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

```
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 lex
er generator
  - Write a \textit{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, tokens).
- 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., $01001 \in \{0, 1\}^*$, $\text{covfefe} \in \{a, \ldots, z\}^*$.
- A **language** over $\Sigma$ is a set of words over $\Sigma$.
  - Integer literals form a language over $\{0, \ldots, 9, -\}$.
  - 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:

\[
\begin{align*}
\text{<RegExp>} & ::= \epsilon & \text{Empty word} \\
& \mid \Sigma & \text{Letter} \\
& \mid <\text{RegExp}> <\text{RegExp}> & \text{Concatenation} \\
& \mid <\text{RegExp}> | <\text{RegExp}> & \text{Alternative} \\
& \mid <\text{RegExp}>^* & \text{Repetition}
\end{align*}
\]
Regular expressions (regex)

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

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

- Meaning of regular expressions:

  \[\mathcal{L}(\epsilon) = \{\epsilon\}\]
  \[\mathcal{L}(a) = \{a\}\]
  \[\mathcal{L}(R_1 R_2) = \{uv : u \in \mathcal{L}(R_1) \land v \in \mathcal{L}(R_2)\}\]
  \[\mathcal{L}(R_1 | R_2) = \mathcal{L}(R_1) \cup \mathcal{L}(R_2)\]
  \[\mathcal{L}(R^*) = \{\epsilon\} \cup \mathcal{L}(R) \cup \mathcal{L}(RR) \cup \mathcal{L}(RRR) \cup \ldots\]
ocamllex regex concrete syntax

- '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:

```
<table>
<thead>
<tr>
<th>token type</th>
<th>pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>identifier</td>
<td></td>
</tr>
</tbody>
</table>

\[a-zA-Z0-9]\* |
| integer     | \[1-9]\[0-9]\* |
| plus        | + |
```

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

```
token type

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

- “foo+42+bar” → **identifier** “foo”, **plus “+”, integer “42”,** **plus “+”, identifier “bar”**

• token type • lexeme
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** = \([a-zA-Z0-9]*\)
- **integer** = \([1-9][0-9]*\)
- **plus** = \(+\)

- “foo+42+bar” → 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:

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

• Input string ifx<10: IDENT “ifx”, LT, INT 10 or IF, IDENT “x”, LT, INT 10?
• Input string if x<9: IF, IDENT “x”, LT, INT 9 or IDENT “if”, IDENT “x”, LT, INT 9?
Disambiguation

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

\[
IF = \text{if} \\
IDENT = [a-zA-Z][a-zA-Z0-9]^* \\
INT = [1-9][0-9]^* \\
LT = <
\]

... 

• Input string if x<10: \text{IDENT “ifx”, LT, INT 10} or \text{IF, IDENT “x”, LT, INT 10}?

• Input string if x<9: \text{IF, IDENT “x”, LT, INT 9} or \text{IDENT “if”, IDENT “x”, LT, INT 9}?

• The lexer is greedy: always prefer longest match

• Order matters: prefer earlier patterns
 Lexer generator pipeline

- Lexical specification is compiled to a *deterministic finite automaton* (DFA), which can be executed efficiently
- Typical pipeline: lexical specification $\rightarrow$ *nondeterministic FA* $\rightarrow$ DFA
- Kleene's theorem: regular expressions, NFAs, and DFAs describe the same class of languages
  - A language is *regular* if it is accepted by a regular expression (equiv., NFA, DFA).
A **deterministic finite automaton** (DFA) $A = (Q, \Sigma, \delta, s, F)$ consists of

- $Q$: finite set of states
- $\Sigma$: finite alphabet
- $\delta : Q \times \Sigma \rightarrow Q$: transition function
  - Every state has *exactly* one outgoing edge per letter
- $s \in Q$: initial state
- $F \subseteq Q$: final states

DFA accepts a string $w = w_1 \ldots w_n \in \Sigma^*$ iff $\delta(...\delta(\delta(s, w_1), w_2), ..., w_n) \in F$. 
A non-deterministic finite automaton (NFA) $A = (Q, \Sigma, \Delta, s, F)$ generalization of a DFA, where

- $\Delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$: transition relation
  - A state can have more than one outgoing edge for a given letter
  - A state can have no outgoing edges for a given letter
  - A state can have $\epsilon$-transitions (read no input, but change state)
A non-deterministic finite automaton (NFA) $A = (Q, \Sigma, \Delta, s, F)$ generalization of a DFA, where

- $\Delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$: transition relation
  - A state can have more than one outgoing edge for a given letter
  - A state can have no outgoing edges for a given letter
  - A state can have $\epsilon$-transitions (read no input, but change state)

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 $(q_0, u_1, q_1), (q_1, u_2, q_2), \ldots, (q_{m-1}, u_m, q_m)$ with $q_0 = s, q_m \in F$, and $u_1u_2...u_m = w$.
Case: $\epsilon$ (empty word)
Regex $\rightarrow$ NFA

