Compilers

Lexical Specification

Syntax - CodeGen
Semantics - Types

Alex Aiken
• At least one: $A^+$ \equiv AA^*$
• Union: $A \mid B$ \equiv $A + B$
• Option: $A?$ \equiv $A + \varepsilon$
• Range: ‘$a$’+‘$b$’+...+‘$z$’ \equiv [a-z]
• Excluded range:
  complement of [a-z] \equiv [^a-z]
• Last lecture: a specification for the predicate 
\[ s \in L(R) \]
• But a yes/no answer is not enough!
  • Instead: partition the input into tokens
    \[ c_1c_2c_3|c_4c_5c_6c_7|... \]
• We adapt regular expressions to this goal
1. Write a rexp for the lexemes of each token class
   - Number = digit*
   - Keyword = ‘if’ + ‘else’ + ...
   - Identifier = letter (letter + digit)*
   - OpenPar = ‘(‘
   - ...

2. Construct $R$, matching all lexemes for all tokens

$$R = \text{Keyword} + \text{Identifier} + \text{Number} + \ldots$$

$$= R_1 + R_2 + \ldots$$
3. Let input be $x_1...x_n$
   For $1 \leq i \leq n$ check
   
   $x_1...x_i \in L(R)$

4. If success, then we know that
   
   $x_1...x_i \in L(R_j)$ for some $j$

5. Remove $x_1...x_i$ from input and go to (3)
• How much input is used?

\[ x_1...x_i \in L(R) \]
\[ x_1...x_j \in L(R) \]
\[ j \neq i \]

• Example

\[ == \]

Rule: Pick longest possible string in \( L(R) \) “Maximal Munch”
Lexical Specification

• Which token is used?

\[ x_1 \ldots x_i \in L(R_j) \quad R = R_1 + R_2 + R_3 + \ldots \]
\[ x_1 \ldots x_i \in L(R_k) \quad j \neq k \]

Keyword = ‘if’ + ‘else’ + ...
Identifier = letter (letter + digit)*

• Use rule listed first (j if j < k)
Treats “if” as a keyword, not an identifier
• What if no rule matches?

\[ x_1 \ldots x_i \notin L(R_j) \]

• Can’t just get stuck …

  • Write a rule matching all “bad” strings
  • Put it last (lowest priority)
Regular expressions are a concise notation for string patterns.

Use in lexical analysis requires small extensions:
- To resolve ambiguities
- To handle errors

Good algorithms known:
- Require only single pass over the input
- Few operations per character (table lookup)
Compilers

Finite Automata
• Regular expressions = specification
• Finite automata = implementation

• A finite automaton consists of
  – An input alphabet $\Sigma$
  – A finite set of states $S$
  – A start state $n$
  – A set of accepting states $F \subseteq S$
  – A set of transitions $\text{state} \rightarrow^{\text{input}} \text{state}$
Finite Automata

• Transition

\[ S_1 \xrightarrow{a} S_2 \]

• Is read

In state \( S_1 \) on input \( a \) go to state \( S_2 \)

• If end of input and in accepting state \( \Rightarrow \) accept

• Otherwise \( \Rightarrow \) reject

  • Terminates in a state \( s \) that is NOT an accepting state \( (s \notin F) \)
  • Gets stuck
Finite Automata

- A state
- The start state
- An accepting state
- A transition
A finite automaton that accepts only “1”

- Accepts ‘1’ : \[\uparrow 1, \uparrow 1\]
- Rejects ‘0’ : \[\uparrow 0\]
- Rejects ’10’ : \[\uparrow 1, 1\uparrow 0\]
• A finite automaton accepting any number of 1’s followed by a single 0
• Alphabet: \{0,1\}

• Accepts ‘110’: ↑110, 1↑10, 11↑0, 110↑
• Rejects ‘100’: ↑100, 1↑00, 10↑0
Select the regular language that denotes the same language as this finite automaton

- \((0 + 1)^*\)
- \((1^* + 0)(1 + 0)\)
- \(1^* + (01)^* + (001)^* + (000*1)^*\)
- \((0 + 1)^*00\)
Select the regular language that denotes the same language as this finite automaton:

