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f, A, B, C, D, E, F
- optionally can add long specifier (`l` or `L`) at end



s_e (error state)

$Q = \{s_0, s_1, s_2, s_3, s_4\}$

$\Sigma = \{0\text{-}9, a\text{-}f, A\text{-}F, x, X, l, L\}$

$\delta =$ use state transition table

$q = s_0$

$F = \{s_3, s_4\}$

Example of
accepted: 0x1f2b7
stuck in start: 7
stuck in final state
but not accepted:
0x1Lk

State transition table

	0	1 – 9	a – f	A – F	x	X	l	L
s_0	s_1	s_e	s_e	s_e	s_e	s_e	s_e	s_e
s_1					s_2	s_2		
s_2	s_3	s_3	s_3	s_3				
s_3	s_3	s_3	s_3	s_3			s_4	s_4
s_4	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e
s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e

To handle empty spaces, create error state s_e

Coding a state transition table

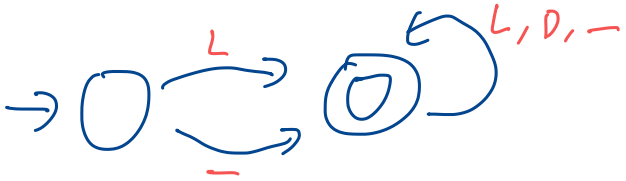
```
curr_state = start_state
done = false
while (!done)
    ch = nextChar()
    next = transition[curr_state][ch]
    if (next == error || ch == EOF)
        done = true
    else
        curr_state = next

return final_states.contains(curr_state) && next != error
```

Works provides FSM is deterministic

Example 3: identifiers in C/C++

A C/C++ identifier L D _

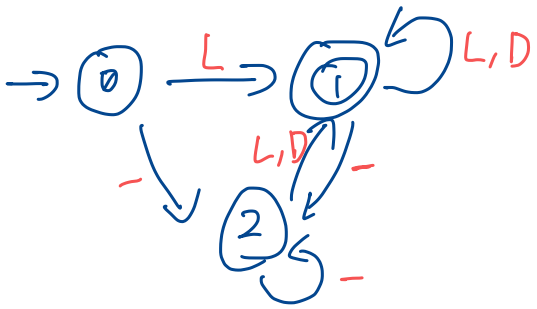
- is a sequence of one or more letters, digits, underscores
- cannot start with a digit



Legal but odd:

_ __ _0_

Add restriction: can't end in underscore



DFA NFA

Deterministic vs non-deterministic FSMs

deterministic

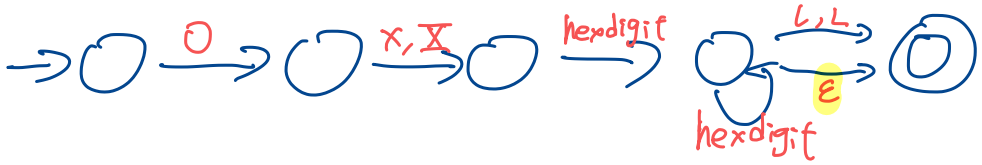
- no state has >1 outgoing edge with same label
- edges can only be labelled with elements of Σ

non-deterministic

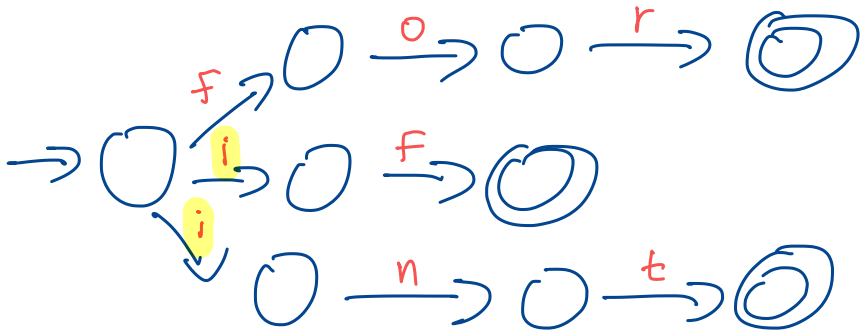
- states may have multiple outgoing edges with same label
- edges may be labelled with special symbol ε (empty string)

ε -transitions can happen without reading input

Example 2 (revisited): hexadecimal integer literals in Java



Example 4: FSM to recognize keywords `for`, `if`, `int`



Recap

- The scanner reads a stream of characters and tokenizes it (i.e., finds tokens)
- Tokens are defined using regular expressions
- Scanners are implemented using (deterministic) FSMs
- FSMs can be non-deterministic

CS 536 Announcements for Wednesday, February 1, 2023

Course websites:

`pages.cs.wisc.edu/~hasti/cs536`
`www.piazza.com/wisc/spring2023/compsci536`

Programming Assignment 1

- test code due Friday, Feb. 3 by 11:59 pm
- other files due Tuesday, Feb. 7 by 11:59 pm

Last Time

- start scanning
- finite state machines
  - formalizing finite state machines
  - coding finite state machines
  - deterministic vs non-deterministic FSMs

Today

- non-deterministic FSMs
- equivalence of NFAs and DFAs
- regular languages
- regular expressions

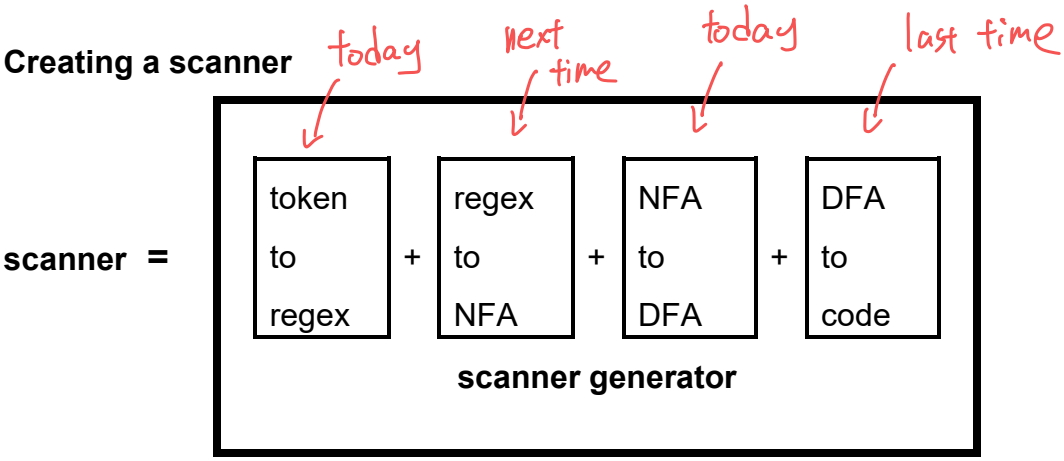
Next Time

- regular expressions → DFAs
- language recognition → tokenizers
- scanner generators
- JLex

Recall

- scanner : converts a sequence of characters to a sequence of tokens
- scanner implemented using FSMs
- FSMs can be DFA or NFA

Creating a scanner



NFAs, formally

finite state machine $M = (Q, \Sigma, \delta, q, F)$

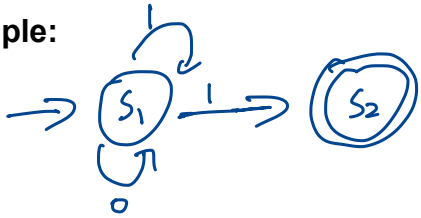
finite set of states — alphabet (symbol - characters) — transition function: $\delta : Q \times \Sigma \Rightarrow \mathcal{P}(Q)$ — start state $q \in Q$ — final states $F \in Q$

$\mathcal{P}()$ = power set = set of all subsets

↳ Q if DFA

L(*M*) = the language of FSM *M* = set of all strings *M* accepts

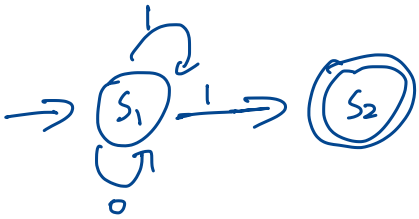
Example:



	0	1
S_1	$\{S_1\}$	$\{S_1, S_2\}$
S_2	$\{\}$	$\{\}$

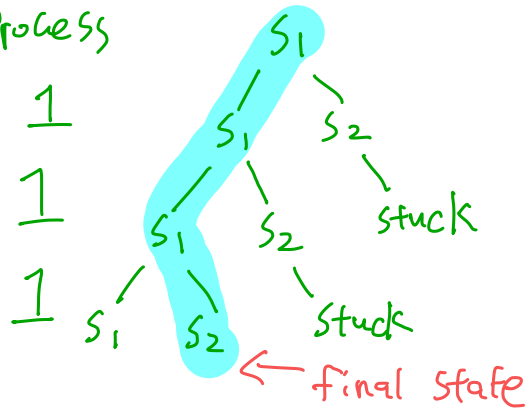
"Running" an NFA

To check if a string is in L(*M*) of NFA *M*, simulate set of choices it could make.



Input : 1 1 1

Process



The string is in L(M) iff there is at least one sequence of transitions that

- consumes all input (without getting stuck)
- ends in one of the final states

NFA and DFA are equivalent

Two automata M and M^* are equivalent iff $L(M) = L(M^*)$

Lemmas to be proven:

Lemma 1: Given a DFA M, one can construct an NFA M^* that recognizes the same language as M, i.e., $L(M^*) = L(M)$

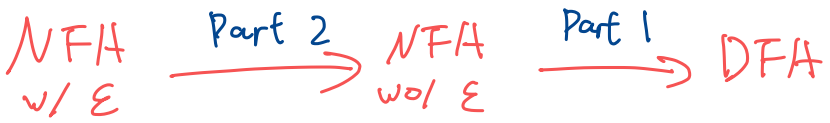
Lemma 2: Given an NFA M, one can construct a DFA M^* that recognizes the same language as M, i.e., $L(M^*) = L(M)$

Proving Lemma 2

Lemma 2: Given an NFA M, one can construct a DFA M^* that recognizes the same language as M, i.e., $L(M^*) = L(M)$

Part 1: Given an NFA M without ε-transitions, one can construct a DFA M^* that recognizes the same language as M

Part 2: Given an NFA M with ε-transitions, one can construct a NFA M^* without ε-transitions that recognizes the same language as M



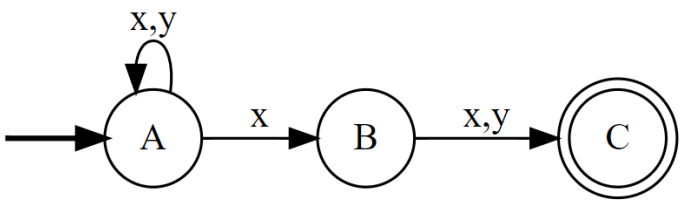
NFA without ε-transitions to DFA

Observation: we can only be in finitely many subsets of states at any one time

Idea: to do NFA *M* → DFA *M**, use a single state in *M** to simulate sets of states in *M*

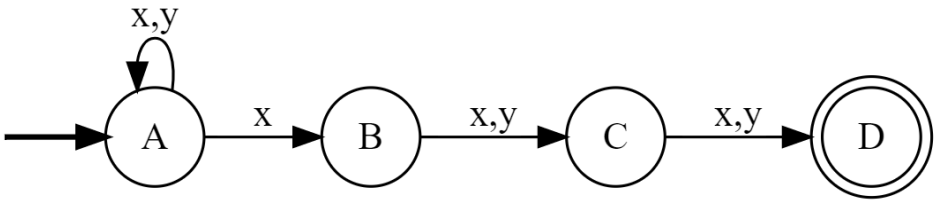
Suppose *M* has |Q| states. Then *M** can have only up to $2^{|Q|}$ states.

Why?



A	B	C	
0	0	0	= { }
0	0	1	= {C}
0	1	0	= {B}
0	1	1	= {B, C}
1	0	0	= {A}
1	0	1	= {A, C}
1	1	0	= {A, B}
1	1	1	= {A, B, C}

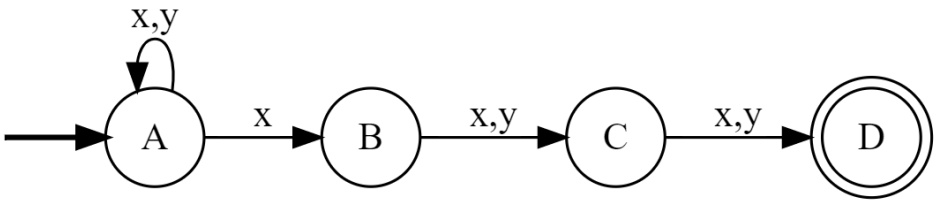
Example



	x	y
A	{A, B}	{A}
B	{C}	{C}
C	{D}	{D}
D	{ }	{ }

NFA without ε-transitions to DFA

Given NFA *M*:

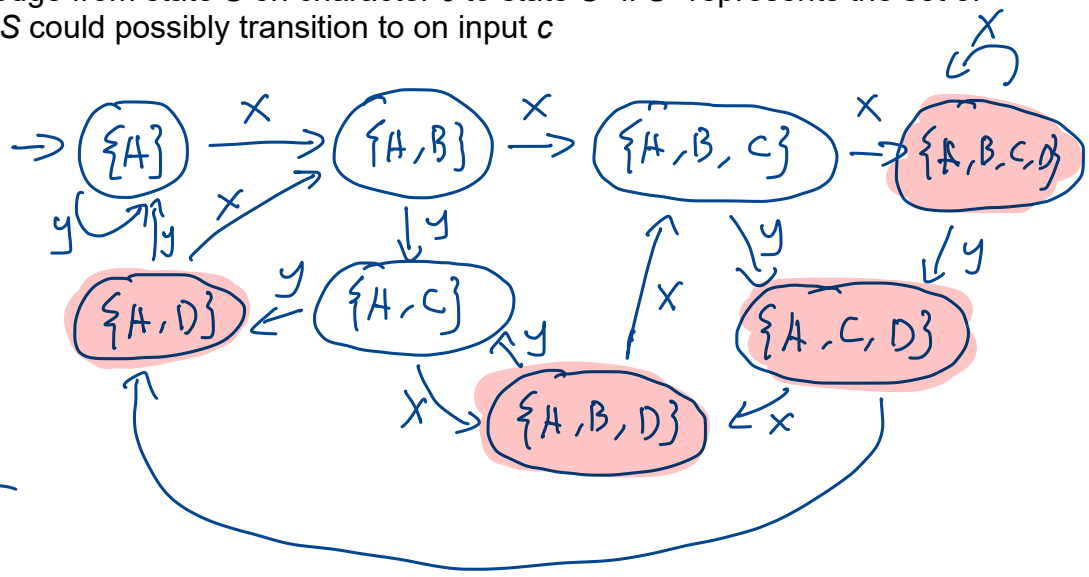


Build new DFA *M** where $Q^* \subseteq \mathcal{P}(Q)$ $|Q^*| \leq 2^{|Q|}$

To build DFA: Add an edge from state *S* on character *c* to state *S** if *S** represents the set of all states that a state in *S* could possibly transition to on input *c*

	x	y
A	{A,B}	{A}
B	{C}	{C}
C	{D}	{D}
D	{}	{}

Final state in M



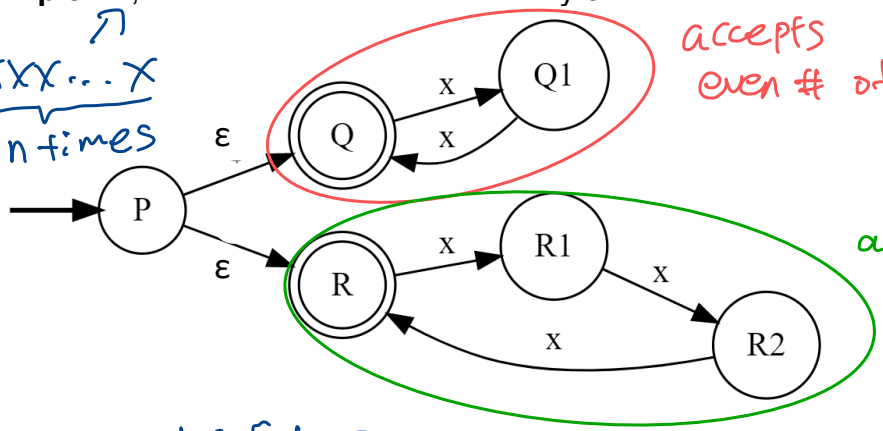
Any state whose subset contains a final state of M is final state in M*

Part I: NFA w/o ε ↓ DFA ✓

ε-transitions

Example: x^n, where *n* is even or divisible by 3

xxx...x n times



accepts even # of x's

accepts #s of x's divisible by 3.

Useful for taking union of 2 FSMs.

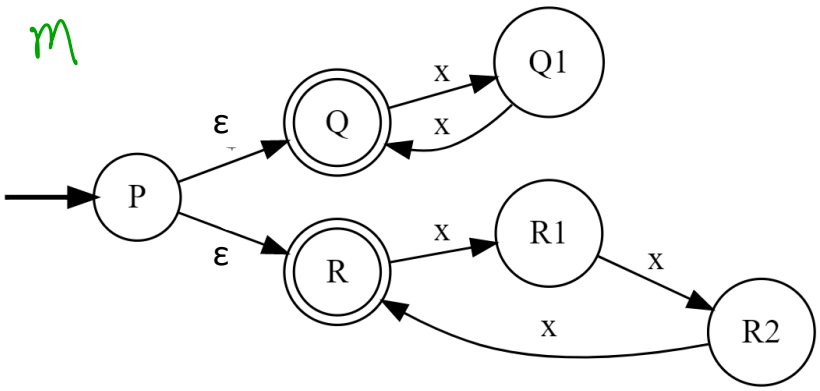
Eliminating ε-transitions

Goal: given NFA *M* with **ε**-transitions, construct an **ε**-free NFA *M** that is equivalent to *M*

Definition: *epsilon closure*

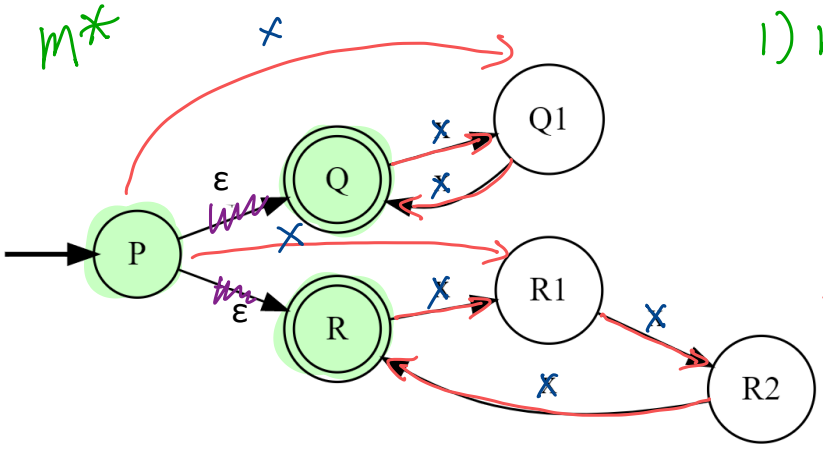
eclose(*s*) = set of all states reachable from *s* using 0 or more epsilon transitions

M



	eclose
P	{P, Q, R}
Q	{Q}
R	{R}
Q1	{Q1}
R1	{R1}
R2	{R2}

M*



1) make S an accepting state of M* iff eclose(S) contains an accepting state of M

2) Add an edge from S to T labelled a iff ∃ edge in M labelled a for some state in eclose(S) to T.

3) Delete all edges labelled with epsilon

✓ Part 2: NFA w/ ε → NFA w/o ε

✓ lemma: NFA → DFA

Summary of FSMs

DFAs and NFAs are equivalent

- an NFA can be converted into a DFA, which can be implemented via the table-drive approach

ε-transitions do not add expressiveness to NFAs

- algorithm to remove ε-transitions

Regular Languages and Regular Expressions

Regular language

Any language recognized by an FSM is a *regular language*

Examples:

- single-line comments beginning with //
- hexadecimal integer literals in Java
- C/C++ identifiers
- {ε, ab, abab, abababab, abababab, …}

↑ empty string

Regular expression (regex)

= a pattern that defines a regular language

regular language: set of (potentially infinite) strings

regular expression: represents a set of (potentially infinite) strings by a single pattern

Example: {ε, ab, abab, abababab, abababab, …} ←→ (ab)*

Why do we need them?

- Each token in a programming language can be defined by a regular language
- Scanner-generator input = one regular expression for each token to be recognized by the scanner

→ regexs are inputs to scanner generator

Formal definition

A **regular expression** over an alphabet Σ is any of the following:

- ∅ (the empty regular expression)
- ε
- *a* (for any *a* ∈ Σ)

Moreover, if R_1 and R_2 are regular expressions over Σ, then so are: $R_1 \mid R_2$, $R_1 \cdot R_2$, R_1^*

Regular expressions (as an expression language)

regular expression = pattern describing a set of strings

operands: single characters, epsilon ε

operators:

alternation ("or"): a | b matches a, matches b

concatenation ("followed by"): a.b ab matches ab

Catenation

iteration ("Kleene star"): a* 0 or more a's

Kleene closure, closure ↳ ε, a, aa, aaa, ...

Precedence: low ↓ high

analogous to: + , * , ^

Conventions

aa	is a.a
a+	is aa*
letter	is a|b|c|d|…|y|z|A|B|…|Z
digit	is 0|1|2|…|9
not(x)	is all characters except x

parentheses for grouping and overriding precedence, e.g., (ab)* ≠ ab* ≡ a(b*)

Example: single-line comments beginning with //

// not('\n')* '\n'

↳ new line

Example: hexadecimal integer literals in Java

- must start 0x or 0X
- followed by at least one hexadecimal digit (hexdigit)
  - hexdigit = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f, A, B, C, D, E, F
- optionally can add long specifier (`l` or `L`) at end

let hexdigit = digit | a | b | c | ... | f | A | B | C | ... | F

0(x | X) hexdigit+ (ε | L | l)

Example: C/C++ identifiers (with one added restriction)

- sequence of letters/digits/underscores
- cannot begin with a digit
- cannot end with an underscore

CS 536 Announcements for Monday, February 6, 2023

Programming Assignment 1

- symbol table files due Tuesday, Feb. 7 by 11:59 pm

Homework 0

- available in schedule
- practice with DFAs, regular expressions

Homework 1

- available soon
- practice with NFA→DFA translation, JLex

Last Time

- non-deterministic FSMs
- equivalence of NFAs and DFAs
- regular languages
- regular expressions

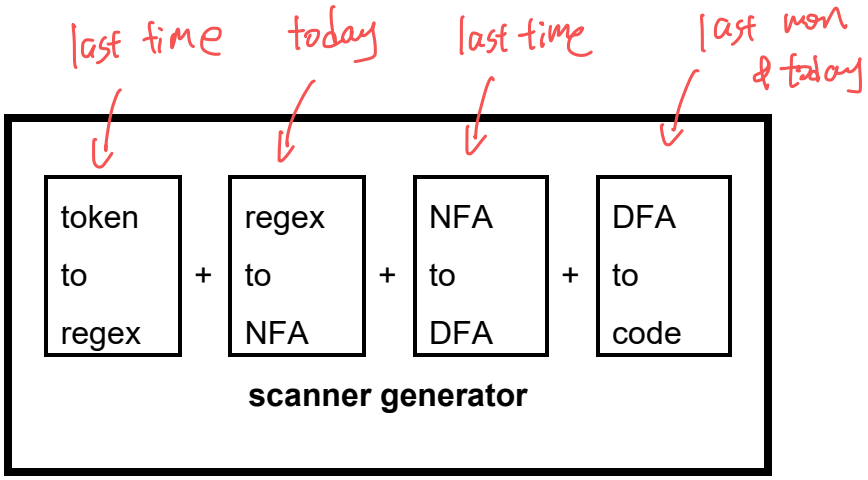
Today

- regular expressions → DFAs
- language recognition → tokenizers
- scanner generators
- JLex

Next Time

- CFGs

Recall



From regular expressions to NFAs

Overview of the process

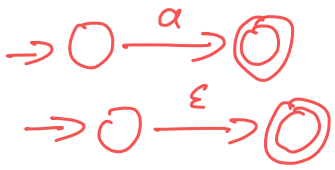
- Conversion of literals and epsilon → simple Finite Automata.
- Conversion of operators
  - Convert operands to NFAs.
  - join NFAs

Regex to NFA rules

Rules for operands

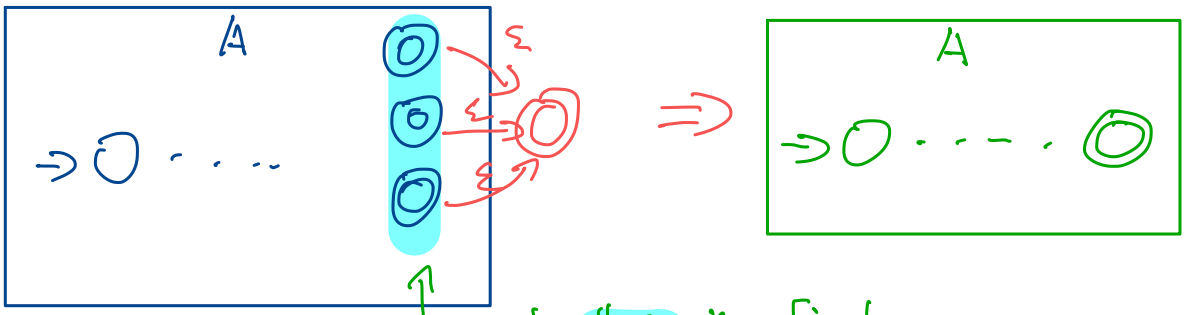
literal 'a'

epsilon ε

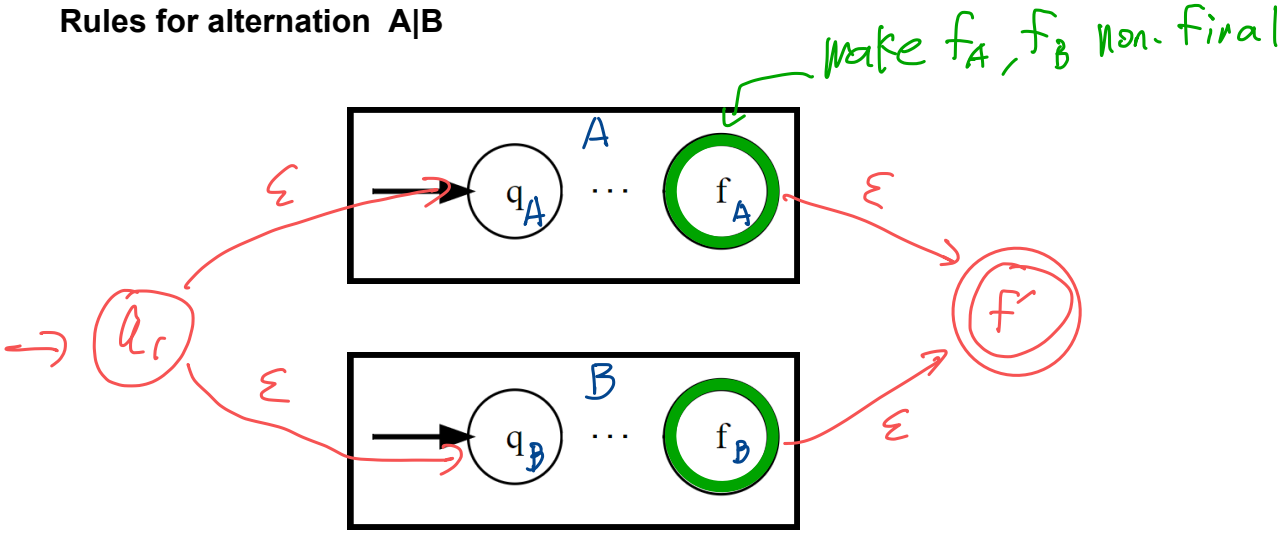


Suppose A is a regex with NFA:

Convert so only 1 final state

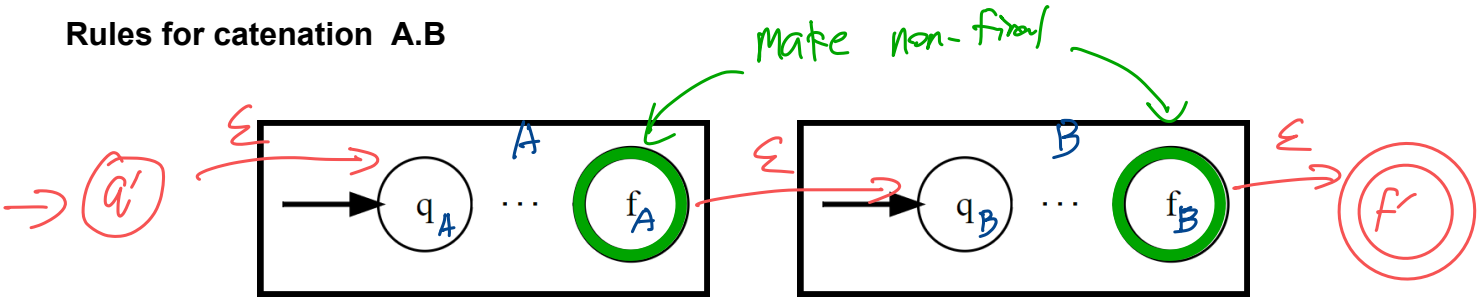


Rules for alternation A|B

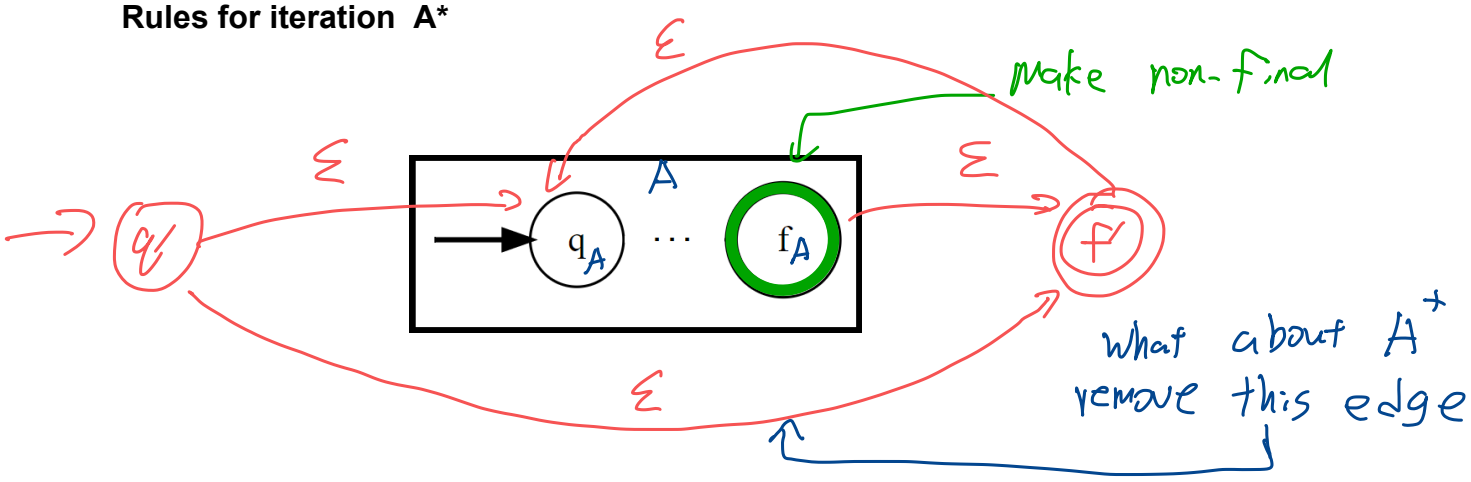


Regex to NFA rules

Rules for catenation A.B

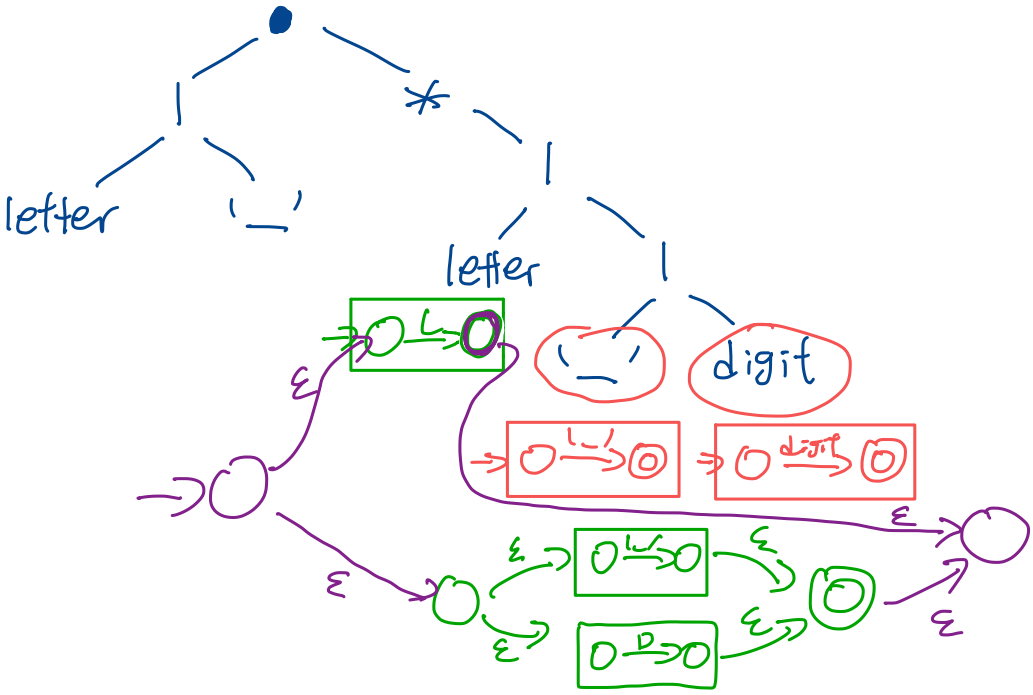


Rules for iteration A*



Tree representation of a regex

Consider regex: (letter | '_') (letter | '_' | digit)*



Regex to DFA

We now can do:

regex → NFA w/ε → NFA w/o ε → DFA

We can add one more step: **optimize DFA**

Theorem: For every DFA *M*, there exists a unique equivalent smallest DFA *M** that recognizes the same language as *M*.

↳ fewest # of states

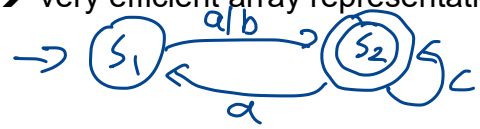
To optimize:

- remove unreachable states — can't get to from start state
- remove dead states — can't get to a final state from it
- merge equivalent states

But what's so great about DFAs?

Recall: state-transition function (δ) can be expressed as a table

➔ very efficient array representation



	a	b	c
s_1	s_2	s_2	
s_2	s_1		s_2

➔ efficient algorithm for running (any) DFA

```
s = start state
while (more input){
    c = read next char
    s = table[s][c]
}
if s is final, accept
else reject
```

What else do we need?

