Lexing
Compiler phases (simplified)

- Source text
  - Lexing
  - Token stream
  - Parsing
  - Abstract syntax tree
  - Translation
  - Intermediate representation
    - Optimization
  - Code generation
  - Assembly
• The *lexing* (or *lexical analysis*) phase of a compiler breaks a stream of characters (source text) into a stream of *tokens*.
  • Whitespace and comments often discarded

• A *token* is a sequence of characters treated as a unit. Each token is associated with a *token type*:
  • *identifier tokens*: x, y, foo, ...
  • *integer tokens*: 0, 1, −14, 512, ...
  • *if tokens*: if
  • ...

• Algebraic datatypes are a convenient representation for tokens

```plaintext
type token = IDENT of string
  | INT of int
  | IF
  | ...
```

// compute absolute value
if (x < 0) {
    return -x;
} else {
    return x;
}
Implementing a lexer

• Option 1: write by hand
• Option 2: use a lexer generator
  • Write a *lexical specification* in a domain-specific language
  • Lexer generator compiles specification to a lexer (in language of choice)
• Many lexer generators available
  • lex, flex, ocamllex, jflex, ...
Formal Languages

- An **alphabet** $\Sigma$ is a finite set of symbols (e.g., \{0, 1\}, ASCII, unicode).
- A **word** (or **string**) over $\Sigma$ is a finite sequence $w = w_1 w_2 w_3 \ldots w_n$, with each $w_i \in \Sigma$.
  - The empty word $\epsilon$ is a word over any alphabet
  - The set of all words over $\Sigma$ is typically denoted $\Sigma^*$
  - E.g., 01001 $\in$ \{0, 1\}*, covfefe $\in$ \{a, ..., z\}*

- A **language** over $\Sigma$ is a set of words over $\Sigma$
  - Integer literals form a language over \{0, ..., 9, −\}
  - The keywords of OCaml form a (finite) language over ASCII
  - Syntactically-valid Java programs forms an (infinite) language over Unicode
Regular expressions (regex)

• Regular expressions are one mechanism for describing languages
• Abstract syntax of regular expressions:

\[
<\text{RegExp}> ::= \epsilon \quad \text{Empty word}
\]
\[
| \Sigma \quad \text{Letter}
\]
\[
| <\text{RegExp}><\text{RegExp}> \quad \text{Concatenation}
\]
\[
| <\text{RegExp}>|<\text{RegExp}> \quad \text{Alternative}
\]
\[
| <\text{RegExp}>^* \quad \text{Repetition}
\]

• Meaning of regular expressions:

\[
\mathcal{L}(\epsilon) = \{\epsilon\}
\]
\[
\mathcal{L}(a) = \{a\}
\]
\[
\mathcal{L}(R_1R_2) = \{uw : u \in \mathcal{L}(R_1) \land v \in \mathcal{L}(R_2)\}
\]
\[
\mathcal{L}(R_1|R_2) = \mathcal{L}(R_1) \cup \mathcal{L}(R_2)
\]
\[
\mathcal{L}(R^*) = \{\epsilon\} \cup \mathcal{L}(R) \cup \mathcal{L}(RR) \cup \mathcal{L}(RRR) \cup \ldots
\]
• ‘a’: letter
• “abc”: string (equiv. ’a”b”c’)
• R+: one or more repetitions of R (equiv. RR*)
• R?: zero or one R (equiv. R|ε)
• [’a’−’z’]: character range (equiv. ’a’ | ’b’ | . . . | ’z’)
• R as x: bind string matched by R to variable x
Lexer generators

Lexer generators take as input a lexical specification, and output code that tokenizes a character stream w.r.t. that specification.

Example lexical specification:

- **identifier** = \([a - zA - Z][a - zA - Z0 - 9]^*\)
- **integer** = \([1 - 9][0 - 9]^*\)
- **plus** = +

- “foo+42+bar” \(\rightarrow\) **identifier** “foo”, **plus** “+”, **integer** “42”, **plus** “+”, **identifier** “bar”

- Typically, lexical spec associates an action to each token type, which is code that is evaluated on the lexeme (often: produce a token value)
Disambiguation

- May be more than one way to lex a string:

\[
IF = if \\
IDENT = [a-zA-Z][a-zA-Z0-9]^* \\
INT = [1-9][0-9]^* \\
\]