Case: $a$ (letter)

\[ \text{start} \xrightarrow{a} s_0 \xrightarrow{a} s_f \]
Case: $R_1 R_2$ (concatenation)
Case: $R_1 R_2$ (concatenation)
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 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 some path accepts
NFA → DFA

\[
\begin{align*}
\text{start} & \rightarrow s_0 \\
& \quad \rightarrow s_1 \quad a \\
& \quad \rightarrow s_2 \quad a \\
& \quad \rightarrow s_f \quad \epsilon
\end{align*}
\]
NFA $\rightarrow$ DFA

Diagram of an NFA and its corresponding DFA.
NFA → DFA

\begin{align*}
\text{start} & \quad s_0 \quad s_1 \quad s_2 \quad s_f \\
& \quad \quad \quad \quad a \quad a \quad \epsilon \\
& \quad \quad \quad \quad b \quad b
\end{align*}
NFA $\rightarrow$ DFA
NFA → DFA
NFA $\rightarrow$ DFA
NFA $\rightarrow$ DFA

\begin{center}
\begin{tikzpicture}

\node[state,initial] (s0) at (0,0) {$s_0$};
\node[state] (s1) at (3,0) {$s_1$};
\node[state] (s2) at (6,0) {$s_2$};
\node[state,accepting] (sf) at (9,0) {$sf$};
\node[state] (0) at (6,-3) {$\emptyset$};

\path[->]
(s0) edge node {$a$} (s1)
(s1) edge node {$a$} (s2)
(s2) edge node {$\epsilon$} (sf)
(s0) edge[loop below] node {$b$} (s0)
(0) edge[loop above] node {$a$} (0)
(0) edge node {$b$} (s1)
(s1) edge node {$b$} (0)
(s1) edge[loop below] node {$a$} (s1);

\end{tikzpicture}
\end{center}
NFA $\rightarrow$ DFA
NFA $\rightarrow$ DFA

![Diagram of an NFA and its equivalent DFA](image-url)
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. $S$)
  
  $\epsilon$-$cl(S) = \text{smallest set that contains } S \text{ and such that } \forall (q, \epsilon, q') \in \Delta, q \in S \Rightarrow q' \in S$
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. $S$)
  
  $\epsilon$-$cl(S) =$ 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 $Q$
  - $\delta'(S, a) =$ $\epsilon$-closure of $\{q_2 : \exists q_1 \in S. (q_1, a, q_2) \in \Delta\}$
  - $s' =$ $\epsilon$-closure of $\{s\}$
  - $F' =$ $\{S \in Q' : S \cap F \neq \emptyset\}$
NFA → 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 $Q$
  - $\delta'(S, a) =$ $\epsilon$-closure of$\{q_2 : \exists q_1 \in S. (q_1, a, q_2) \in \Delta\}$
  - $s' =$ $\epsilon$-closure of $\{s\}$
  - $F' =$ $\{S \in Q' : S \cap F \neq \emptyset\}$
- Crucial optimization: only construct states that are reachable from $s'$
NFA $\rightarrow$ 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 $Q$
  - $\delta'(S, a) =$ $\epsilon$-closure of $\{q_2 : \exists q_1 \in S. (q_1, a, q_2) \in \Delta\}$
  - $s' =$ $\epsilon$-closure of $\{s\}$
  - $F' = \{S \in Q' : S \cap F \neq \emptyset\}$
- Crucial optimization: only construct states that are reachable from $s'$
- Less crucial, still important: minimize DFA (Hopcroft’s algorithm, $O(n \log n)$)
Lexical specification → String classifier

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

\[
\begin{align*}
\text{identifier} &= [a - zA - Z][a - zA - Z0 - 9]^* \\
\text{integer} &= [1 - 9][0 - 9]^* \\
\text{float} &= ([1 - 9][0 - 9]^*|0).[0 - 9]^+
\end{align*}
\]
\(\{i_0, n_0, f_0\}\)
The image contains a diagram with nodes and edges. The nodes are labeled with sets such as \( \{i_0, n_0, f_0\} \), \( \{i_1\} \), \( \{n_1, f_1\} \), and \( \{f_2\} \). The edges are labeled with characters from \( [a-zA-Z0-9] \) and \( [1-9] \). The diagram includes arrows indicating transitions between these sets. The diagram also includes labels such as "identifier" and "float".
\[ [a - zA - Z0 - 9] \]

**Diagram:**

- **Node \( \{i_0, n_0, f_0\} \):**
  - Input: \( [1 - 9] \)
  - Output: \( [0 - 9] \)
- **Node \( \{i_1\} \):**
  - Input: \( [a - zA - Z] \)
  - Output: \( \{f_1\} \)
- **Node \( \{f_1\} \):**
  - Input: \( 0 \)
  - Output: \( \{f_2\} \)
- **Node \( \{n_1, f_1\} \):**
  - Input: \( [0 - 9] \)
  - Output: \( \text{int} \)
- **Node \( \{f_2\} \):**
  - Input: \( [0 - 9] \)
  - Output: \( \text{float} \)

**Identifiers:**
- **identifier**
- **int**
- **float**