FSMs – only check for language membership of a string

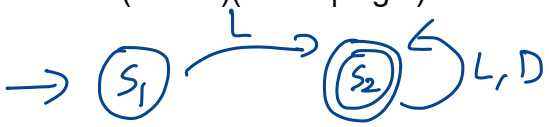
scanner needs to

- recognize a stream of many different tokens using the longest match
- know what was matched

Table-driven DFA → tokenizer

Idea: augment states with actions that will be executed when state is reached

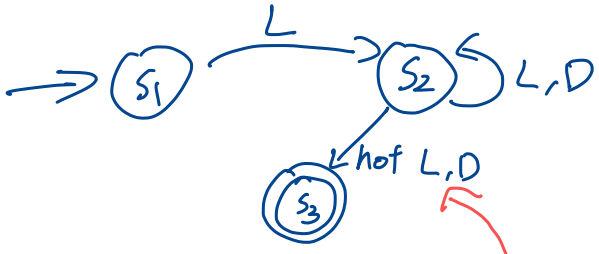
Consider: (letter)(letter | digit)*



state	action
S_2	return ID ← token

avg = 7

Problem: Don't get longest match



state	action
S_3	~~return ID~~

avg = 7

To fix:
Put back 1 char;
Return ID

Problem: maybe we need this char

Actions needed:
- return a token
- Put back a character
- report an error

Also add EOF token, EOF symbol to alphabet Σ

Scanner Generator Example

Language description:
consider a language consisting of two statements

- assignment statements: `ID = expr`
- increment statements: `ID += expr`

where `expr` is of the form:

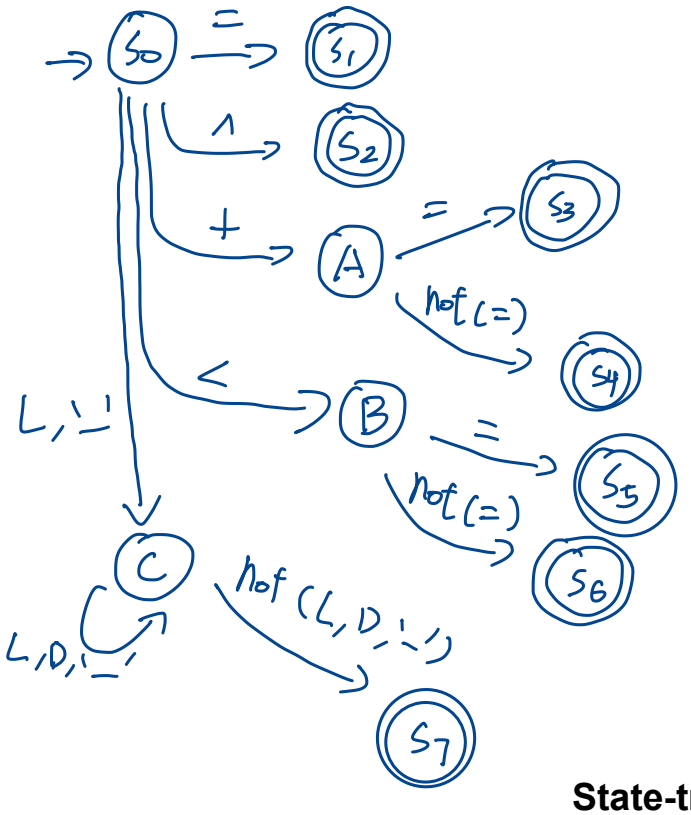
- `ID + ID`
- `ID ^ ID`
- `ID < ID`
- `ID <= ID`

and `ID` are identifiers following C/C++ rules (can contain only letters, digits, and underscores; can't start with a digit)

Tokens:

Token	Regular expression
ASSIGN	"="
INCR	"+="
PLUS	"+"
EXP	"^"
LESSTHAN	"<"
LEQ	"<="
ID	(letter | '_')(letter | '_' | digit)*

Combined DFA



Actions

S_1: return ASSIGN

S_2: return EXP

S_3: return INC

S_4: Put 1 back; return PLUS

S_5: return LEQ

S_6: Put 1 back; return LESSTHAN

S_7: Put 1 back; return ID

State-transition table

	=	+	^	<	_	letter	digit	EOF	none of these
S_0	ret ASSIGN	A	ret EXP	B	C	C		ret EOF	
A	ret INC	put 1 back, ret PLUS							
B	ret LEQ	put 1 back, ret LESSTHAN							
C	put 1 back, ret ID				C	C	C	put 1 back, ret ID	

```
do {
    read char
    perform action / update state
    if (action was to return a token)
        start again in start state
} while not(EOF or stuck)
```

Lexical analyzer generators
(aka scanner generators)

Formally define transformation from regex to scanner

Tools written to synthesize a lexer automatically

- Lex : UNIX scanner generator, builds scanner in C
- Flex : faster version of Lex
- JLex : Java version of Lex

JLex

Declarative specification (non-procedural)

- you don't tell JLex how to scan / how to match tokens
- you tell JLex what you want scanned (tokens) & what to do when a token is matched

Input: set of regular expressions + associated actions

JLex specification — .jlex eg xyz.jlex

Output: Java source code for a scanner

→ xyz.jlex.java

JLex.main + xyz.jlex → compile to ge-

Yylex.class

- Constructer: takes input stream
- next_token: returns next token in input

Format of JLex specification

3 sections separated by %%

- user code section
- directives
- regular expression rules

Regular expression rules section

Format: `<regex>{code}` where `<regex>` is a regular expression for a single token

- can use macros from Directives section – surround with curly braces { }
- characters represent themselves (except special characters)
- characters inside " " represent themselves (except \")
- . matches anything

→ \n \t ^ $

Regular expression operators: | * + ? ( )

1 or more

0 or 1 instances

( ) for grouping

Character class operators: - ^ \

denote closing [ ] matches on char

- range
- ^ not
- \ escape

JLex example

```
// This file contains a complete JLex specification for a very
// small example.

// User Code section:  For right now, we will not use it.

%%

DIGIT=       [0-9]
LETTER=      [a-zA-Z]
WHITESPACE=  [\040\t\n]

%state SPECIALINTSTATE

%implements java_cup.runtime.Scanner
%function next_token
%type java_cup.runtime.Symbol

%eofval{
System.out.println("All done");
return null;
%eofval}

%line

%%

({LETTER}|"_")({DIGIT}|{LETTER}|"_")* {
                        System.out.println(yyline+1 + ": ID "
                                + yytext()); }

"="             { System.out.println(yyline+1 + ": ASSIGN"); }
"+"             { System.out.println(yyline+1 + ": PLUS"); }
"^"             { System.out.println(yyline+1 + ": EXP"); }
"<"             { System.out.println(yyline+1 + ": LESSTHAN"); }
"+="            { System.out.println(yyline+1 + ": INCR"); }
"<="            { System.out.println(yyline+1 + ": LEQ"); }
{WHITESPACE}*   { }
.               { System.out.println(yyline+1 + ": bad char"); }
```

Using scanner generated by JLex in a program

```
// inFile is a FileReader initialized to read from the
// file to be scanned
Yylex scanner = new Yylex(inFile);
try {
    scanner.next_token();
} catch (IOException ex) {
    System.err.println(
              "unexpected IOException thrown by the scanner");
    System.exit(-1);
}
```

CS 536 Announcements for Wednesday, February 8, 2023

Programming Assignment 2

- released today
- due Wednesday, February 22

Last Time

- regular expressions → DFAs
- language recognition → tokenizers
- scanner generators
- JLex

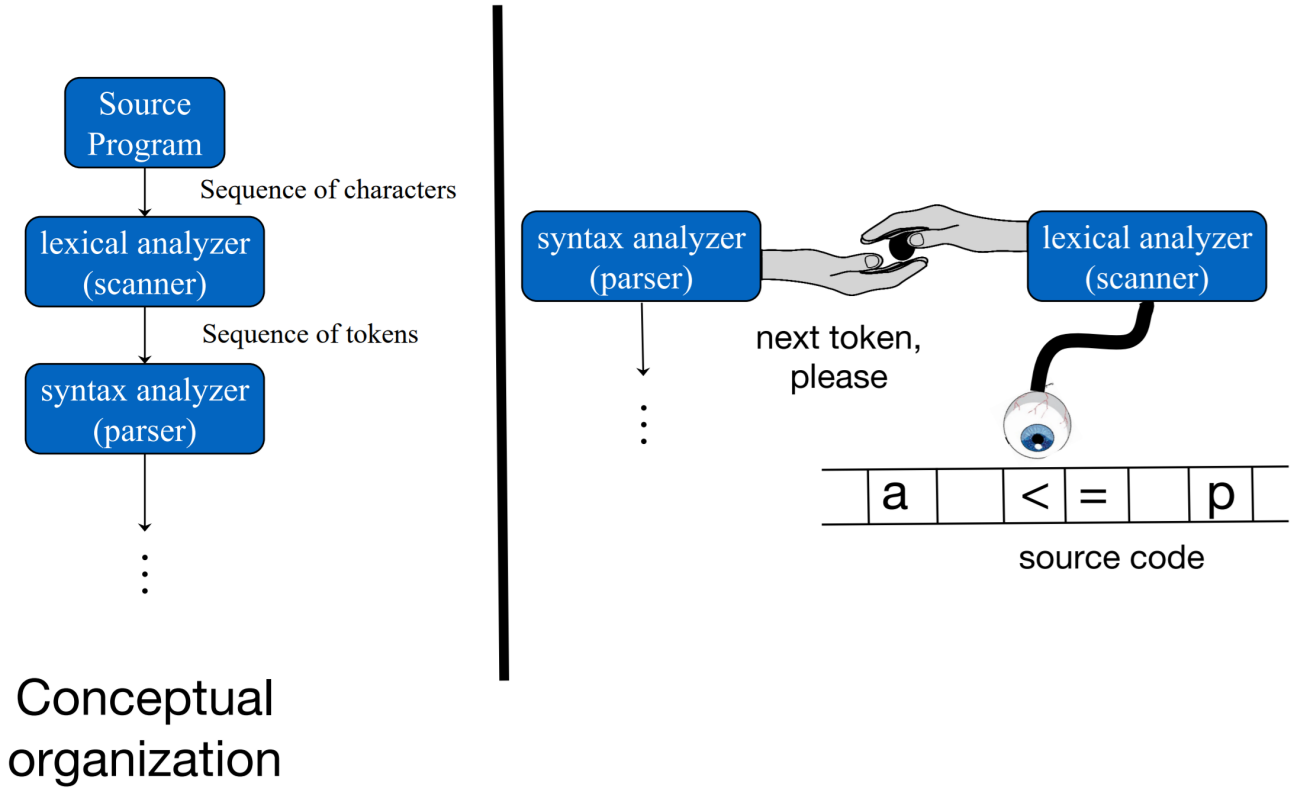
Today

- CFGs

Next Time

- CFG ambiguity

Recall big picture



Conceptual organization

Why regular expressions are not good enough

Regular expression wrap-up

+ perfect for tokenizing a language

– limitations

- define only limited family of languages
  - can't be used to specify all the programming constructs we need
- no notion of structure

Regexs cannot handle "matching"

Example: $L_{()} = \{ (^n)^n \text{ where } n > 0\}$ $= \{"()", "(())", "((()))", \dots\}$

Theorem: No regex/DFA can describe the language $L_{()}$

Proof by contradiction: Suppose there exists a DFA A for $L_{()}$ where A has N states.

Then A has to accept the string $(^N)^N$ with some sequence of states

$q_0 \xrightarrow{(} q_1 \xrightarrow{(} q_2 \xrightarrow{(} \dots \xrightarrow{(} q_n \xrightarrow{)} q_{N+1} \xrightarrow{)} q_{N+2} \xrightarrow{)} \dots \xrightarrow{)} q_{2N+1}$

accept $(^N$ accept $)^N$

By the pigeonhole principle, there exists $i, j \le N$ where $i < j$ such that $q_i = q_j$

So

$q_0 \xrightarrow{(} q_1 \xrightarrow{(} \dots q_i \xrightarrow{(} q_{j+1} \xrightarrow{(} \dots \xrightarrow{(} q_N \xrightarrow{)} q_{N+1} \xrightarrow{)} q_{N+2} \xrightarrow{)} \dots \xrightarrow{)} q_{2N+1}$

accept $(^{i+(N-j)}$ accept $)^N$

In other words, A accepts $(^{N-j+i})^N$ but $(^{N-j+i})^N \notin L_{()}$

which is a contradiction

No notion of structure

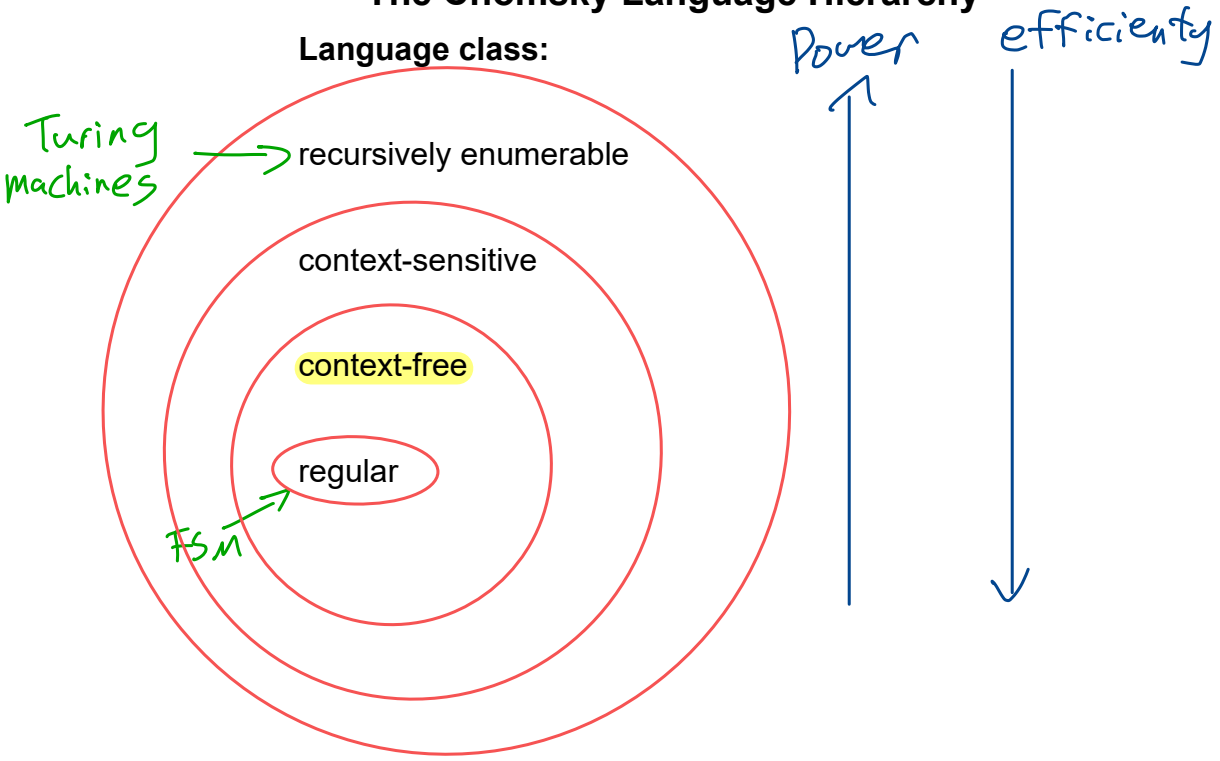
$x = y + z$

Consider the following stream of tokens: ID ASSIGN ID PLUS ID

What should be done 1st? $(x=y)+z$ or $x=(y+z)$

What about precedence & associativity?

The Chomsky Language Hierarchy



Context-free grammar (CFG)

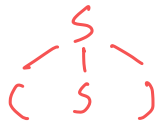
= a set of recursive rewriting rules to generate patterns of strings

S → '(' S ')'

rule for rewriting S

Before applying: S

After applying:



Formal definition: A CFG is a 4-tuple (N, ∑, P, S)

- N = set of **non-terminals** - placeholders (interior nodes in parse tree)
- ∑ = set of **terminals** - tokens from scanner
- P = set of **productions** - rules for re-writing non-terminals. (for deriving)
- S = initial non-terminal symbol ("start symbol"), S ϵ N
  - if not otherwise specified, use non-term on LHS of 1st production as start symbol.

Productions

Production syntax : LHS → RHS

single non-terminal ← expression (seq of terms & non-terms) or ε

non-term → expression S → '(' S ')'

non-term → ε S → ε

or (assuming non-terms on LHSs are the same)

non-term → expression | ε S → '(' S ')' | ε

non-term → expression | ε S → '(' S ')' | ε

Language defined by a CFG

= set of strings (i.e., sequences of terminals) that can be derived from the start non-terminal

To derive a string (of terminal symbols):

- set Curr_Seq to start symbol
- repeat
  - find a non-terminal x in Curr_Seq
  - find production of the form x → α
  - "apply" production: create new Curr_Seq by replacing x with α
- until Curr_Seq contains no non-terminals

Derivation notation

- derives $\Rightarrow$
- derives in one or more steps $\Rightarrow^+$ or $\overset{+}{\Rightarrow}$
- derives in zero or more steps $\Rightarrow^*$ or $\overset{*}{\Rightarrow}$

L(G) = language defined by CFG G

= $\{w \mid S \overset{+}{\Rightarrow} w$ where S is start non-term & w is a seq of term or $\varepsilon\}$

Example grammar

Terminals

BEGIN
END — Program boundry

SEMICOLON — ";" to seperate statements

ASSIGN — "=" in assignment statements

ID — identifier (variable name)

PLUS — "+" operator in expressions.

Non-terminals

prog — start non-terminal (root of parse tree)

stmts — list of statements

stmt — a single statement

expr — a (mathmatical) expression

Productions — define syntax of legal programs

1) prog → BEGIN stmts END

2) stmts → stmts SEMICOLON stmt
3) | stmt

4) stmt → ID ASSIGN expr

5) expr → ID
6) | expr PLUS ID

↑ #s for refrence only

Example derivation

Productions

1) prog → BEGIN stmts END
2) stmts → stmts SEMICOLON stmt
3) | stmt
4) stmt → ID ASSIGN expr
5) expr → ID
6) | expr PLUS ID

using production

prog -> BEGIN stmts END (1)
-> BEGIN stmts SEMICOLON stmt END (2)
-> BEGIN stmt SEMICOLON stmt END (3)
-> BEGIN ID ASSIGN expr SEMICOLON stmt END (4)
-> BEGIN ID ASSIGN expr SEMICOLON ID ASSIGN expr END (4)
-> BEGIN ID ASSIGN ID SEMICOLON ID ASSIGN expr END (5)
-> BEGIN ID ASSIGN ID SEMICOLON ID ASSIGN expr PLUS ID END (6)
-> BEGIN ID ASSIGN ID SEMICOLON ID ASSIGN ID PLUS ID END (5)

or BEGIN ID = ID ; ID = ID + ID END

Prog $\stackrel{+}{\Rightarrow}$ BEGIN ID = ID ; ID = ID + ID END

Is BEGIN ID = ID ; END a valid program?

No but it would be if we add production

stmt → ε

& then BEGIN END would also be a valid program

Parse trees

= way to visualize a derivation

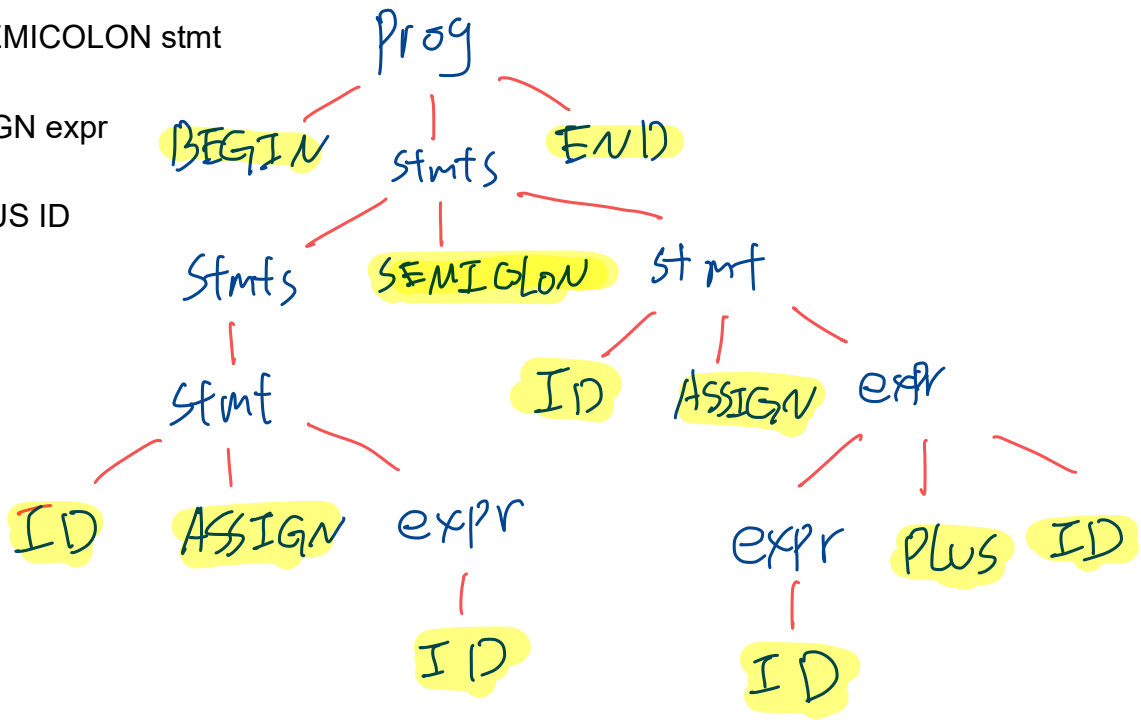
To derive a string (of terminal symbols):

- set root of parse tree to start symbol
- repeat
  - find a leaf non-terminal x
  - find production of the form x → α
  - "apply" production: symbols in α become the children of x
- until there are no more leaf non-terminals

Derived sequence determined from leaves, from left to right

Productions

1) prog → BEGIN stmts END
2) stmts → stmts SEMICOLON stmt
3) | stmt
4) stmt → ID ASSIGN expr
5) expr → ID
6) | expr PLUS ID



CS 536 Announcements for Monday, February 13, 2023

Programming Assignment 2 – due Wednesday, February 22

Last Time

- why regular expressions aren't enough
- CFGs
  - formal definition
  - examples
  - language defined by a CFG
- parse trees

Today

- Makefiles
- resolving ambiguity
- expression grammars
- list grammars

Next Time

- syntax-directed translation

Makefiles

Basic structure

```
<target>: <dependency list>
     <command to satisfy target)
```

Example

```
Example.class: Example.java IO.class
    javac Example.java
IO.class: IO.java
    javac IO.java
```

Make creates an internal **dependency graph**

- a file is rebuilt if one of its dependencies changes

Variables – for common configuration values to use throughout your makefile

Example

```
JC = /s/std/bin/javac
JFLAGS = -g

Example.class: Example.java IO.class
    $(JC) $(JFLAGS) Example.java

IO.class: IO.java
    $(JC) $(JFLAGS) IO.java
```

Phony targets

- target with no dependencies = "phony"
- use make to run commands:

Example

force remove

```
clean:
	rm -f *.class
```

test:

```
java Example inFile1.txt outFile2.txt
java Example inFile2.txt outFile2.txt
```

Programming Assignment 2

Modify:

also modify allTokens.in

- brevis.jlex
- P2.java
- Makefile

Makefile

```
###
# testing - add more here to run your tester and compare
# its results to expected results
###
test:
	java -cp $(CP) P2
	diff allTokens.in allTokens.out

###
# clean up
###

clean:
	rm -f *~ *.class brevis.jlex.java

cleantest:
	rm -f allTokens.out
```

Run make to compile P2
(by default make does 1st target in Makefile)

Running the tester

```
royal-12(53)% make test
java -cp ./deps:. P2
3:1 ****ERROR**** ignoring illegal character: a
diff allTokens.in allTokens.out
3d2
< a
make: *** [Makefile:40: test] Error 1
```

classpath

error msg produced by brevis scanner when P2 is run.

-> output of diff command.

<- from make

CFG review

formal definition: CFG $G = (N, \Sigma, P, S)$ — terminals = tokens

CFG generates a string by applying productions until no non-terminals remain

$\Rightarrow+$ means "derives in 1 or more steps"

language defined by a CFG G

$L(G)$ = { w | s $\Rightarrow+$ w} where

s = start is the start non-terminal of G, an

w = sequence consisting of (only) terminal symbols or ε

Example: nested parens

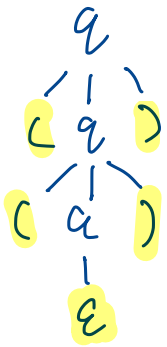
$N = \{q\}$

$\Sigma = \{(,)\}$

$P = q \to (q) \mid \varepsilon$

$S = q$

$L(G) = \{\varepsilon, (), [()], [[()]], \ldots\}$



parse tree

$q \Rightarrow (q) \Rightarrow (\varepsilon)$

$q \Rightarrow (\varepsilon)$

$q \Rightarrow (q) \Rightarrow ((q)) \Rightarrow ((\varepsilon))$

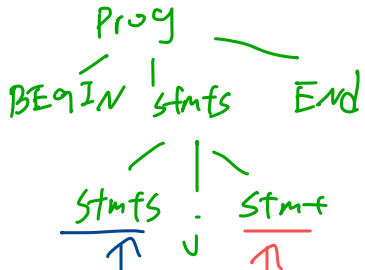
Derivation order

1) prog → BEGIN stmts END

2) stmts → stmts SEMICOLON stmt
3) | stmt

4) stmt → ID ASSIGN expr

5) expr → ID
6) | expr PLUS ID



Leftmost derivation : left most non-terminal is always expanded

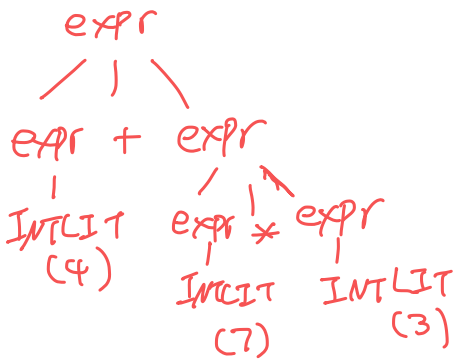
Rightmost derivation : rightmost non-terminal is always expanded

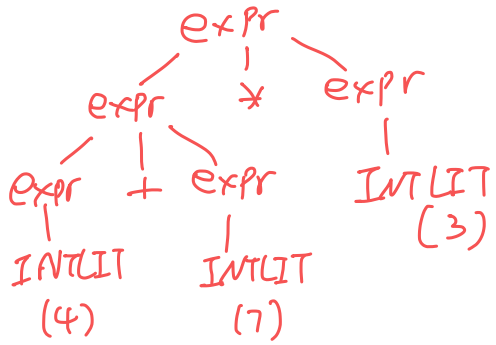
Expression Grammar Example

1) expr → INTLIT
2) | expr PLUS expr
3) | expr TIMES expr
4) | LPAREN expr RPAREN

Goal: Create a CFG for arithmetic expressions involving only +, *, parens id integer literals.

Derive: 4 + 7 * 3





ambiguous grammar!

For grammar G and string w, G is **ambiguous** if there is

> 1 left most derivation of w
OR
> 1 right most derivation of w
OR
> 1 parse tree for w

these are all equivalent

Grammars for expressions

Goal: write a grammar that correctly reflects precedences and associativities

a + b * c → a + (b * c) ↳ a+b+c ⟷ (a+b)+c

a = b = c ⟷ a = (b = c)

Precedence

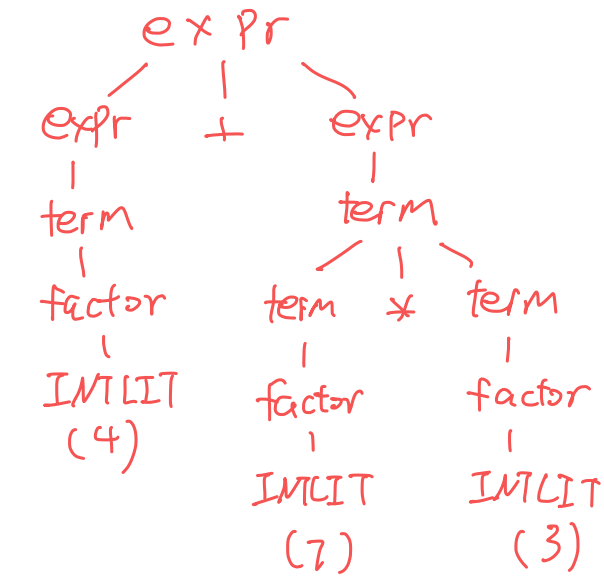
- use different non-terminal for each precedence level
- start by re-writing production for lowest precedence operator first

Example

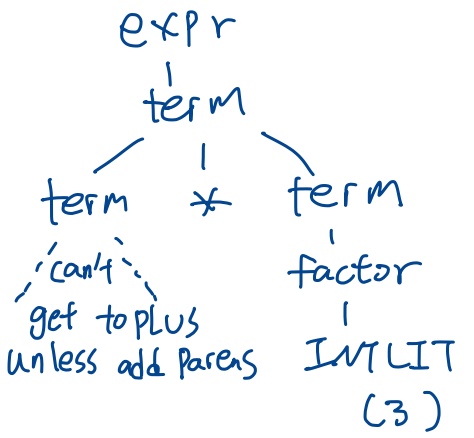
1) expr → INTLIT
2) | expr PLUS expr ← + has lowest precedence
3) | expr TIMES expr
4) | LPAREN expr RPAREN

expr → expr + expr
| term
term → term * term
| factor
factor → INTLIT
| (expr)

4 + 7 * 3



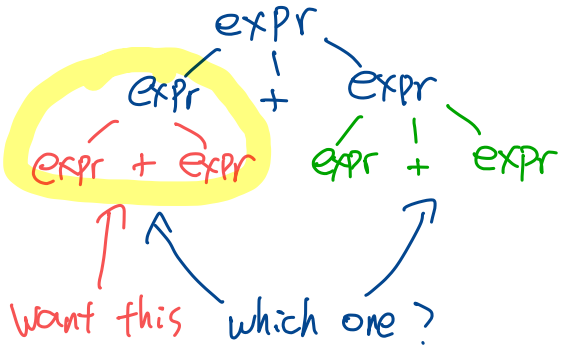
Try to get * eval'd last



Grammars for expressions (cont.)

What about associativity? Consider 1 + 2 + 3

equiv to (1+2)+3



ambiguous grammar

left associative:
+ − * /

right associative:
= ^

2 ^ 3 ^ 4 ≡ 2^{3^4}

Definition: recursion in grammars

A grammar is ***recursive*** **in non-terminal** ***x*** if
x ⟹+ α *x* γ for non-empty strings of symbols α and γ

A grammar is ***left-recursive*** **in non-terminal** ***x***
if *x* ⟹+ *x* γ for non-empty string of symbols γ

A grammar is ***right-recursive*** **in non-terminal** ***x*** if
x ⟹+ α *x* for non-empty string of symbols α

In expression grammars

for left associativity, use left recursion

for right associativity, use right recursion

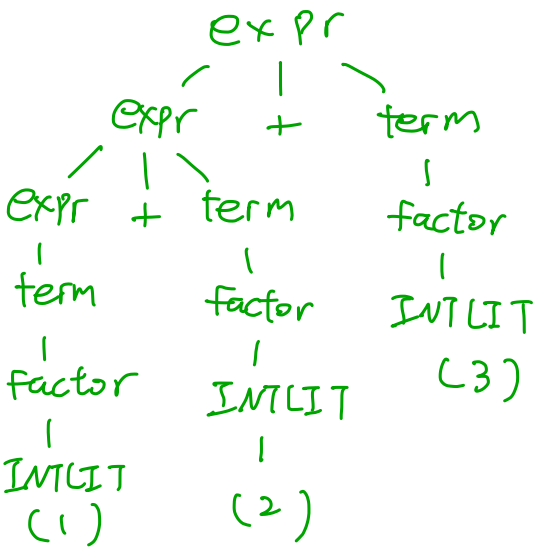
Example

expr → expr PLUS ~~expr~~ term
| term

term → term TIMES factor
| factor

factor → INTLIT
| LPAREN expr RPAREN

1 + 2 + 3



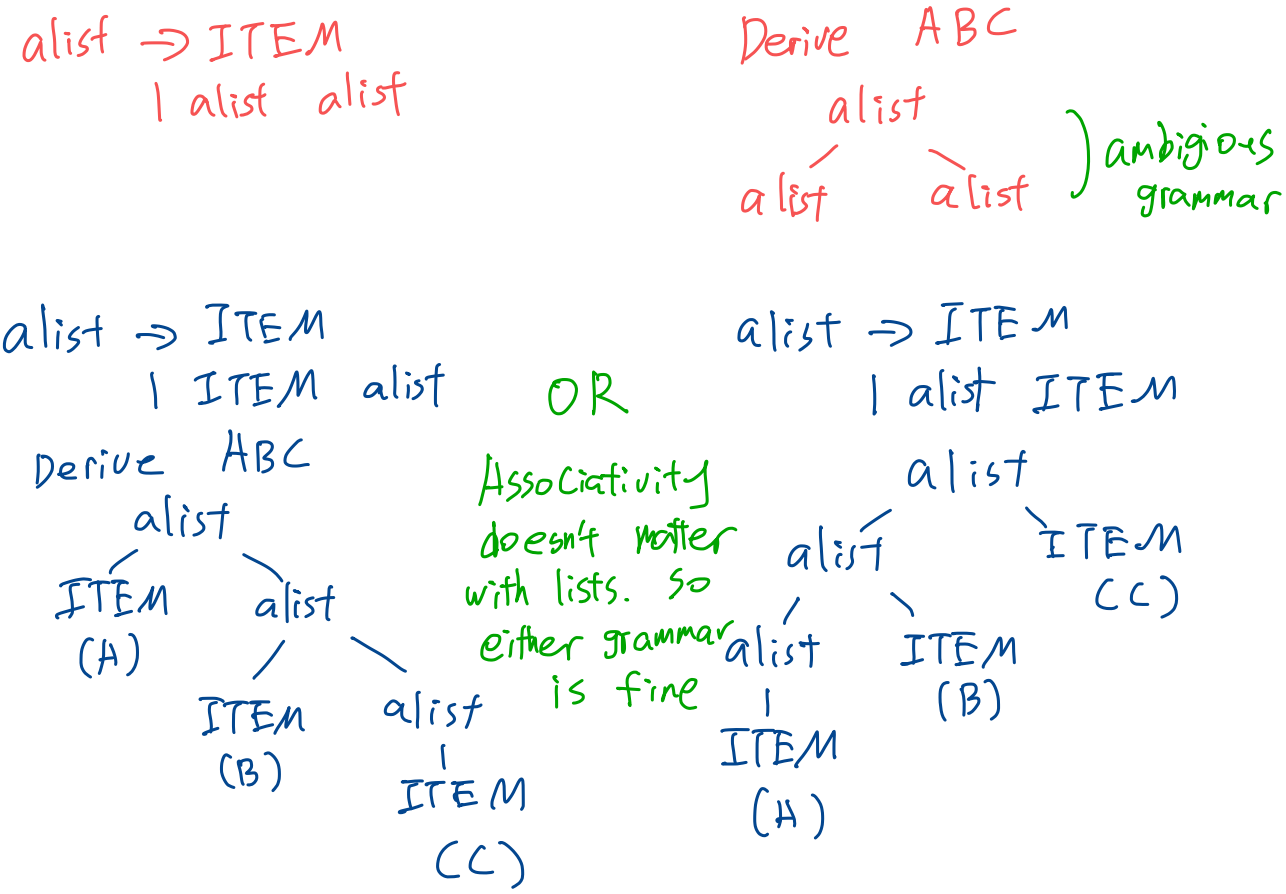
You try: add exponentiation (POW)

Add exponentiation (POW) to this grammar, with the correct precedence and associativity.

```
expr    → expr PLUS term
        | term
term    → term TIMES factor
        | factor
factor  → INTLIT
        | LPAREN expr RPAREN
```

List grammars

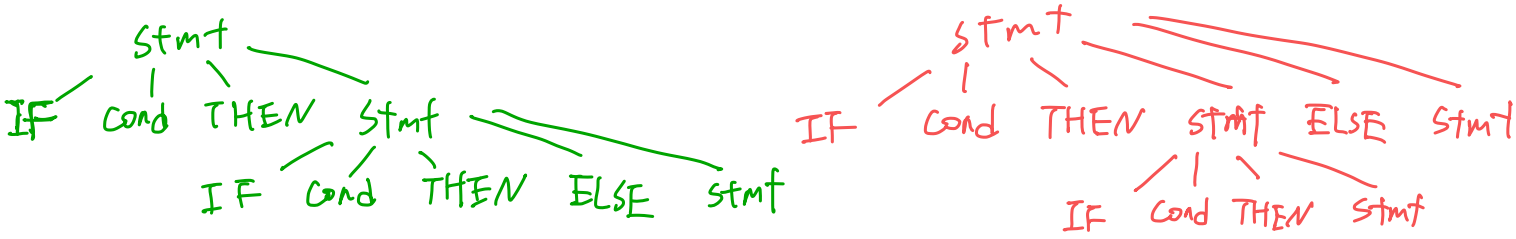
Example a list with no separators, e.g., A B C D E F G



Another ambiguous example

stmt → IF cond THEN stmt
| IF cond THEN stmt ELSE stmt
| . . .

Given this word in this grammar: **`if`** a **`then`** **`if`** b **`then`** s1 **`else`** s2
How would you derive it?



CS 536 Announcements for Wednesday, February 15, 2023

Programming Assignment 2

- due Wednesday, February 22

Last Time

- Makefiles
- ambiguous grammars
- grammars for expressions
  - precedence
  - associativity
- grammars for lists

Today

- syntax-directed translation
- intro to abstract syntax trees

Next Time

- implementing ASTs

Recall our expression grammar

Write an unambiguous grammar for integer expressions involving only addition, multiplication, and parentheses thate correctly handles precedence and associativity.

```
expr   → expr PLUS term
       |  term

term   → term TIMES factor
       |  factor

factor → INTLIT
       |  LPAREN expr RPAREN
```

this notation for CFG is called BNF or extended BNF (Backus-Naur Form)

exponentiation
- has highest precedence
- right associative

$2 \uparrow 3 \uparrow 4 = 2^{3^4}$

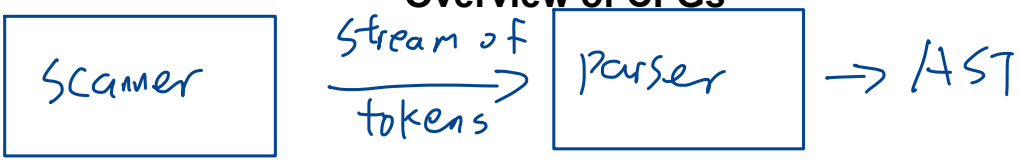
$2 \uparrow (3 \uparrow 4)$

Extend this grammar to add exponentiation (POW)

Add exponentiation (POW) to this grammar, with the correct precedence and associativity.

```
factor → exponent POW factor   ← for associativity
       |  exponent

exponent → INTLIT
         |  LPAREN expr RPAREN
```

Overview of CFGs



CFGs for language definition

- the CFGs we've discussed can generate/define languages of valid strings

start by building parse tree & end with some valid string $w \in L(G)$

CFGs for language recognition

start with string w & end with yes/no answer depending on whether $w \in L(G)$

CFGs for parsing

start with string w & end with parse tree for w if $w \in L(G)$

generally use AST instead of parse tree

- need to translate sequence of tokens (w)

Syntax-directed translation (SDT)

= translating from a sequence of tokens into a sequence of actions/other form, based on underlying syntax

could be: AST, value, type, etc.

To define a syntax-directed translation

Augment CFG with ***translation rules*** (at most 1 rule per Production)

↳ LHS → RHS

- define translation of LHS non-terminal as a function of
  - Constants
  - translations of RHS non-terminals
  - values of tokens (terminals) on RHS

To translate a sequence of tokens using SDT *

- build parse tree
- use translation rules to compute translation of each non-terminal in parse tree bottom-up ⇐ handle children of node before node
- translation of sequence of tokens is the translation of the parse tree's root non-terminal (ie start symbol)

The **type** of the translation can be anything: numeric, string, set, tree, ...

* Note: above is how to understand the translation, not how a compiler actually does it.

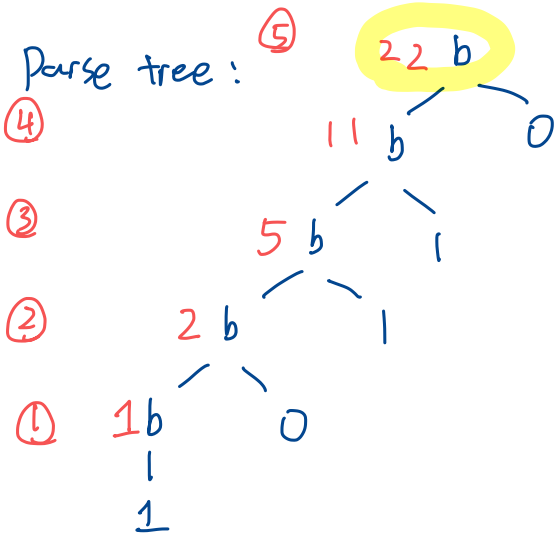
Example: grammar for language of binary numbers

SDT to compute the decimal equivalent of a binary number

CFG	translation rules
b → 0	b.trans = 0
b → 1	b.trans = 1
b → b 0	b_1.trans = b_2.trans * 2
b → b 1	b_1.trans = b_2.trans * 2 + 1

Example: input string is 10110

Parse tree:



Translation is 22

Example: grammar for language of variable declarations

CFG			Translation rules
			declList.trans = " "
declList	→	ε	
	|	decl declList	declList$_1$.trans = decl.trans + " " + declList$_2$.trans
decl	→	type ID ;	decl.trans = ID.value
type	→	INT	
	|	BOOL	

Write a syntax-directed translation for the CFG given above so that the translation of a sequence of tokens is a string containing the ID's that have been declared.

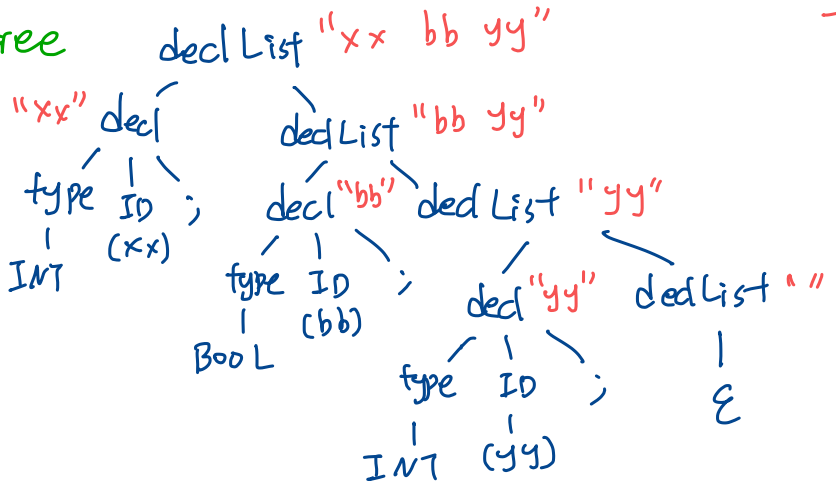
Example Input

```
int xx;
bool bb;
int yy;
```

Output

"xx yy bb"
(in any order)

Parse tree



Translation is "xx bb yy"

Example: grammar for language of variable declarations

CFG			Translation rules
declList	→	ε	declList.trans = ""
	|	decl declList	declList$_1$.trans = decl.trans + " " + declList$_2$.trans
decl	→	type ID ;	decl.trans = type.trans ? ID.value : ""
type	→	INT	type.trans = true
	|	BOOL	type.transe = false

if type.trans
then decl.trans = ID.value
else decl.trans = ""

Modify the previous syntax-directed translation so that only declarations of type `int` are added to the output string.