- Input string \textit{if}x<10: \begin{aligned} & \text{IDENT “ifx”, LT, INT 10} \quad \text{or} \quad & \text{IF, IDENT “x”, LT, INT 10} \\
\end{aligned}

- Input string \textit{if} \textit{x}<9: \begin{aligned} & \text{IF, IDENT “x”, LT, INT 9} \quad \text{or} \quad & \text{IDENT “if”, IDENT “x”, LT, INT 9} \\
\end{aligned}
Disambiguation

- May be more than one way to lex a string:

\[
\begin{align*}
IF &= \text{if} \\
IDENT &= [a-zA-Z][a-zA-Z0-9]^* \\
INT &= [1-9][0-9]^*
\end{align*}
\]

- Input string if\(x<10\):
  - IDENT “ifx”, LT, INT 10
  - IF, IDENT “x”, LT, INT 10

- Input string if \(x<9\):
  - IF, IDENT “x”, LT, INT 9
  - IDENT “if”, IDENT “x”, LT, INT 9

- The lexer is greedy: always prefer longest match
- Order matters: prefer earlier patterns
 Lexer generator pipeline

• Typically: lexical specification $\rightarrow$ NFA $\rightarrow$ DFA
  • Kleene's theorem: regular expressions, NFAs, and DFAs describe the same class of languages

• DFA: *Deterministic finite automaton* $A = (Q, \Sigma, \delta, s, F)$ consists of
  • $Q$: finite set of states
  • $\Sigma$: finite alphabet
  • $\delta : Q \times \Sigma \rightarrow Q$: transition function
  • $s \in Q$: initial state
  • $F \subseteq Q$: final states

DFA accepts a string $w = w_1...w_n \in \Sigma^*$ iff $\delta(...\delta(\delta(s, w_1), w_2), ..., w_n) \in F$. 
Lexer generator pipeline

- Typically: lexical specification $\rightarrow$ NFA $\rightarrow$ DFA
  - Kleene's theorem: regular expressions, NFAs, and DFAs describe the same class of languages
- DFA: *Deterministic finite automaton* $A = (Q, \Sigma, \delta, s, F)$ consists of
  - $Q$: finite set of states
  - $\Sigma$: finite alphabet
  - $\delta : Q \times \Sigma \rightarrow Q$: transition function
  - $s \in Q$: initial state
  - $F \subseteq Q$: final states

DFA accepts a string $w = w_1...w_n \in \Sigma^*$ iff $\delta(...\delta(\delta(s, w_1), w_2), ..., w_n) \in F$.

- NFA: *Non-deterministic finite automaton* $A = (Q, \Sigma, \delta, s, F)$ generalization of a DFA, where
  - $\Delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$: transition relation

NFA accepts a string $w = w_1...w_n \in \Sigma^*$ iff there exists a $w$-labeled path from $q_0$ to an accepting state (i.e., there is some sequence $(q_0, u_1, q_1), (q_1, u_2, q_2), ..., (q_{m-1}, u_m, q_m)$ with $q_0 = s$, $q_m \in F$, and $u_1u_2...u_m = w$.)
Case: $\epsilon$ (empty word)
Case: $a$ (letter)
Case: $R_1 R_2$ (concatenation)
Regex $\rightarrow$ NFA

Case: $R_1 R_2$ (concatenation)
Regex → NFA

Case: $R_1 | R_2$ (alternative)
Case: $R_1 | R_2$ (alternative)
Case: $R^*$ (iteration)
Case: $R^*$ (iteration)
• For any NFA, there is a DFA that recognizes the same language
• Intuition: the DFA simulates all possible paths of the NFA simultaneously
  • There is an unbounded number of paths \textit{but} we only care about the “end state” of each path, not its history
  • States of the DFA track the set of possible states the NFA could be in
  • DFA accepts when \textit{some} path accepts
NFA → DFA, formally

- Have: NFA $A = (Q, \Sigma, \delta, s, F)$. Want: DFA $A' = (Q', \Sigma, \delta', s', F')$ that accepts same language.
- For any $S \subseteq Q$, define the $\epsilon$-closure of $S$ to be the set of states reachable from $S$ by $\epsilon$ transitions (incl. $S$)

$\epsilon$-cl($S$) = smallest set that contains $S$ and such that $\forall (q, \epsilon, q') \in \Delta, q \in S \Rightarrow q' \in S$
NFA $\rightarrow$ DFA, formally

• Have: NFA $A = (Q, \Sigma, \delta, s, F)$. Want: DFA $A' = (Q', \Sigma, \delta', s', F')$ that accepts same language.

