COS320: Compiling Techniques

Zak Kincaid

February 13, 2022
Lexing
Compiler phases (simplified)

1. Source text
   - Lexing
2. Token stream
   - Parsing
3. Abstract syntax tree
   - Translation
4. Intermediate representation
   - Optimization
   - Code generation
5. 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 (a *lexeme*) along with an *token type*:
  - *identifier tokens*: \( x, y, \text{foo}, \ldots \)
  - *integer tokens*: \( 0, 1, -14, 512, \ldots \)
  - *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 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, 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., $01001 \in \{0, 1\}^*$, *embiggen* $\in \{a, \ldots, z\}^*$.
- 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 form 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}
  
  \text{\mid } \Sigma \quad \text{Letter}
  
  \text{\mid <RegExp><RegExp>} \quad \text{Concatenation}
  
  \text{\mid <RegExp>|<RegExp>} \quad \text{Alternative}
  
  \text{\mid <RegExp>\ast} \quad \text{Repetition}
  \]

- Meaning of regular expressions:

  \[
  L(\epsilon) = \{\epsilon\}
  
  L(a) = \{a\}
  
  L(R_1R_2) = \{uv: u \in L(R_1) \land v \in L(R_2)\}
  
  L(R_1|R_2) = L(R_1) \cup L(R_2)
  
  L(R^\ast) = \{\epsilon\} \cup L(R) \cup L(RR) \cup L(RRR) \cup \ldots
  \]
Regular expressions (regex)

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

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

- Meaning of regular expressions:

\[
\mathcal{L}(\epsilon) = \{\epsilon\}
\]
\[
\mathcal{L}(a) = \{a\}
\]
\[
\mathcal{L}(R_1 R_2) = \{uv : u \in \mathcal{L}(R_1) \land v \in \mathcal{L}(R_2)\}
\]
\[
\mathcal{L}(R_1 | R_2) = \mathcal{L}(R_1) \cup \mathcal{L}(R_2)
\]
\[
\mathcal{L}(R^*) = \{\epsilon\} \cup \mathcal{L}(R) \cup \mathcal{L}(RR) \cup \mathcal{L}(RRR) \cup \ldots
\]
• ‘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 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 = + }
```

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:

- **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)
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:

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

- “foo+42+bar” \( \rightarrow \) \texttt{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:

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

\[
\ldots
\]

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

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

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

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

\[
\ldots
\]

• Input string if x<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?

• Two rules sufficient to disambiguate (remember these!)
  1. **The lexer is greedy**: always prefer longest match
  2. **Order matters**: prefer earlier patterns
• Lexical specification is compiled to a \textit{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 \textit{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 (accepting) states

**DFA accepts a string** $w = w_1...w_n \in \Sigma^*$ iff $\delta(\delta(\delta(s, w_1), w_2), ..., w_n) \in F$. 
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 $w = w_1 \ldots w_n \in \Sigma^*$ iff there exists a $w$-labeled path from $s$ to an final 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 $u_1 u_2 \ldots u_m = w$.
Case: $\epsilon$ (empty word)
Case: $a$ (letter)
Regex → NFA

Case: $R_1 R_2$ (concatenation)
Case: $R_1R_2$ (concatenation)
Case: $R_1|R_2$ (alternative)
Case: $R_1 | R_2$ (alternative)
Case: $R^*$ (iteration)
Case: $R^*$ (iteration)
NFA $\rightarrow$ DFA

- 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

start → $s_0$ → $s_1$ → $s_2$ → $s_f$
NFA → DFA

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

start

\[ s_0 \quad \xrightarrow{a} \quad s_1 \quad \xrightarrow{a} \quad s_2 \quad \xrightarrow{\epsilon} \quad s_f \]

\[ s_0 \quad \xrightarrow{a} \quad s_1 \quad \xrightarrow{b} \quad \emptyset \quad \xrightarrow{a} \quad s_1, s_f \quad \xrightarrow{a} \quad s_2, s_f \]
NFA → DFA

Start

\[ \begin{align*}
    s_0 & \xrightarrow{a} s_1 \\
    s_1 & \xrightarrow{a} s_2 \\
    s_2 & \xrightarrow{\epsilon} s_f
\end{align*} \]

\[ s_0 \xrightarrow{b} \emptyset \xrightarrow{b} s_0 \]

\[ s_0 \xrightarrow{a} s_1, s_f \xrightarrow{b} s_2, s_f \]
NFA $\rightarrow$ DFA
NFA → DFA

start → $s_0$ → $s_1$ → $s_2$ → $s_f$

$b$

$\emptyset$

$a$

$b$

$a$

$b$

$a$

$b$

$a$
NFA → DFA
NFA → DFA

Start

$s_0$ → $s_1$ → $s_2$ → $s_f$

$a$

$b$

$\emptyset$

$a$

$b$

$a$

$b$

$a$

$b$

$a$

$b$

$a$

$b$

$a$

$b$

$a$

$b$

$a$

$b$

$a$

$b$

$a$
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. $S$)

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

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

- Construct DFA as follows:
  - $Q' = \text{set of all } \epsilon$-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, $O(n \log n)$)
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. $S$)
  $\epsilon$-$\text{cl}(S) = \text{smallest set that contains } S \text{ and such that } \forall (q, \epsilon, q') \in \Delta, q \in S \Rightarrow q' \in S$
- Construct DFA as follows:
  - $Q' = \text{set of all } \epsilon$-closed subsets of $Q$
  - $\delta'(S, a) = \epsilon$-$\text{closure of } \{ q_2 : \exists q_1 \in S. (q_1, a, q_2) \in \Delta \}$
  - $s' = \epsilon$-$\text{closure of } \{ s \}$
  - $F' = \{ S \in Q' : S \cap F \neq \emptyset \}$
- Crucial optimization: only construct states that are reachable from $s'$
- Less crucial, still important: minimize DFA (Hopcroft’s algorithm, $O(n \log n)$)
Lexical specification → String classifier

• Want: partial function \( \text{match} \) 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.

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

\[i_0 \rightarrow i_1\]

\[a-zA-Z] \quad [a-zA-Z0-9]\]

integer

\[n_0 \rightarrow n_1\]

\[1-9\] \quad [0-9]\]

float

\[f_0 \rightarrow f_1 \rightarrow f_2\]

\[f_0 \rightarrow f_1 \rightarrow f_2\]

\[0 \rightarrow [1-9] \rightarrow [0-9]\]}
\{i_0, n_0, f_0\}
$\{i_0, n_0, f_0\} \rightarrow \{i_1\}$  \hspace{1cm} identifier

$\{i_0, n_0, f_0\} \rightarrow \{f_1\}$  \hspace{1cm} \text{float}$[0-9]$

$\{i_0, n_0, f_0\} \rightarrow \{n_1, f_1\}$  \hspace{1cm} \text{int}$[1-9]$

$[a-zA-Z]$$[0-9]$
\[ [a - zA - Z0 - 9] \]
\[
\begin{align*}
&[a - zA - Z0 - 9] \\
&\{i_0, n_0, f_0\} \\
&\{i_1\} \quad \text{identifier} \\
&\{n_1, f_1\} \quad \text{int} \\
&\{f_1\} \\
&\{f_2\} \quad \text{float}
\end{align*}
\]
\[ [a - zA - Z0 - 9] \]

\[ [a - zA - Z] \]

\[ [a - zA - 1] \]

\[ [0 - 9] \]

\[ [1 - 9] \]

\[ [0 - 9] \]
$$[a - zA - Z0 - 9]$$