Example input

```
int xx;
bool bb;
int yy;
```

Output

"xx yy"
(in any order)

Note:
1) different nonterms can have different types as their translation
2) translation rules can be conditional

SDT for parsing

Previous examples showed SDT process assigning different types to the translation

- translate tokenized stream to an integer value
- translate tokenized stream to a string

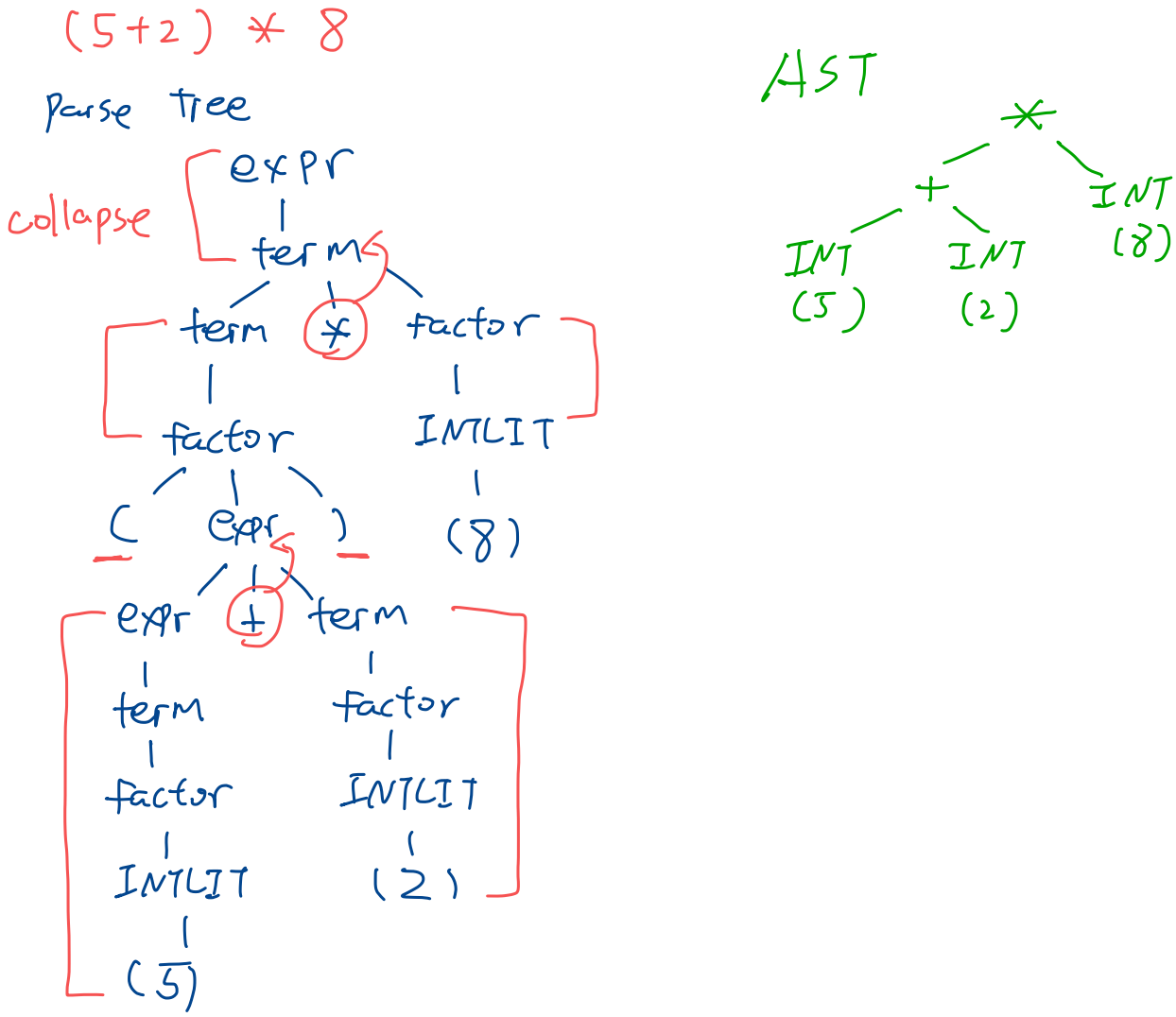
For parsing, we'll need to translate a tokenized stream to an **abstract-syntax tree (AST)**

Abstract syntax trees

AST = condensed form of parse tree

- Operators as internal nodes (not leaves)
- Chains of productions are collapsed.
- list are flattened
- syntactic details are omitted.
  - ↳ eg, ; parens

AST Example


(5+2) * 8
Parse tree
collapse
expr
term
term
*
factor
factor
INTLIT
(
expr
)
(8)
expr
+
term
term
factor
factor
INTLIT
INTLIT
(5)
(2)
AST
*
+
INT
(8)
INT
(5)
INT
(2)

CS 536 Announcements for Monday, February 20, 2023

Programming Assignment 2
- due Wednesday, February 22

Last Time
- syntax-directed translation
- abstract syntax trees

Today
- implementing ASTs — End of midterm 1 material

Next Time
- Java CUP

SDT review

SDT = translating from a sequence of tokens into a sequence of actions/other form, based on underlying syntax

To define a syntax-directed translation
- augment CFG with ***translation rules*** — lhs → rhs ∈ (contains terms, non-terms, ε)
  - define translation of LHS non-terminal as a function of: — lhs.trans =
    - constants — 2, " "
    - translations of RHS non-terminals — rhs.trans
    - values of terminals (tokens) on RHS — TOKEN.value

To translate a sequence of tokens using SDT (conceptually)
- build parse tree
- use translation rules to compute translation of each non-terminal (bottom-up)
- translation of sequence of tokens = translation of parse tree's root non-terminal

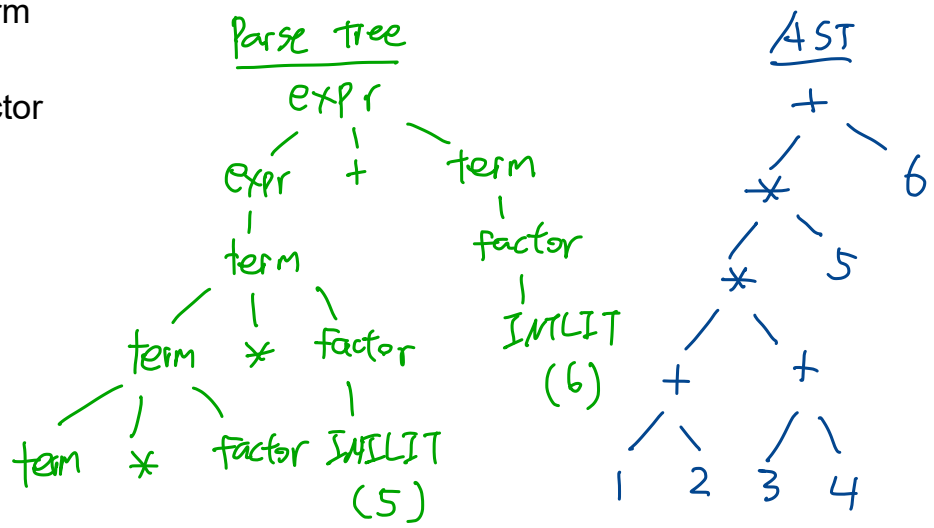
For parsing, we'll need to translate tokenized stream to **abstract-syntax tree (AST)**

Example (1+2) * (3+4) *5 *6

expr → expr + term
| term

term → term * factor
| factor

factor → INTLIT
| (expr)



translation rule:

expr1.trans = MkPlusNode (expr2.trans, term.trans).

AST for parsing

We've been showing the translation in two steps:

token stream → parse tree → AST then throw away parse tree

In practice we'll do

token stream → AST

Why have an AST?
- captures essential structure
- easier to work with

AST implementation

expr → expr + term

expr1.trans = MkPlusNode(expr2.trans, term.trans)

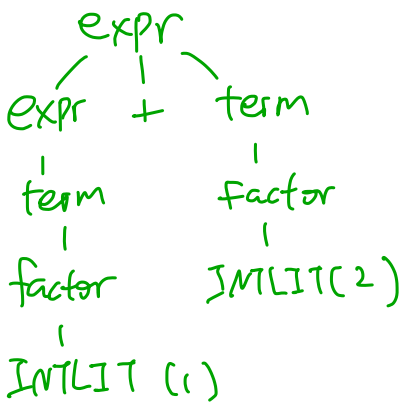
Define a class for each kind of AST node

Create a new node object in some rules

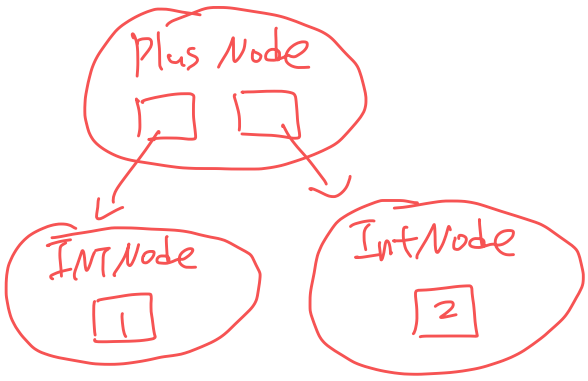
- new node object is the value of LHS.trans
- fields of node object come from translations of RHS non-terminals

Given 1 + 2

Parse tree



AST




```
class PlusNode {
    IntNode left;
    IntNode right;
}

class IntNode {
    int value;
}
```

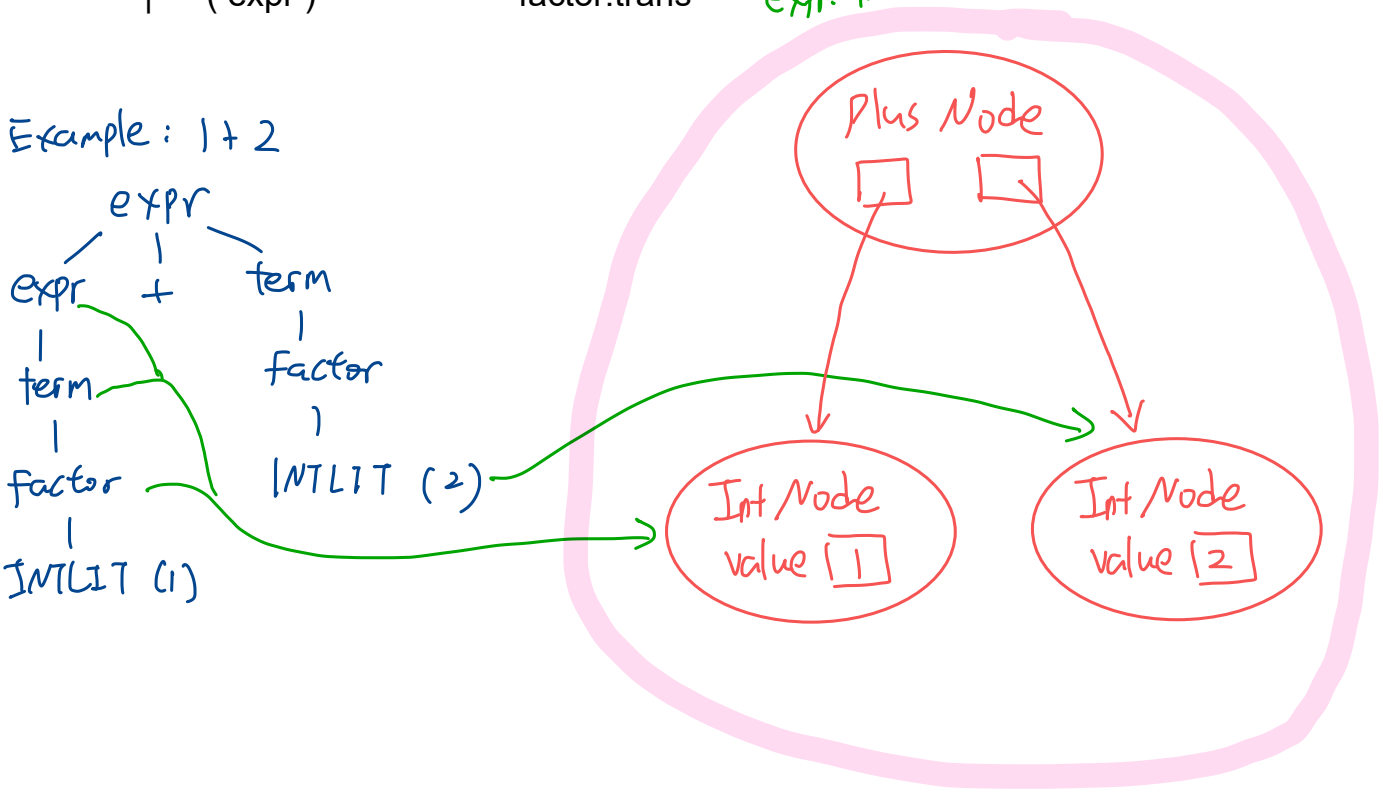
Need class hierarchy & make these subclasses of ExpNode

```
class PlusNode extends ExpNode {
    ExpNode left;
    ExpNode right;
}

class IntNode extends ExpNode {
    int value;
}
```

→ put int ExpNode

Translation rules to build ASTs for expressions

CFG			Translation rules		
expr	→	expr + term	$expr_1$.trans	=	new PlusNode (expr2.trans, Term.trans)
	|	term	expr.trans	=	term.trans
term	→	term * factor	$term_1$.trans	=	new TimesNode (expr2.trans, factor.trans)
	|	factor	term.trans	=	factor.trans
factor	→	INTLIT	factor.trans	=	new IntNode (INTLIT.value)
	|	(expr)	factor.trans	=	expr.trans

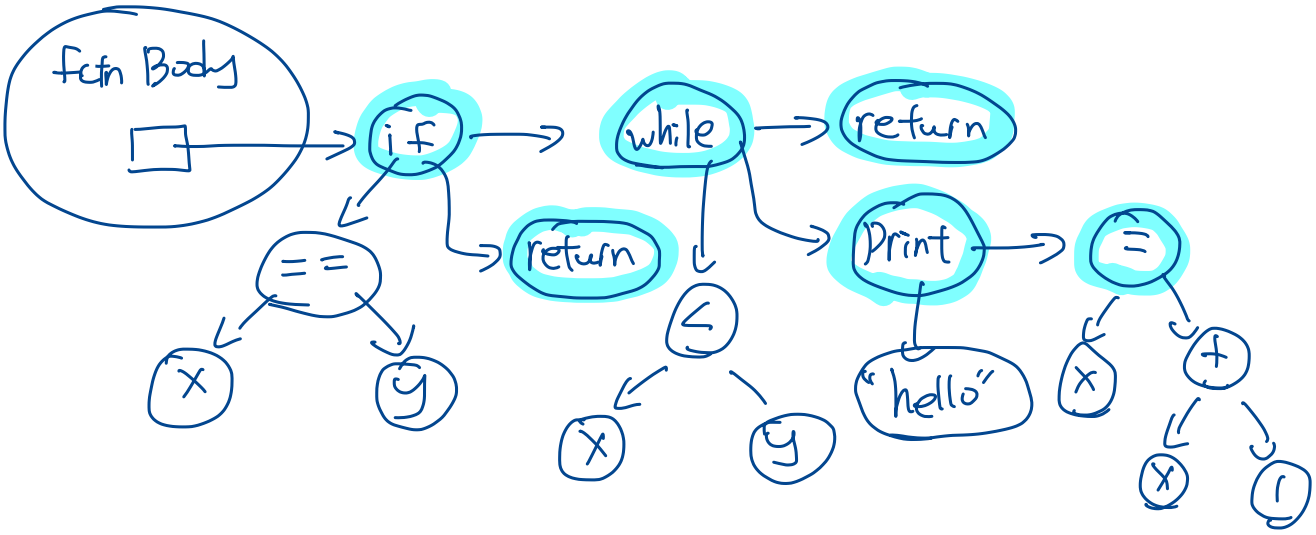


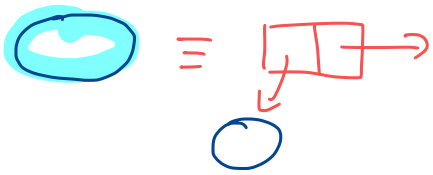
ASTs for non-expressions

Example

```
void foo(int x, int y) {
    if (x == y) {
        return;
    }
    while (x < y) {
        cout << "hello";
        x = x + 1;
    }
    return;
}
```

AST for function body





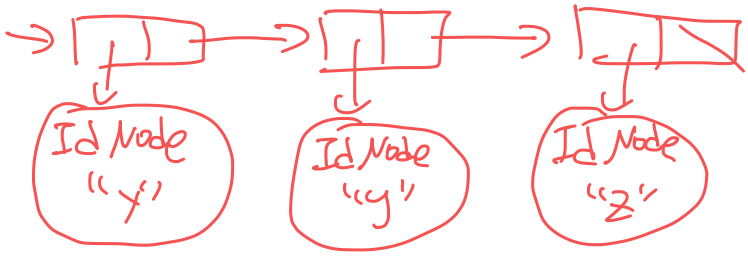
ASTs for lists

CFG

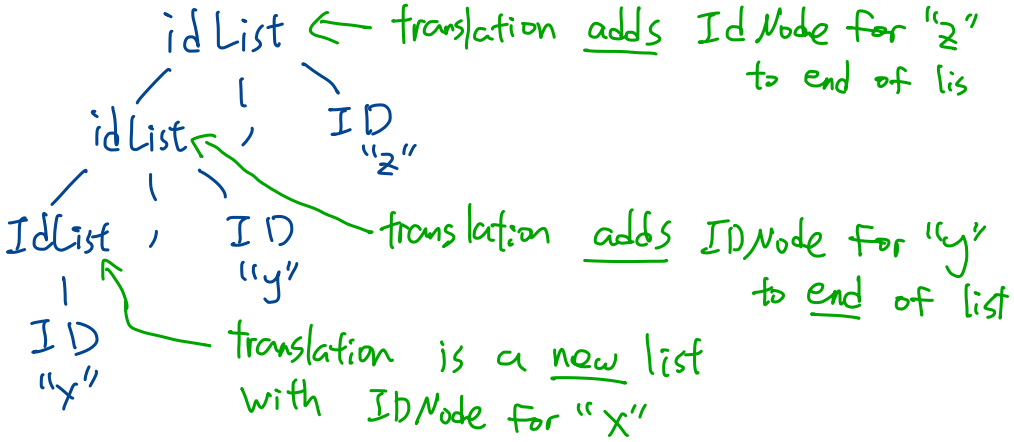
idList → idList COMMA ID
| ID

Input x, y, z

Want AST to be



Parse tree



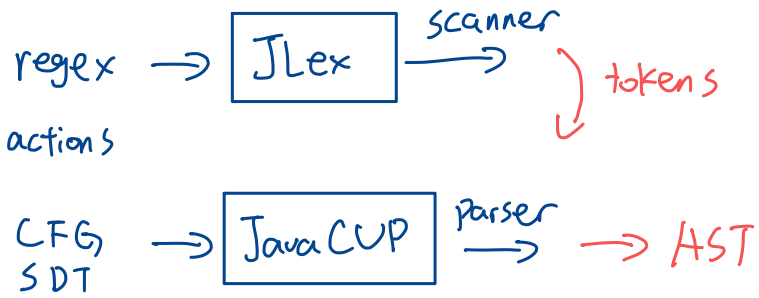
The bigger picture

Scanner

- **Language abstraction**: regular expressions
- **Output**: token stream
- **Tool**: JLex
- **Implementation**: interpret DFA using table (for δ), recording most_recent_accepted_position & most_recent_token

Parser

- **Language abstraction**: CFG
- **Output**: AST (by way of a syntax-directed translation)
- **Tool**: Java CUP ← next time
- **Implementation**: ??? ← in a couple weeks



CS 536 Announcements for Wednesday, February 22, 2023

Programming Assignment 2
- due Wednesday, February 22

Midterm 1
- Wednesday, March 1, 7:30 – 9 pm
- B10 Ingraham Hall
- bring your student ID

Last Time
- implementing ASTs

Today
- Java CUP

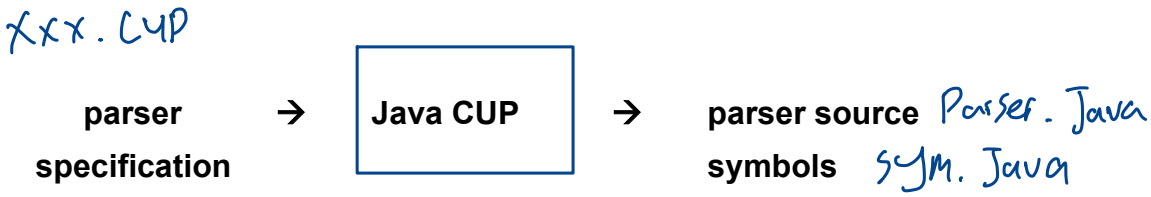
Next Time
- review for Midterm 1

Parser generators

Tools that take an SDT spec and build an AST

- **YACC** Yet Another C Compiler - creates a parser in C
- **Java CUP** Constructor of Useful Parsers - creates a parser in Java.

Conceptually similar to JLex:
- Input: language rules + actions
- Output: Java code



Java CUP

`parser.java`

- constructor takes argument of type `Yylex` ← from scanner
- `parse` method
  - if input correct, returns `Symbol` whose `value` field contains translation of root nonterm
  - if input incorrect, quits on first syntax error
- uses output of JLex ← in brevis.jlex
  - depends on scanner and `TokenVal` classes
  - `sym.java` defines the communication language = defines token names used by both jlex & JavaCup
- uses definitions of AST classes (in `ast.java`)

Parts of Java CUP specification

Grammar rules with actions: not shown (yet)

```
expr ::= INTLITERAL
       |  ID
       |  expr PLUS expr
       |  expr TIMES expr
       |  LPAREN expr RPAREN
       ;
```

Terminal and nonterminal declarations:

```
terminal     INTLITERAL;
terminal     ID;
terminal     PLUS;
terminal     TIMES;
terminal     LPAREN;
terminal     RPAREN;

non terminal expr;
```

Precedence and associativity declarations:

```
precedence left PLUS;
precedence left TIMES;
```

↑ associativity

order (in xxx.cup file) indicates precedence

low ↓ high

can also do:

```
precedence nonassoc LESS;
```

Java CUP Example

defined in ast.java

Assume:

- Java class `ExpNode` with subclasses `IntLitNode`, `IdNode`, `PlusNode`, `TimesNode`
- `PlusNode` and `TimesNode` each have two children
- `IdNode` has a `String` field (for the identifier)
- `IntLitNode` has an `int` field (for the integer value)
- `INTLITERAL` token is represented by `IntLitTokenVal` class and has field `intVal`
- `ID` token is represented by `IdTokenVal` class and has field `idVal`

defined in
brevis.jlex

Step 1: add types to terminals and nonterminals

```
/*
 * Terminal declarations
 */
terminal INTLITERAL;
terminal ID;
terminal PLUS;
terminal TIMES;
terminal LPAREN;
terminal RPAREN;

/*
 * Nonterminal declarations
 */
non terminal expr;
```

Need type if want to use value associated with token

terminal IntLitTokenVal INTLITERAL;

terminal IDTokenVal ID;

└ from scanner (... .jlex)

Type required for all nonterms

non terminal ExpNode expr;

└ from ast.java

Step 2: add precedences and associativities

```
/*
 * Precedence and associativity declarations
 */
precedence left PLUS;
precedence left TIMES;
```

Java CUP Example (cont.)

Step 3: add actions to CFG rules

```
/*
 * Grammar rules with actions
 */
expr ::= INTLITERAL:i
         {:
            RESULT = new IntLitNode(i.intVal);
         :}
       | ID:i
         {:
            RESULT = new IdNode(i.idVal);
         :}
       | expr:e1 PLUS expr:e2
         {:
            RESULT= new PlusNode(e1, e2);
         :}
       | expr:e1 TIMES expr:e2
         {:
            RESULT = new TimesNode(e1, e2);
         :}
       | LPAREN expr:e RPAREN
         {:
            RESULT = e;
         :}
       ;
```

type is IntLitTokenVal

Exp Nodes

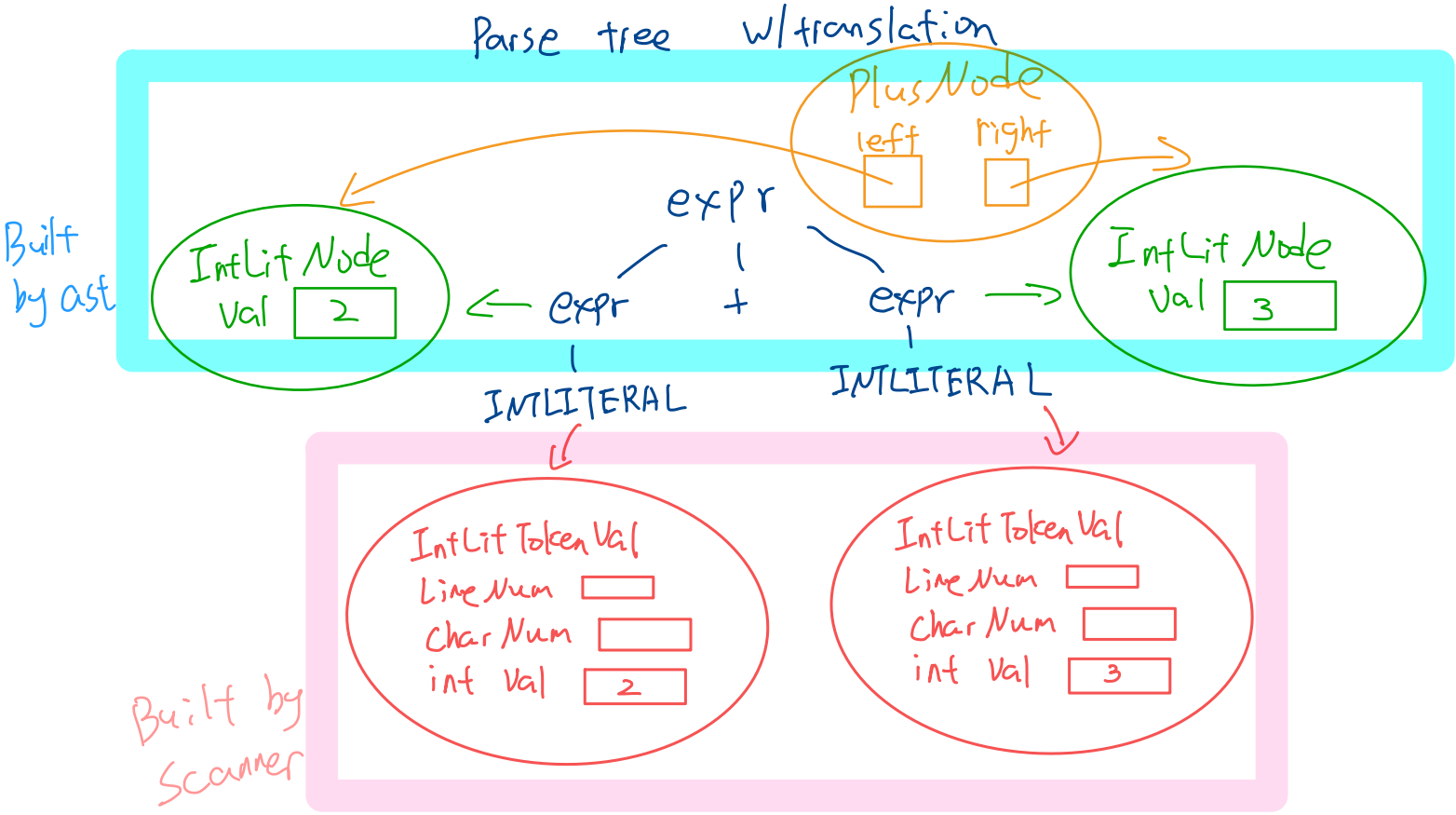
subclasses of ExpNode

Format:

```
nonterm ::= rule 1
            {: // action for rule 1
               RESULT = ---;
            :}
          | rule 2
            {:
               RESULT = ...;
            :}
          ;
```

Java CUP Example (cont.)

Input: 2 + 3



Translating lists

Example ← left-recursive

idList → idList COMMA ID | ID

Left-recursion or right-recursion?

- for top-down parsers must use right recursion
  left-recursion leads to infinite loop
- for Java CUP use left recursion
  → bottom up parser

Example

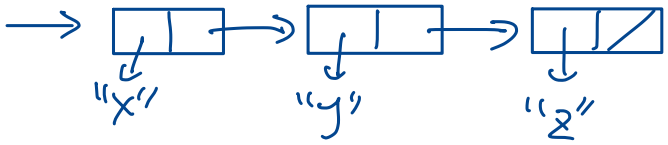
CFG: idList → idList COMMA ID | ID

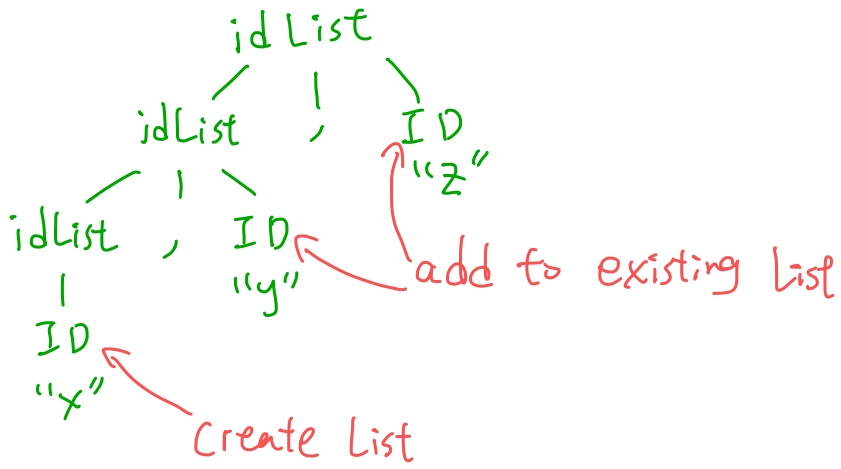
Goal: the translation of an `idList` is a `LinkedList` of `String`s

Example

Input: `x , y , z`

Output:





Example (cont.)

Java CUP specification for this syntax-directed translation idList → idList COMMA ID | ID

Terminal and nonterminal declarations:

```
terminal IdTokenVal ID;
terminal            COMMA;
non terminal LinkedList<String> idList;
```

Grammar rules and actions:

```
idList ::= idList:L    COMMA       ID:i
           {: L.addLast(i.idVal);
              RESULT = L;

           :}
         | ID:i
           {: LinkedList<String> L = new LinkedList<String>();
              L.add(i.idVal);
              RESULT = L;

           :}
         ;
```

Handling unary minus

```
/*
 * precedences and associativities of operators
 */
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE;
precedence nonassoc UMINUS;

/*
 * grammar rules
 */
exp  ::= . . .
     |    MINUS exp:e
       {: RESULT = new UnaryMinusNode(e);
       :} %prec UMINUS
     |    exp:e1 PLUS exp:e2
       {: RESULT = new PlusNode(e1, e2);
       :}
     |    exp:e1 MINUS exp:e2
       {: RESULT = new MinusNode(e1, e2);
       :}
        . . .
     ;
```

binary minus has lowest precedence

"phony" token (never returned by scanner)

unary minus has highest precedence

Precedence of a rule is that of the last token of the rule, unless assigned a specific precedence via %prec

CS 536 Announcements for Monday, February 27, 2023

Last Time

- Java CUP
  - specification format
  - handling precedence and associativity
  - translating lists
  - handling unary minus

Today

- review for Midterm 1

No class on Wednesday, March 1

Midterm 1,
Wednesday, March 1, 7:30 – 9 pm

Covers

- lectures through 2/20
- programming assignments 1 & 2

Additional practice

- Homeworks 0, 1, & 2
- sample midterm (esp questions 1, 3a, 3b, 4)

Format

- closed-book, closed-notes
- paper and pencil/pen
- question formats
  - multiple-choice
  - short-answer
  - written questions

Make sure to bring your student ID

See also Exam Information page

Midterm 1 Topics

Scanning

- general :
  - what does a scanner do
  - how does it fit into the design of a compiler
- underlying model :
  - FSMs, DFAs vs NFAs,
  - translating regex → NFA
  - translating NFA → DFA
- specification of a scanner :
  - regular expressions, JLex specifications*
    *you do not need to know all of JLex's special characters

Context-Free Grammars

- specification of a language's syntax via a CFG
- derivations (left-most, right-most)
- parse trees
- expression grammars (precedence, associativity)
- list grammars
- ambiguous grammars
- recursive grammar (left recursive, right recursive)

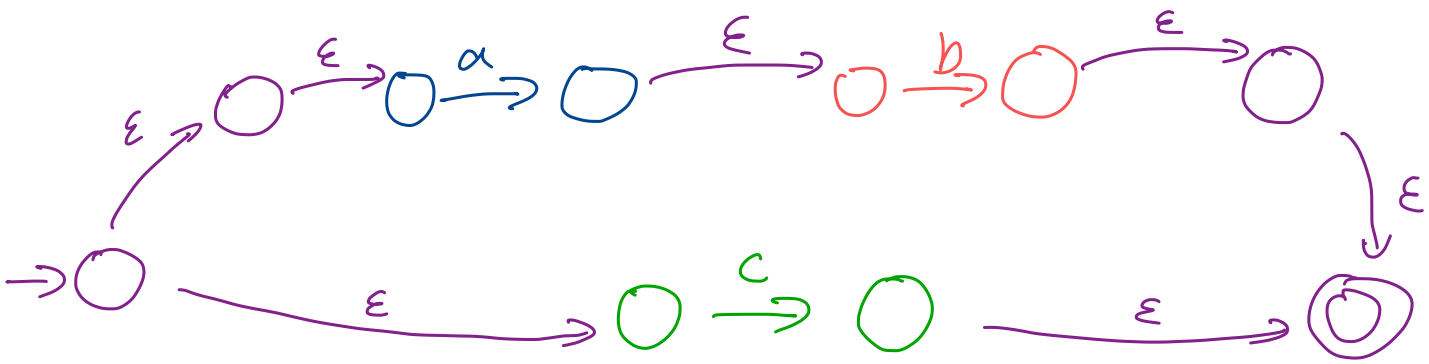
Syntax-Directed Translation

- "plain" translations
  - writing rules of the form "s1.trans ="
- being able to define translations of any types (integer, AST nodes, etc.)

Translating a regular expression into a DFA

Example: ab|c

Translate regex → NFA (with ϵ-transitions)



Removing ϵ-transitions from NFAs

Let M be an NFA with ϵ-transitions. Goal: construct ϵ-free NFA M* that is equivalent to M

Recall: eclose(s) = set of all states reachable from s using 0 or more ϵ - transitions

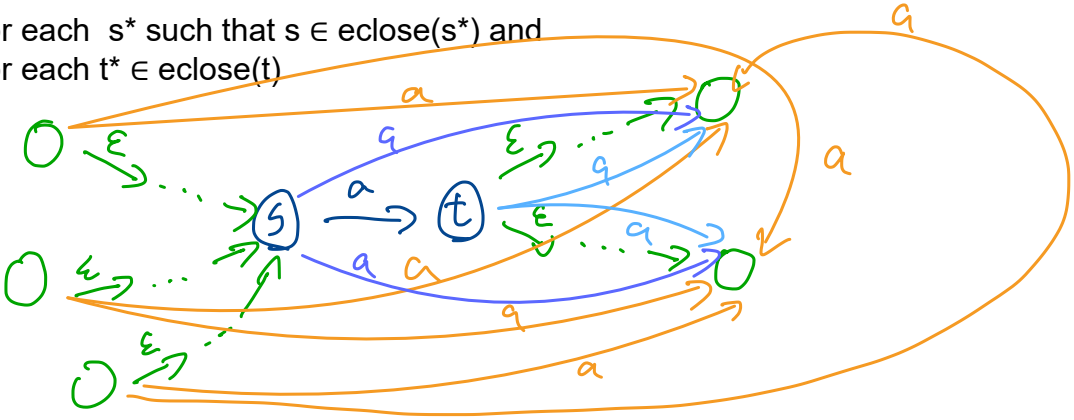
Overview:

1) determine eclose(s) for each state s ∈ M
2) initialize M* to M
3) determine additional final states of M* – using eclose
4) add edges to M*
5) remove ϵ-transitions

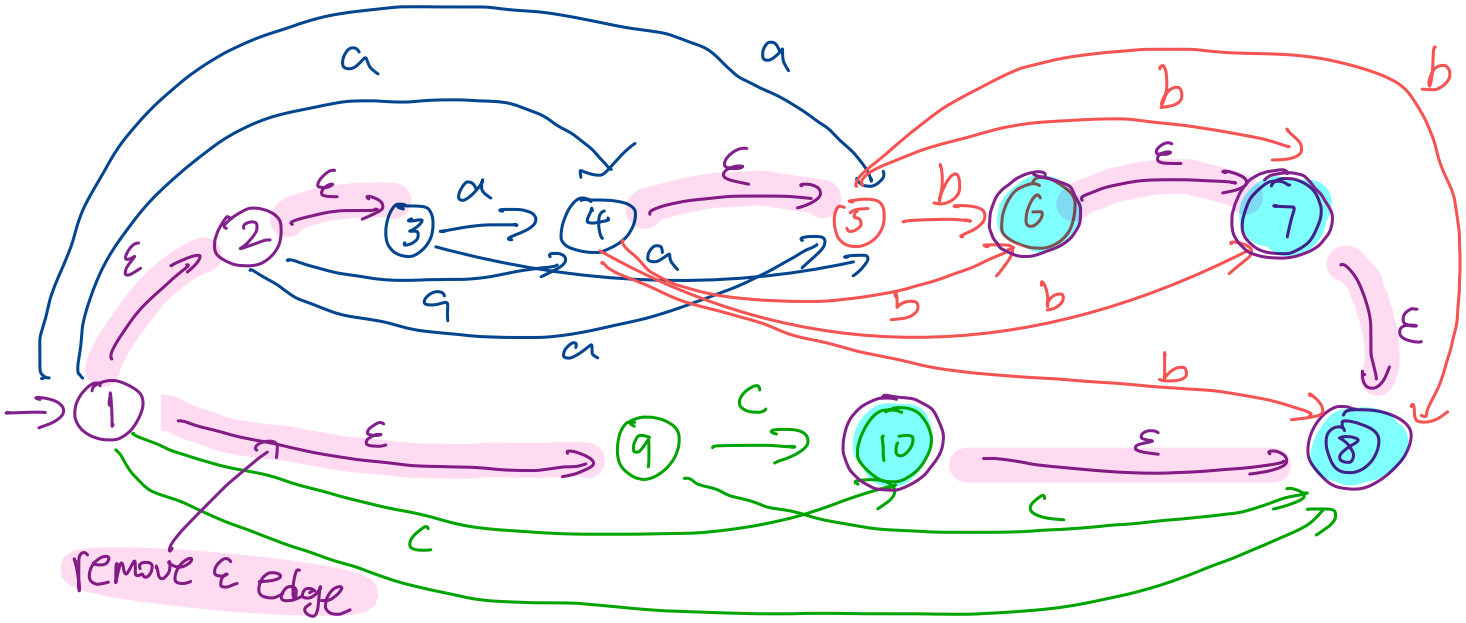
Final states of M*: If eclose(s) contains a final state in M, then s is a final state in M*

Adding edges to M*: If there is edge s –^{a}→ t in M (a ≠ ϵ), then add edge s* –^{a}→ t* in M*

- for each s* such that s ∈ eclose(s*) and
- for each t* ∈ eclose(t)



Translate NFA w/ ϵ-transitions → NFA w/o ϵ-transitions



	eclose
1	1, 2, 3, 9
2	2, 3
3	3
4	4, 5
5	5
6	6, 7, 8
7	7, 8
8	8
9	9
10	10, 8

consider (3) $\xrightarrow{a}$ (4) $\quad 3 \in$ eclose(x) for $x = \{1, 2, 3\}$

eclose(4) = {4, 5}

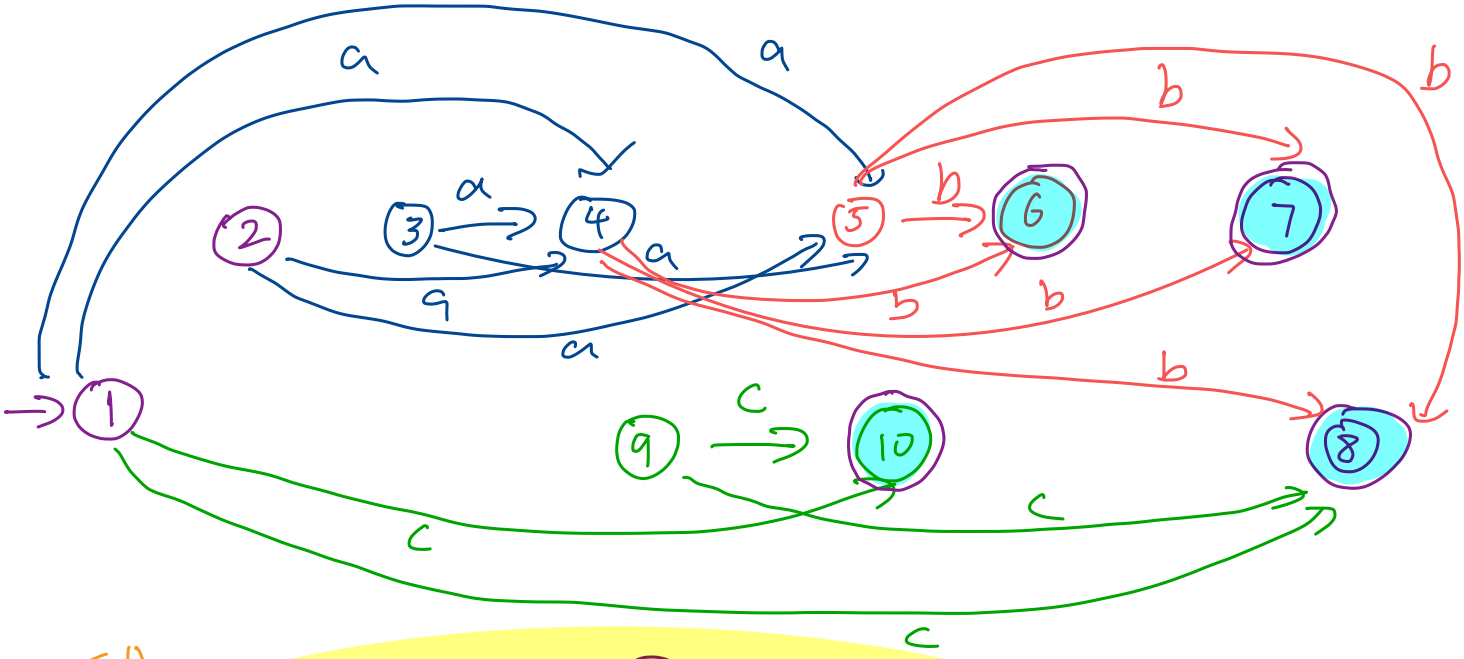
consider (5) $\xrightarrow{b}$ (6) $\quad 5 \in$ eclose(x) for $x = \{4, 5\}$

