EECS 483: Compiler Construction

Lecture 20:
Intro to Frontend, Lexing 1

March 31
Winter Semester 2025

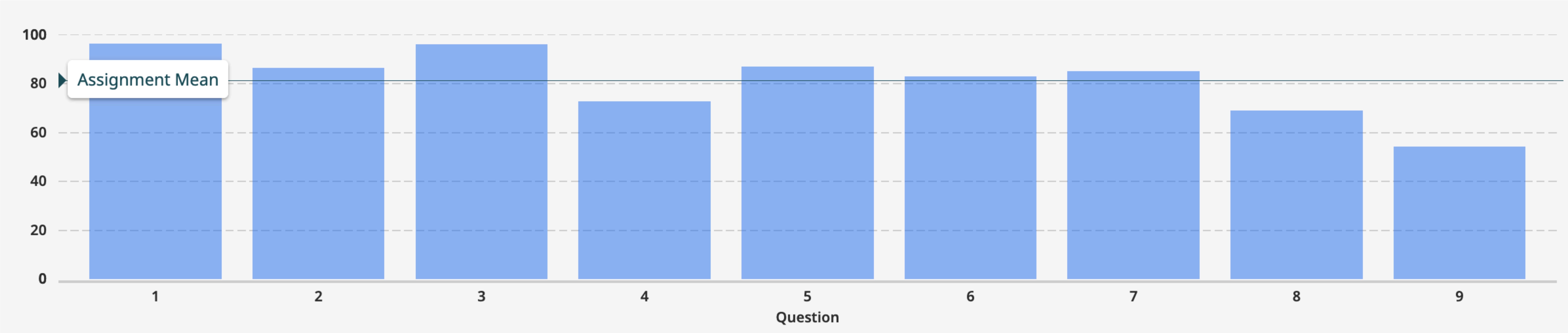
Midterm

- Raw Grades released on Gradescope, curved grades on Canvas.
  - Median 75/90 ~ 83%
  - Mean 73/90 ~ 81%
  - Std dev. ~ 12

Curved to a Mean of 85%

Submit any regrade requests this week.

Midterm by Q



Lowest averages:

- Unfamiliar calling convention
- Minimal SSA form
- Translating Imperative to Functional Code

Assignments

Due this Friday, get on it!

Assignment 5: optimization released in 1 week

Compiler Frontend

Compiler Frontends

The task of the compiler frontend is take the input program as a string and

1. Validate that it is a well-formed program

2. Output an **Abstract Syntax Tree** that is more convenient for the rest of the compiler pipeline to use

Compiler Frontends

So far in class we have only implemented a small part of the frontend: the "semantic analysis" phase. For Snake programs this meant checking variables and functions are used properly.

Remainder of the semester: first two components of the frontend **lexing/lexical analysis** and **parsing/syntactic analysis**

Compiler Frontends

The task of the lexing and parsing phases is to **find** structure (abstract syntax trees) in an unstructured representation (strings of characters).

Works differently from passes we've seen so far, which all had tree-structured programs as inputs.

Lexical analysis, tokens, regular expressions, automata

LEXING

First Step: Lexical Analysis

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

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

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

- Token: data type that represents indivisible “chunks” of text:
  - Identifiers: `a y11 elsex _100`
  - Keywords: `if else while`
  - Integers: `2 200 -500 5L`
  - Floating point: `2.0 .02 1e5`
  - Symbols: `+ * ` { } ( ) ++ << >> >>>`
  - Strings: `“x” “He said, \”Are you?\””`
  - Comments: `// 483: Project 1 … /* foo */`

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

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

DEMO: LEXING BY HAND

Lexing By Hand

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

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

PRINCIPLED SOLUTION TO LEXING

Making Lexing Less Painful

- Lexers are
  - tedious to write
  - easy to mess up, hard to read
  - repetitive: most lexers are essentially the same algorithm but different specifics
- Solution: make a new, high-level **domain-specific language** for writing lexers
  - Easier for humans to read, write, update
  - Efficient implementation strategy implemented once and for all
    - limited computational power -> Rice's theorem no longer applies, can get "perfect" optimization
- Examples:
  - lex/flex
  - antlr
  - ocamllex
  - In Rust: logos, lalrpop

A Lexer Compiler

- Now we have reduced lexing to a mini-compiler task. So let's do what we've been doing all semester!
  - Design a **language** for lexers
  - Describe its **semantics**
  - Transform that language into **intermediate representations**
  - **Optimize** the intermediate representation
  - **Generate code** that implements our optimized IR.

A Language for Lexers

- What language should we use to describe a lexer?
- What does a lexer need to do?
- A lexer needs to specify
  - What strings make up the "tokens" of our language
  - How to turn these abstract tokens into data that our compiler pipeline can use
- Need to make a language for describing sets of strings

Formal Languages

- First we fix the "alphabet" of characters Σ.
  - Common alphabets { 0 , 1 } for bitstrings
  - 0-255 for ASCII characters
  - very large set of Unicode "characters"
- A string (over Σ) is a finite sequence of characters (i.e., elements of Σ)
- A **formal language** is a **subset** of strings.

- Examples that we use in lexing:
  - Booleans { "true" , "false" }
  - The keywords of a language { "def", "let", "in", "extern", ... }
  - The set of all number literals { 0, -1, +1, 199239190, ... }
  - The set of all valid variable names { "x", "y", "z",... but not "def", "extern" etc }

Formal Languages

- First we fix the "alphabet" of characters Σ.
  - Common alphabets { 0 , 1 } for bitstrings
  - 0-255 for ASCII characters
  - very large set of Unicode "characters"
- A string (over Σ) is a finite sequence of characters (i.e., elements of Σ)
- A **formal language** is a **subset** of strings.

- Examples that we use in lexing:
  - Singletons for particular keywords { "def" } {"let"} {"extern"} or syntactic tokens { ")" } { "(" } { ":" }
  - Booleans { "true" , "false" }
  - The set of all number literals { 0, -1, +1, 199239190, ... }
  - The set of all valid variable names { "x", "y", "z",... but not "def", "extern" etc }

- A lexer generator then needs a **syntax** for describing such formal languages
  - A language of expressions
  - Which are given a **semantics** as formal languages

Regular Expressions

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

Example Regular Expressions

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

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

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

How to Match?

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

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

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

Lexer Generators

- Reads a list of regular expressions: $\mathtt{R_1,\ldots,R_n}$, one per token.
- Each token has an attached "action" $\mathtt{A_i}$ (just a piece of code to run when the regular expression is matched):

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

token
regular expressions

actions

- Generates scanning code that:
  1. Decides whether the input is of the form $\mathtt{(R_1|\ldots|R_n)^*}$
  2. Whenever the scanner matches a (longest) token, it runs the associated action
  3. Most typically: adds a token to the output stream

lexlex.mll

DEMO: OCAMLLEX