Lecture 5-6 — Lexing

- **Compilers always start by doing two things with your program: *lexing* and *parsing*. These are really useful things to know how to do, not just for writing compilers, but for writing any kind of program where the input is highly structured. In many cases, parsing is not necessary, but lexing almost always is.**

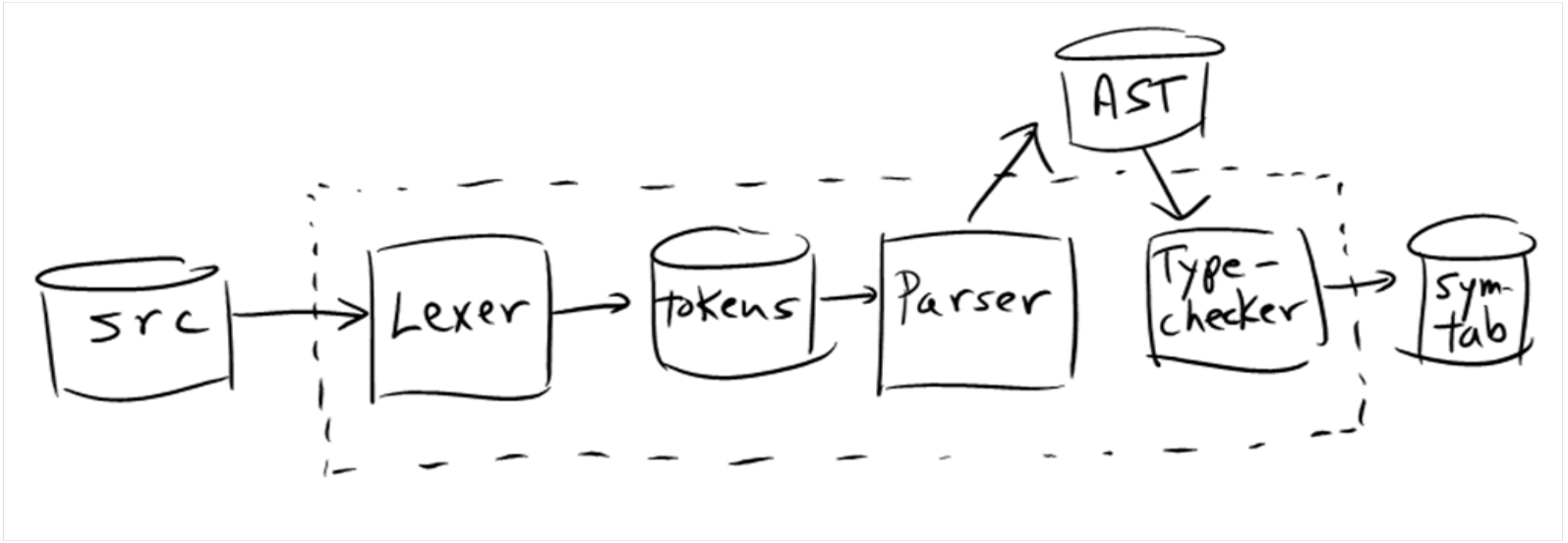
  There are two ways to implement a lexer: (1) write a program (i.e. do it manually), or (2) prepare a lexer specification and submit it to a lexer generator. We will do both.

  Here are the topics for these two lectures:

  - **Overview of compilers**
  - **Lexing by implementing a DFA**
  - **Lexing with a generator (`ocamllex`)**

Structure of a compiler

Detail of compiler front-end:



Lexing and tokens

- **Job of *lexer* (aka *lexical analyzer*, *scanner*, *tokenizer*) is to divide input into *tokens*, and discard whitespace and comments.**
- ***Token* is smallest unit useful for parsing.**
- **Think of the lexer as a function from a list of characters (the source file) to a list of tokens.**

Types of tokens

- **Tokens for programming languages are of three types:**
  - ***Special characters and character sequences:*** `=`, `==`, `>=`, `{`, `}`, `;`, **etc.**
  - ***Keywords:*** `class`, `if`, `int`, **etc.**
  - ***Token categories:*** **For parsing purposes, all integers can be treated the same, and similarly for floats, doubles, characters, strings, and identifiers.**
- **Comments and whitespace are also handled by the lexer, but these are not tokens because they are ignored by the parser; the lexer deletes them.**
- **In Java, three identifiers are *reserved* (like keywords) but are treated as identifiers by the parser:** `true`, `false`, `null`.