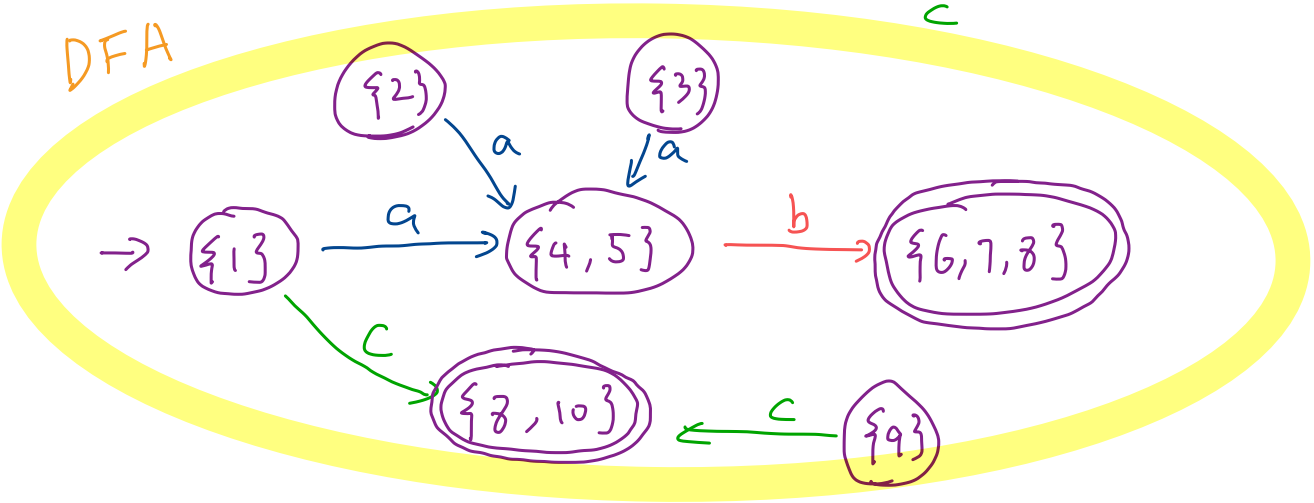
eclose(6) = {6, 7, 8}

consider (9) $\xrightarrow{c}$ (10) $\quad 9 \in$ eclos(x) for $x = \{1, 9\}$

eclose(10) = {10, 8}

NFA w/o ϵ-transitions → DFA

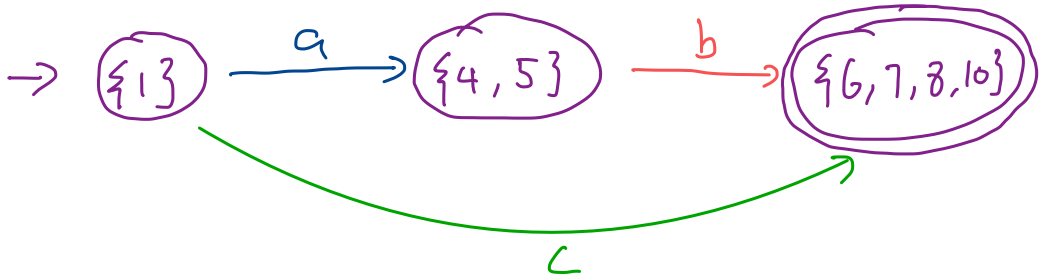




Optimize DFA

{2}, {3}, {9} are unreachable → remove them

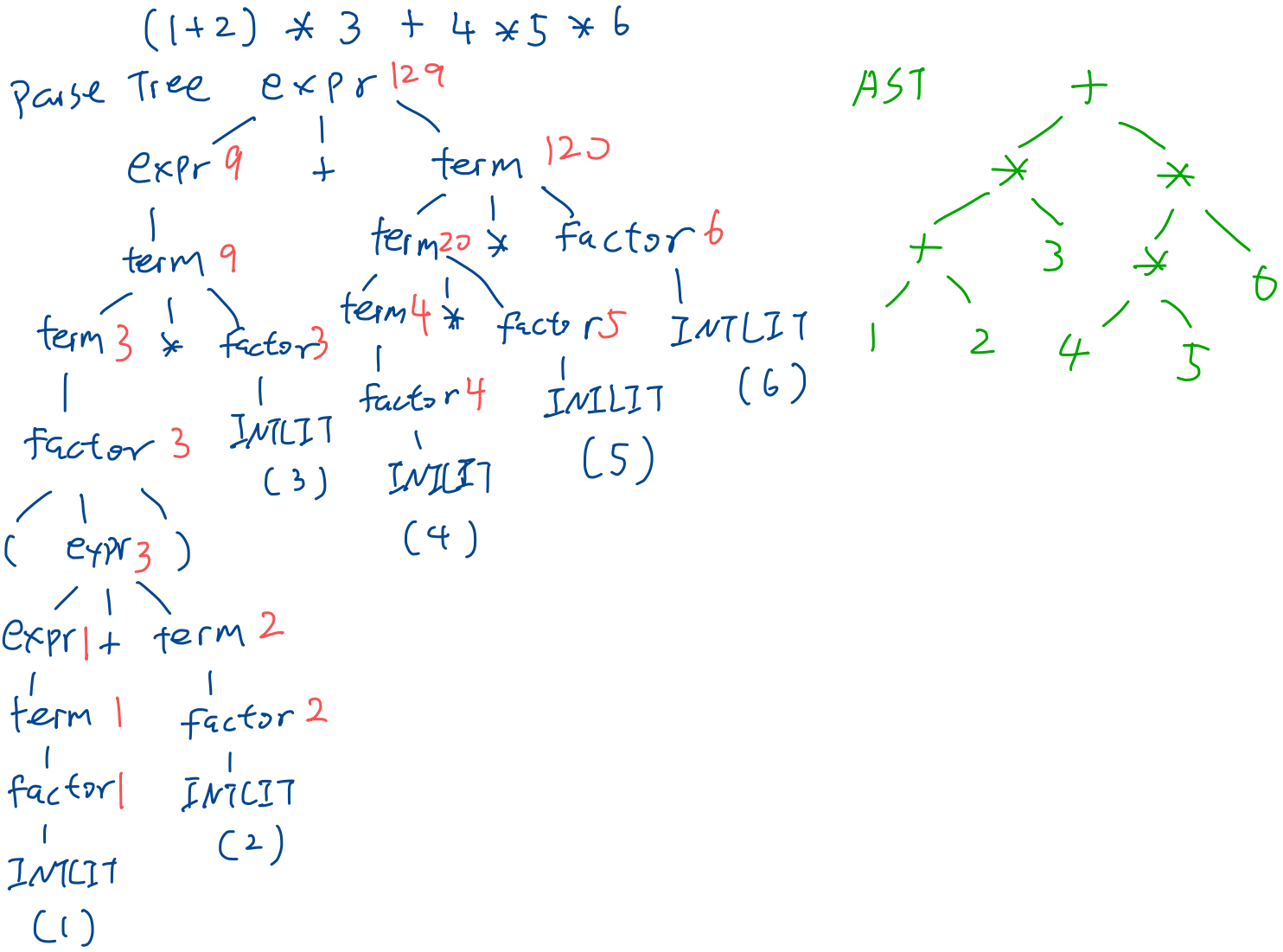
{6,7,8}, {8,10} are equivalent → merge



Syntax-directed translation

CFG			Translation rules
expr	→	expr + term	$expr_1.trans = expr_2.trans + term.trans$
	|	term	$expr.trans = term.trans$
term	→	term * factor	$term_1.trans = term_2.trans * factor.trans$
	|	factor	$term.trans = factor.trans$
factor	→	INTLIT	$factor.trans = INTLIT.value$
	|	(expr)	$factor.trans = expr.trans$

Write a syntax-directed translation for the CFG given above so that the translation of a sequence of tokens is numeric value of the expression (i.e., the expression evaluated).



CS 536 Announcements for Monday, March 6, 2023

Last Time

- review for Midterm 1

Today

- approaches to parsing
- bottom-up parsing
- CFG transformations
  - removing useless non-terminals
  - Chomsky normal form (CNF)
- CYK algorithm

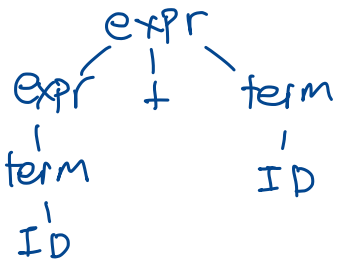
Next Time

- wrap up CYK
- classes of grammars
- top-down parsing

Parsing: two approaches

Top-down / "goal driven"

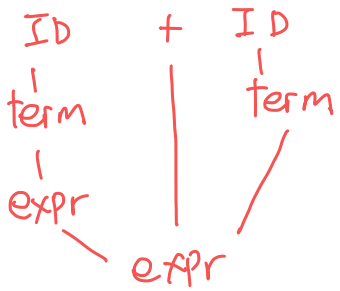
- start at start nonterminal
- grow parse tree downward until entire sequence is matched



Bottom-up / "data driven"

- start with terminals (sequence)
- generate ever larger subtrees until get to single tree whose root is the start nonterminal

(note: parse tree is upside down)



Example:

CFG: expr → expr + term | term

term → term * ID | ID

Derive: ID + ID

Cocke – Younger – Kasami (CYK) algorithm

- Works bottom-up
- Time complexity : $O(n^3)$ n = length of input (# of tokens in sequence)
- Requires grammar to be in Chomsky Normal Form

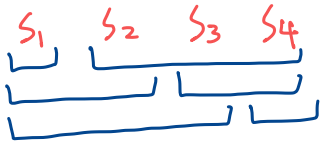
Chomsky Normal Form (CNF)

- all rules must be in one of two forms
  - $x \rightarrow \mathbf{T}$ (T is a terminal)
  - $x \rightarrow a\ b$
- only rule allowed to derive epsilon is the start symbol *s*

Why CNF is helpful?

- nonterminals in pairs → can think of a subtree as a subspan of our input
- nonterminals (except start) can't derive epsilon → each subspan has at least 1 token

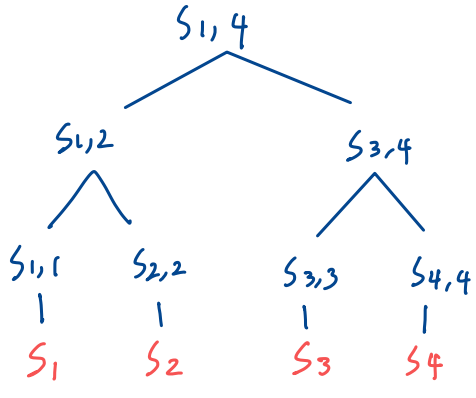
input s_1 s_2 s_3 s_4

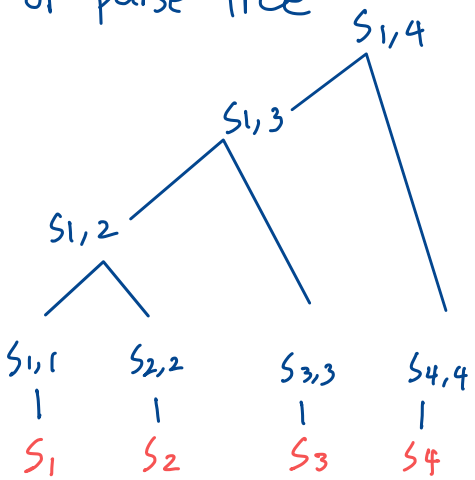


CYK : Dynamic Programming

$x \rightarrow \mathbf{T}$ → forms leaf of parse tree

$x \rightarrow a\ b$ → binary interior nodes of parse tree

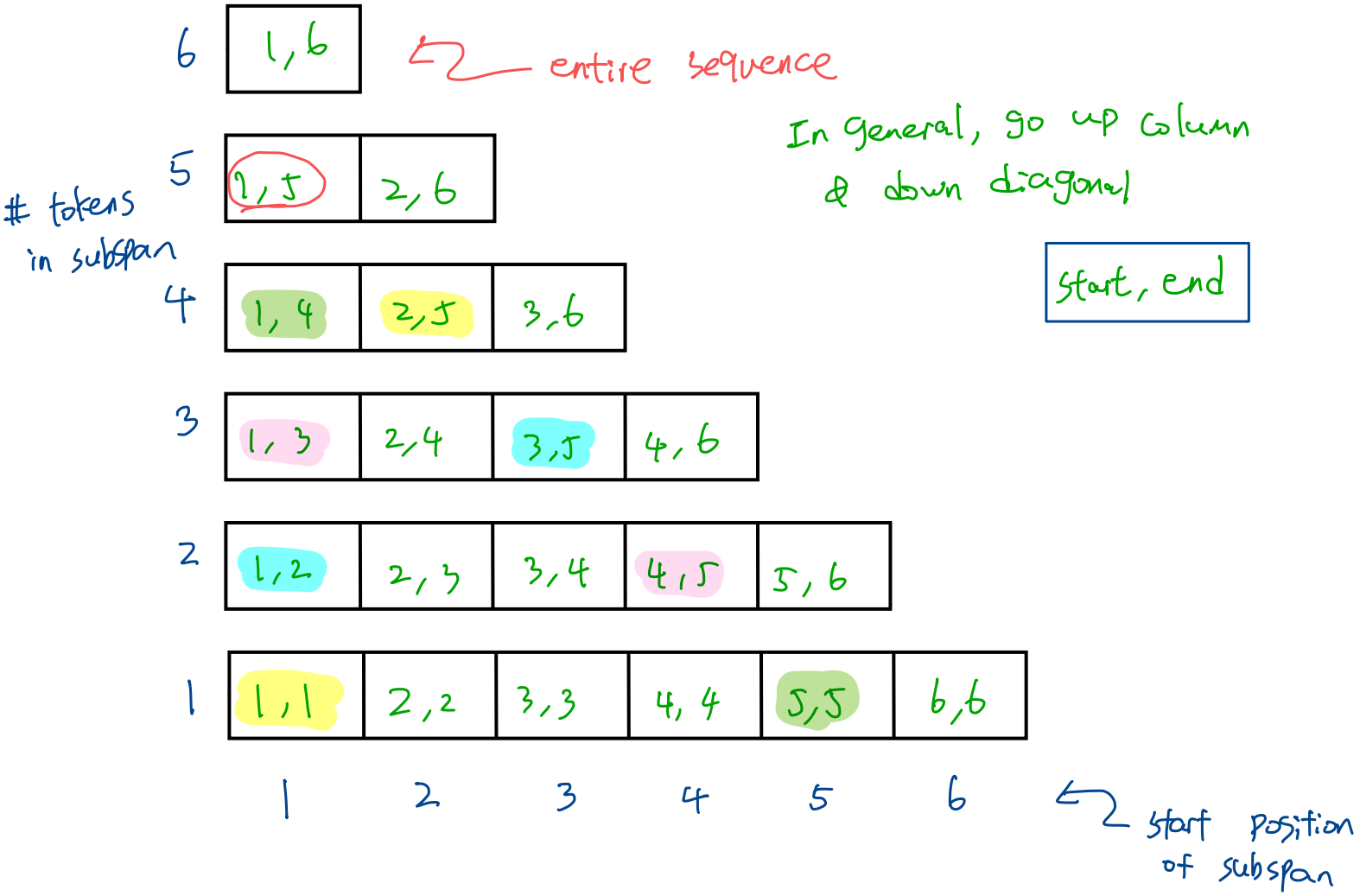




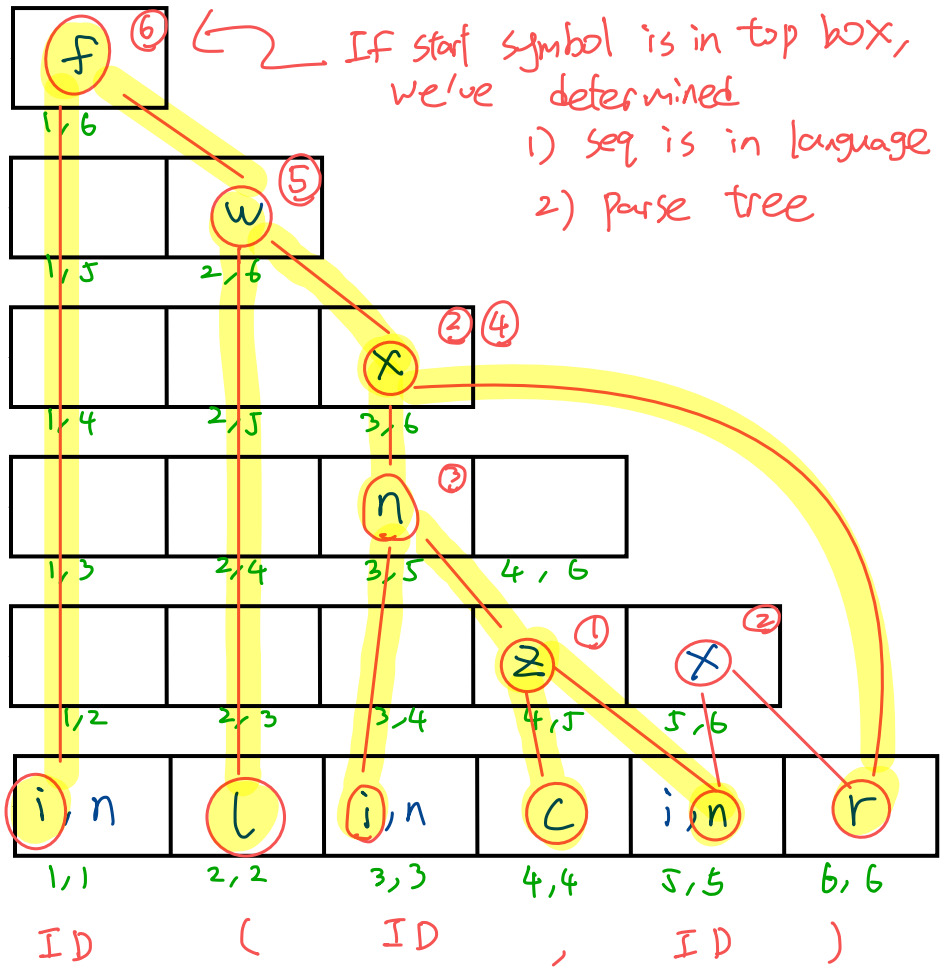
Running CYK

Track every viable subtree from leaf to root.

All subspans for a sequence (string) with 6 terminals



CYK Example



$f \rightarrow i\ w$ (5)

$f \rightarrow i\ y$

$w \rightarrow l\ x$ (5)

$x \rightarrow n\ r$ (2) (4)

$y \rightarrow l\ r$

$n \rightarrow$ **ID**

$n \rightarrow i\ z$ (3)

$z \rightarrow c\ n$ (1)

$i \rightarrow$ **ID**

$l \rightarrow$ **(**

$r \rightarrow$ **)**

$c \rightarrow$ **,**

Eliminating useless nonterminals

Avoid unnecessary work – remove *useless* rules

1. If a nonterminal cannot derive a sequence of terminal symbols, then it is *useless*
2. If a nonterminal cannot be derived from the start symbol, then it is *useless*

Nonterminals that cannot derive a sequence of terminal symbols

```
mark all terminal symbols
repeat
    if all symbols on the RHS of a production are marked
        mark the LHS nonterminal
until no more nonterminals can be marked
```

(everywhere it shows up)

Example

s → *x* | *y*
x → **( )**
y → **(** *y* **)**

y is useless

Nonterminals that cannot be derived from the start symbol

```
mark the start symbol
repeat
    if the LHS of a production is marked
        mark all RHS nonterminals
until no more nonterminals can be marked
```

(whereever they show up)

Example

s → *a b*
a → **+** | **−** | ε
b → **digit** | *b* **digit**
c → *b*

c is useless

Chomsky Normal Form

Four steps

- eliminate epsilon productions
- eliminate unit productions
- fix productions with terminal on RHS (along w/ other stuff)
- fix productions with > 2 nonterminals on RHS

← ok to have start → ε

Eliminate (most) epsilon productions

If nonterminal a immediately derives epsilon
- make copies of all rules with a on RHS & delete all combinations of a in copies.

Example 1

f → **ID (*a*)**
a → ε
a → *n*
n → **ID**
n → **ID , *n***

f → ID (a)
f → ID (a) ⇒ f → ID ( )
a → n
n → ID
n → ID, n

Example 2

x → *a* **X** *a* **Y** *a*
a → ε
a → **Z**

x → a X a Y a
| a x a Y
| a x Y a
| x a Y a
| a X Y
| x a Y
| x Y a
| X Y
a → Z

Chomsky Normal Form (cont.)

Eliminate unit productions

Productions of the form $a \rightarrow b$ are called *unit productions*

- Place b anywhere a could have appeared. & remove unit production

Example

f → **ID (*a*)**
f → **ID ( )**
a → *n*
n → **ID**
n → **ID , *n***

f → ID (n)
f → ID ( )
n → ID
n → ID , n

Fix RHS terminals

For productions with terminals and something else on the RHS

- For each terminal T, add rule $X \rightarrow T$ where X is a new non-terminal.
- replace T with X in those productions.

Example

f → **ID (*n*)**
f → **ID ( )**
n → **ID**
n → **ID , *n***

this is ok

i → ID
l → (
r →)
c → ,
f → i l n r
f → i l r
n → ID
n → i c n

Chomsky Normal Form (cont.)

Fix RHS nonterminals

For productions with > 2 nonterminals on the RHS

- replace all but 1st non terminal with new non-terminal
- add a rule from new non-terminal to replaced non-terminal sequence.
- repeat

Example

$f \to i\ l\ n\ r \Rightarrow f \to i\ w,\ w \to l\ n\ r \Rightarrow f \to i\ w,\ w \to l\ x,\ x \to n\ r$

Left for you to try

$f \to i\ l\ r$

$n \to i\ C\ n$

CS 536 Announcements for Wednesday, March 8, 2023

Last Time

- approaches to parsing
- bottom-up parsing
- CFG transformations
  - removing useless non-terminals
  - Chomsky normal form (CNF)
- CYK algorithm

can't derive a seq of tokens
can't be derived from start nonterm

Today

- wrap up CYK
- classes of grammars
- top-down parsing

Next Time (after Spring Break)

- building a predictive parser
- FIRST and FOLLOW sets

Parsing (big picture)

Context-free grammars (CFGs) Given CFG G

- language generation: G → $w \in L(G)$
- language recognition: given w, is $w \in L(G)$?

Translation

- given $w \in L(G)$, create a parse tree for w
- given $w \in L(G)$, create an AST for w
  - ↳ passed on to next phase of our compiler

CYK algorithm

Step 1: get grammar in Chomsky Normal Form (CNF)

Step 2: build all possible parse trees bottom-up

- start with runs of 1 terminal
- connect 1-terminal runs into 2-terminal runs
- connect 1- and 2-terminal runs into 3-terminal runs
- connect 1- and 3- or 2- and 2-terminal runs into 4-runs
- …
- if we can connect entire tree, rooted at start symbol, we've found a valid parse

Pros: able to parse an arbitrary CFG

Cons: $O(n^3)$ time complexity ← too slow !

For special classes of grammars, we can parse in $O(n)$ time

↳ eg LL(1) & LALR(1)

Classes of grammars

LL(1)

scan from L to R | L 1 token lookahead
leftmost derivation

LALR(1)

lookahead (technicle detail of LR Parsers) | L 1 token lookahead
rightmost derivation (in reverse)
scan from L to R

Both are accepted by parser generators

LALR(1)

- parsed by bottom-up parsers
- harder to understand

Java CUP generates a LALR(1) parser

LL(1)

- parsed by top-down parsers

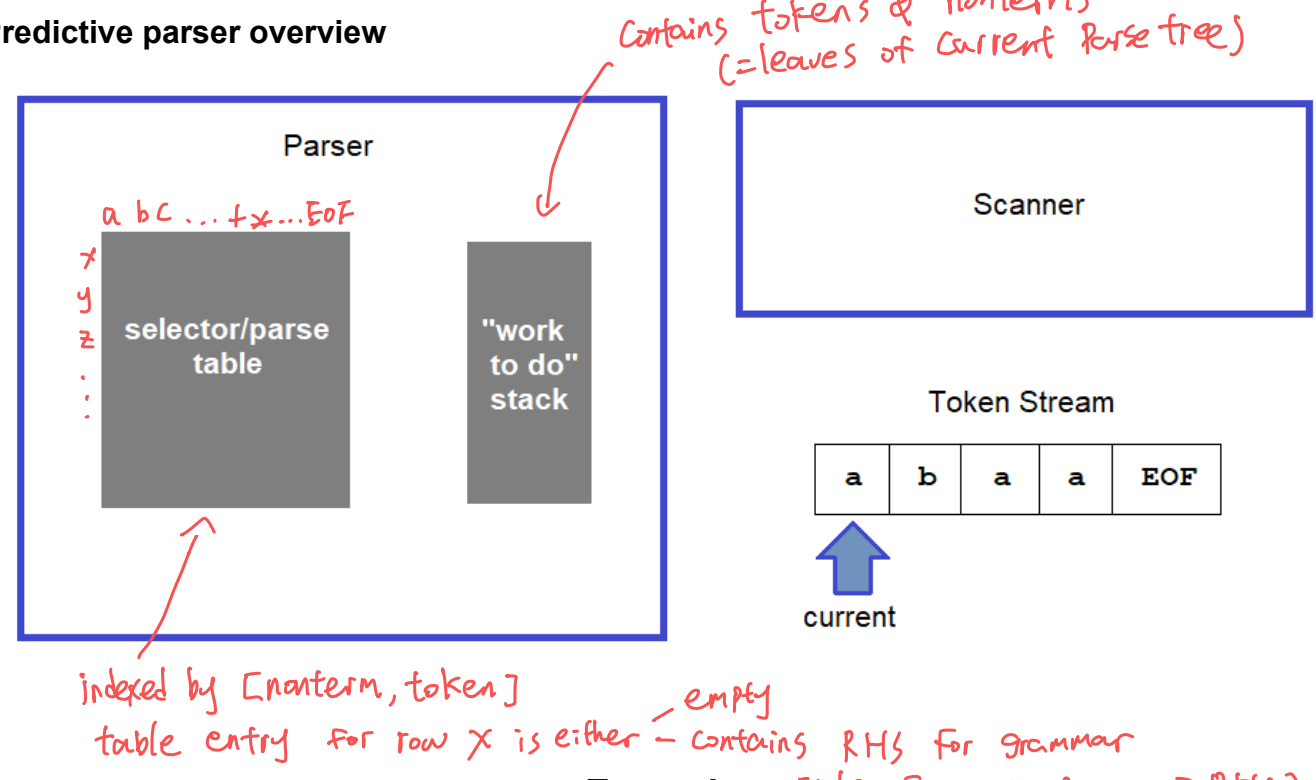
↳ or Predictive Parsers
or recursive descent parser

CFG
∪
LR(1)
∪
LALR(1)
∪
LL(1)
∪
RG
= regular grammars
- langs that can be recognized by DFAs

Top-down parsers

- Start at start symbol
- Repeatedly "predict" what production to use

Predictive parser overview



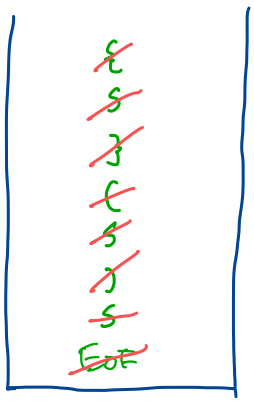
Example

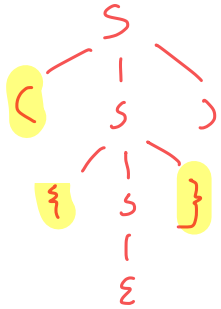
CFG: s → (s) | { s } | ε

Parse table:

	(	)	{	}	EOF
s	(s)	ε	{ s }	ε	ε

Input: ({ }) EOF





Predictive parser algorithm

```
stack.push(EOF)
stack.push(start nonterm)
T = scanner.getToken( )

repeat

     if stack.top is terminal Y
         match Y with T
         pop Y from stack
         T = scanner.getToken()

     if stack.top is nonterminal x
         get table[x, current token T]
         pop x from stack
         push production's RHS (each symbol from R to L)

until one of the following:
     stack is empty
     stack.top is a terminal that does not match T
     stack.top is a nonterm and parse-table entry is empty
```

initial stack

start EOF

note: don't push ε

accept input

reject input

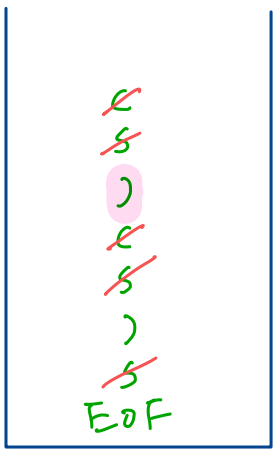
Example

CFG: s → (s) | { s } | ε

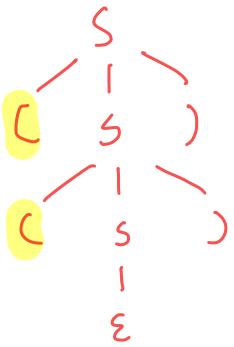
Parse table:

	(	)	{	}	EOF
s	(s)	ε	{ s }	ε	ε

Input: ((} EOF



no match → reject input



Consider

CFG: s → (s) | { s } | ( ) | { } | ε

Parse table:

	(	)	{	}	EOF
S					

(S) or ( ) ?

If could look ahead 2 tokens, we could make a good choice

This grammar is not LL(1), but it is LL(2)

Some grammars are not LL(k) for any k.

Two issues

1) How do we know if the language is LL(1)?
2) How do we build the selector table?

Answer: If we can build a parse / selector table & each entry has only 1 production in it, then the grammar is LL(1)

Converting non-LL(1) grammars to LL(1) grammars

Necessary (but not sufficient conditions) for LL(1) parsing

- free of left recursion – no left-recursive rules
- left-factored – no rules with a common prefix, for any nonterminal

Left recursion

- A grammar *G* is *recursive* in nonterm *X* iff *X* =>+ α *X* β
- A grammar *G* is *left recursive* in nonterm *X* iff *X* =>+ *X* β
- A grammar G is *immediately left recursive* in *X* iff *X* => *X* β

Why left-recursion is a problem

Consider: xlist → xlist ID | ID

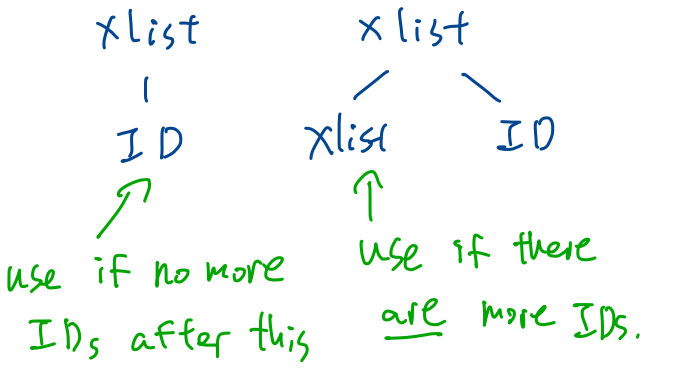
Current parse tree

xlist

Current token

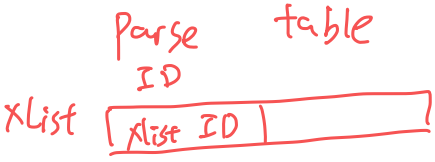
ID

How to grow tree (top down)?



Don't know without lookahead

Parse table

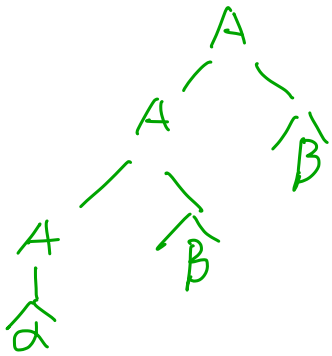


Removing left-recursion

We can remove immediate left recursion without "changing" the grammar:

Consider: $A \rightarrow A\,\beta \mid \alpha$

doesn't start with nonterm A

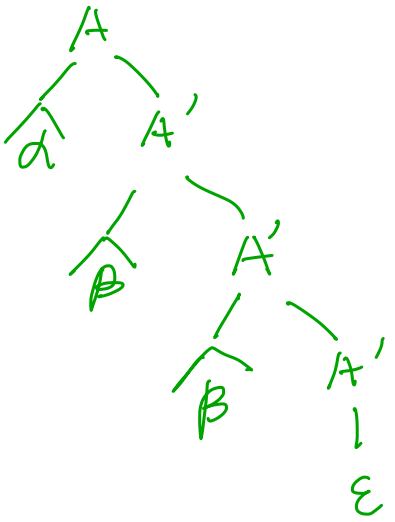


Solution: introduce new nonterminal A' and new productions:

$A \Rightarrow \alpha A'$

$A' \Rightarrow \beta A' \mid \varepsilon$

messes up associativity of parse tree (we'll fix this when we build the AST)



More generally,

$A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n \mid A\,\beta_1 \mid A\,\beta_2 \mid \ldots \mid A\,\beta_p$

transforms to

$A \Rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_n A'$

$A' \Rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_p A' \mid \varepsilon$

Grammars that are not left-factored

If a nonterminal has two productions whose right-hand sides have a common prefix, the grammar is not left-factored.

Example: s → (s) | ( )

Given: A → α β_1 | α β_2

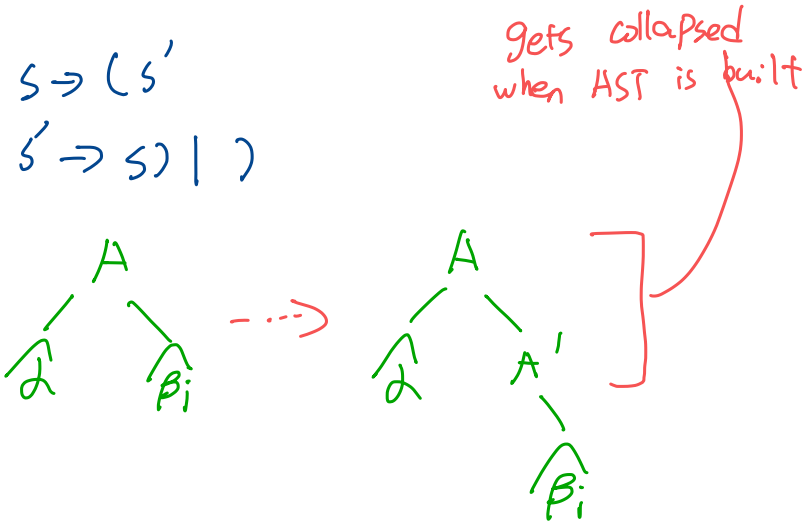
transform it to

A → α A′

A′ → β_1 | β_2

S → (S′

S′ → S) |)



More generally,

A → α β_1 | α β_2 | . . . | α β_n | δ_1 | δ_2 | . . . | δ_p

transforms to

A → α A′ | δ_1 | δ_2 | . . . | δ_p

A′ → β_1 | β_2 | . . . | β_n

Combined example

exp → (exp) exp | ( ) exp′

exp′ → exp exp′ | ε

↓

exp → (exp″

exp″ → exp) exp′ |) exp′

exp′ → exp exp′ | ε

exp → (exp) exp′ | ( ) exp′

exp′ → exp exp′ | ε

CS 536 Announcements for Monday, March 20, 2023

Last Time

- wrap up CYK
- classes of grammars
- top-down parsing

Today

- review grammar transformations
- building a predictive parser
- FIRST and FOLLOW sets

Next Time

- predictive parsing and syntax-directed translation

LL(1) Predictive Parser

Predict the parse tree top-down

Parser structure

- 1 token lookahead
- parse/selector table
- stack tracking current parse tree's frontier

Necessary conditions

- left-factored
- free of left-recursion

Review of LL(1) grammar transformations

Necessary (but not sufficient conditions) for LL(1) parsing

- free of left recursion – no left-recursive rules
- left-factored – no rules with a common prefix, for any nonterminal

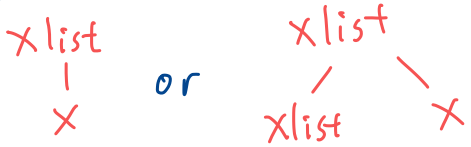
Why left-recursion is a problem

Outside/high-level view

CFG snippet: xlist → xlist X | X

Current parse tree: xlist Current token: X

How to grow parse tree?



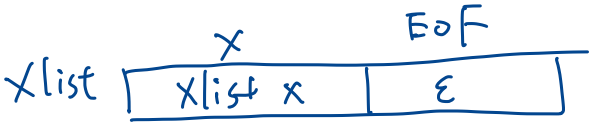
Depends on if there are more X's
⇒ need more lookahead

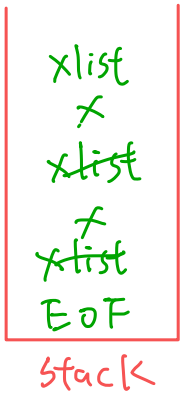
Inside/algorithmic-level view

CFG snippet: xlist → xlist X | X

Current parse tree: xlist Current token: X

Parse table





→ stack overflow

Removing left-recursion (review)

Replace

$A \rightarrow A\,\alpha \mid \beta$ ← head of list

with

$A \rightarrow \beta\,A'$

$A' \rightarrow \alpha\,A' \mid \varepsilon$

where β does not start with A (or may not be present)

Preserves the language (as a list of α's, starting with a β), but uses right recursion

Example

xlist → xlist X | ε

xlist → ε xlist' → x list'

xlist' → X xlist' | ε

) = xlist → x xlist | ε

Left factoring (review)

Removing a common prefix from a grammar

Replace

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_m$$

with

$$A \rightarrow \alpha A' \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_m$$
$$A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

where β_i and γ_i are sequence of symbols with no common prefix

Note: γ_i may not be present, and one of the β_i may be ε

Idea: combine all "problematic" rules that start with α into one rule αA'
A' now represents the suffix of the problematic rules

Example 1

exp → < A > | < B > | < C > | D

exp → < exp' | D
exp' → A > | B > | C >

Example 2

stmt → ID ASSIGN exp | ID (elist) | return
exp → INTLIT | ID
elist → exp | exp COMMA elist

stmt → ID stmt' | return
stmt' → ASSIGN exp | (elist)
exp → INTLIT | ID
elist → exp elist'
elist' → ε | COMMA elist

Building the parse table

Goal: given production *lhs* → *rhs*, determine what terminals would lead us to choose that production

ie, figure out T such that table[lhs][T] = rhs

- also what terminals could indicate an error at this point

- what terminals could *rhs* possibly start with?
- What terminals could possibly come after *lhs*?