- $(0 + 1)^*$
- $(1^* + 0)(1 + 0)$
- $1^* + (01)^* + (001)^* + (000^*1)^*$
- $(0 + 1)^*00$
• Another kind of transition: $\varepsilon$-moves

$\varepsilon$

A \rightarrow B

• Machine can move from state A to state B without reading input
Finite Automata

- **Deterministic Finite Automata (DFA)**
  - One transition per input per state
  - No $\varepsilon$-moves

- **Nondeterministic Finite Automata (NFA)**
  - Can have multiple transitions for one input in a given state
  - Can have $\varepsilon$-moves
• **A DFA takes only one path through the state graph**
  - Completely determined by input

• **An NFA can choose**
  - Whether to make ε-moves
  - Which of multiple transitions for a single input to take
• An NFA can get into multiple states

• Input:

• Possible States:

• Rule: NFA accepts if it can get to a final state
• An NFA can get into multiple states

• Input:

• Possible States: \{A\} \{A,B\} \{A,B,C\}

• Rule: NFA accepts if it can get to a final state
Finite Automata

- NFAs and DFAs recognize the same set of languages
  - regular languages

- DFAs are faster to execute
  - There are no choices to consider

- NFAs are, in general, smaller
  - Sometimes exponentially smaller
Compilers

Regular Expressions to NFAs

Alex Aiken
For each kind of rexp, define an equivalent NFA

- Notation: \( \text{NFA}_M \) for rexp \( M \)

- For \( \varepsilon \)

- For input \( a \)
• For $AB$

• For $A + B$
• For $A^*$
• Consider the regular expression \((1+0)^*1\)
Choose the NFA that accepts the following regular expression: $1^* + 0$
Choose the NFA that accepts the following regular expression: $1^* + 0$
Compilers

Syntax  CodeGen
Semantics  Types

NFA to DFA
NFA to DFA

NFA

Regular expressions

Lexical Specification

DFA

Table-driven Implementation of DFA

Alex Aiken
\(\varepsilon\)-closure of a state

\(\varepsilon\)-closure(B) = \{B, C, D\} \quad \varepsilon\)-closure(G) = \{A, B, C, D, G, H, I\}
An NFA may be in many states at any time

How many different states?

If there are $N$ states, the NFA must be in some subset of those $N$ states

How many non-empty subsets are there?

$2^N - 1 = \text{finitely many}$
NFA to DFA

- Simulate the NFA
- Each state of DFA
  - a non-empty subset of states of the NFA
- Start state
  - $\varepsilon$-closure of the start state of NFA
- Add a transition $S \xrightarrow{a} S'$ to DFA iff
  - $S'$ is the set of NFA states reachable from any state in $S$ after seeing the input $a$, considering $\varepsilon$ moves as well
- Final states
  - Subsets that include at least one final state of NFA
NFA to DFA
NFA to DFA

Alex Aiken
Choose the DFA that represents the same language as the given NFA.
Choose the DFA that represents the same language as the given NFA.
Compilers
Implementing Finite Automata

Syntax
CodeGen
Semantics
Types
Implementing FA

NFA

Regular expressions

Lexical Specification

DFA

Table-driven Implementation of DFA
• A DFA can be implemented by a 2D table $T$
  – One dimension is states
  – Other dimension is input symbol
  – For every transition $S_i \xrightarrow{a} S_k$ define $T[i,a] = k$

• DFA “execution”
  – If in state $S_i$ and input $a$, read $T[i,a] = k$ and skip to state $S_k$
  – Very efficient
Implementing FA

\[\begin{array}{c|cc}
 & 0 & 1 \\
\hline
S & T & U \\
T & T & U \\
U & T & U \\
\end{array} \]

\[
i = 0; \\
state = 0; \\
while (input[i]) { \\
\quad state = A[state, input[i++]]; \\
}\]
Implementing FA

• NFA -> DFA conversion is at the heart of tools such as flex

• But, DFAs can be huge
  In the worst case, there is $2^N - 1$ DFA states for an NFA with N states

• In practice, flex-like tools trade off speed for space in the choice of NFA and DFA