Tokens of Java

- **Here is a (partial) declaration of the tokens of Java:**

```
type token = EOF | EQ | PLUS | SEMICOLON
  | CLASS | INT | LPAREN | RPAREN | ...
  | INTLIT of int | FLOATLIT of float
  | IDENT of string | STRINGLIT of string
```

- **E.g. Lexer would transform "`class C { int process ( ...`" to**

```
[CLASS; IDENT "C"; LBRACE; INT; IDENT "process"; LPAREN; ... ]
```

- **Now you try it: "`x1 = (3 + y);`" becomes:**

```
[                ;        ;          ;              ;

                                ;          ;          ;              ]
```

Lexing vs. parsing

- Lexing is a simple, efficient pre-processing step
  - Tokens can be recognized by finite automata, which are easily implemented.
  - Significant reduction in size from source file to token list
- Parsing is more complicated
  - Program cannot be parsed using DFAs. (Why?)
- Many input problems can be solved by lexing alone — which is why regular expressions are so widely used. (From 373: finite automata $\equiv$ regular expressions.)

Two methods of writing a lexer

- **Manual:**
  - **Design a DFA for the tokens, and implement it**
- **Automatic:**
  - **Write regular expressions for the tokens. Run a program ("lexer-generator") that converts regular expressions to a DFA, and then simulates the DFA.**
- **Automatic methods usually easier, but sometimes not available**

Building a lexer by hand

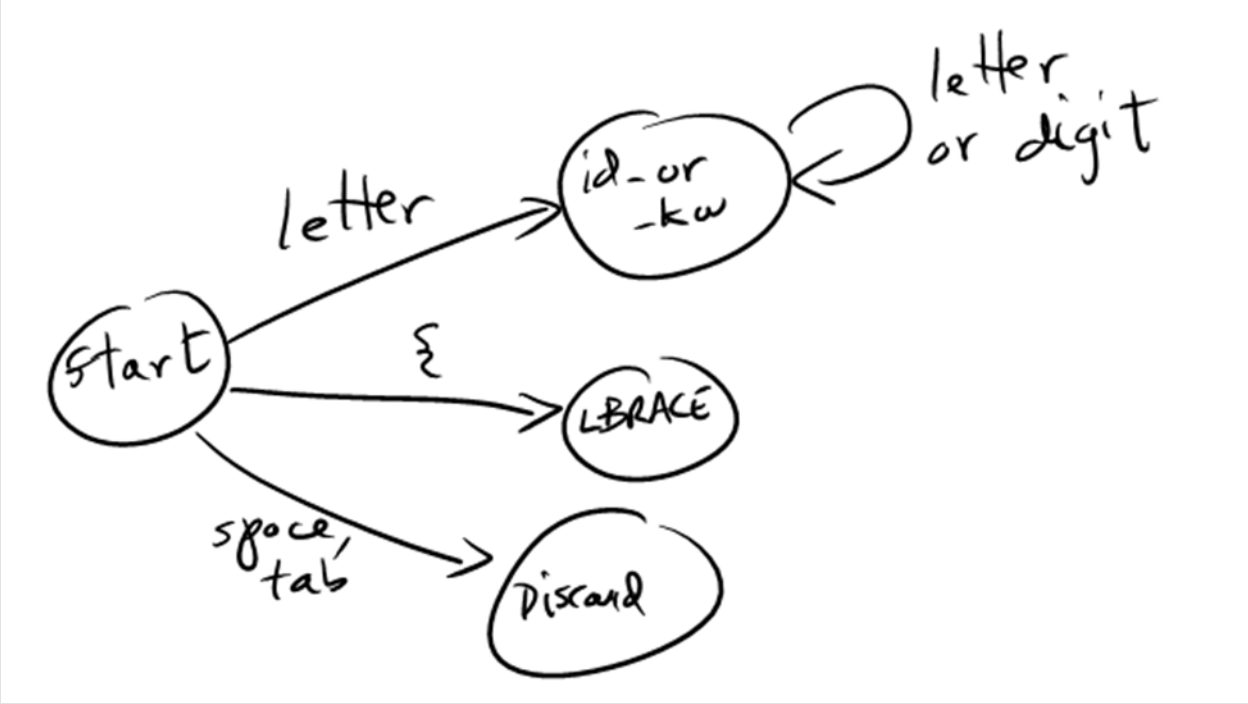
- **DFA (non-standard definition): A directed graph with vertices labelled from the set $Tokens \cup \{Error, Discard, Id\_or\_KW\}$, and edges labelled with sets of characters. The outgoing edges from any one vertex are labelled with disjoint sets. One vertex is specified as the *start state*.**
- **Note differences from standard definition:**
  - ***DFA is not "complete"*: You can get to a state where there are input characters left, but no edge that contains the next character. *This is deliberate.***
  - ***No "accept" states*: Or rather, states labelled with $Error$ are reject states and every other state is an accept state.**

Building a lexer by hand

- Given a DFA as above:
  - Open source file, then:
    1. From start state, run the DFA as long as possible.
    2. When no more transitions are possible, do whatever the label on the ending state says:
       - *Token*: add to token list
       - *Error*: print error message
       - *Discard*: do nothing at all
       - *Id_or_KW*: check for keyword; add keyword or ident to token list.
    3. Go to (1).