Idea: FIRST(*rhs*) = set of terminals that begin sequences derivable from *rhs*

Suppose top-of-stack symbol is nonterminal *p* and the current token is **A** and we have

- Production 1: *p* → α
- Production 2: *p* → β

FIRST lets us disambiguate

if $A \in FIRST(\alpha)$, then Prod 1 is a viable choice

if $A \in FIRST(\beta)$, then Prod 2 is a viable choice

if in just 1 of them, then we can predict which Prod to use

FIRST sets

FIRST(α) is the set of terminals that begin the strings derivable from α, and also, if α can derive ε, then ε is in FIRST(α).

Formally,

FIRST(α) = $\{T \mid T \in \Sigma \wedge \alpha \Rightarrow {}^*T\beta) \vee (T = \varepsilon \wedge \alpha \Rightarrow {}^*\varepsilon)\}$ (terminals)

For a symbol X

- if X is terminal: FIRST(X) = {X}
- if X is ε : FIRST(X) = {ε}
- if X is nonterminal : for each production X → $Y_1Y_2Y_3..Y_n$
  - put FIRST(Y_1) – ε into FIRST(X)
  - if ε is in FIRST(Y_1), put FIRST(Y_2) – ε into FIRST(X)
  - if ε is in FIRST(Y_2), put FIRST(Y_3) – ε into FIRST(X)
  - ...
  - if ε is in FIRST(Y_i) for all i, put ε into FIRST(X)

repeat until there are no changes in any non-terminal's FIRST set

Example

Original CFG

expr → expr + term
| term
term → term * factor
| factor
factor → exponent ^ factor
| exponent
exponent → INTLIT
| (expr)

Transformed CFG

expr → term expr'
expr' → + term expr' | ε
term → factor term'
term' → * factor term' | ε
factor → exponent factor'
factor' → ^ factor | ε
exponent → INTLIT | (expr)

	FIRST	FOLLOW
expr	INTLIT (	EOF)
expr'	+ ε	= Follow(expr) EOF)
term	INTLIT (	+ EOF) ← Follow(expr')
term'	* ε	= Follow(term) + EOF)
factor	INTLIT (	* + EOF)
factor'	^ ε	* + EOF)
exponent	INTLIT (	^ * + EOF)

		FIRST
expr	→ term expr'	INTLIT (
expr'	→ + term expr'	+
expr'	→ ε	ε
term	→ factor term'	INTLIT (
term'	→ * factor term'	*
term'	→ ε	ε
factor	→ exponent factor'	INTLIT (
factor'	→ ^ factor	^
factor'	→ ε	ε
exponent	→ INTLIT	INTLIT
exponent	→ (expr)	(

Computing FIRST(α) (continued)

Extend FIRST to strings of symbols α

- want to define FIRST for all RHS of Productions

Let α = $Y_1Y_2Y_3..Y_n$

- put FIRST(Y_1) – ε into FIRST(α)
  - if ε is in FIRST(Y_1), put FIRST(Y_2) – ε into FIRST(α)
  - if ε is in FIRST(Y_2), put FIRST(Y_3) – ε into FIRST(α)
  - ...
  - if ε is in FIRST(Y_i) for all i, put ε into FIRST(α)

Given two productions for nonterminal *p*

- Production 1: *p* → α
- Production 2: *p* → β

FIRST(α) FIRST(β) ← look for current token

If only 1 has it, pick that production

If both have it, grammar is not LL(1)

If neither have it, if one FIRST set has ε in it, look at what terminals can follow p

FOLLOW sets

For single nonterminal *a*, FOLLOW(*a*) is the set of terminals that can appear immediately to the right of *a*

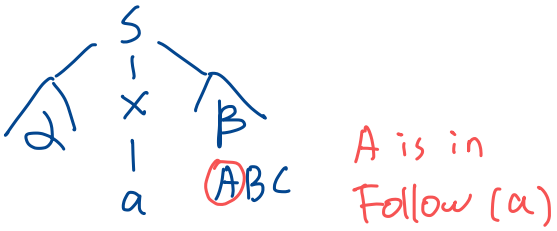
Formally,

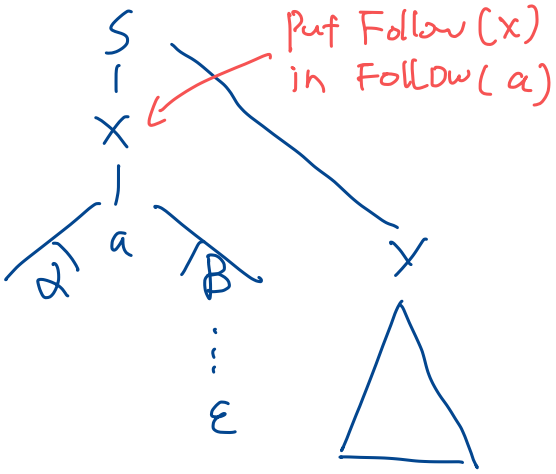
FOLLOW(*a*) = $\{ T \mid (T \in \Sigma \wedge S \Rightarrow^* \alpha a T \beta) \vee (T = EOF \wedge S \Rightarrow^* \alpha a) \}$ (terminals)

Computing FOLLOW sets

To build FOLLOW(a)

- if a is the start non-term, put EOF in FOLLOW(a)
- for each production x → α a β
  - put FIRST(β) – ε into FOLLOW(a)
  - if ε is in FIRST(β), put FOLLOW(x) into FOLLOW(a)
- for each production x → α a
  - put FOLLOW(x) into FOLLOW(a)





Building the parse table

```
for each production x → α {
    for each terminal T in FIRST(α) {
        put α in table[x][T]
    }
    if ε is in FIRST(α) {
        for each terminal T in FOLLOW(x) {
            put α in table[x][T]
        }
    }
}
```

CS 536 Announcements for Wednesday, March 22, 2023

Programming Assignment 3 – due Thursday, March 23

Midterm 2 – Wednesday, March 29

Last Time

- review grammar transformations
- building a predictive parser
- FIRST and FOLLOW sets

Today

- review parse table construction
- predictive parsing and syntax-directed translation

Next Time

- static semantic analysis
- exam review

Recap of where we are

Predictive parser builds the parse tree top-down

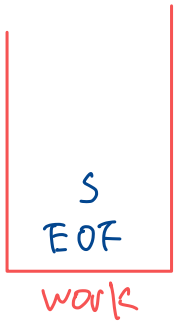
- 1 token lookahead
- parse/selector table
- stack tracking current parse tree's frontier

Building the parse table – given production *lhs* → *rhs*, determine what terminals would lead us to choose that production

ie figure out T so that Table[Lhs][T] = rhs

FIRST(α) = { T | (T ϵ Σ ∧ α =>* Tβ) ∨ (T = ε ∧ α =>* ε) }

FOLLOW(a) = { T | (T ϵ Σ ∧ s =>* αaTβ) ∨ (T = EOF ∧ s =>* αa) }



Curr token: D

S → a C
| b a

Look at FIRST(aC) & FIRST(ba)

If D is on neither, & ba ⇒* ε, then look at FOLLOW(ba)

FIRST and FOLLOW sets

FIRST(α) for α = $y_1\ y_2 \ldots y_k$

Add FIRST(y_1) – { ε }

If ε is in FIRST($y_{1 \text{ to } i-1}$), add FIRST(y_i) – { ε }

If ε is in all RHS symbols, add ε

FOLLOW(a) for x → α a β

If a is the start, add EOF

Add FIRST(β) – { ε }

Add FOLLOW(x) if ε is in FIRST(β) or β is empty

Note that

FIRST sets

- only contain alphabet terminals and ε
- defined for arbitrary RHS and nonterminals
- constructed by started at the beginning of a production

↳ at begining of RHS (for FIRST (LHS))

FOLLOW sets

- only contain alphabet terminals and EOF
- defined for nonterminals only
- constructed by jumping into production

Putting it all together

- Build FIRST sets for each nonterminal
- Build FIRST sets for each production's RHS
- Build FOLLOW sets for each nonterminal
- Use FIRST and FOLLOW sets to fill parse table for each production

Building the parse table

```
for each production x → α {
    for each terminal T in FIRST(α) {
        put α in table[x][T]
    }
    if ε is in FIRST(α) {
        for each terminal T in FOLLOW(x) {
            put α in table[x][T]
        }
    }
}
```

Example

CFG

s → a C | b a
a → A B | C s
b → D | ε

FIRST and FOLLOW sets

	FIRST sets	FOLLOW sets
s	A , C, D	EoF, C
a	A , C	C , EoF
b	D , ε	A , C
s → a C	A , C	
s → b a	D, A , C	
a → A B	A	
a → C s	C	
b → D	D	
b → ε	ε	

Parse table

```
for each production x → α
    for each terminal T in FIRST(α)
        put α in table[x][T]

    if ε is in FIRST(α)
        for each terminal T in FOLLOW(x)
            put α in table[x][T]
```

not LL(1)

	A	B	C	D	EOF
s	aC, ba		aC, ba	ba	
a	AB		Cs		
b	ε		ε	D	

Example

CFG

s → (s) | { s } | ε

FIRST and FOLLOW sets

	FIRST sets	FOLLOW sets
s	({ ε	EoF) }
s → (s)	(	
s → { s }	{	
s → ε	ε	

Parse table

```
for each production x → α
    for each terminal T in FIRST(α)
        put α in table[x][T]

    if ε is in FIRST(α)
        for each terminal T in FOLLOW(x)
            put α in table[x][T]
```

	(	)	{	}	EOF
s	(s)	ε	{ s }	ε	ε

Parsing and syntax-directed translation

Recall syntax-directed tranlation (SDT)

To translate a sequence of tokens

- build the parse tree
- use translation rules to compute the translation of each non-terminal in the parse tree, bottom up
- the translation of the sequence is the translation of the parse tree's root non-terminal

Goal: evaluate expression

CFG:		SDT rules:
expr →	expr + term	$expr_1$.trans = $expr_2$.trans + term.trans
	| term	expr.trans = term.trans
term →	term * factor	$term_1$.trans = $term_2$.trans * factor.trans
	| factor	term.trans = factor.trans
factor →	INTLIT	factor.trans = INTLIT.value
	| (expr)	factor.trans = expr.trans

The LL(1) parser never needed to explicitly build the parse tree
– it was implicitly tracked via the stack.

Instead of building parse tree, give parser a second, ***semantic*** stack

- holds translations of nonterms

SDT ***rules*** are converted to ***actions***

- Pop translations of RHS nonterms
- Push computed translation of LHS nonterm

translations are popped off the stack R-to-L

CFG:		SDT actions:
expr →	expr + term	tTrans = pop; eTrans = pop; push(eTrans + tTrans)
	| term	✗ tTrans = pop; push(tTrans)
term →	term * factor	fTrans = pop; tTrans = pop; push(tTrans * fTrans)
	| factor	✗ fTrans = pop; push(fTrans)
factor →	INTLIT	push(INTLIT.value)
	| (expr)	✗ eTrans = pop; push(eTrans)

useless rules

Parsing and syntax-directed translation (cont.)

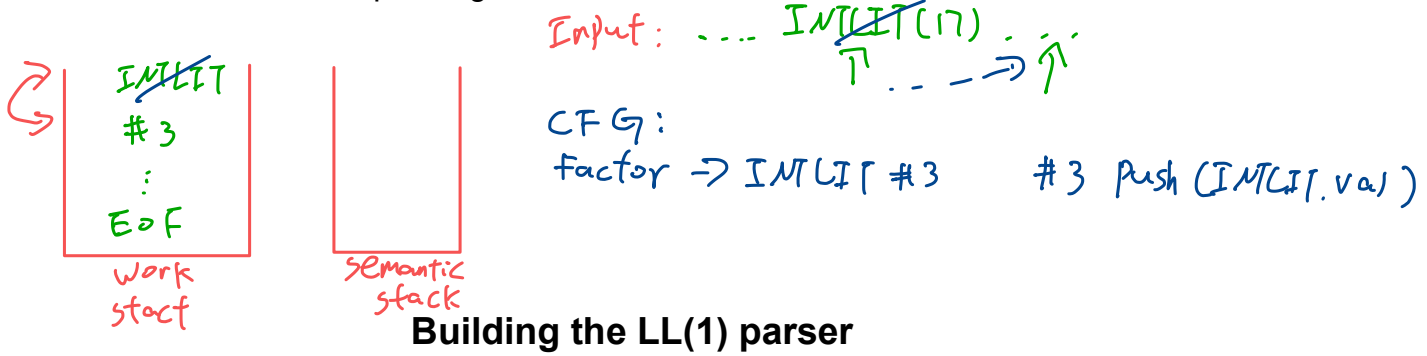
Augment the parsing algorithm

- number the actions (work)
- when RHS of production is pushed onto symbol stack, include the actions
- when action is the top of symbol stack, pop & perform the action

CFG:			SDT actions:
expr →	expr + term #1	#1	tTrans = pop; eTrans = pop; push(eTrans + tTrans)
|	term		
term →	term * factor #2	#2	fTrans = pop; tTrans = pop; push(tTrans * fTrans)
|	factor		
factor → #3	INTLIT	#3	push(INTLIT.value)
|	(expr)		

Placing the action numbers in the productions

- action numbers go
  - after their corresponding non-terminals
  - before their corresponding terminal



Building the LL(1) parser

1) Define SDT using the original grammar

- write translation rules
- convert translation rules to actions that push/pop using semantic stack
- incorporate action #s into grammar rules

2) Transform grammar to LL(1)

– treating actions # like terminals

3) Compute FIRST and FOLLOW sets

– treating action #s like ε

4) Build the parse table

Example SDT on transformed grammar

Original CFG:

expr → expr + term #1
| term

term → term * factor #2
| factor

factor → #3 INTLIT
| (expr)

Transformed CFG:

expr → term expr'
expr' → + term #1 expr'
| ε
term → factor term'
term' → * factor #2 term'
| ε
factor → #3 INTLIT | (expr)

Transformed CFG:

expr → term expr'

expr' → + term #1 expr' | ε

term → factor term'

term' → * factor #2 term' | ε

factor → #3 INTLIT | (expr)

SDT actions:

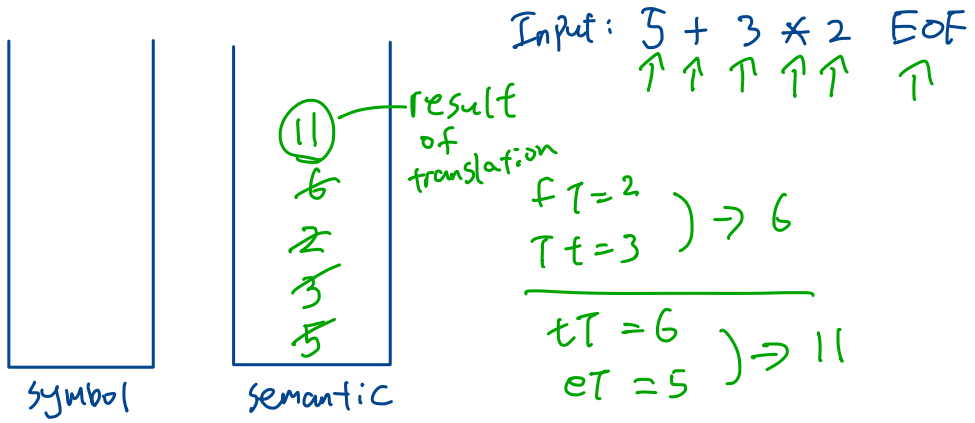
#1 : tTrans = pop;
eTrans = pop;
push(eTrans + tTrans)

#2 : fTrans = pop;
tTrans = pop;
push(tTrans * fTrans)

#3 : push(INTLIT.val)

Parse table

	+	*	(	)	INTLIT	EOF
expr			term expr'		term expr'	
expr'	+ term #1 expr'			ε		ε
term			factor term'		factor term'	
term'	ε	* factor #2 term'		ε		ε
factor			(expr)		#3 INTLIT	



What about ASTs?

Push and pop AST nodes on the semantic stack

Keep references to nodes that we pop

Original CFG:

expr → expr + term #1
| term

term → #2 INTLIT

Transformed CFG:

expr → term expr'
expr' → + term #1 expr'
| ε
term → #2 INTLIT

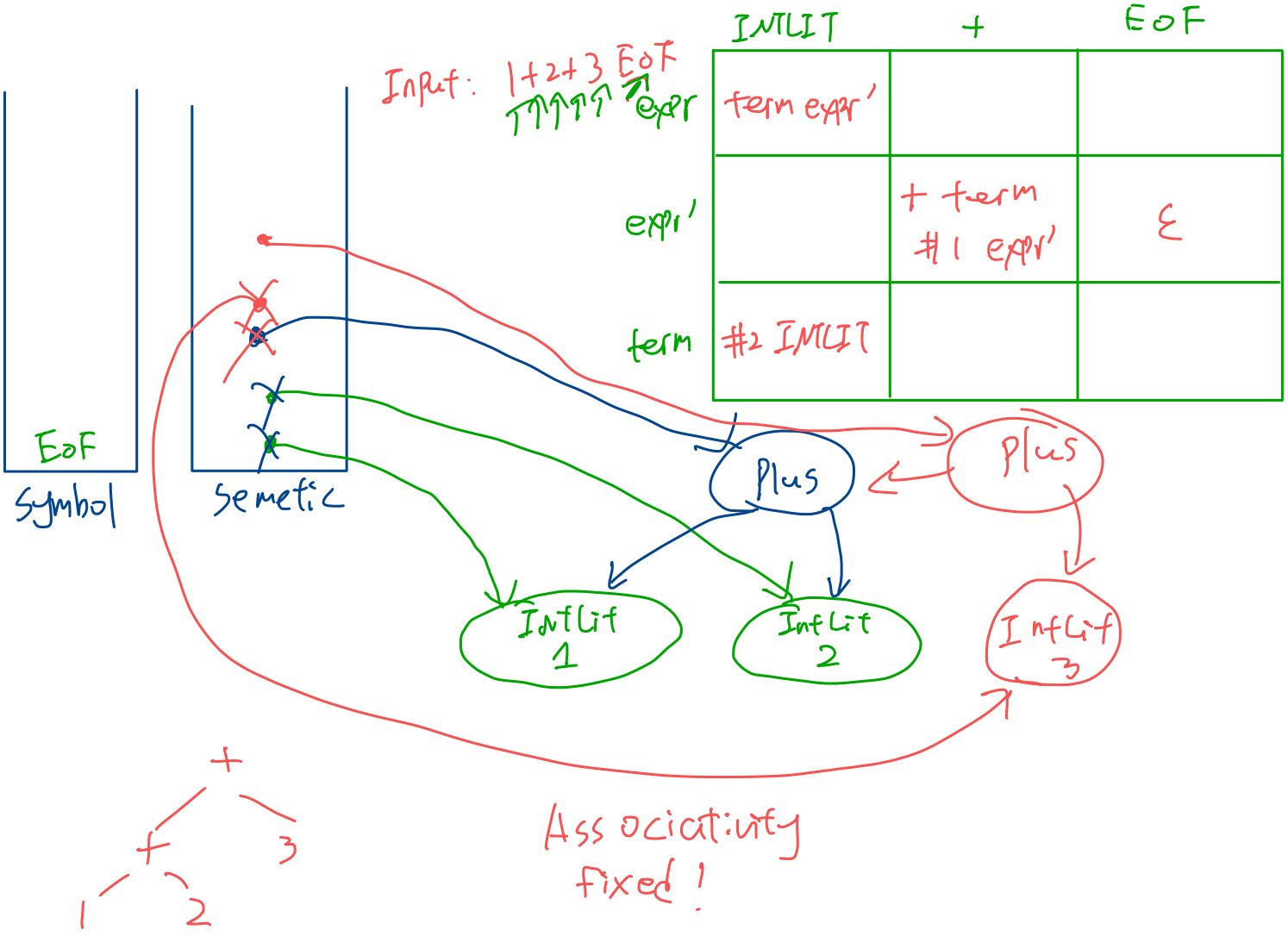
SDT actions:

#1 : tTrans = pop;
eTrans = pop;
push(new PlusNode (eTrans, tTrans))

#2 : push(new IntLitNode(INTLIT.value))

Parse table:

	INTLIT	+	EOF
expr	term expr'		
expr'		+ term #1 expr'	ε
term	#2 INTLIT		



Associativity fixed!

CS 536 Announcements for Monday, March 27, 2023

Midterm 2

- Wednesday, March 29, 7:30 – 9 pm
- B10 Ingraham Hall
- bring your student ID

Last Time

- building a predictive parser
- predictive parsing and syntax-directed translation

Today

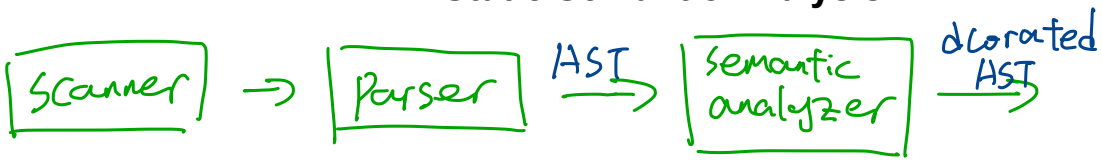
- static semantic analysis
- start name analysis
- exam review

Next Time

- Wednesday, March 2 – no lecture
- Monday, April 3 – continue name analysis

Static Semantic Analysis

Known at compile time



Two phases

- **name analysis** (aka **name resolution**)
  - for each scope
    - process declarations
      – add entries to symbol table
      – report multiply-declared names.
    - process statements
      – update IdNodes to point to appropriate symbol table entry
      – find uses of undeclared variables.
- **type checking**
  - process statements
    – use symbol table to find types of each expression & sub-expression
    – find type errors

Why do we need this phase?

Code generation

- different operations use different instructions
  - consistent variable access
  - integer addition vs floating-point addition
  - operator overloading

Optimization

- symbol table entry serves to identify which variable is used
  - can help in removing dead code (with some further analysis)
  - note: pointers can make these tasks hard

Error checking

Semantic error analysis

For non-trivial programming languages, we run into fundamental undecidability problems:

- does the program halt?
- does the program crash?

Even with simplifying assumptions (sometimes infeasible in practice) as well

- combinations of thread interleavings
- inter-procedural data analysis

In general - can't guarantee the absence of errors.

Goal of static semantic analysis: catch some obvious errors

- undeclared identifiers
- multiply-declared identifiers
- ill-typed terms.

Name analysis

Associating IDs with their uses

Need to bind names before we can do type analysis

Questions to consider:

- What definitions do we need about identifiers? → symbol table
- How do we bind definitions and uses together? → scope

Symbol Table

= (structured) dictionary that binds a name to information we need

Each entry in the symbol table stores a set of attributes:

- kind – record, variable, function, class
- type – integer, integer x string → boolean, record
- nesting level
- runtime location – where in memory is it stored

Symbol table operations

- insert entry
- lookup name
- add new sub-table
- remove/forget a sub-table

When do we do these operations?

Implementation considerations

efficiency of access is important

size unknown ahead of time → need expansion to be graceful & efficient.

don't need to delete entries

⇒ use hashtables.

Scoping

scope = block of code in which a name is visible/valid = life time of a name

No scope (flat name scope)

assembly, FORTRAN name is visible throughout program.

Static/most-nested scope – starting with ALGOL 60

- block structure
  - nested visibility
  - easy to tell which def of a name applies.
  - new decls apply to local scope
- name-scopes - limit region of definition

Kinds of scoping

static – can tell at compile time the correspondence between use & declaration

dynamic – correspondence determined at run time.

Scoping issues to consider

- Can the same name be used in multiple scopes? including nested scopes (variable shadowing)
- Can the same name be used multiple times in a single scope (if the names are of different kinds)? what about overloading?
- Where does declaration have to occur relative to use? are forward refrences allowed?
- How do we match up uses to declarations? static vs. dynamic
- What are the boundaries of scopes, e.g., are method/function parameters and local variables in the same scope?

CFG

s → a C | b a

a → A B | C s

b → D | ε

CYK example

aa → A

b b → B

cc → C

~~dd → D~~ useless

s → a cc | b a

a → aa bb | cc s | cc a

b → D

f → aa bb cc

↓

f → aa ss

ss → bb cc

transformation not needed in this example

Convert to CNF

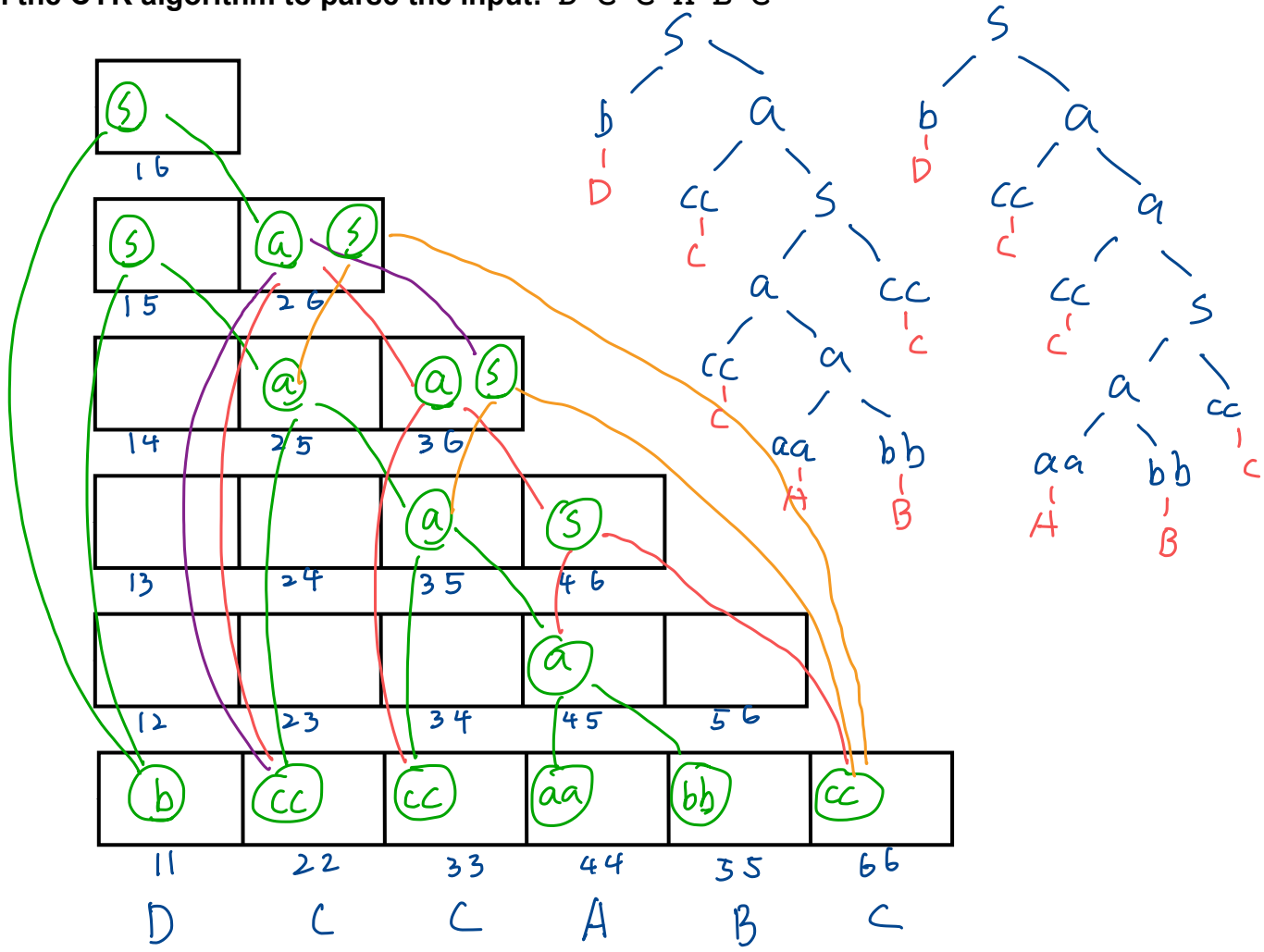
remove ε productions

eliminate unit productions

fix prods w/ terminals on RHS

fix prods w > 2 terminals on RHS

Run the CYK algorithm to parse the input: D C C A B C



FIRST/FOLLOW Example

Original CFG
expr → expr + term
| term
term → term * factor
| factor
factor → INTLIT
| (expr)

Transformed CFG
expr → term expr'
expr' → + term expr' | ε
term → factor term'
term' → * factor term' | ε
factor → INTLIT | (expr)

	FIRST	FOLLOW
expr	INTLIT (	EOF)
expr'	+ ε	EOF)
term	INTLIT (	+ EOF)
term'	* ε	+ EOF)
factor	INTLIT (	* + EOF)

Parse table

	+	*	(	)	INTLIT	EOF
expr			term expr'		term expr'	
expr'	+ term expr'			ε		ε
term			factor term'		factor term'	
term'	ε	* factor term'		ε		ε
factor			(expr)		INTLIT	

Building the parse table

```
for each production x → α
    for each terminal T in FIRST(α)
        put α in table[x][T]

    if ε is in FIRST(α)
        for each terminal T in FOLLOW(x)
            put α in table[x][T]
```

Question 3 (27 points)

Part (a) Suppose we have defined two new binary operators `?` and `$`. **Write an unambiguous context-free grammar** for the language of expressions with integer literal operands . . .

Solution:

exp → exp ? term
| term
term → factor $ term
| factor
factor → INTLIT
| (exp)

Part (c) If the grammar you wrote for part (a) has any immediate left recursion, apply the transformations learned in class to remove it, and write the result below. Give the *entire* grammar, not just the rules you transformed.

Part (d) If the grammar you wrote for part (b) needs left factoring, do that and write the result below. Give the *entire* grammar, not just the rules you transformed.

c) exp → term exp'
exp' → ? term exp' | ε
term → factor $ term
| factor
factor → INTLIT
| (exp)

term → factor term'
term' → $ term | ε

Question 5 (34 points)

Below is a context-free grammar for a language of simple blocks of code:

```
block     → LCURLY declList stmtList RCURLY
declList  → decl declList | ε
stmtList  → stmt stmtList | ε
decl      → TYPE ID SEMI
stmt      → ID EQUALS exp SEMI
          | IF LPAREN exp RPAREN block ELSE block
          | RETURN exp SEMI
exp       → ID | INT
```

Part (a) Fill in the *FIRST* and *FOLLOW* sets for the nonterminals below:

Non-terminal *X*	*FIRST*(*X*)	*FOLLOW*(*X*)
block	LCURLY	EOF
declList	TYPE ε	ID IF RETURN RCURLY
stmtList	ID IF RETURN ε	RCURLY
decl	TYPE	TYPE ID IF RETURN RCURLY
stmt	ID IF RETURN	ID IF RETURN RCURLY
exp	ID INT	SEMI RPAREN

CS 536 Announcements for Monday, April 3, 2023

Last Monday

- static semantic analysis
- name analysis
  - symbol tables
  - scoping
- exam review

Today

- name analysis

Next Time

- type checking

Static Semantic Analysis

Two phases

- name analysis → P4
- type checking → P5

Output: annotated AST

Name analysis

- for each scope
  - process declarations – add entries to symbol table
  - process statements – update IdNodes to point to appropriate symbol table entry
- each entry in symbol table keeps track of: kind, type, nesting level, runtime location
- identify errors
  - multiply-declared names
  - uses of undeclared variables
  - bad `record` accesses
  - bad declarations

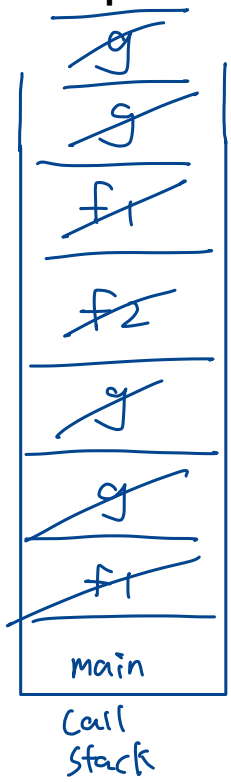
Scoping

- **scope** = block of code in which a name is visible/valid
- kinds of scoping
  - **static** – correspondence between use & declaration made at compile time
  - **dynamic** – correspondence between use & declaration made at run time

Dynamic scoping example

What does this print, assuming dynamic scoping?

```
void main() {
    int x = 10;
    f1();
    g();
    f2();
}
void f1() {
    String x = "hello";
    g();
}
void f2() {
    double x = 2.5;
    f1();
    g();
}
void g() {
    print(x);
}
```



output
hello
10
hello
2.5

Scope example

What uses and declarations are OK in this Java code?

```
class animal {
    // methods
    void attack(int animal) {
        for (int animal = 0; animal < 10; animal++) {
            int attack;
        }
    }
    int attack(int x) {
        for (int attack = 0; attack < 10; attack++) {
            int animal;
        }
    }
    void animal() { }

    //fields
    double attack;
    int attack;
    int animal;
}
```

not allowed

overloaded method can't only differ in return type

can't have multiple fields with same name

Scoping issues to consider

Can the same name be used in multiple scopes?

variable shadowing

Do we allow names to be reused in nesting relations?

```
void verse(int a) {
    int a;
    if (a) {
        int a;
        if (a)
            int a;
        }
    }
}
```

What about when the kinds are different?

```
void chorus(int a) {
    int chorus;
}
```

overloading

Same name; different type

```
int bridge(int a) { … }
bool bridge(int a) { … }
bool bridge(bool a) { … }
int bridge(bool a, bool b) { … }
```

← not allowed in Java

Where does declaration have to appear relatative to use?

forward references

How do we implement it?

```
void music(){
    lyrics();
}
void lyrics() {
    music();
}
```

Requires 2 passes
- 1 Pass to fill sym tab
- 1 Pass to use sym tab

Scoping issues to consider (cont.)

How do we match up uses to declarations?

Determine which uses correspond to which declarations

```
int 1 k = 10,  2 x = 20;
void 3 foo(int 4 k) {
    int 5 a = x 2 ;
    int 6 x = k 4 ;
    int 7 b = x 6 ;
    while (...) {
        int 8 x;
        if (x 8 == k 4 ) {
            int 9 k, 10 y;
            k 9 = y 10 = x 8 ;
        }
        if (x 8 == k 4 ) {
            int 11 x = y ;   <- error
        }
    }
}
```

Name analysis for brevis

brevis is designed for ease of symbol table use

- statically scoped
- global scope plus nested scopes
- all declarations are made at the top of a scope
- declarations can always be removed from table at end of scope

brevis scoping rules

- use most deeply nested scope to determine binding
- variable shadowing allowed
- formal parameters of function are in same scope as function body

Walk the AST

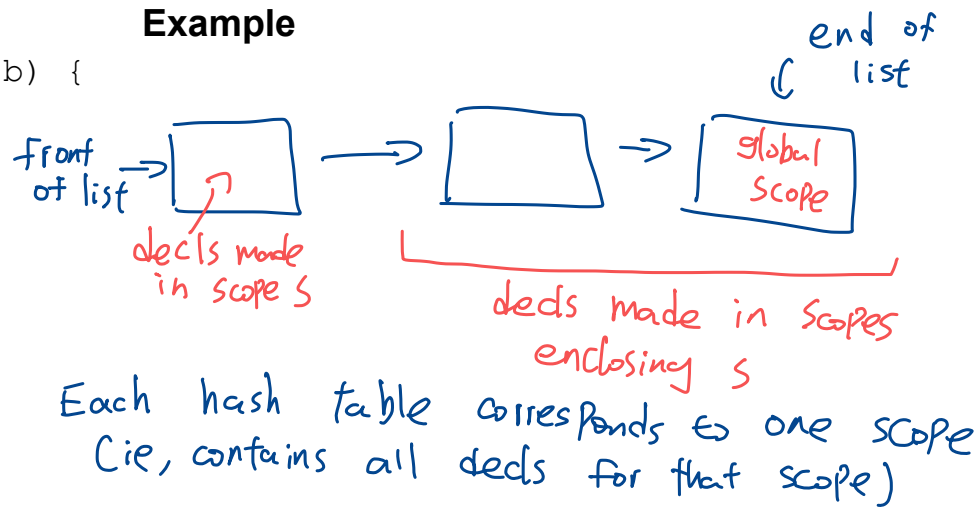
- put new entries into the symbol table when a declaration is encountered
- augment AST nodes where names appear (both declarations & uses) with a link to the relevant object in the symbol table

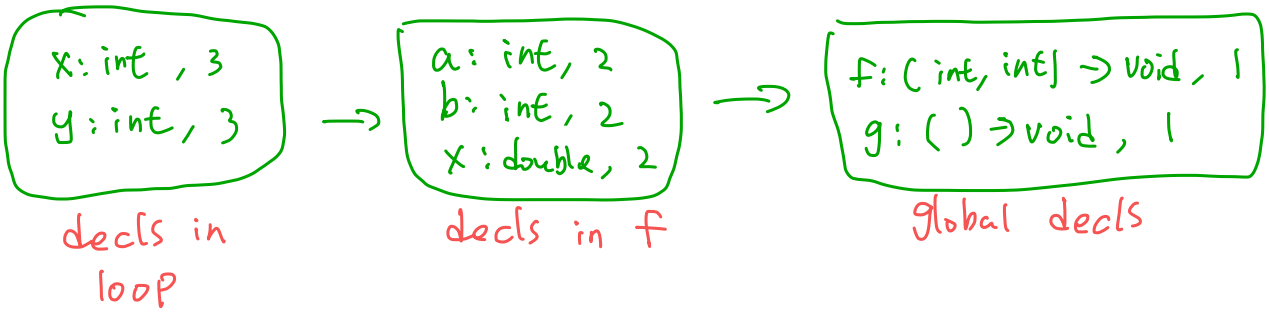
Symbol-table implementation

- use a list of hashmaps

Example

```
void f(int a, int b) {
    double x;
    while (...) {
        int x, y;
        ...
    }
}
void g() {
    f();
}
```





Symbol kinds

Symbol kinds (= types of identifiers)

- variable
  has a name, type
- function declaration
  has a name, return type, list of parameter types
- record declaration
  has a name, list of field (type w/ names), size

Implementation of Sym class

Many options, here's one suggestion

- `Sym` class for variable definitions
- `FnSym` subclass for function declarations
- `RecordDefSym` subclass for `record` type definitions
- `RecordSym` subclass for when you want an instance of a `record`

Name analysis and `records`

Symbol tables and `records`

- Compiler needs to
  - for each field: determine type, size, and offset with the record
  - determine overall size of record
  - verify declarations and uses of something of a `record` type are valid
- Idea: each `record` type definition contains its own symbol table for its field declarations
  - associated with the main symbol table entry for that `record`'s name

Relevant brevis grammar rules

```
decl          ::= varDecl
              | fnDecl
              | recordDecl      // record defs only at top level
              ;
varDeclList   ::= varDeclList varDecl
              | /* epsilon */
              ;
varDecl       ::= type id SEMICOLON
              | RECORD id id SEMICOLON
              ;
...
recordDecl    ::= RECORD id LPAREN recordBody RPAREN SEMICOLON
              ;
recordBody    ::= recordBody varDecl
              | varDecl
              ;
...
type          ::= BOOL
              | INT
              | VOID
              ;
loc           ::= id
              | loc DOT id
id            ::= ID
              ;
```

Definition of a `record` type

```
record Point (
    int x;
    int y;
);

record Color (
    int r;
    int g;
    int b;
);

record ColorPoint (
    record Color color;
    record Point point;
);
```

- make sure not already in sym tab
- Create a symtab for this record & store in sym for record's name
- for each varDecl in body of record
  - if type is record, make sure record type is in global (main) sym tab
  - make sure field is not in record's sym tab (& then add it)

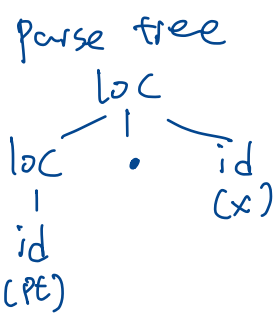
Declaring a variable of type `record`

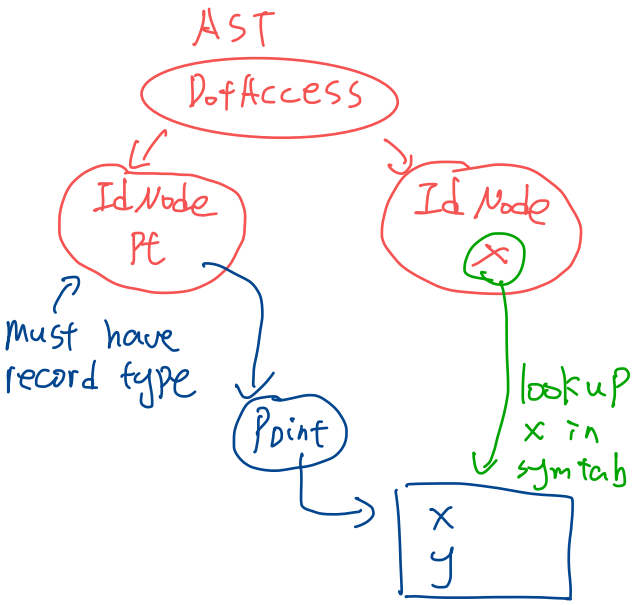
```
record Point pt;
record Color red;
record ColorPoint cpt;
```

- lookup (globally)
  - make sure if exists & is a record
- lookup (locally)
  - make sure if doesn't exist

Accessing fields of a `record`

```
pt.x = 7;
pt.y = 8;
pt.z = 10;

red.r = 255;
red.g = 0;
red.b = 0;

((cpt).point).x = pt.x;
cpt.color.r = red.r;
cpt.color.g = 34;
```

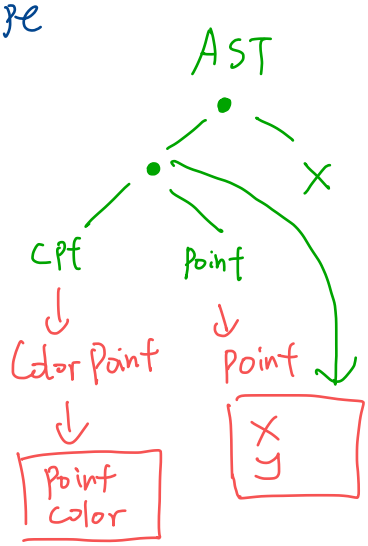




<u>If L child is an identifier</u>

- check identifier has been declared of record type
- get symbol table for that record type
- lookup R child in symtab

Recursively handle L child



<u>If L child is a dot-access</u>

- recursively process L child
- if symbol table in • access node is null

  then error

  else lookup R child in sym tab

<u>If R child is a record type</u>

- then set ref in • access to record's sym tab
- else set ref to null

CS 536 Announcements for Wednesday, April 5, 2023

Last Time

- name analysis
- static vs dynamic scoping
- scoping issues to consider
- name analysis in brevis
  - scoping rules
  - symbol table
  - handling `records`

Today

- type checking
- type-system concepts
- type-system vocabulary
- brevis
  - type rules
  - how to apply type rules

Next Time

- runtime environments

Name analysis: handling classes

Similar to handling aggregate data structures

- also need to be able to search the class hierarchy
  to see if uses are to inherited fields & methods

Idea:

Symbol table for each class with two nesting hierarchies (ie "regular" sym tab)

1) for lexical scoping within methods
2) for inheritance hierarchy
   - not just a list of hash tables ⇒ hierarchy not necessarily linear

To resolve a name

- first look in lexical scoping sym tab (ie "regular" one)
- then search inheritance hierarchy.

