EECS 483: Compiler Construction

Lecture 20:
Intro to Frontend, Lexing 1

March 30
Winter Semester 2026

Assignments

Assignment 4: due Friday

Assignment 5: optimization (liveness analysis, register allocation, possible values analysis, assertion removal) released in 1 week

Midterm grading in progress, will be done by the end of the week.

Learning Objectives

What are the common components of the compiler frontend?

What is lexical analysis?

What is hard about implementing lexical analysis?

How do we give a formal specification for a lexical analysis?

What are **regular expressions**?

What is the formal semantics of a regular expression?

How and why we use **regular expressions** for specification and implementation of lexical analysis.

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.

LEXING

First Step: Lexical Analysis

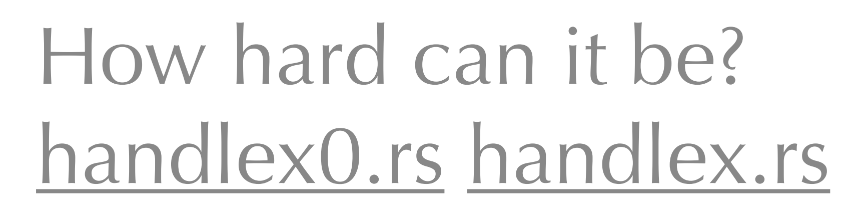
- Change the *character stream* “`if (b == 0) a = 0;`” into *tokens*:

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?
handlex0.rs handlex.rs

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

Lexer Generators

- 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
  - 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 two things
  1. What strings make up the "tokens" of our language
  2. How to turn these abstract tokens into data that our compiler pipeline can use
- 1 is the hard part: we need 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.

Formal Languages

- 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 — { "" } contains only the empty string
  - ∅ — { } Empty set
  - `'a'` — { "a" } An ordinary character stands for itself
  - $R_1 \mid R_2$ — Alternatives, stands for **union** of R_1 or R_2
    $Sem(R_1) \cup Sem(R_2)$
  - R_1R_2 — Concatenation, stands for R_1 followed by R_2
    $\{ w1\ w2 \mid w1 \in Sem(R_1), w2 \in Sem(R_2) \}$
  - $R*$ — Kleene star, stands for *zero or more* repetitions of R
    $\{ w1\ w2 \dots \mid w1 \in R, w2 \in R, w3 \in R, \dots \}$
    (includes the base case that there are no substrings)

Regular Expressions

- *Useful extensions:*
  - `“foo”` Strings, equivalent to `'f''o''o'`
  - `R+` One or more repetitions of `R`, equivalent to `RR*`
  - `R?` Zero or one occurrences of `R`, equivalent to `(ε|R)`
  - `['a'-'z']` One of `a` or `b` or `c` or … `z`, equivalent to `(a|b|…|z)`
  - `[^'0'-'9']` Any character except `0` through `9`

Example Regular Expressions

- Recognize the keyword "if":

  `"if"`
- Recognize a digit:
  - '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
  - using abbreviations: `['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
```

Example Regular Expressions

- Exercise: Write a regular expression for string literals that allow escaping.
- Alphabet: just { a , b , " , \ }
- Examples of valid inputs:
  - "abababab"
  - "\""
  - "\\"
  - "\a"
- Examples of invalid inputs:
  - "aba
  - abaa"
  - "\"
- Solution:
  - " (a | b | \(a | b | \ | "))* "

Overlapping Regexes

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

- The regular expressions alone are overlapping, need to disambiguate somehow.
- 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

DEMO: REGEX-BASED LEXING