Lexing DFA

- **The lexing DFA looks something like this:**



- **To build this, start by building separate DFAs for each token...**

Ex: DFA for operators

- `;`
- `}`
- `+`, `++`, `+=`
- `<`, `<=`, `<<`, `<<=`, `<<<`

Ex: DFA for integer constants

Ex: DFA for integer or floating-point constants

Putting DFAs together

- **Combine individual DFAs by putting together their start states (then making adjustments as necessary to ensure determinism).**
- **Following slides show how to implement the DFA.**
  - **Number the nodes with positive numbers (0 for the start state)**
  - **Define `transition` : int * char $\rightarrow$ int. `transition`(s, c) gives the transition function of the DFA; it returns -1 if there is no transition from state s on character c.**

Lexing functions (in pseudocode)

```
(state * string) get_next_token() {
    s = 0;  thistoken = "";
    while (true) {
        c = peek at next char
        if (transition(s,c) == -1)
             return (s, thistoken)
        move c from input to thistoken
        s = transition(s,c)
    }

token list get_all_tokens() {
    tokenlis = []
    while (next char not eof) {
        (s, thistoken) = get_next_token()
        tokenlis = lexing_action(s, thistoken, tokenlis)
    }
    return tokenlis;
}
```

Lexing actions

```
lexing_action (state, thistoken, tokenlis) {
    switch (label(state)) {
    case ERROR:


    case DISCARD:


    case ID_OR_KW:



    default:


    }
}
```

DFAs for comments

- **C++ comments**
- **C comments**
- **OCaml comments**

ocamllex

- A *lexer generator* accepts a list of regular expressions, each with an associated token; creates a DFA like the one we built above; and outputs a program that implements the DFA.
- `ocamllex` is a parser generator whose output is an OCaml program.
- In the following slides, we describe the input specifications read by `ocamllex`, and how to solve various lexing problems in `ocamllex`.
- For full details, consult `ocamllex` manual.

Regular expressions in `ocamllex`

$'c'$	character c
$"c_1c_2\ldots c_n"$	string ($n \geq 0$)
$r_1\ r_2\ \ldots$	sequence of regular expressions
$r_1\ \vert r_2\ \vert\ldots$	alternation of regular expressions
$r*$	Kleene star
$r+$	same as $rr*$
$r?$	same as $r\vert""$
_	any character
`eof`	special eof character
`['`c_1`'-'`c_2`']`	range of characters
`[^'`c_1`'-'`c_2`']`	complement of a range of characters
r `as` id	same as r, but id is bound to the characters matched by r

Regular expression examples

- **Identifiers:**
- **Integer literals**
- **The keyword `if`**
- **The left-shift operator**
- **Floating-point literals**

Ocamllex specifications

```
{ header }                    (auxiliary ocaml defs)
let ident = regexp            (regexp abbreviations)
...
rule main = parse
       regexp { action }      (action is value returned
     | ...                    when regexp is matched)
     | regexp { action }
{ trailer }                   (auxiliary ocaml defs)
```

- *header* **can define names used in actions**
- **In addition to ordinary regular expression operators,** *regexps* **can include the names given as abbreviations; these are just replaced by their definitions.**
- *actions* **are ocaml expressions; all must be of the same type**

Ocamllex specifications (cont.)

- `ocamllex` **transforms this specification into a file that defines the function** `main` **: Lexing.lexbuf** $\rightarrow$ ***type-of-actions***. **The name of** `main`**'s argument is** `lexbuf`. `lexbuf` **is not a list, but a "stream" that delivers characters one at a time; the** `Lexing` **module contains a number of functions on** `lexbuf`**s, but you won't need to use them.**

- `trailer` **can define functions that call** `main`.

- ***actions*** **can call** `main`**, and can refer to variable** `lexbuf` **(the input stream).**

Ocamllex specifications (cont.)

- ocamllex puts all the regular expressions together using alternation, then transforms it to an NFA and then a DFA (as in 373). The DFA looks like the one we designed by hand: Each state is associated with one of the regular expressions (or it's an error state). It runs the DFA as long as possible (just as we did), and then takes the action associated with the state it ended in.
- If two regexps match:
  - If one matches a *longer* part of the input, that one wins.
  - If they both match the exact same characters, the one that occurs first in the specification wins. (E.g. keywords should precede regexp for identifiers.)