What is a type?

Short for **data type**

- classification identifying kinds of data
- a set of possible values that a variable can possess
- operations that can be done on member values
- a representation (perhaps in memory)

Type intuition – is the following allowed?

```
int a = 0;
int *pointer = &a;
float fraction = 1.2;
a = pointer + fraction;
```

Components of a type system

base types (built-in/primitive) integer boolean void

rules for constructing types record struct class enum

means of determining if types are compatible or equivalent

Can values with different types be combined? If so, how?

Can we consider 2 types the same? If so, how?

```
record Point (
    integer x;
    integer y;
);
```

```
record Pair (
    integer a;
    integer b;
);
```

rules for inferring the type of an expression

Type rules of a language specify

What types the operands of an operator must be (+, >, ==, =)

```
double a;
int b;
a = b; ← allowed in Java, C++
b = a; ← not allowed in Java, legal in C++
```

What type the result of an operator is

Type coercion

- implicit cast from one data type to another

  int j = 3.0; ← may have information loss

- type promotion – destination type can represent source type

  double f = 123;

Places where certain types are expected

```
if (x = 4) {
  ...
}
```

ok if we allow assignment to return a value

(eg x = y = z = 7;)

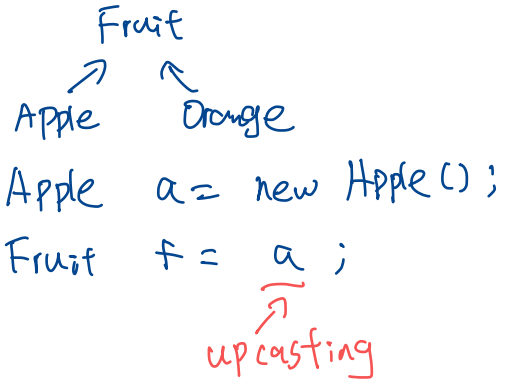
In brevis condition of if must eval to boolean

Type checking: when do we check?

static typing – type checking at compile time

dynamic typing – type checking done at runtime

combination of the two ← Java does this




```
Apple a = new Apple();
Orange o = (orange) a;
```

Cross-casting – not allowed, static check

```
Fruit f = new Apple();
if(...) {
    f = new Orange();
}
Apple two = (Apple) f;
```

down casting – compiles

runtime error if f is Orange

Type checking: when do we check? (cont.)

Static vs dynamic trade-offs

- static
  + compile time error checking
  + compile-time optimization
- dynamic
  + avoid dealing with errors that don't matter
  + some added flexibility
  - failure can happen at runtime

Duck typing - type is defined by methods and properties

```
class bird:
    def quack() : print("quack")
class robobird
    def quack() : print("0100101101")
```

often dynamically checked

Type checking: what do we check?

strong vs weak typing - continuum with no percise defs.

- degree to which type checks are performed
- degree to which type errors are allowed to happened at runtime

General principles

- statically typed → stronger (fewer type errors possible)
- more implicit casting allowed → weak
- fewer checks performed at runtime → weaker

Example

C (weaker)

```
union either {
    int i;
    float f;
} u;

u.i = 12;

float val = u.f;
```

Standard ML (stronger)

```
real(2) + 2.0
```

Type safety

- All successful operations must be allowed by the type system
- Java is explicitly designed to be type safe – if you have a variable with some type, then it is guaranteed to be of that type.
- C is not (allows type safety violations)

```
printf("%s", 1);
struct big {
    int a[100000];
};
struct big *b = malloc(1);
```

memory safety issue

- C++ is a little better

```
class T1 { char a; }
class T2 { int b; }
int main() {
    T1 *myT1 = new T1();
    T2 *myT2 = new T2();
    myT1 = (T1 *)myT2;
}
```

allows unchecked casts

Type system of brevis

brevis's type system

- primitive types integer, boolean, void, string ← literals only
- type constructors record
- coercion boolean cannot be used as an integer & vice-versa

Type errors in brevis

Operators applied to operands of wrong type

- arithmetic operators must have integer operands
- logical operators must have boolean operands
- equality operators == \=
  - must have operands of the same type
  - can't be applied to
    - function names (but can be applied to function results)
    - record names
    - record variables
- other relational operators must have integer operands
- assignment operator =
  - must have operands of the same type
  - can't be applied to
    - function names
    - record names
    - record values

Expressions that, because of context, must be a particular type but are not

- expressions that must be boolean (in brevis)
  condition of if, condition of while
- reading scan -> X;
  X can't be a function, record name, record variable
- writing print <- X;
  X can't be a function, record name, record variable
  - but can be string, expression eg (7+3)

Related to function calls

- invoking (i.e., calling) something that is not a function
- invoking a function with
  - wrong number of arguments
  - wrong types of arguments
- returning a value from a `void` function ie can't have return X;
- not returning a value from a non-`void` function ie can't have return;
- returning wrong type of value in a non-`void` function

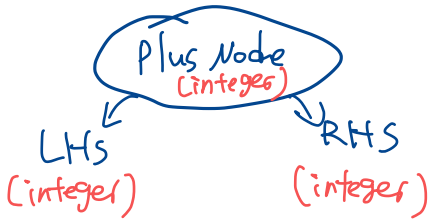
Type checking

Recursively walks the AST to

- determine the type of each expression and sub-expression using the type rules of the language
- find type errors (& report them)

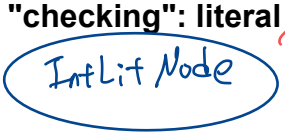
Add a `typeCheck` method to AST nodes

Type checking: binary operator



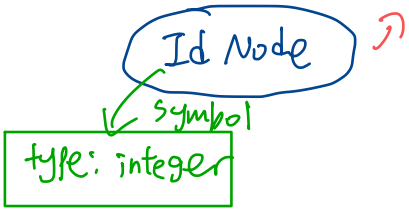
Get type of LHS
Get type of RHS
Check that types are compatible for operator
Set kind of node to be a value
set type of node to be the type of operator's result.

Type "checking": literal



Cannot be wrong
- just pass type of literal up the tree

Type checking: IdNode



Lookup type of declaration
- Should be a symbol linked to node (from name analysis)

Pass symbol type up the tree

Type checking: others

- call to function *f*
  - get type of each actual parameter of *f*
  - match against type of corresponding formal parameter of *f*
  - pass *f*'s return type up the tree
- statement *s*
  - type check constituents of *s*

use symbol table entry for f to get info

— nothing to pass up the tree — statement don't produce a value ⇒ s has no "return type"

Type checking (cont.)

Type checking: errors

Goals:

- report as many *distinct* errors as possible – don't give up after 1st error
- don't report *same* error multiple times – avoid error cascading – internally need to know if an error has already been reported.

Introduce internal `error` type

- when type incompatibility is discovered
  - report the error
  - pass `error` up the tree
- when a type check gets `error` as an operand
  - don't (re)report an error
  - pass `error` up the tree

Example:

```
integer a;
boolean b;
a = true + 1 + 2 + b;
b = 2;
```

CS 536 Announcements for Monday, April 10, 2023

Last Time

- type checking
- type-system concepts
- type-system vocabulary
- brevis
  - type rules
  - how to apply type rules

Today

- runtime environments
- runtime storage layout
- activation records
- static allocation
- stack allocation
- what happens on function call, entry, return

Next Time

- parameter passing

Type checking in brevis

brevis' type system

- primitive types: `integer` `boolean` `void`
- type constructors: `record`
- coercion: a `boolean` cannot be used as an `integer` is expected and vice versa

Type errors in brevis

- operators applied to operands of wrong type
- expressions that, because of context, must be a particular type but are not
- related to function calls

Type checking

- Recursively walks the AST to
  - determine the type of each expression and sub-expression using the type rules of the language
  - find type errors
- Add a `typeCheck` method to AST nodes

Type checking (cont.)

Type checking: errors

Goals:

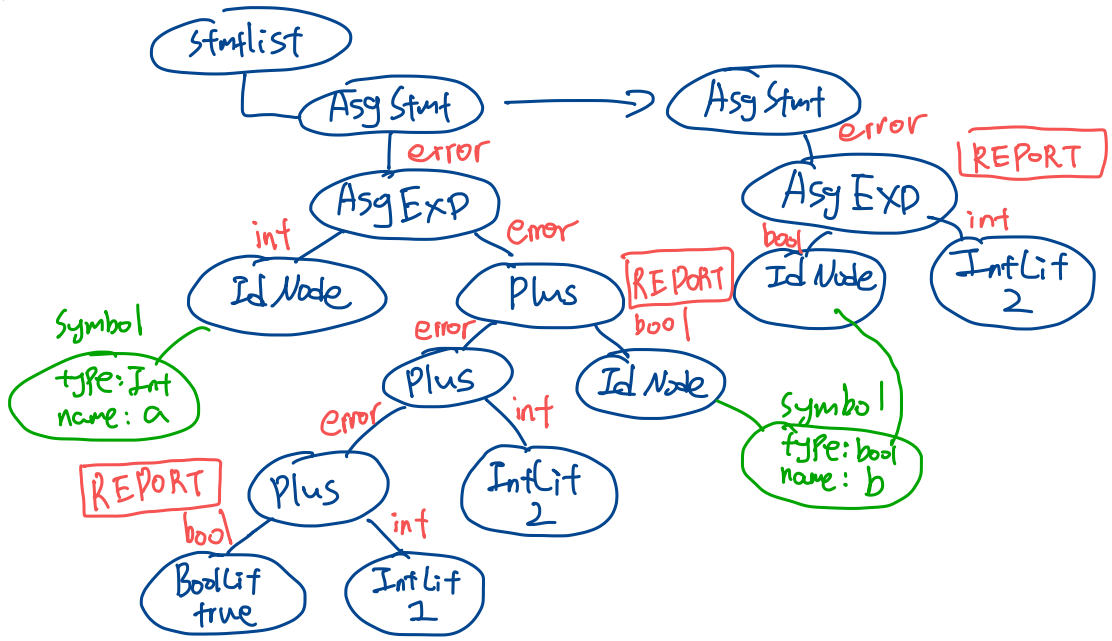
- report as many *distinct* errors as possible – don't give up after 1st error
- don't report *same* error multiple times – avoid error cascading – internally need to know if an error has already been reported.

Introduce internal `error` type

- when type incompatibility is discovered
  - report the error
  - pass `error` up the tree
- when a type check gets `error` as an operand
  - don't (re)report an error
  - pass `error` up the tree

Example:

```
integer a;
boolean b;
a = true + 1 + 2 + b;
b = 2;
```



Back to the big picture

Before code generation, we need to consider the ***runtime environment***:
= underlying software & hardware configuration assumed by the program

Program piggybacks on the operating system (OS)

- provides functions access to hardware
- provides illusion of uniqueness
- enforces some boundaries on what is allowed

Compiler must use runtime environment as best it can

- limited # of very fast registers to do computation
- comparatively large region of memory to hold data
- some basic instructions from which to build more complex behaviors

We need to create/impose conventions on the way our program accesses memory

- assembly code enforces very few rules
- conventions help to guarantee separately developed code works together

- allow modularity
- increases programmer efficiency

Issues to consider

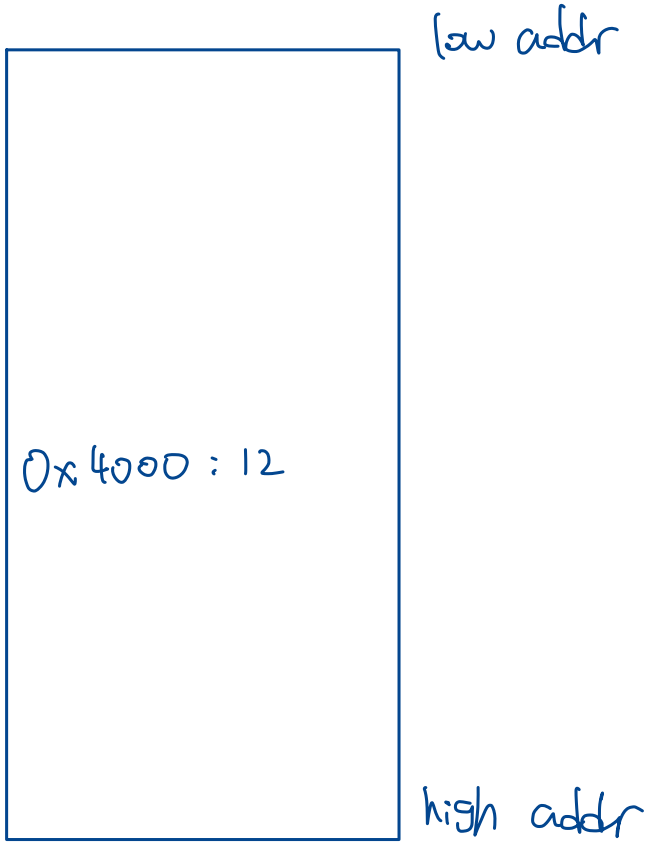
Variables

- How are they stored?
- What happens when a variable's value is needed?

How do functions work?

- What information should be stored for each function?
- What should happen when client code calls a function?
- What should happen when a function is entered?
- What should happened when a function returns?

General memory layout



Memory layout: static allocation

Region for global memory

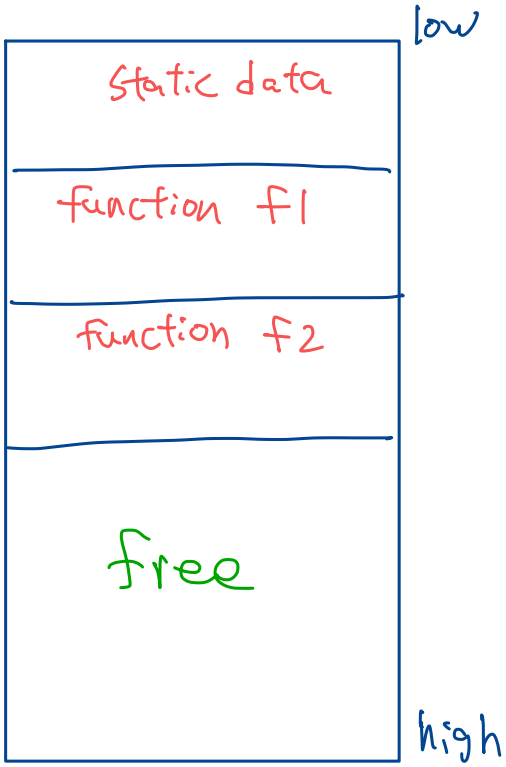
One "frame" for each procedure
- memory "slot" for each local, parameter
- memory "slot" for caller

Every time a function is called,
its names (local varibles & parameters)
refer to the ***same*** location in memory

+ fast access to all names

+ no overhead of stack/heap manipulation

– no recursion



frame

Memory layout: stack allocation

Allocate one ***activation record*** (AR) per invocation

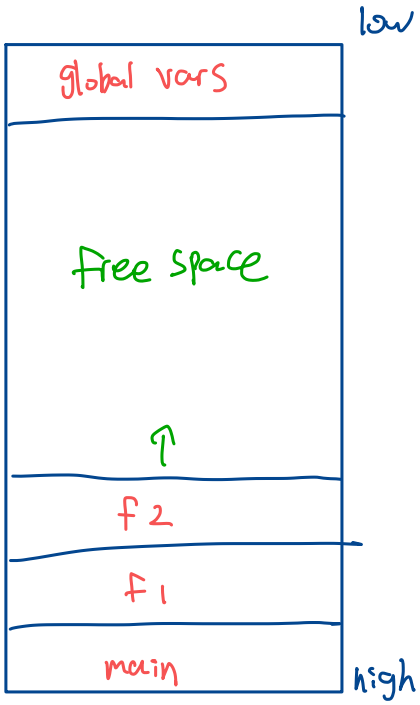
- use the stack — conceptually allows infinite recursion
- push a new AR on function entry
- pop AR on function exit
- to reduce the size, put static data in the global area

Stack size not known at compile time

- don't know (at compile-time) how many ARs there will be
- size of local variables may not be known
- each AR keeps track of the previous AR's boundaries

Activation record keeps track of

- local variables
- info about the call made by the caller
  - data context
    enough info to determine boundries of AR in use when current func was called
  - control context
    enough info to know code invoked the current function



Non-local dynamic memory

Don't always want all data allocated in a function call to disappear on return

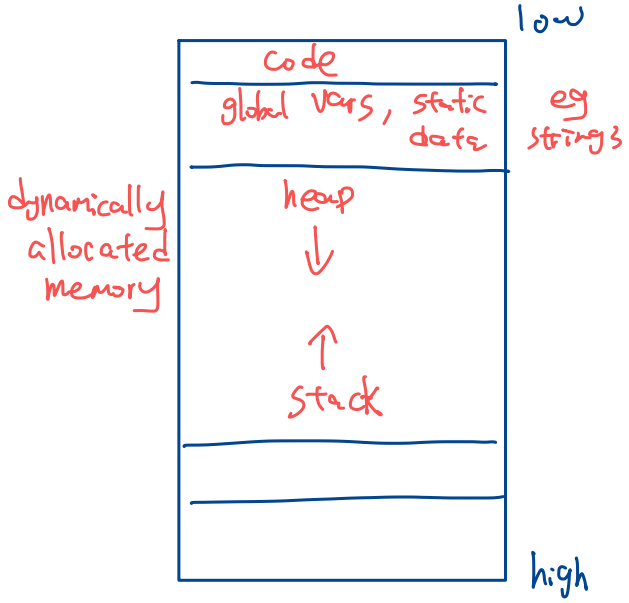
want to be able to create, eg, linked lists

Don't know how much space we'll need

allocate many such objects of varying sizes

The Heap

- region of memory independent of the stack
- allocated according to calls in the program
- how is memory "given back"?
  - programmer specifies when no longer in use (C)
  - runtime enviroment determines automatically when no longer in use. (Java)



Function calls

Instruction pointer (**$ip**) tracks the line (address) of code that it is executing

- if **$ip** points to code generated for some function, we'll say we are ***in*** that function

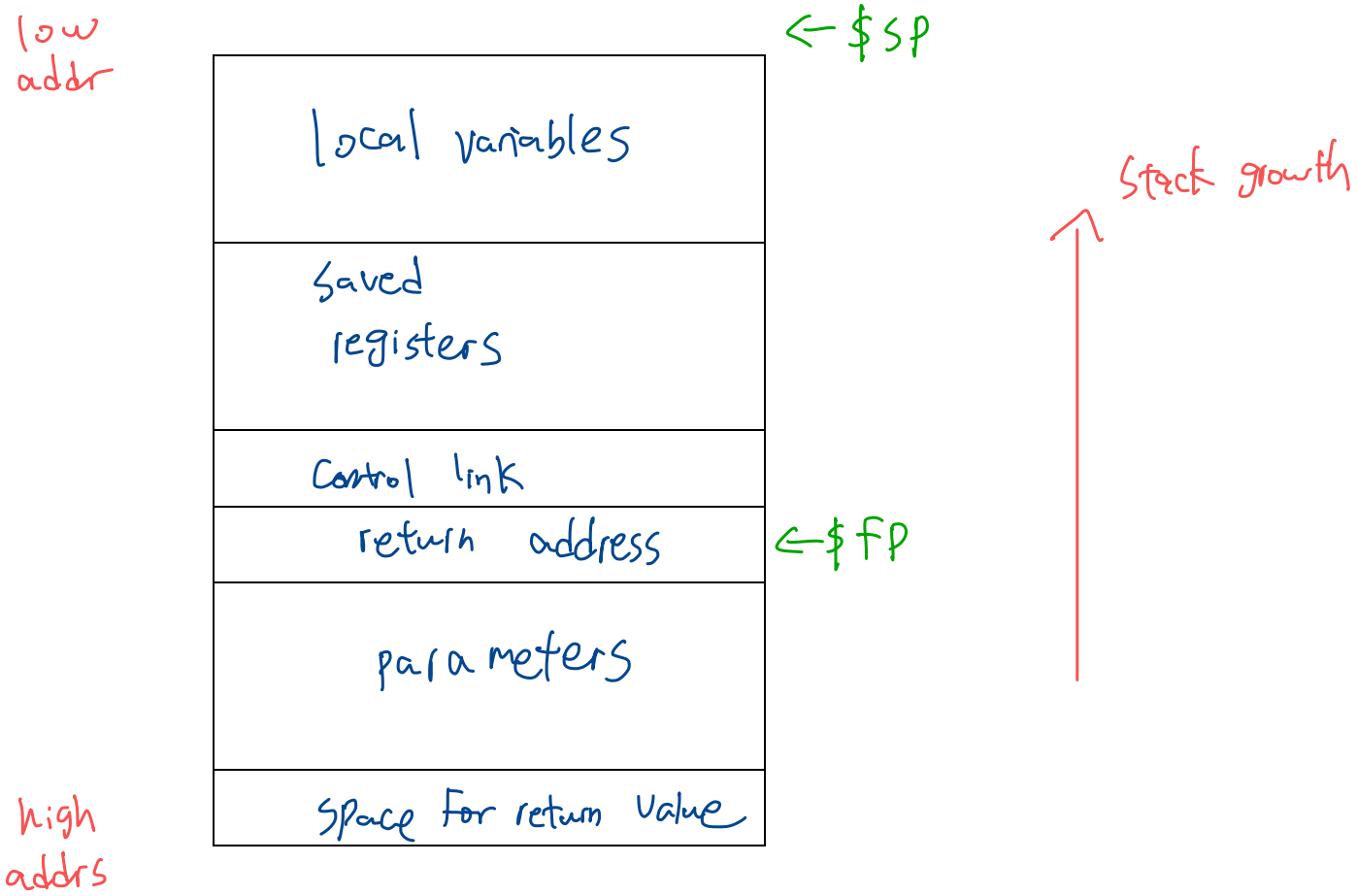
caller = function doing the invocation

callee = function being invoked

$sp (**stack pointer**) – points to top of stack (1st unused location)

$fp (**frame pointer**) – points to bottom of current AR

Activation records revisited



Function entry: caller responsibilities

Store the *caller-saved* registers in it's own AR

Set up the actual parameters

- set aside slot for the return value
- push parameters onto the stack

Copy return address out of **$ip**

Jump to first instruction of the callee

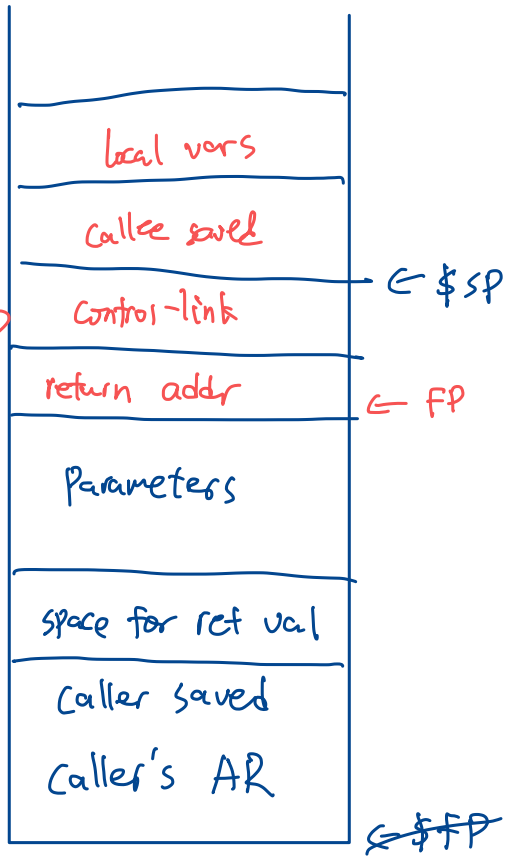
Function entry: callee responsibilities

Save **$fp** (it will need to be restored when the callee returns)

Update the base of the new AR to be the end of the old AR

Save *callee-saved* registers (if necessary)

Make space for locals



Function exit: callee responsibilities

Set the return value

Restore callee-saved registers

Grab stored return address

Restore *old* **$sp**

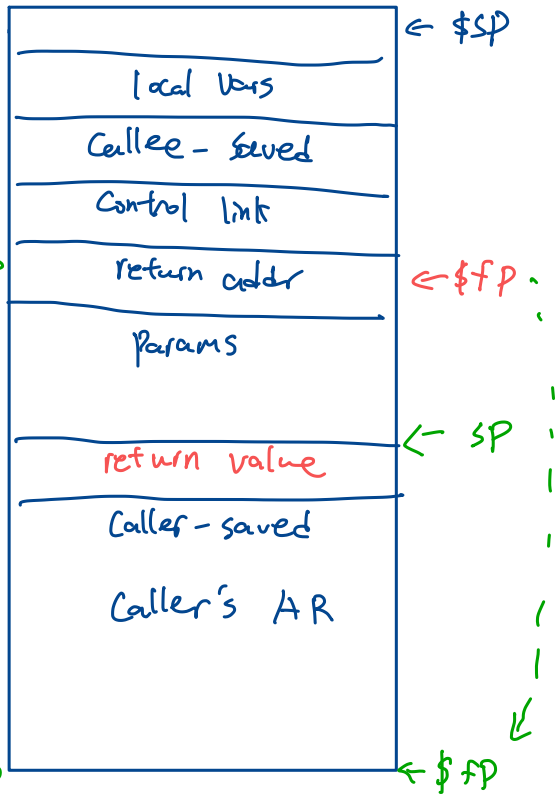
Restore *old* **$fp**

Jump to the stored return address

Function exit: caller responsibilities

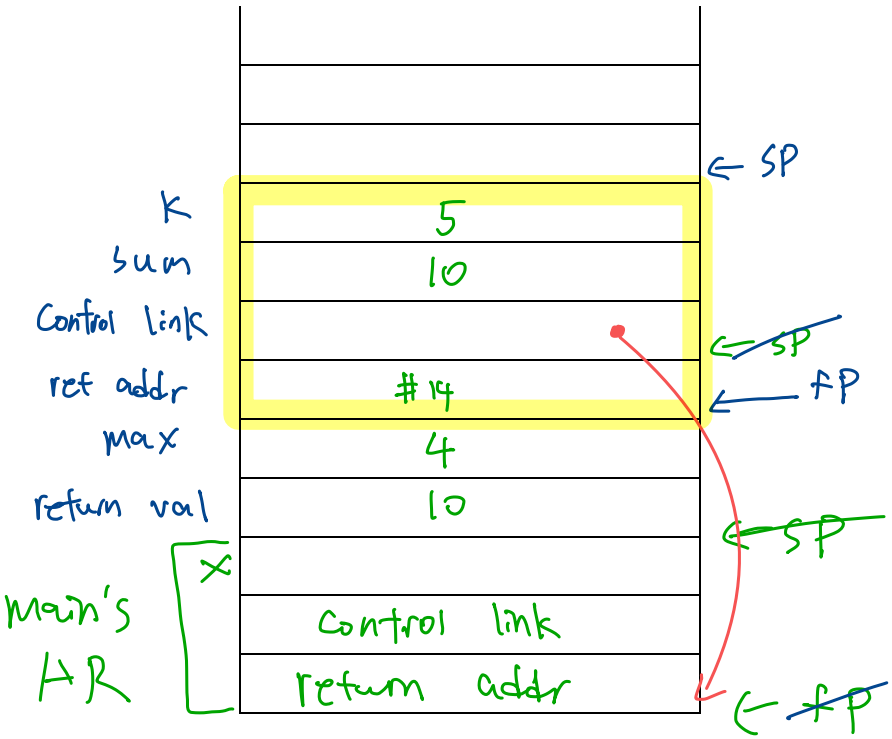
Pop the return value (or copy from register)

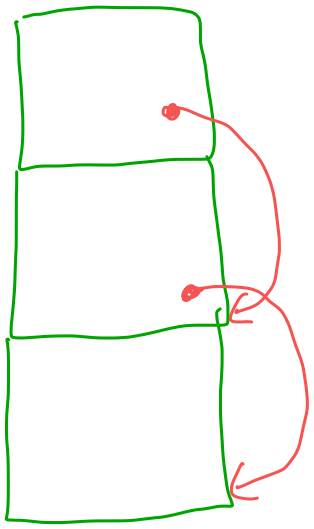
Restore caller-saved registers



Example

```
#1     integer summation(integer max) {
#2         integer sum;
#3         integer k;
#4         sum = 0;
#5         k = 1;
#6         while (k <= max) {
#7             sum = sum + k;
#8             k++;
#9         }
#10         return sum;
#11     }
#12    void main() {
#13        integer x;
#14        x = summation(4);
#15        print <- x;
#16    }
```





CS 536 Announcements for Wednesday, April 12, 2023

Last Time

- runtime environments
- runtime storage layout
- static vs stack allocation
- activation records
- what happens on function call, entry, return

Today

- parameter passing
- terminology
- different styles
  - what they mean
  - how they look on the stack
  - compare and contrast

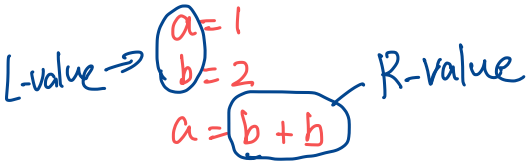
Next Time

- runtime access to variables in different scopes

Parameter passing: terminology

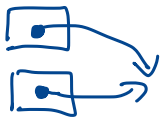
R-value – value of an expression

L-value – value with with a location



pointer – a variable whose value is a memory address

aliasing – when two (or more) variables hold the same address



In **definition** of function/method/procedure

```
void f(int x, int y, bool b) { . . . }
```

formals, formal parameters, parameters

In **call** to function/method/procedure

```
f(x + y, 7, true)
```

actuals, actual parameters, arguments

Types of parameter passing

pass by value

- when a procedure is called, the *values* of the actuals are copied into the formals

Java & C always use pass by value
C++ & Pascal can do this

pass by reference

- when a procedure is called, the *address* of the actuals are copied into the formals

C++ & Pascal can do this
C can simulate this in C by passing pointers

pass by value-result

- when a procedure is called, the values of actuals are passed
- when procedure is ready to return, final values of formals are copied back to the actuals

- actuals must be variables (ie have L-value), not an arbitrary expression
- used in Fortran IV & Ada (ie not very modern)

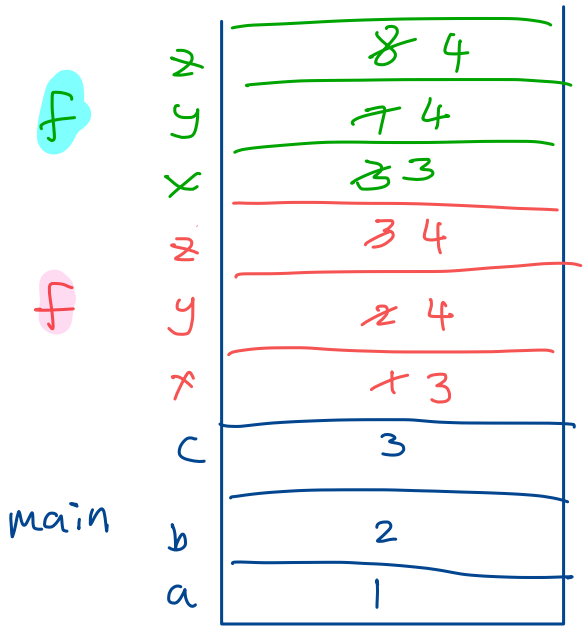
pass by name

- (conceptually) each time a procedure is called, the body of the procedure (the callee) is rewritten with the actual text of the actual parameters
- like macros in C/C++, but conceptually the rewriting occurs at runtime

- used in Algol
- hard to understand / debug

Example: pass by value

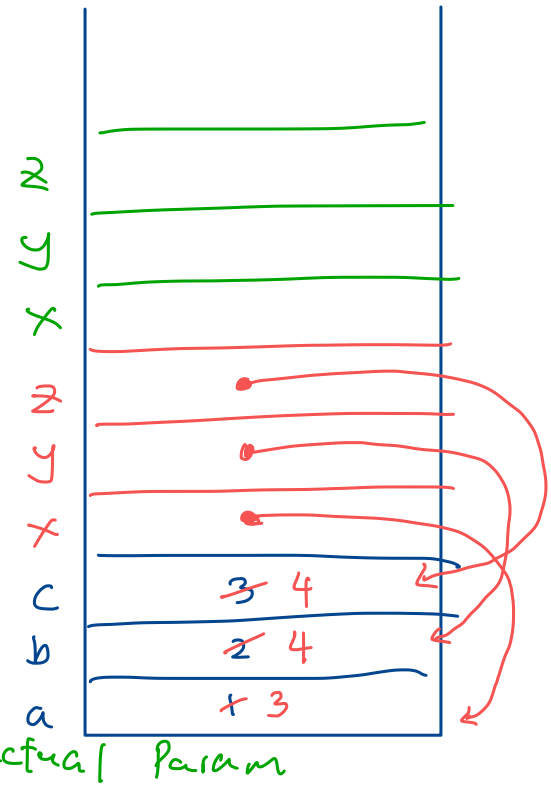
```
void f(int x, int y, int z) {
    x = 3;
    y = 4;
    z = y;
}

void main() {
    int a = 1, b = 2, c = 3;
    f(a, b, c);
    f(a+b, 7, 8);
}
```



Example: pass by reference

```
void f(int x, int y, int z) {
    x = 3;
    y = 4;
    z = y;
}

void main() {
    int a = 1, b = 2, c = 3;
    f(a, b, c);
    f(a+b, 7, 8);
}
```

error: actuals have R-value but don't have L-values, i.e., don't have a location



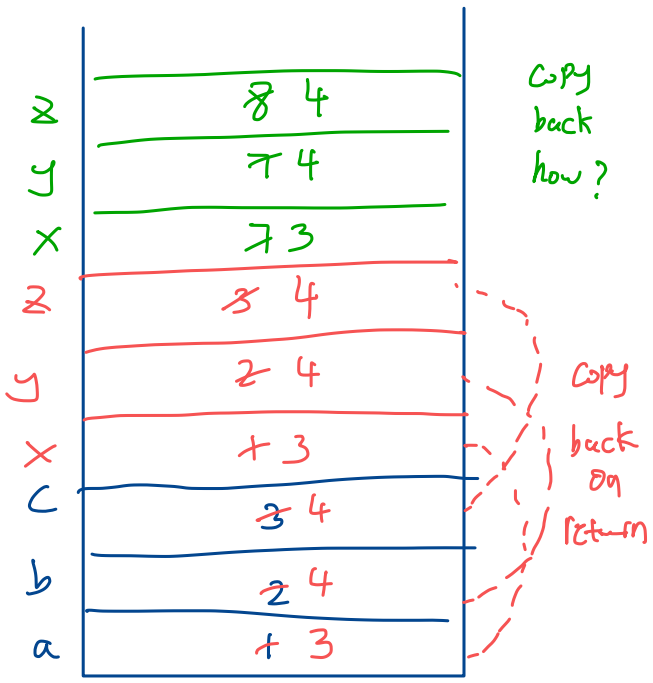
Note: typechecker would catch this error
- function type includes Param passing mode for each formal
- type checker would ensure that each actual Param passed by ref has an L-value.

Example: pass by value-result

```
void f(int x, int y, int z) {
    x = 3;
    y = 4;
    z = y;
}

void main() {
    int a = 1, b = 2, c = 3;
    f(a, b, c);
    f(a+b, 7, 8);
}
```

error: just like for pass by refrence (caught by typechecker)

Effect: Same as pass by refrence unless we have aliasing



Parameter passing example

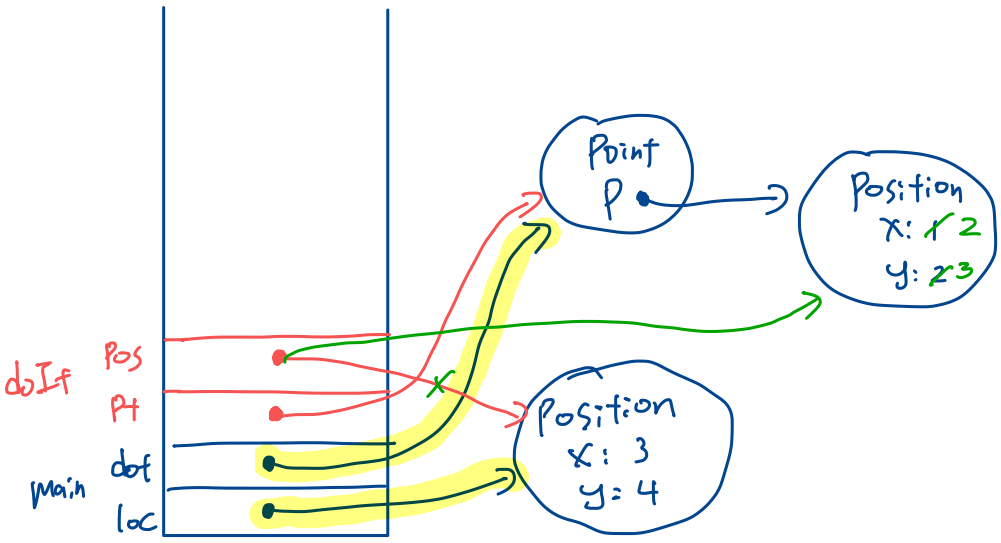
```
class Point {
     Position p;
     ...
}
class Position {
     int x, y;
     ...
}

void doIt(Point pt, Position pos) {
    pos = pt.p;
    pos.x++;
    pos.y++;
}

void main() {
    Position loc;
    Point dot;
    // code to initialize Point dot with position (1, 2)
    // code to initialize Position loc at (3, 4)
    doIt(dot, loc);
}
```

In Java, `loc` & `dot` are *references* to objects (in the heap)

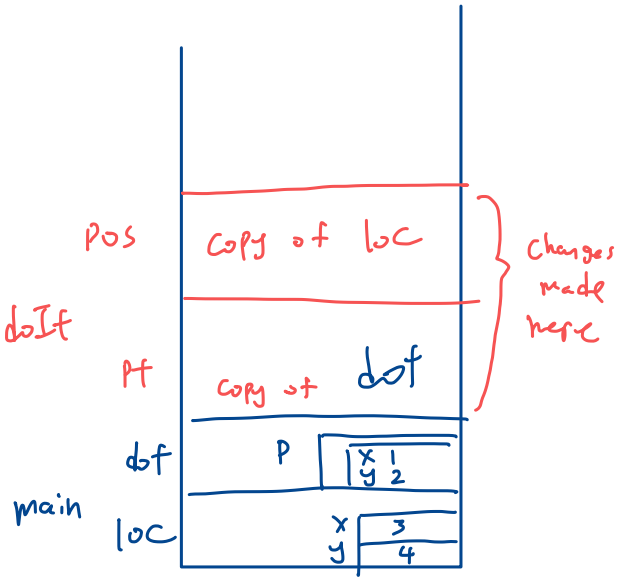
In C++, `loc` & `dot` *are* objects (in the AR of `main`)

Parameter passing example (cont.)