• For any $S \subseteq Q$, define the $\epsilon$-closure of $S$ to be the set of states reachable from $S$ by $\epsilon$ transitions (incl. $S$)

  $\epsilon\text{-cl}(S) =$ smallest set that contains $S$ and such that $\forall (q, \epsilon, q') \in \Delta, q \in S \Rightarrow q' \in S$

• Construct DFA as follows:
  • $Q' =$ set of all $\epsilon$-closed subsets of $Q$
  • $\delta'(S, a) = \{q_2 : \exists q_1 \in S. (q_1, a, q_2) \in \Delta\}$
  • $s' =$ $\epsilon$-closure of $\{s\}$
  • $F' = \{S \in Q' : S \cap F \neq \emptyset\}$
NFA $\rightarrow$ DFA, formally

- Have: NFA $A = (Q, \Sigma, \delta, s, F)$. Want: DFA $A' = (Q', \Sigma, \delta', s', F')$ that accepts same language.
- For any $S \subseteq Q$, define the $\epsilon$-closure of $S$ to be the set of states reachable from $S$ by $\epsilon$ transitions (incl. $S$)
  $\epsilon$-$cl(S) =$ smallest set that contains $S$ and such that $\forall (q, \epsilon, q') \in \Delta, q \in S \Rightarrow q' \in S$
- Construct DFA as follows:
  - $Q' =$ set of all $\epsilon$-closed subsets of $Q$
  - $\delta'(S, a) = \{ q_2 : \exists q_1 \in S. (q_1, a, q_2) \in \Delta \}$
  - $s' =$ $\epsilon$-closure of $\{s\}$
  - $F' = \{ S \in Q' : S \cap F \neq \emptyset \}$
- Crucial optimization: only construct states that are reachable from $s'$
NFA → DFA, formally

• Have: NFA $A = (Q, \Sigma, \delta, s, F)$. Want: DFA $A' = (Q', \Sigma, \delta', s', F')$ that accepts same language.

• For any $S \subseteq Q$, define the $\epsilon$-closure of $S$ to be the set of states reachable from $S$ by $\epsilon$ transitions (incl. $S$)

$$\epsilon-cl(S) = \text{smallest set that contains } S \text{ and such that } \forall (q, \epsilon, q') \in \Delta, q \in S \Rightarrow q' \in S$$

• Construct DFA as follows:
  • $Q' = \text{set of all } \epsilon\text{-closed subsets of } Q$
  • $\delta'(S, a) = \{ q_2 : \exists q_1 \in S. (q_1, a, q_2) \in \Delta \}$
  • $s' = \epsilon\text{-closure of } \{ s \}$
  • $F' = \{ S \in Q' : S \cap F \neq \emptyset \}$

• Crucial optimization: only construct states that are reachable from $s'$

• Less crucial, still important: minimize DFA
Lexical specification → String classifier

- Want: partial function *match* mapping strings to token types
  - \( \text{match}(s) = \) highest-priority token type whose pattern matches \( s \) (undef otherwise)

- Process:
  1. Convert each pattern to an NFA. Label accepting states w/ token types.
  2. Take the union of all NFAs
  3. Convert to DFA
     - States of the DFA labeled with sets of token types.
     - Take highest priority.

\[
\begin{align*}
\text{identifier} &= [a-zA-Z][a-zA-Z0-9]^* \\
\text{integer} &= [1-9][0-9]^* \\
\text{float} &= ([1-9][0-9]^*|0).[0-9]^+ 
\end{align*}
\]
\{i_0, n_0, f_0\}
\{i_0, n_0, f_0\} \rightarrow \{i_1\} \quad \text{identifier}

\{i_0, n_0, f_0\} \rightarrow \{f_1\} \quad \text{int}

\{i_0, n_0, f_0\} \rightarrow \{n_1, f_1\} \quad [0-9]

[a-zA-Z]
f_i \in \{f \mid \text{int}\} \cup \{f \mid \text{float}\} \cup \{f \mid \text{identifier}\}

\{f\} \rightarrow \{f, u\} \rightarrow \{f, 0, u, v\}

\{I\} \rightarrow 0

\{I\} \rightarrow [Z - Vz - v]

[6 - 0Z - Vz - v]