Lexing functions

- `ocamllex` **will define function** `main`**, but this is not a convenient functions to use. Instead, in** *trailer* **define more useful functions that call** `main`**, e.g.**

```
(* Remember: main: Lexing.lexbuf -> type-of-actions *)

(* Get first token in string s *)
let get_token s = let b = Lexing.from_string (s)
                  in main b

(* Get list of all tokens in buffer b *)
let rec get_tokens b =
     match main b with
            EOF -> []
          | t -> t :: get_tokens b

(* Get list of all tokens in string s *)
let get_all_tokens s =
   get_tokens (Lexing.from_string s)
```

Ocamllex example 1

- **In this and following examples, the trailer is omitted; it will contain definitions like the ones on the previous slide.**
- **What does this do?**

```
rule main = parse
    ['0'-'9']+                    { "Int" }
  | ['0'-'9']+'.'['0'-'9']+       { "Float" }
  | ['a'-'z']+                    { "String" }
```

Ocamllex examples 2 and 3

```
let digit = ['0'-'9']
rule main = parse
   digit+                { "Int" }
 | digit+'.'digit+       { "Float" }
 | ['a'-'z']+            { "String" }
```

```
{   type token = Int | Float | Ident    }
rule main = parse
   ['0'-'9']+                    { Int }
 | ['0'-'9']+'.'['0'-'9']+       { Float }
 | ['a'-'z']+                    { Ident }
```

Ocamllex examples 4 and 5

```
{  type token = Int | Float | Ident  }
rule main = parse
   ['0'-'9']+                    { Int }
 | ['0'-'9']+'.'['0'-'9']+       { Float }
 | ['a'-'z']+                    { Ident }
 | _                             { main lexbuf }
```

```
{ type token = Int of int | Float of float | Ident of string }
rule main = parse
   ['0'-'9']+ as x                    { Int (int_of_string x) }
 | ['0'-'9']+'.'['0'-'9']+ as x       { Float (float_of_string x) }
 | ['a'-'z']+ as id                   { Ident id }
 | _                                  { main lexbuf }
```

Ocamllex example 6

```
{ type token = Int of int | Float of float | Ident of string | EOF }

rule main = parse
    ['0'-'9']+ as x                  { Int (int_of_string x) }
  | ['0'-'9']+'.'['0'-'9']+ as x     { Float (float_of_string x) }
  | ['a'-'z']+ as id                 { Ident id }
  | _                                { main lexbuf }
  | eof                              { EOF }
```

Difficult cases...

- **Write regular expressions for various kinds of comments:**
  - **C++-style (`//...`)**
  - **C-style (`/*...  */`, no nesting)**
  - **OCaml-style (`(*...  *)`, with nesting)**

Simulating DFAs using lexing functions in ocamllex

- **Ocamllex specifications can have more than one function:**

```
rule main = parse
    ... regexp's and actions ...
and string = parse
    ... regexp's and actions ...
and comment = parse
    ... regexp's and actions ...
```

- **Defines functions** `main`, `string`, **and** `comment`.
- **Each function has type** `Lexing.lexbuf` $\rightarrow$ **type-of-its-actions.**
- **Each function can call the other functions. E.g.** `main` **might include:**

```
"/*"  { comment lexbuf }
```

Handling C-style comments

```
let open_comment = "/*"
let close_comment = "*/"
rule main = parse
     digits '.' digits as f    { Float (float_of_string f) }
   | digits as n               { Int (int_of_string n) }
   | letters as s              { Ident s }
   | open_comment              { comment  lexbuf }
   | eof                       { EOF }
   | _                         { main lexbuf  }
and comment = parse
     close_comment             { main lexbuf }
   | _                         { comment lexbuf }
```

Handling OCaml-style comments

- **Every lexing function has an argument named `lexbuf` of type `Lexing.lexbuf`, but you can add additional arguments. This allows you to handle nested comments (even though they are not finite-state).**

```
rule main = parse
    | open_comment          { comment 1 lexbuf}
    | eof                   { [] }
    | _                     { main lexbuf }
and comment  depth = parse
      open_comment          { comment (depth+1) lexbuf }
    | close_comment         { if depth = 1
                              then main lexbuf
                              else comment (depth - 1) lexbuf }
    | _                     { comment depth lexbuf }
```

Wrap-up

- **On Tuesday and today, we discussed:**
  - **What a lexer does**
  - **How to write a lexer by hand**
  - **How to write a lexer using ocamllex**
- **We discussed it because:**
  - **This is the first step in a compiler**
  - **Knowing how to write a lexer is a useful skill**
- **In the next class, we will:**
  - **Talk about parsing**
- **What to do now:**
  - ***MP3***
  - ***Important*: Review context-free grammars from CS 373**