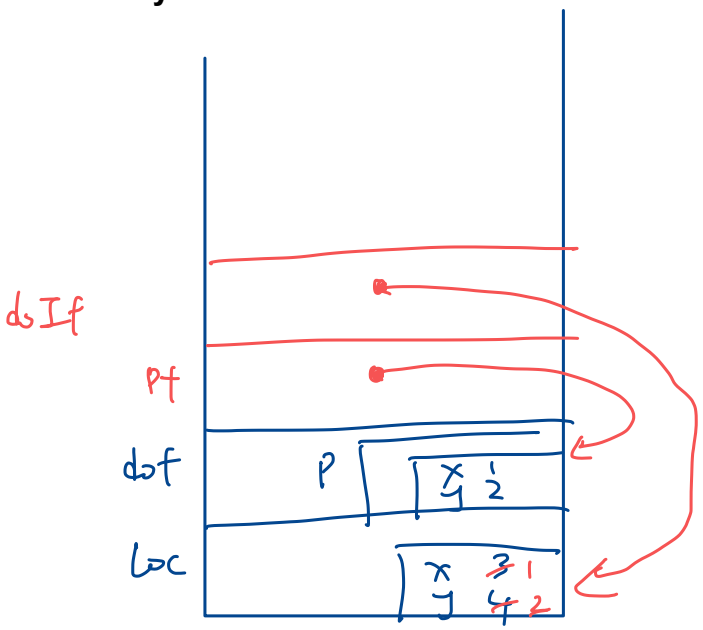
Pass by value in Java



Pass by value in C++



Pass by reference in C++



What are the (*x*,*y*) coordinates of `dot` and `loc` after the call to `doIt`?

	Pass by value (Java)	Pass by value (C++)	Pass by reference (C++)
`dot`	(2, 3)	(1, 2)	(1, 2)
`loc`	(3, 4)	(3, 4)	(2, 3)

Aliasing and parameter passing

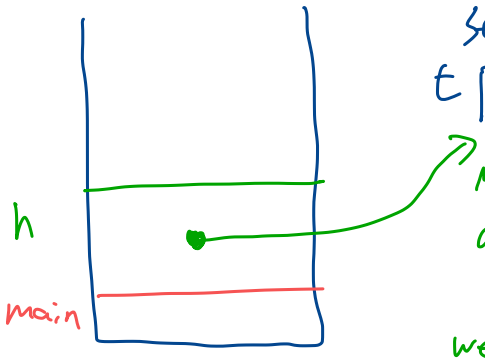
How aliasing can happen

- via pointers (in pass by value) – aliasing of actuals and formals – not really of interest here.

```
doIt(dot, loc);  // in Java
```

- when a global variable is passed by reference

```
int t = 0;

void h(int x) {
    x = 7;
    t = 4;
}

void main() {
    h(t);
}
```



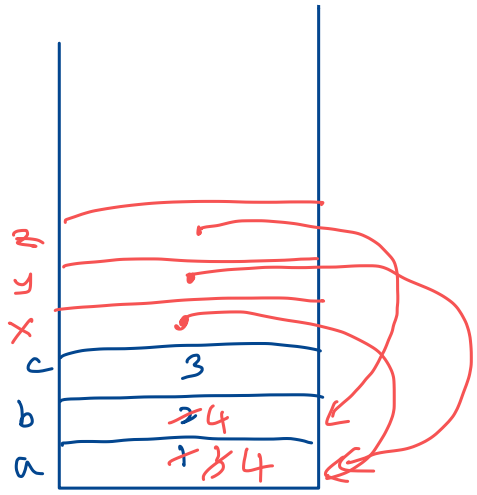
Static data

t 0 ↛ 4

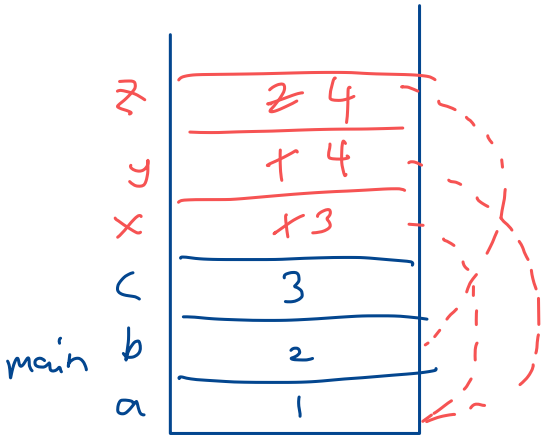
Note: if switch order of asgns in: t=4; x=7;
we can't guarantee t has the value 4 at end of h.

- when a parameter is passed by reference more than once

```
void f(int x, int y, int z) {
    x = 3;
    y = 4;
    z = y;
}

void main() {
    int a = 1, b = 2, c = 3;
    f(a, a, b);
}
```



What happens in pass by value-result?



when returning from f, what order are values copied back to actuals.

Options for handling this

1) compile error
2) order defined by the language (eg like Java)
3) order is implementation dependent (eg like C/C++)

Code generation and parameter passing

Efficiency considerations (calls, accesses by callee, return)

Pass by value

- Copy values into callee's AR (slow)
- callee directly access AR locations (fast)

Pass by reference

- copy addresses into callee's AR (fast)
- access in callee via indirection (slow)

Pass by value-result

- strictly slower than pass by value
  - need to know where to copy values back on return

Handling objects

```
class Point {
    Position p;
    ...
}

class Position {
    int x, y;
    ...
}

void doIt(Point pt, Position pos) {
    pos = pt.p;
    pos.x++;
    pos.y++;
}

void main() {
    Position loc;
    Point dot;
    // ... initialize dot with position (1, 2)
    // ... initialize loc at (3, 4)
    doIt(dot, loc);
}
```

In Java, `loc` and `dot` hold the addresses of objects

no overhead of copying entire object - just the address.

In C++, `loc` and `dot` are objects in the stack

use pointers to objects in heap for efficiency.

Compare and contrast

Pass by value

No aliasing - fewer unwanted side effects

Easier for static analysis (esp optimization)

called function (callee) is faster - no indirection
but the call (&copying of the values) may take time

Pass by reference

More efficient when passing large objects

Can modify actuals

Const ref in C++ - pass by ref but not allowed to be modified - Compiler checks & gives warning / error.

Pass by value-result

more efficient than pass by reference for small objects (no indirection)

If no aliasing, can be implemented as pass by reference for large objects (so still efficient)

BUT determining *if* there is aliasing (and *what* is aliased) is a challenging task (in general)

CS 536 Announcements for Monday, April 17, 2023

Last Time

- parameter passing
- terminology
- different styles
  - what they mean
  - how they look on the stack
  - compare and contrast

Today

- how do we deal with variables and scope?
- how do we organize activation records?
- how do we retrieve values of variables from activation records?

Next Time

- code generation

Accessing variables at runtime

local variables

- declared and used in the same function
- further divided into "block" scope in brevis

global variables

- declared at the outermost level of the program

in C/C++/brevis - globals integer x;

in Java - class (static) data members;
↑ Java keyword.

non-local variables (i.e., from nested scopes)

- for static scope: variables declared in an outer scope
- for dynamic scope: variables declared in the calling context

Compile-vs-runtime

nested classes (Java)

nested procedures (Pascal)

Accessing local variables at runtime

Local variables

- includes parameters and all local variables in a function
- stored in activation record of function in which they are declared
- accessed using offset from frame pointer

Accessing the stack

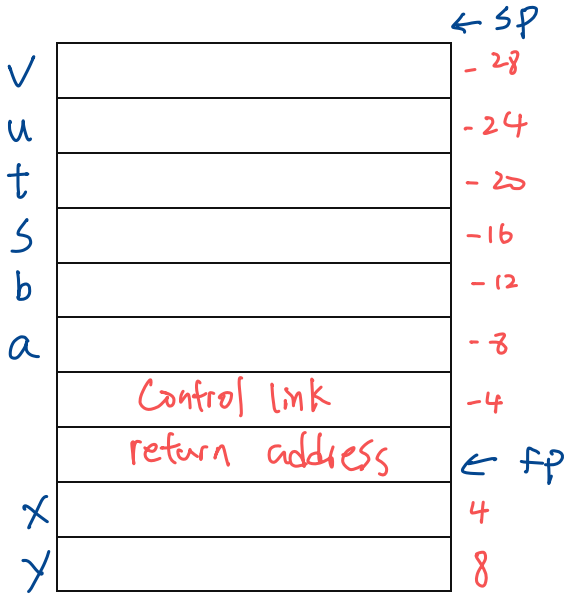
- general anatomy of MIPS instruction

  opcode operand 1 operand 2

- use "load" and "store" instructions
  - every memory cell has an address
  - calculate that memory address, then move data from/to that address

```
void test(int x, int y) {
    int a, b;
    ...
    if (...) {
        int s;
        ...
    }
    else {
        int t, u, v;
        ...
        u = b + y;
    }
}
```

Activation record for `test`



MIPS code for `u = b + y`

```
lw $t1, -12($fp)   # load b
```

comment symbol

lw: load word; $t1: reg; -12($fp): offset + framepointer

load contents of a given address into given register

```
lw $t2, 8($fp)   # load y
add $t3, $t1, $t2   # b+y
sw $t3, -24($fp)   # store u
```

Simple memory-allocation scheme

Reserve a slot for each variable in the function

Algorithm

For each function

```
set offset = +4
```

← start of 1st param

```
for each parameter
    add name to symbol table
    offset += size of parameter
```

w/ current offset

```
offset = -4
```

to account for control link

```
offset -= size of callee saved registers

for each local
    offset -= size of variable
    add name to symbol table
```

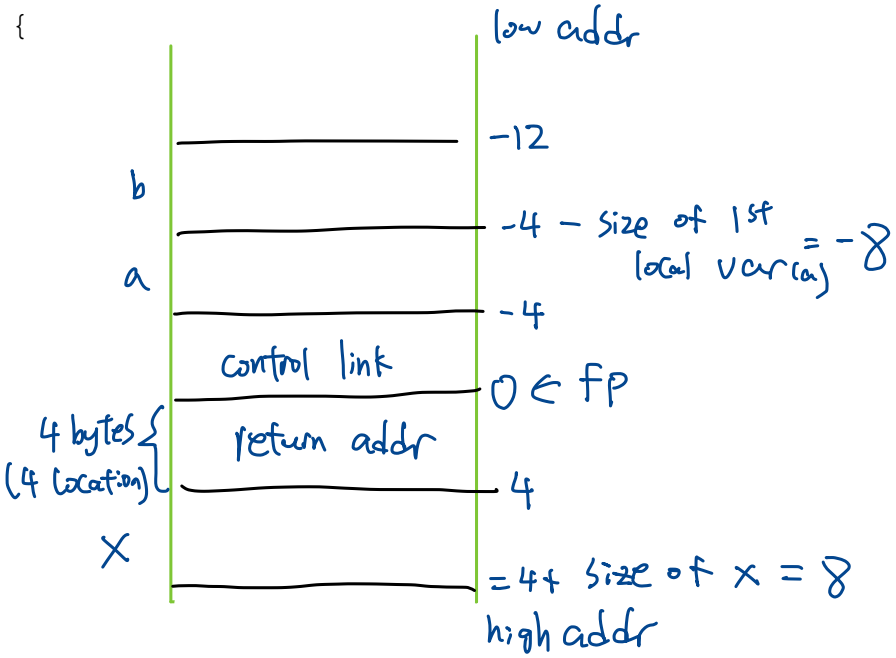
w/ current offset

Implementation (in P6)

- add an offset field to each symbol table entry
- during name analysis, add the offset along with the name
- walk the AST performing decrements at each declaration node

Example

```
void test(int x, int y) {
    int a, b;
    if (...) {
        int s;
    }
    else {
        int t, u, v;
        u = b + y;
    }
}
```



Accessing global variables at runtime

Place in static data area

- in MIPS, handled with a special storage directive
- variables referred to by name, not address

. data
. text directive for code

Note

- space allocated directly at compile time
- never needs to be deallocated

instead of indirectly through $fp, $sp registers

Example

initial value

```
.data
_x: .word 10
_y: .byte 1
_z: .asciiz "this is a string"

.text
lw $t0, _x  # load from x into $t0
sw $t0, _x  # store from $t0 into x
```

Accessing non-local variables at runtime

Two situations

- static scope
  - variable declared in one procedure and accessed in a nested one
- dynamic scope
  - any variable x that is not declared locally resolves to instance of x in the AR closest to the current AR

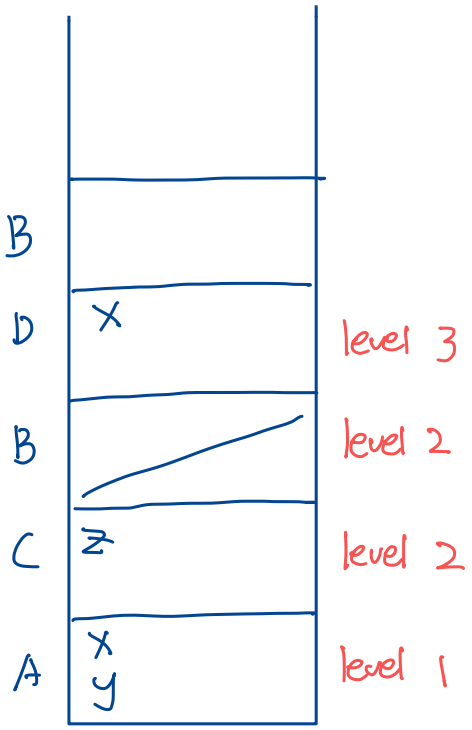
- each function has its own AR
- Variable stored in AR of procedure that declared it
- inner function access outer functions AR at runtime

Example: static non-local scope

```
function main() {
    int a = 0;

    function subprog() {
        a = a + 1;
    }
}
```

Example: static non-local scope

```
void procA() {  // level 1
   int x, y;
   void procB() {  // level 2
      print x;   x1 (always)
   }
   void procC() {  // level 2
      int z;
      void procD() {  // level 3
         int x;
         x = z + y;  x3 = z2 + y1
         procB();
      }

      x = 4;   x1 = 4
      z = 2;   z2 = 2
      procB();
      procD();
   }
   x = 3;   x1 = 3
   y = 5;   y1 = 5
   procC();
}
```



Access links

Add additional field in the AR (called access link, or static link)

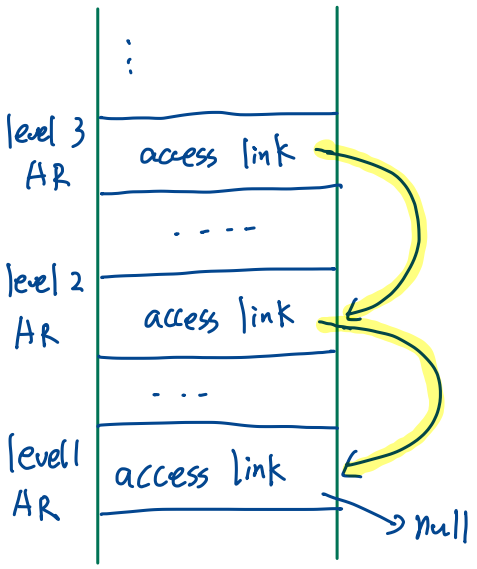
points to locals area (or fp) of enclosing function

How access links work

- we know how many *levels* to traverse statically

current scope is at nesting level 3
& variable to access is at level 1

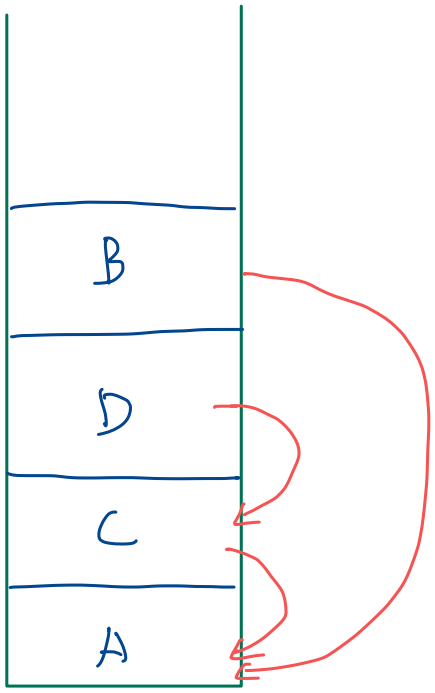
- go back 2 access levels
  3 - 1 = 2 levels



Access links (cont.)

Setting up access links

```
void procA() {  // level 1
   int x, y;
   void procB() {  // level 2
      print x;
   }
   void procC() {  // level 2
      int z;
      void procD() {  // level 3
         int x;
         x = z + y;
         procB();
      }

      x = 4;
      z = 2;
      procB();
      procD();
   }
   x = 3;
   y = 5;
   procC();
}
```



Handling use of non-local variable x (at compile time)

- each variable keeps track of nesting level in which it is declared
- when x is used in procedure P
  - follow predetermined # of links to get to AR for procedure in which x is declared

Lx = level of x's decl, Lp = level of p

links to follow is Lp - Lx

make $fp be location of access link

MIPS

```
lw $t0, 0($fp)   # 1 link followed
lw $t0, ($t0)    # 2 links followed
...
lw $t0, -12($t0) # use x's offset in AR of declaring procedure
```

Using a display

Idea: avoid run-time overhead of following access links by having a global array (called the display) containing links to the procedures that lexically enclose the current procedure

How it works

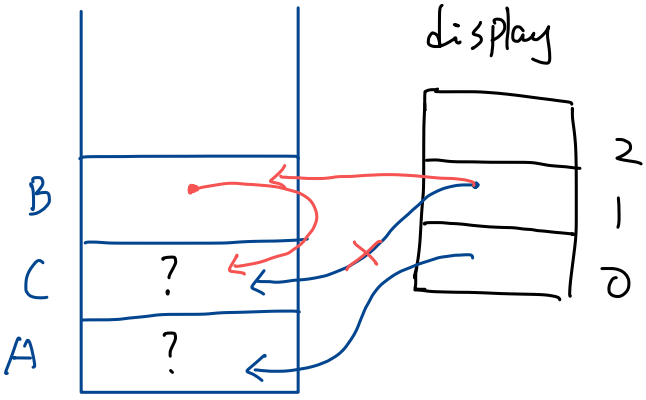
- given procedure `P` at nesting level `k` is currently executing
- `display[0], display[1], ..., display[k-2]` hold pointers to ARs of the most recent activations of the `k-1` procedures that enclose `P`
- `display[k-1]` holds pointer to `P` 's AR
- to access non-local variable `x` declared in nesting level `n`
  - use `display[n-1]` to get to AR that holds `x`
  - then use regular offset (within AR) to get to `x`

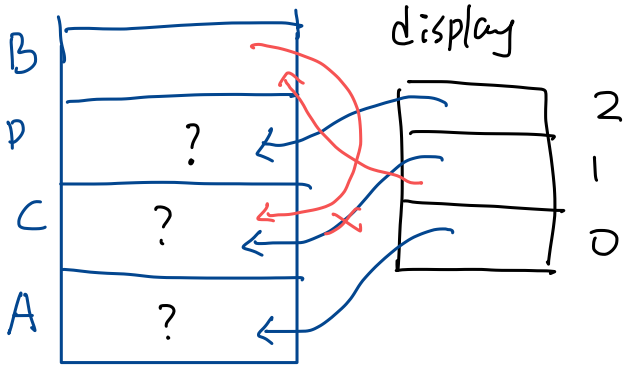
How to maintain the display in the code

- add new "save-display" field to AR
- when procedure `P` at nesting level `k` is called
  - save current value of `display[k-1]` in save-display field of `P` 's AR
  - set `display[k-1]` to point to save-display field of `P` 's AR
- when procedure `P` is ready to return
  - restore `display[k-1]` using value in save-display field

Example

```
void procA() {
   int x, y;
   void procB() {
      print x;
   }
   void procC() {
      int z;
      void procD() {
         int x;
         x = z + y;
         procB();
      }

      x = 4;
      z = 2;
      procB();
      procD();
   }
   x = 3;
   y = 5;
   procC();
}
```

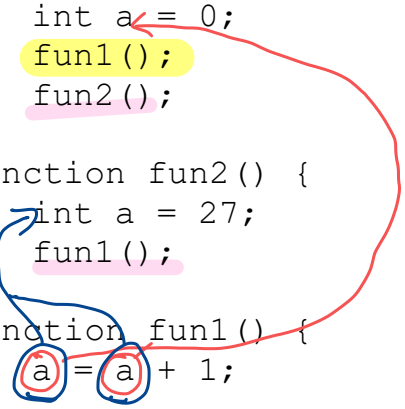




Dynamic non-local scope

Example

```
function main() {
    int a = 0;
    fun1();
    fun2();
}
function fun2() {
    int a = 27;
    fun1();
}
function fun1() {
    a = a + 1;
}
```



Key point – we don't know *which* non-local variable we are refering to (at compile-time)

Two ways to set up dynamic access

- deep access – somewhat similar to access links
- shallow access – somewhat similar to displays

Deep access

- if the variable isn't local
  - follow control link to caller's AR
  - check to see if it defines the variable
  - if not, follow the next control link down the stack
- note that we need to know if a variable is defined *with that name* in an AR
  - usually means we'll have to associate a name with a stack slot

Shallow access

- keep a table with an entry for each variable declaration
- compile a direct reference to that entry
- at function call on entry to function F
  - F saves (in its AR) the current values of all variables that F declares itself
  - F restores these values when it finishes

CS 536 Announcements for Wednesday, April 19, 2023

Last Time

- variable access at runtime
  - local vs global variables
  - static vs dynamic scopes

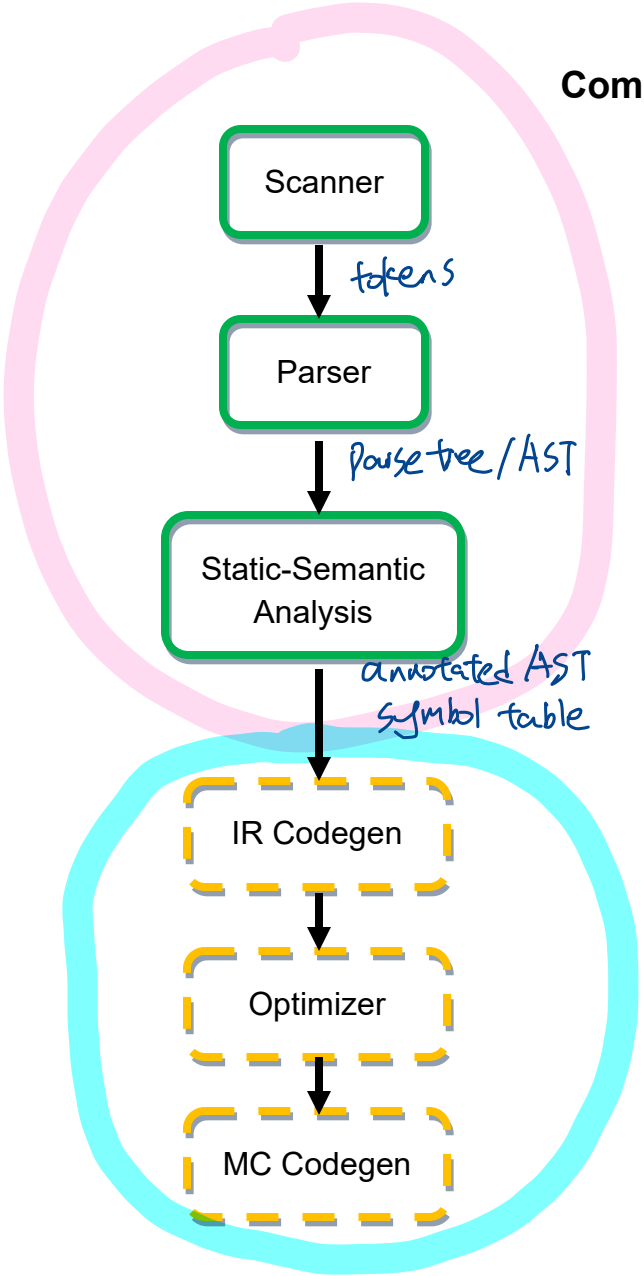
Today

- start looking at details of MIPS
- code generation

Next Time

- continue code generation

Compiler Big Picture



Front End

Back End

(unlike in front end)
- can skip phases without sacrificing correctness

what phases do we do?
How do we order the phases

Compiler Back End: Design Decisions

When do we generate?

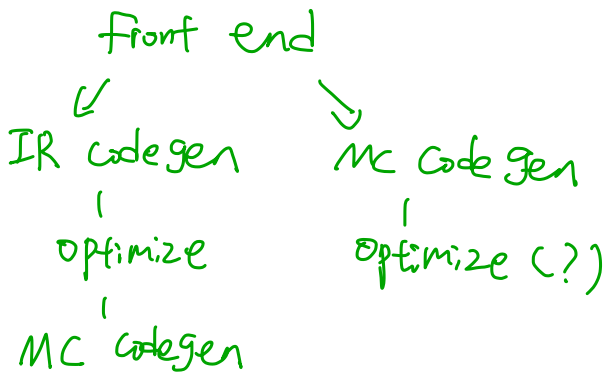
- directly from AST
- during SDT eg, during parsing
  - still need to do name analysis & type checking
  - → extra pass(es)?

How many passes?

- fewer passes
  - faster compilation
  - less storage required
  - increases burden on programmer
- more passes
  - heavyweight
  - can lead to better modularity

What do we generate?

- machine code
  - much faster to generate
  - less enginering in the compiler
- intermediate representation (IR)
  - more amenable to optimization
  - more flexible output options.
  - can reduce the complexity of code gen



Possible IRs

- CFG (control-flow graph)
- 3AC (three-address code)
  - instruction set for a fictional machine
  - every operator has at most 3 operands
  - provides illusion of infinitely many registers
  - "flatten out" expressions

3AC Example

3AC instruction set

Assignment
- x = y op z
- x = op y
- x = y

Jumps
- if (x op y) goto *L*

Indirection
- x = y[z]
- y[z] = x
- x = &y
- x = *y
- *y = x

Call/Return
- param x,k
- retval x
- call p
- enter p
- leave p
- return
- retrieve x

Type Conversion
- x = AtoB y

Labeling
- label L

Basic Math
- times, plus, etc.

Example

source code

```
if   (x + y * z > x * y + z)
     a = 0;
b = 2;
```

3AC code

```
tmp1 = y * z
tmp2 = x + tmp1
tmp3 = x * y
tmp4 = tmp3 + z
if (tmp2 <= tmp4) goto L
     a = 0
L: b = 2
```

3AC representation

- each instruction represented using a structure called a “quad”
  - space for the operator
  - space for each operand
  - pointer to auxilary info (label, succesor quad, etc.)
- chain of quads sent to an architecture-specific machine-code-generation phase

Code Generation

For brevis

- skip building a separate IR
- generate code by traversing the AST
  - add codeGen methods to AST nodes
  - directly emit corresponding code into file

Two high-level goals

- generate correct code
- generate *efficient* code

} hard to achieve both at same time

Simplified strategy

Make sure we don't have to worry about running out of registers

- for each operation, put all arguments on the stack
  ie for built-in ops, eg plus (as well as for user-defined functions)
- make use of the stack for computation
- only use two registers for computation $t0, $t1

Different AST nodes have different responsibilities

Many nodes simply "direct traffic"

- ProgramNode.codeGen - call codegen on its child
- List-node types - call codeGen on each element in list
- DeclNode
  - RecordDeclNode - no code to generate
  - FnDeclNode
  - VarDeclNode - what code to generate depends on context - global or local

Program
|
DeclList
↳ VarDecl → RecordDecl → FnDecl → ...
|
generate
code for global vars

↳ no code to generate

Code Generation for Global Variable Declarations

Source code:

```
integer name;
record MyRecord instance;
```

In AST: VarDeclNode

Generate:

```
        .data
        .align 2    # align on word boundaries
_name:  .space N    # N is the size of variable (in bytes)
```

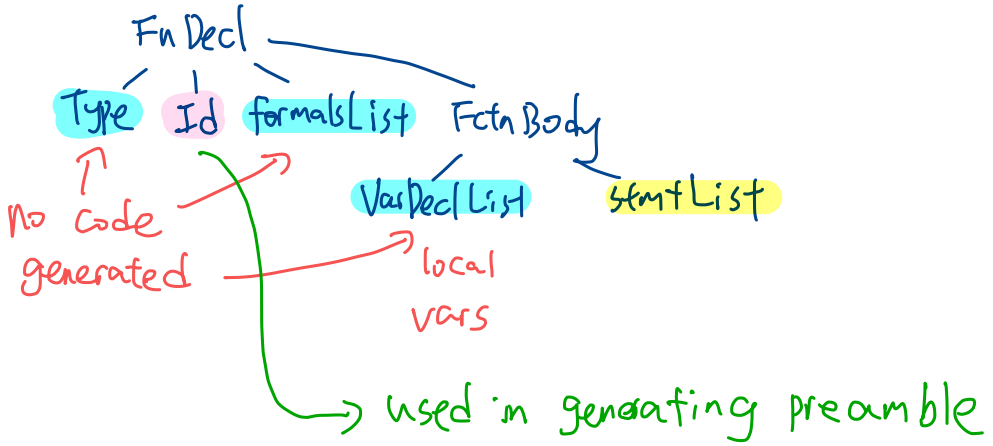
Size of variable – we can calculate this during name analysis

- for scalars, well-defined: integer, boolean are 4 bytes
- for records: 4*size of records

Code Generation for Function Declarations

Need to generate

- preamble – like a function signature
- prologue – sets up function's AR
- body – generate code for the function's statements
- epilogue – tear down the AR



MIPS Crash Course

Registers

Also has $LO and $HI, special purpose for multiplication & division

Register	Purpose
$sp	stack pointer
$fp	frame pointer
$ra	return address
$v0	used for system calls and to return int values from function calls, including the syscall that reads an int
$f0	used to return double values from function calls, including the syscall that reads a double
$a0	used for output of int and string values
$f12	used for output of double values
$t0 - $t7	temporaries for ints
$f0 - $f30	registers for doubles (used in pairs; i.e., use $f0 for the pair $f0, $f1)

$t0, $t1,

Program structure

Data

- label: `.data`
- variable names & size; heap storage

Code

- label: `.text`
- program instructions
- starting location: **`main`**

Preamble

For main function, generate
- .text
- .globl main

main:

For all other functions, generate
- .text

_fctnName:

↑ replaced with actual name of function

MIPS Crash Course (cont.)

Data

e.g.,

```
        name:   type        value(s)
v1:     .word   10
a1:     .byte   'a' , 'b'
a2:     .space  40
```

single integer initialized to 10

2 element char array with elts initialized to 'a' & 'b'

40 here is allocated space – no value is initialized

– allocates 40 consecutive bytes

Memory instructions

`lw` `register_destination, RAM_source`

- copy word (4 bytes) at source RAM location to destination register.

`lb` `register_destination, RAM_source`

- copy byte at source RAM location to low-order byte of destination register

`li` `register_destination, value`

- load immediate value into destination register

li $t0, 5

`sw` `register_source, RAM_dest`

- store word in source register into RAM destination

`sb` `register_source, RAM_dest`

- store byte in source register into RAM destination

Arithmetic instructions

```
add     $t0,$t1,$t2
sub     $t2,$t3,$t4
addi    $t2,$t3, 5
addu    $t1,$t6,$t7
subu    $t1,$t6,$t7

mult    $t3,$t4

div     $t5,$t6

mfhi    $t0
mflo    $t1
```

add/sub of signed (2's compliment) integers

add immediate

add/sub of un-signed integers

result is in $LO

stores result in $LO and remainder in $HI

move from $HI to $t0

move from $LO to $t1

MIPS Crash Course (cont.)

Control instructions

```
b       target
beq     $t0,$t1,target
blt     $t0,$t1,target
ble     $t0,$t1,target
bgt     $t0,$t1,target
bge     $t0,$t1,target
bne     $t0,$t1,target
```

unconditional branch to target

program label

branch to target if $t0 == $t1

```
j       target
jr      $t3
```

unconditional jump to target

indirect jump - jump to address in $t3

```
jal     sub_label       #  "jump and link"
```

jump to sub_label & store the return location in $ra

Check out: MIPS tutorial

https://minnie.tuhs.org/CompArch/Resources/mips_quick_tutorial.html

CS 536 Announcements for Monday, April 24, 2023

Last Time

- compiler backend design issues
  - we're going directly from AST to machine code
- start looking at code generation
  - global variables
  - function preamble
- start looking at details of MIPS

Today

- continue code generation
- function declaration
- function call and return
- expressions
- literals
- assignment
- I/O
- dot-access

Next Time

- wrap up code generation

Recall

Global variables

- one way

```
        .data
        .align 2
_name:  .space 4
```

- simpler form for primitives ← fine for P6 (brevis)

```
        .data
_name:  .word value
```

↑ initial value

Function Declarations

Need to generate

- preamble
- prologue
- body
- epilogue

See last lecture for special handling of main's Preamble

Preamble

```
integer f(integer a, integer b){
   integer c;
   c = a + b - 7;
   return c;
}
```

```
.text
_f:
# ... function body ...
```

This label gives us a place to jump to:

jal _f

Prologue

Need to

1. save the return address

```
sw $ra, 0($sp)
subu $sp, $sp, 4
```

} push $ra onto stack

2. save the frame pointer

```
sw $fp, 0($sp)
subu $sp, $sp, 4
```

} push $fp

3. update the frame pointer

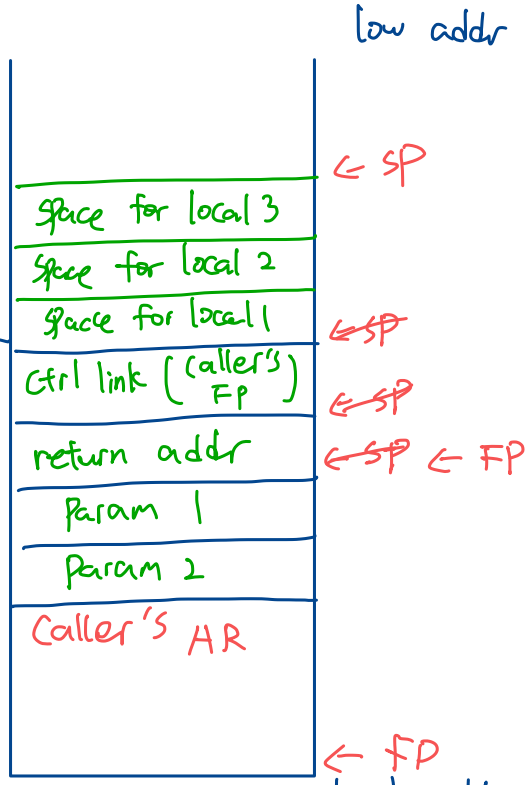
```
addu $fp, $sp, 8
```

$fp = $sp + 8

4. make space for locals

```
subu $sp, $sp, size
```

not initialized



During name analysis

- compute offsets of params & locals
- extend to compute size of all locals

Function Declarations (cont.)

Epilogue

Need to

1. restore return address

```
lw $ra, 0($fp)
```

2. restore the frame pointer

```
a  move $t0, $fp
b  lw $fp, -4($fp)
```

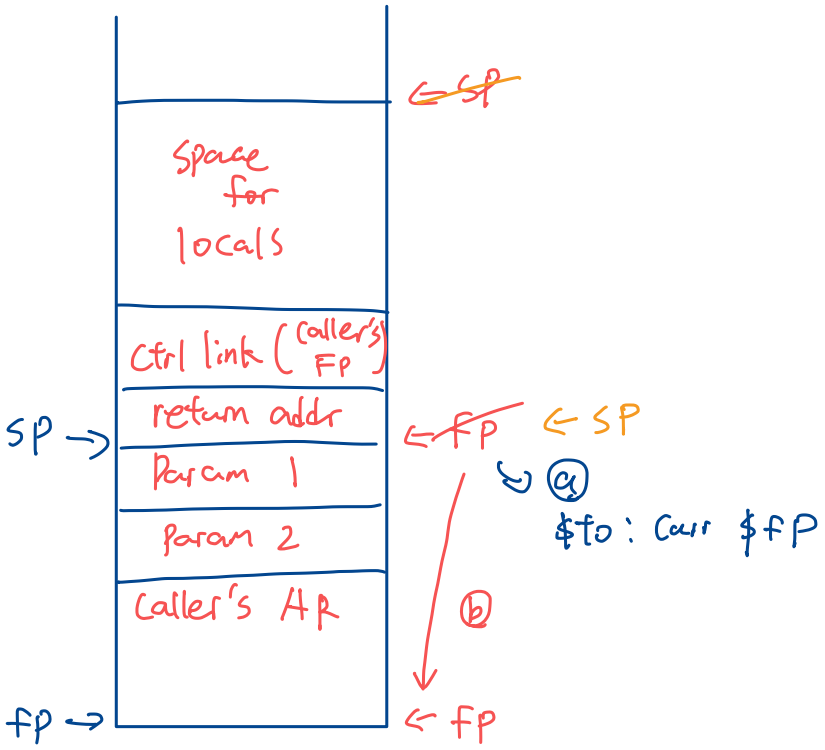
3. restore the stack pointer

```
move $sp, $t0
```

4. return control

```
jr $ra
```

① $ra : addr to return to



Body of function

Generate code for each statement in StmtListNode

- higher-level data constructs
  - loading parameters, setting return
  - evaluating expressions
- higher-level control constructs
  - performing a function call
  - while loops
  - if-then and if-then-else statements

Fn Body Node

DeclListNode — no code to generate

StmtListNode

Accessing local variables and parameters

```
lw $t0, offset($fp)
```

sw $t0, offset($fp)

↳ positive for params
negative for locals

Function Returns

Function returns when

- hit a return statement
- "fall off" end of function body

Approach

- label epilogue

```
_fctnName_exit:
# ... epilogue ... #
```

- have each return jump to label

```
# ... prologue ... #
    ...
# ... function body ... #
    ...
# code for evaluating return expression
    ...
lw $v0, 4($sp)
addu $sp, $sp, 4

j _fctnName_exit
```

if have a return value

& leaving result on stack

pop stack into $v0

jump to epilogue

About functions that return a value…

```
void main {
    integer x;
    x = f();
}
```

Consider 3 possibilities for function `f`

①
```
integer f() {

}
```
Code flow analysis required to verify every path through fctn w/non-void return type actually returns a value

②
```
integer f() {
    return;
}
```
non-void fctn & no return value in return statement

③
```
integer f() {
    return true;
}
```
wrong type of return value

type checker catches these

Code Generation for Expressions

Categories of expression nodes

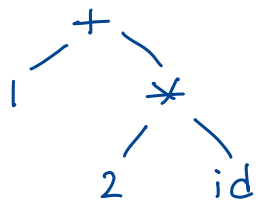
- literals integer literals, string literals, true, false
- IDs
- dot-access - not for P6
- call
- assignment
- non-short-circuited operators
- short-circuited operators (&&, ||)

Goal: evaluate expression leaving result on the stack

To do this, linearize ("flatten" expression tree)

- use a work stack and post-order traversal - like SDT during Parsing
- at operand: push value onto stack
- at operator: pop source values from stack, push result

Example: `1 + 2 * id`



eval + node
  eval L child
    Push 1 (on stack)
  eval R child (* node)
    eval L child
      Push 2
    eval R child ie value of id
      Push id
    Pop into $t1
    Pop into $t0
    multiply: $t0 * $t1 into $t1
    Push $t1
  Pop into $t1
  Pop into $t0
  add: $t0 + $t1 into $t0
  Push $t0

1 + 2 * id
~~2 * val-id~~
~~val of id~~
~~2~~
~~1~~

Code Generation for Literals

Integer (and boolean) literals

```
li $t0, value
# code to push $t0 on stack
```

String literals

- stored in static data area

```
    .data
label: .asciiz string_value
```

must include quotes

label: unique label

- to access, push *address* on to stack

load addr (la $t0, label), push $t0 onto stack

- two strings with same sequence of characters are considered equal (ie, using ==)

brevis code: if ("abc" = "abc") ...

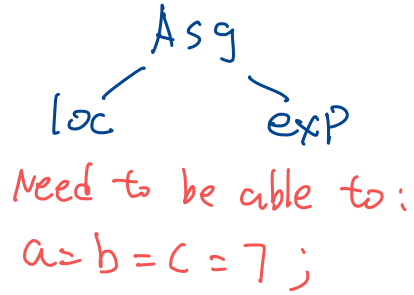
generate only one copy of .data

_L35: .asciiz "abc"

Code Generation for Assignments

Code generation for AssignExpNode

- compute address of LHS location; leave result on stack
- compute value of RHS expr; leave result on stack
- pop RHS into $t1
- pop LHS into $t0
- store value in $t1 at address held in $t0
- push $t1 on stack



Need to be able to:
a = b = c = 7;

Code generation for AssignStmtNode

AsgStmtNode
|
AsgExpNode

1) call CodeGen on AsgExpNode
2) Pop stack

Code Generation for Function Calls

Precall

- put argument *values* on the stack
  pass-by value semantics
- save *live* registers
  note: we don't have any in a stack machine
- jump to callee preamble label
  jal _fctn Name (jump and link)

call Exp Node
IdNode — ExpList Node

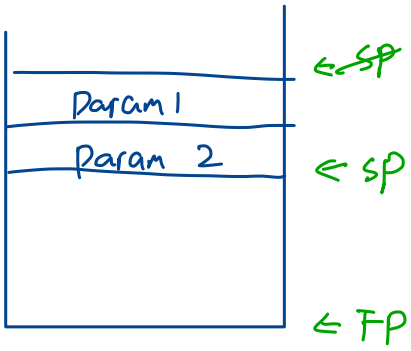
Postcall

- tear down the actual parameters ⇒ move sp
- retrieve and push result value ⇒ onto stack
  ↳ from $v0

Call Stmt Node
|
Call Exp Node

Handle like Asg Stmt Node

- Pop stack after CodeGen on Call Exp Node



x = f(y) + 4 * z ; Call Exp Node

g(a,b,c); Call Stmt Node

Code Generation for I/O

MIPS I/O is done using `syscall`

Algorithm

- load system call code into $v0
  - 1 to print integer
  - 4 to print string
  - 5 to read integer
- put argument into $a0
  eg, if printing, code Gen on exp & pop into $a0
- do syscall
  ie generate MIPS instruction: syscall

Code Generation for Dot-access

Offset from base of record to certain field is known statically

- compiler can do the math for the slot address
- not true for languages with pointers!

Example

```
record Inner (
    boolean hi;
    integer there;
    integer c;
);

record Demo (
    record Inner b;
    integer val;
);

void f(){
    record Demo inst;

    ... = inst.b.c;

    inst.b.c = ... ;
```

CS 536 Announcements for Wednesday, April 26, 2023

Last Time

- continue code generation
- function declaration, call, and return
- expressions
- literals
- assignment
- I/O

Today

- wrap up code generation
- dot-access
- introduce control flow graphs
- control-flow constructs

Next Time

- optimization

P6 : Codegen class

Constants for registers and boolean constants

e.g., `FP` , `SP` , `T0` , `T1`

Codegen.FP → "$fp"

Methods to help automatically generate code

`generate(opcode, ... args ... )`

e.g., `generate("add", "$t0", "$t0", "$t1")`

writes out `add $t0, $t0, $t1`

versions for fewer args as well

`generateIndexed(opcode,arg1, arg2, offset)`

e.g., `generateIndexed("lw", "$t0", $t1", -12)`

writes out `lw $t0, -12($t1)`

`genPush(reg) / genPop(reg)`

`nextLabel()` – returns a unique string to use as a label → of the form .L x (x: unique int)

`genLabel(L)` – places a label (L: string) generate the code .L37: if this is contents of L

Code Generation for Dot-access

Offset from base of record to certain field is known statically

- compiler can do the math for the slot address
- not true for languages with pointers!

Example

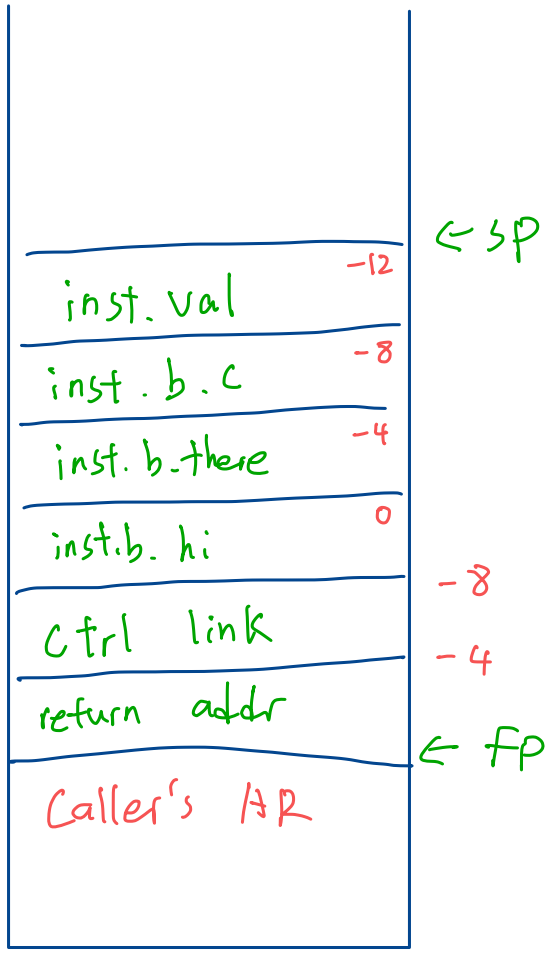
```
record Inner (
    boolean hi;
    integer there;
    integer c;
);

record Demo (
    record Inner b;
    integer val;
);

void f(){
    record Demo inst;

    ... = inst.b.c;

    inst.b.c = ... ;
```

12 bytes

16 bytes

space for inst
inst based at $fp - 8

Field b.c is -8 off the base of inst



RHS - Put value on stack

```
lw $t0, -16($fp)  # $t0 = value stored at $fp-16
Push $t0
```

LHS - Put address on stack

```
subu $t0, $fp, 16  # $t0 = $fp - 16
Push $t0
```

Control flow graphs

Kinds of control flow

- function calls – saw last lecture (jal, jr)
- selection – if, if-else, if-elseif, switch
- repetition – while, do-while, repeat-until, for
- short-circuited operators &&, ||

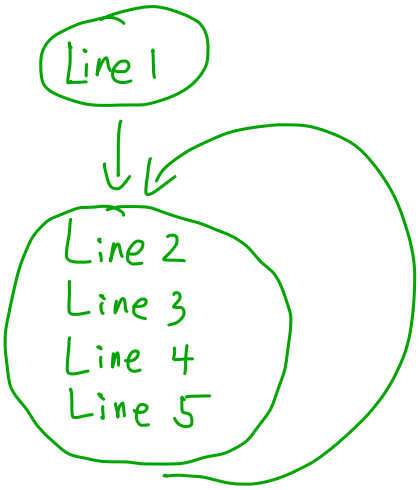
Control flow graph (CFG)

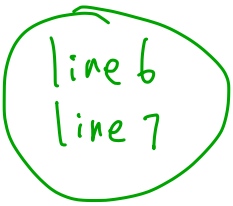
- important representation for program optimization
- helpful way to visualize source code

Example

```
Line1: li $t0, 4
Line2: li $t1, 3
Line3: add $t0, $t0, $t1
Line4: sw $t0, val
Line5: b Line2
Line6: sw $t0, 0($sp)
Line7: subu $sp, $sp, 4
```

CFG





line 2:
t0 = 4
t1 = 3
t0 = t0 + t1
val = t0
goto line 2
Push t0 on stack

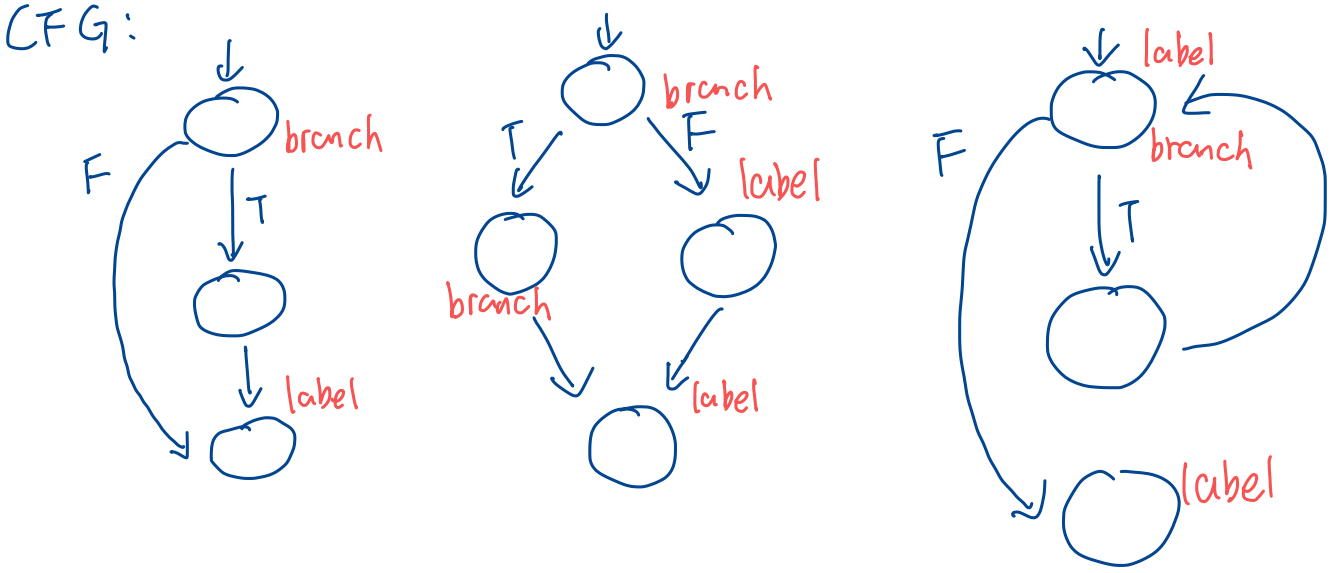
Kinds of control flow in brevis

```
if (exp) {
    ...
}
```

```
if (exp) {
    ...
} else {
    ...
}
```

```
while (exp) {
    ...
}
```

CFG:



What is needed at the assembly-code level

- branching
  - unconditional
  - conditional
- labels

MIPS

`b label` ← use branch in if/while control structure (rather than jump)

`beq r1, src, label`

↑ register or immediate value

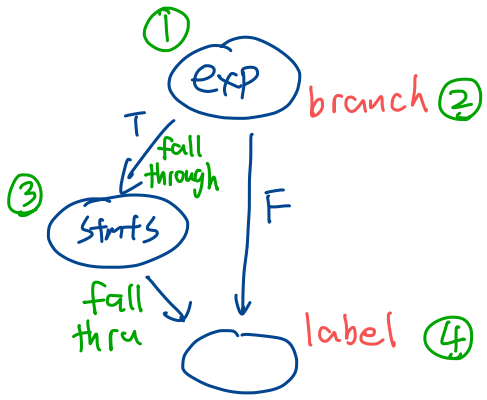
Also: `bne`, `blt`, `ble`, `bgt`, `bge`

Code generation for `if` statements

brevis code example:

```
if (a == b) {
    // body of if
}
```

Need to linearize – output a sequence of instructions



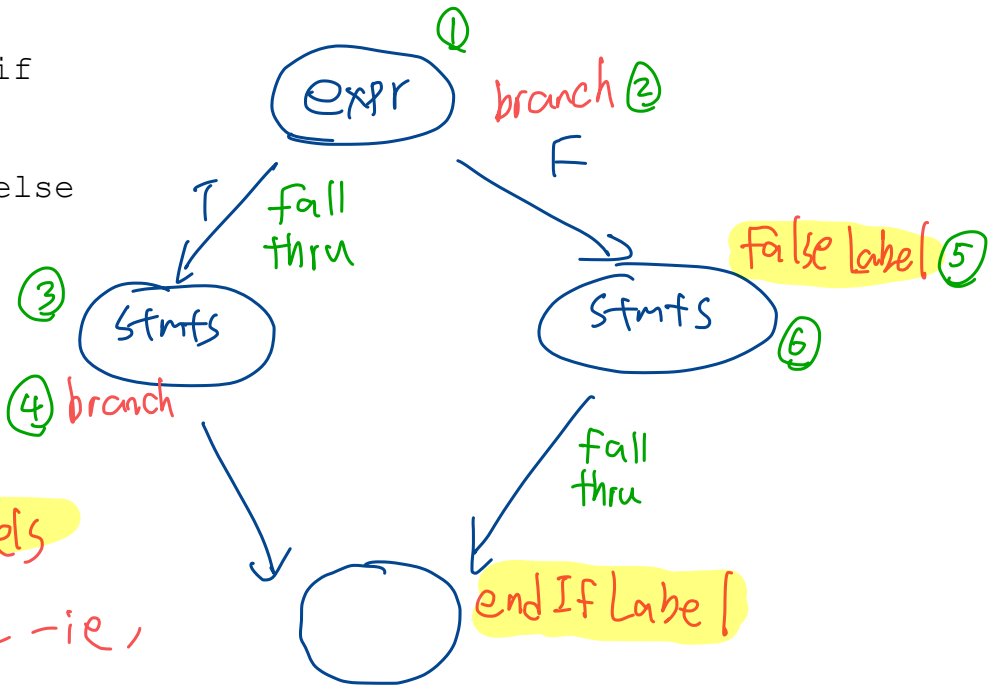
Code generation steps:

- get a label for end of construct
- (1) generate code for expression – leaving result on stack
- (2) generate conditional branch – to label (not yet placed!)
- (3) generate body of `if`
- (4) place end-of-construct label

Code generation for `if-else` statements

brevis code example:

```
if (a > b) {
    // body of if
}
else {
    // body of else
}
```



Need these labels to be unique - ie, generated by CodeGen.nextLabel()

Code generation for `if-else` statements (cont.)

brevis code:

```
if (a > b) {
    // body of if
}
else {
    // body of else
}
```

MIPS code outline:

```
(1) lw $t0, addr_a      )- CodeGen on Id(a)
    push $t0

    lw $t0, addr_b      )- CodeGen on Id(b)
    push $t0

    pop $t1
    pop $t0             )- CodeGen on >
    sgt $t0, $t0, $t1
    push $t0

(2) pop $t0
    beq $t0, FALSE, falseLabel
    .
(3) .  # body of if
    .
(4) b doneIfLabel

(5) falseLabel:
    .
(6) .  # body of else
    .

(7) doneIfLabel:
```

use Codegen.java has push/pop methods to generate MIPS code

sgt R2, R0, R1
set R2 to 1 if R0 > R1 to 0 otherwise

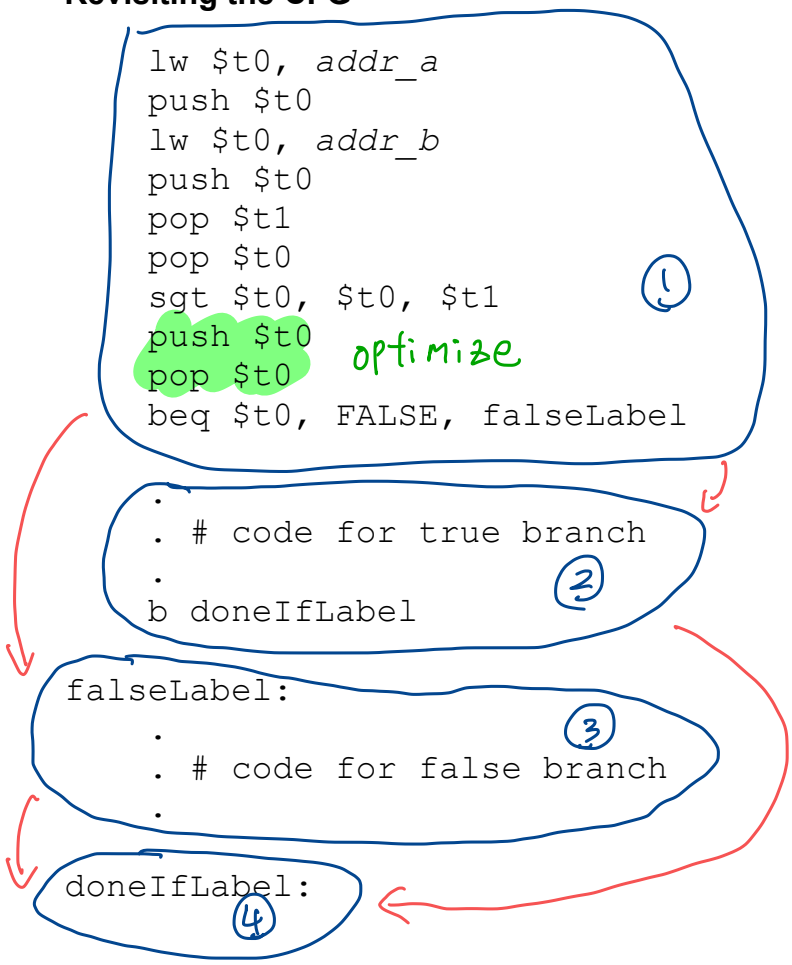
Also have: sge, slt, sle, seq, sne

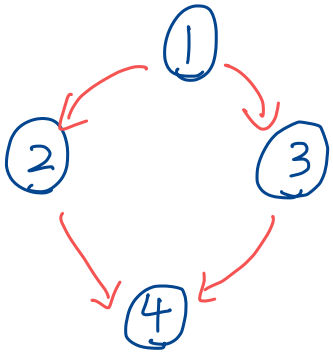
L constant in Codegen.java

note: only ended up using beq & b branching instr

Code generation for `if-else` statements (cont.)

Revisiting the CFG

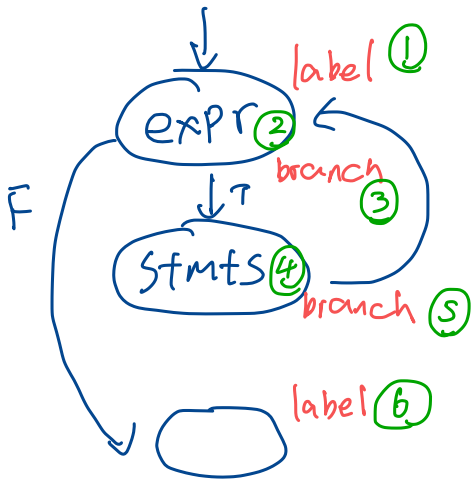




Code generation for `while` statements

brevis code example:

```
while (a == b) {
    // body of while
}
```



MIPS tips

It's really easy to get confused with assembly

Some suggestions

- start simple: main procedure that prints the value 1
  - get procedure `main` to compile and run
    - function prologue and epilogue
  - trivial case of expressions: evaluating the constant 1, which pushes a 1 on the stack
  - printing: `print <- 1;`
- then grow your compiler incrementally
  - expressions
  - control constructs
  - call/return

Create super simple test cases

- main procedure: print the value of some expression
- create more and more complicated expressions

Regression suite

- rerun **all** test cases to check whether you introduced a bug
- more suggestions
  - try writing desired assembly code by hand before having the compiler generate it
  - draw pictures of program flow
  - have your compiler put in detailed comments in the assembly code it emits

CS 536 Announcements for Monday, May 1, 2023

Last Time

- wrap up code generation
- dot-access
- introduce control flow graphs
- control-flow constructs and code generation

Today

- optimization overview
- peephole optimization
- loop optimizations
- copy propagation

Next Time

- wrap up optimization
- wrap up course / review

Recall example from last time

MIPS code outline:

```
    lw $t0, addr_a
    push $t0

    lw $t0, addr_b
    push $t0

    pop $t1
    pop $t0
    sgt $t0, $t0, $t1
    push $t0

    pop $t0
    beq $t0, FALSE, falseLabel
    .
    . # code for true branch
    .
    b doneIfLabel

falseLabel:
    .
    . # code for false branch
    .

doneIfLabel:
```

Optimization Overview

Goals

Informally: Produce "better" code that does the "same thing" as the original code.

What are we trying to accomplish?

- faster code
- fewer instructions
- lower power
- smaller footprint
- bug resilience?

Safety guarantee

Informally: Don't change the program's output (observable behavior)

- the same input produces the same output
- if the original program produces an error on a given input, so will the transformed code
- if the original program does not produce an error on a given input, neither will the transformed code

Does order need to be preserved?
- when output is generated
- different order of ops in floating-point arithmetic may produce different results.

Aside: evaluating polynomials: $Ax^7 + Bx^6 + Cx^5 + \ldots$ $O(n)$ adds $O(n^2)$ mults

can be evaluated as

$(((Ax + B)x + C) + D) + \ldots$ $O(n)$ adds $O(n)$ mults

However… There's no perfect way to check equivalence of two arbitrary programs

- if there was, we could use it to solve the halting problem
- we'll attempt to perform behavior-preserving transformations

Program Analysis

A perspective on optimization

- recognize some behavior in a program
- replace it with a "better" version

However, halting problem keeps arising:

- we can only use approximate algorithms to recognize behavior

Two properties of program-analysis/behavior detection algorithms

- **soundness**: all results that are output are valid
- **completeness**: all results that are valid are output

Analysis algorithms with these properties are mutually exclusive:

- if an algorithm was sound *and* complete, it would either:
  - solve the halting problem, or
  - detect a trivial property

Optimization Overview (cont.)

We want our optimizations to be *sound* transformations

- they are always valid
- but some opportunities for applying a transformation will be missed

Our techniques

- can detect many *practical* instances of the behavior
- won't cause any harm
- but we still want to consider efficiency

Peephole optimization

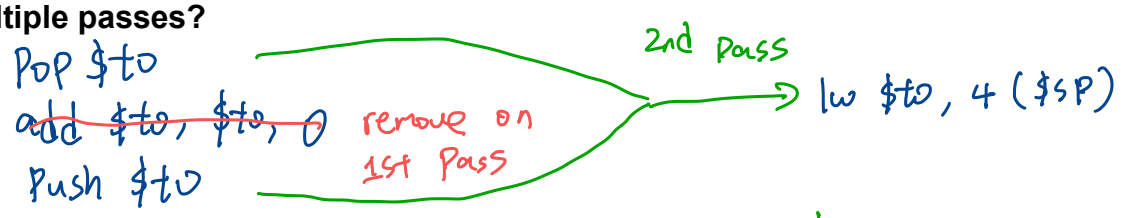
- naïve code generator errs on the side of correctness over efficiency
- use pattern-matching to find the most obvious places where code can be improved
- look at only a few instructions at a time

- done after code is generated

Peephole optimization

What can be optimized		Replaced with
push followed by pop 4 MIPS instrs	push $t0 pop $t0 (same reg)	nothing
Note: can't do optimization if have a label associated with pop	push $t0 pop $t1 (diff reg)	move $t1, $t0
pop followed by push	pop $t0 push $t0	load value from top of stack directly into $t0
branch to next instruction	b label label:	label:
* jump to a jump extra conditions are required	b L1 ... L1: b L2	bL2 ... L1: b L2
* jump around a jump	beq $t0, $t1, L1 b L2 L1:	bne $t0, $t1, L2 L1:
store followed by load same reg, same addr	sw $t0, addr lw $t0, addr	sw $t0, addr
load followed by store same reg, same addr	lw $t0, addr sw $t0, addr	lw $t0, addr
useless operations add 0	add $t0, $t0, 0 add $t0, $t1, 0	(nothing) move $t0, $t1
similarly for multiplying by 1		
multiplication by 2	shift-left (faster)	

Do multiple passes?



Fixed # of passes? Or until no more changes?

Loop-Invariant Code Motion (LICM)

Idea: Don't duplicate effort in a loop

Goal: Pull code out of the loop ("loop hoisting")

Important because of "hot spots"

- most execution time due to small regions of deeply-nested loops

Example

```
for (i=0; i<100; i++) {
    for (j=0; j<100; j++ {
        for (k=0; k<100; k++) {
            A[i][j][k] = i*j*k;
        }
    }
}
```

subexpression is invariant with respect to the inner most loop

becomes

```
for (i=0; i<100; i++) {
    for (j=0; j<100; j++ {
        temp = i*j;
        for (k=0; k<100; k++) {
            A[i][j][k] = temp*k;
        }
    }
}
```

Suppose `A` is on the stack.

To compute the address of `A[i][j][k]`:

```
FP - offset_of_A[0][0][0]
+ (i*10000*4)
+ (j*100*4)
+ (k*4)
```

tmp 0 = FP - offset A
for (i=0; ...
  tmp1 = tmp0 + i * 40000;
  for (j=0; ...
    tmp2 = tmp1 + j * 400;
    tmp = i * j;
    for (k=0; ...
      T0 = tem * k;
      T1 = tmp2 + k * 4;
      store T0, 0(T1)

Loop-Invariant Code Motion (cont.)

When should we do LICM?

- at IR level, more candidate operations
- assemby might be *too* low-level
  - need guarantee that the loop is *natural* – no jumps into middle of the loop

How should we do LICM? Factors to consider

- safety – is the transformation semantics-preserving?

  make sure – operation is truly loop invariant
  – ordering of events is preserved

- profitability – is there any advantage to moving the instruction?

  may end up – moving instrs that are never executed
  – performing more intermediate computation than necessary.

Other Loop Optimizations

Strength reduction in for-loops

- replace multiplications with additions

Loop unrolling

- for a loop with a small, constant number of iterations, may actually take less time to execute by just placing every copy of the loop body in sequence – no jumps
- may also consider doing multiple iterations within the body

Loop fusion

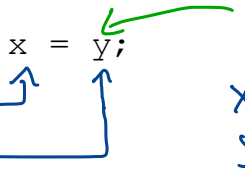
- merge 2 sequential, independent loops into a single loop body – fewer jumps

Copy propagation

copy statement

definition of x

use of y



can be a variable or literal (on RHS of =)

x is an L-value

y is an R-value

Idea: Suppose we are at *use* U of *x* and a *definition* D of *x* (of the form *x* = *y*) reaches U

- If
  1) no other definition of *x* reaches U **and**
  2) *y* does not change between D and U
- then we can replace the use of *x* at U with *y*

Example

```
x = 3;
y = 5;
p = x;
if (w*x > 9) {
    x = 4;
    z = x + w*y;
}
else {
    z = 2*y + x;
}
q = 5*p;
s = z + x;
t = s + y;
```

y = 5; ← useless definition (can be removed)

p = x; 3

if (w*x > 9) { 3

x = 4;

z = x + w*y; 4 5

} else { 5 3

z = 2*y + x; → 13 constant folding

} q = 5*p; → change to 3 on 2nd pass

s = z + x;

t = s + y; 5

Copy propagation (cont.)

How is this an optimization?

- can create useless code (which can then be removed)

  if all uses of x reached by D are replaced,
  then definition D can be removed. (eg, y=5)

- can create improved code

  t = 5+y; RHS requires (at a minimum) 2 loads & one add

  t = 5+5; RHS requires only 1 load & one add
  (can use immediate value in add instr)

- constant folding

  z = 2*5 +3; → z=10+3 → z=13
  now can propagate this ↑ def of z

- if done before other optimizations, can improve results

  x=2;
  if (x<7) → x=2;
  //stmts if (2<7) → x=2;
  //stmts //stmts

CS 536 Announcements for Wednesday, May 3, 2023

Course evaluation – log into aefis.wisc.edu using your NetID

Last Time

- optimization overview
- peephole optimization
- loop optimizations
- copy propagation

Today

- wrap up optimization
  - copy propagation
- wrap up course / review

Optimization Review

Goal: Produce "better" code that does the "same thing" as the original code.

- better = faster code, fewer instructions

When?

- before code generation (i.e., on intermediate representation)
- after code generation (i.e., on generated machine code)

Important considerations

- **performance/profitability** – want to be sure optimization is "worth it"
- **safety** – orginal source code, non-optimized target code, and optimized target code all do the "same thing" / have the same "meaning"

Look at optimizations that

- are *sound* transformations sound = all results that are output are valid
- recognize a behavior in a program & replace it with a "better" version

Copy propagation

Idea: Suppose we are at *use* U of *x* and a *definition* D of *x* (of the form x = y) reaches U

- If
  1) no other definition of *x* reaches U **and**
  2) *y* does not change between D and U
- then we can replace the use of *x* at U with *y*

Optimization opportunities

- can create useless code (which can then be removed)
- can create improved code
- constant folding
- if done before other optimizations, can improve results

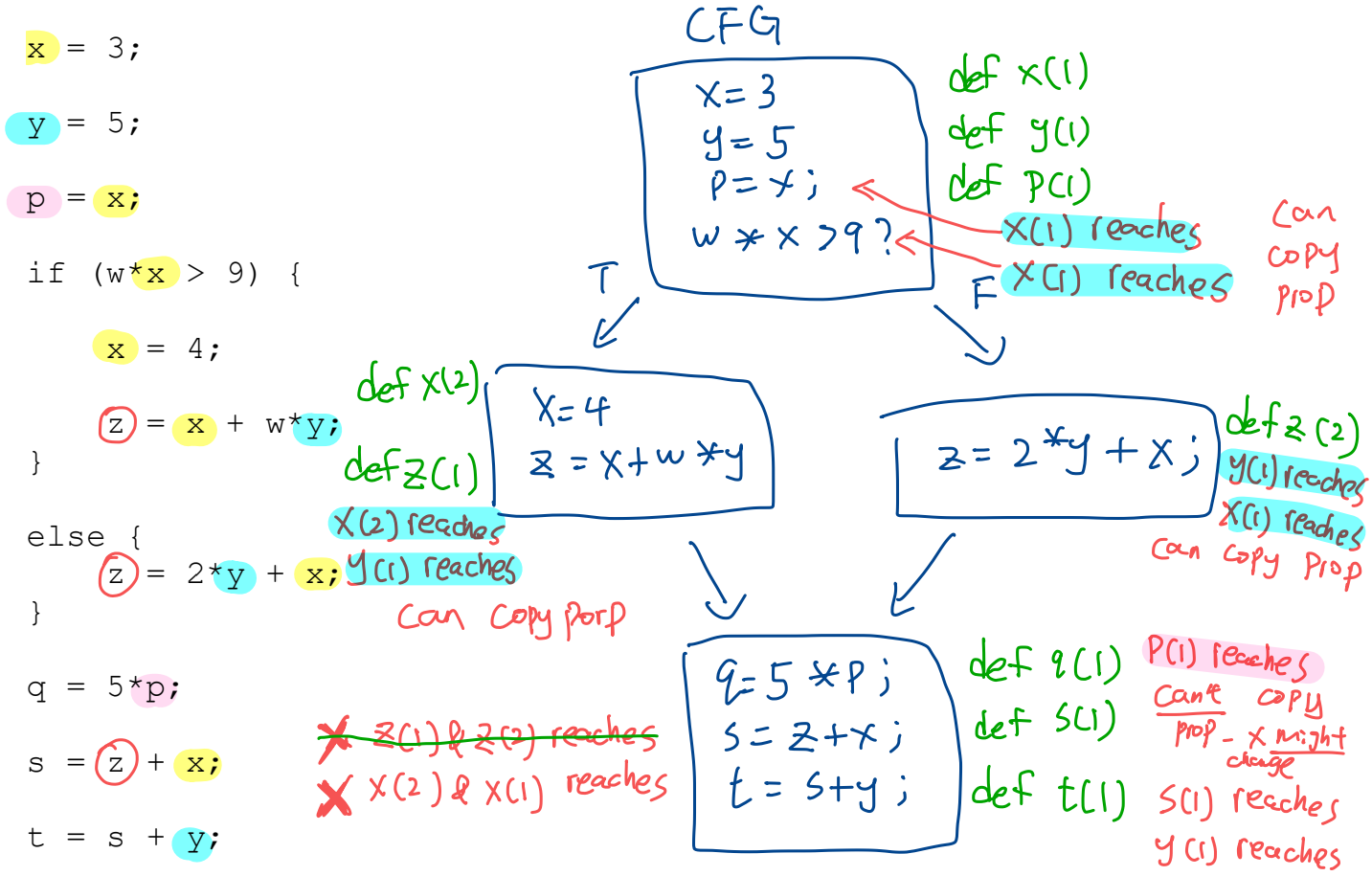
To do copy propagation, we must make sure two properties hold:

Property 1) No other definition of *x* reaches U

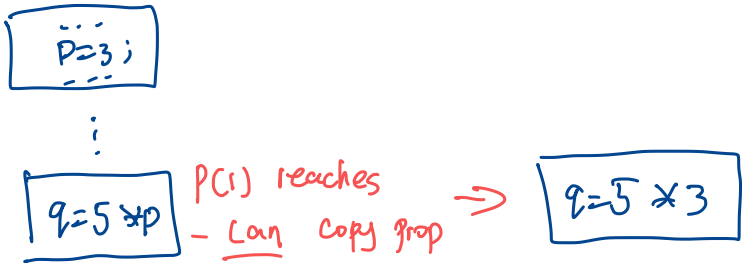
- How? Do a *reaching-definitions* analysis
  - one way: data flow analysis
  - another way: create control flow graph (CFG)
    - do "backward" search starting at U
    - stop exploring a branch of a search when we find a def of x (but continue overall search)

Example

```
x = 3;
y = 5;
p = x;
if (w*x > 9) {
    x = 4;
    z = x + w*y;
}
else {
    z = 2*y + x;
}
q = 5*p;
s = z + x;
t = s + y;
```



After one pass:



Copy Propagation (cont.)

Property 2) *y* does not change between D and U

- If *y* is a constant, then this is trivially true.
- If on any path through the CFG from D to U there is def of y, then y might change.
- If y & z aliases (refer to same memory location) & there is a def of z between D & U, then y might change

```
x = y;
// code to make y & z aliases
z = 5;
w = x + 4;
```

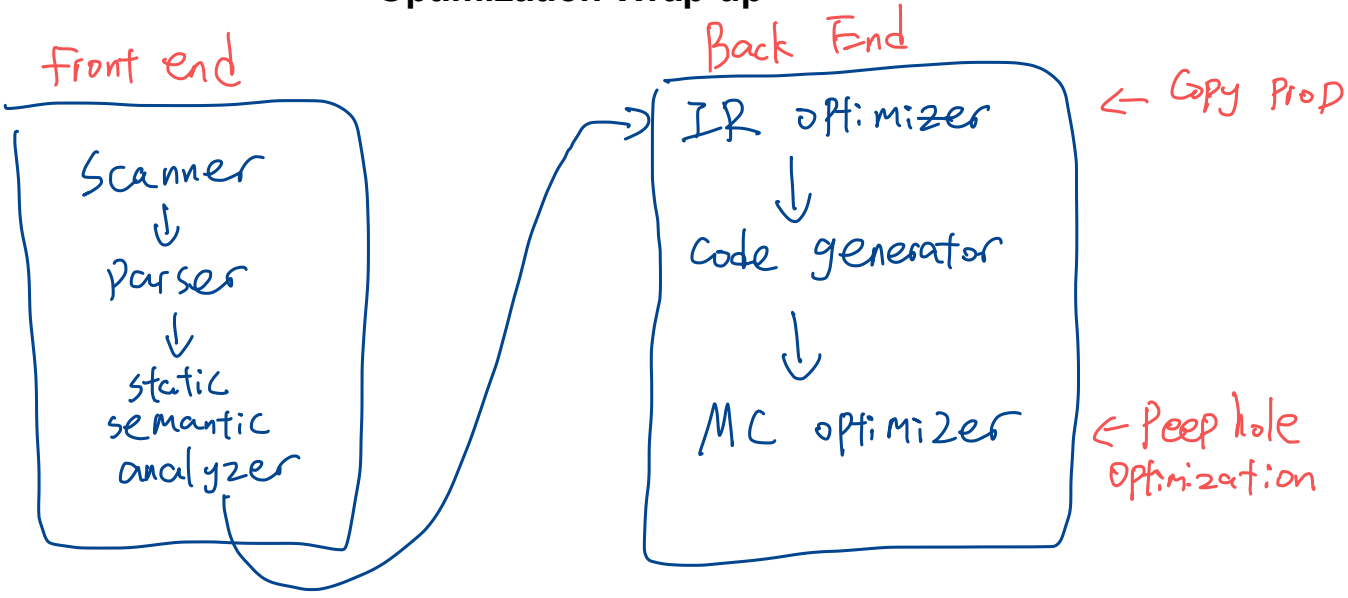
↑ can't change x to y

In C/C++

```
x = y;
int *z = &y;
*z = 5;
w = x + 7;
```

Same place in memory

Optimization Wrap-up



Where have we been?

CS 536: Introduction to Programming Languages and Compilers

What does a programming language consist of?

- tokens
- grammar
- static semantic analysis

What else? What choices are made?

- scoping rules
  How do we match variable defs to variable uses?
  What is allowed? nested functions, nested var decls?

- types
  What types are there?
  How do the types relate to each other?

- parameter passing
  What ways are there to get info to a called procedure?
  What impacts are there on the calling procedure?

- when do we check for things?
  at compile time → statically
  at runtime time → dynamic
  or both?

Where have we been?

CS 536: Introduction to Programming Languages and Compilers

How do we translate a PL into something a computer can run? i.e., compilers

- recognizing tokens
  reg exprs & FSMs
  tools for translating
- recognizing languages - context-free grammars, parsing
  what can be parsed
  how? top-down vs bottom-up

- enforcing scoping and typing rules
- developing data structures that assist our translation/representation/translation
  AST, Parse tree, symbol tables
- how do we organize and manage memory
  variables - where stored, how accessed
  - local vs. global vs non-local
  using register vs stack

- handling control flow within a program
  - interprocedural - how fctn calls & returns are implemented
  - intraprocedural - how are loops & selection stmts implemented.

How can we make our translation better?

- intermediate representations
- IR optimizations
  copy propagation, LICM & other loop optimization
- MC optimizations
  peephole optimizations

Course wrap-up

Covered a broad range of topics

- some formal concepts
- some practical concepts

What we skipped

- object-oriented language features
- dynamically-allocated memory management
- linking and loading
- interpreters
- register allocation
- dataflow analysis
- performance analysis
- proofs

Final Exam, Thursday, May 11, 2:45 pm
19 Ingraham Hall

Bring your UW Student ID

Reference material provided along with exam:

- copy of the brevis grammar
- compiler class reference with selected class, methods, fields

Topic overview

Basic ideas of scanning & parsing

Symbol-table management / name analysis

- static scoping
- dynamic scoping

Type checking

Runtime storage management

- general storage layout
- activation records
- access to variables at runtime (parameters, locals, globals, non-locals)

Parameter-passing modes

Code generation

Optimization

- goals
- optimization techniques (e.g., peephole optimization, copy propagation)

Extending

- grammar
- AST
- name analysis,
- type checking
- code generation

to handle new constructs