Implementation of Lexical Analysis (Scanning)

Adapted from material by:
Prof. Alan Aiken and Prof. George Necula (UCB)
Prof. Saman Amarasinghe (MIT)

Outline

• Specifying lexical structure using regular expressions
• Recognizing tokens using finite automata
  ➢ Deterministic Finite Automata (DFAs)
  ➢ Non-deterministic Finite Automata (NFAs)
• Implementation of regular expressions
  RegExp => NFA => DFA => Tables

Regular Expressions => Lexical Spec.

1. Write a regular expression for the lexemes of each token
   • Number = digit (Kleene plus)
   • Keyword = ‘if’ | ‘else’ | … (Union)
   • Identifier = letter (letter | digit)*
   • OpenPar = ‘(’
   • …
2. Construct R, matching all lexemes for all tokens
   R = Keyword | Identifier | Number | …
      = R₁ | R₂ | …

(Cont’d)

3. Let input be the sequence of characters
   x₁…xₙ
   • For each 1 ≤ i ≤ n, check if
     x₁…xᵢ ∈ L(R)
4. It must be that
   x₁…xᵢ ∈ L(R) for some j
5. Remove x₁…xᵢ from input and go to (3)

Problem:
There are ambiguities in the algorithm.
Ambiguities

- How much input is used? What if
  - $x_1 \ldots x_i \in L(R)$ and also
  - $x_1 \ldots x_K \in L(R)$
  - Rule: Pick longest possible string in $L(R)$
    - The “maximal munch” $\subseteq \forall <, \not= \forall$

- Which token is used? What if
  - $x_1 \ldots x_i \in L(R_j)$ and also
    - $x_1 \ldots x_i \in L(R_k)$
    - Rule: use rule listed first (if $j < k$)
        - Treats “if” as a keyword not an identifier

Error Handling

- What if
  - No rule matches a prefix of input?
- Problem: Can get stuck …

- Solution:
  - Write a rule matching all “bad” strings
  - Put it last (catch-all clause)

Summary

- Regular expressions provide 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)

Parity Problem

- $\Sigma = \{0,1\}, \omega \in \Sigma^*$
- $\text{parity}: \Sigma^* \to \text{boolean}$
- $\text{parity}(\omega) \iff \omega$ contains even number of 1s

Finite automaton = Recognizer
Basic Features

- Consumes the entire input string.
- Remembers the parity of the bit string by abstracting from the number of 1s in the string.
- Finite amount of memory required for this purpose.
  - Observe that counting requires unbounded memory, while computing the parity requires very small and fixed amount of memory.
- Accepts/Rejects the input in a deterministic fashion.

Deterministic Finite State Automaton (DFA)

\[ M = (Q, \Sigma, \delta, q_0, F) \]

- \( Q \): Finite set of states
- \( \Sigma \): Finite Alphabet
- \( \delta \): Transition function total function from \( Q \times \Sigma \) to \( Q \)
- \( q_0 \): Initial/Start State
- \( F \): Set of final/accepting state

Finite Automata State Graphs

- State
- The start state
- An accepting state
- A transition

Operation of the machine

- Read the current letter of input under the tape head.
- Transit to a new state depending on the current input and the current state, as dictated by the transition function.
- Halt after consuming the entire input.
**Associating Language with DFA**

- **Machine configuration:**
  \[ [q, \omega] \text{ where } q \in Q, \omega \in \Sigma^* \]

- **Yields relation:**
  \[ [q, a\omega] \mapsto_M [\delta(q, a), \omega] \]

- **Language:**
  \[ \{ \omega \in \Sigma^* | [q_0, \omega] \mapsto_M [q, \varepsilon] \land q \in F \} \]

**Example**

- **Set of strings over \( \{a, b\} \) that contain \( bb \)
  \((a | b)^* bb(a | b)^*\)

- **Design states by partitioning \( \Sigma^* \).**
  - Strings containing \( bb \) \( q_2 \)
  - Strings not containing \( bb \)
    - Strings that end in \( b \) \( q_1 \)
    - Strings that do not end in \( b \) \( q_0 \)

- **Initial state:** \( q_0 \)
- **Final state:** \( q_2 \)

**State Diagram and Table**

<table>
<thead>
<tr>
<th>State</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_0 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( q_1 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( q_2 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transition</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_0 )</td>
<td>( q_0 )</td>
<td>( q_1 )</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>( q_0 )</td>
<td>( q_2 )</td>
</tr>
<tr>
<td>( q_2 )</td>
<td>( q_2 )</td>
<td>( q_2 )</td>
</tr>
</tbody>
</table>

| String over \( \{a, b\} \) that do not contain \( bb \) |
|-----------------|---|---|
| \( q_0 \) |   |   |
| \( q_1 \) |   |   |
| \( q_2 \) |   |   |

<table>
<thead>
<tr>
<th>Transition</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_0 )</td>
<td>( q_0 )</td>
<td>( q_1 )</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>( q_0 )</td>
<td>( q_2 )</td>
</tr>
<tr>
<td>( q_2 )</td>
<td>( q_2 )</td>
<td>( q_2 )</td>
</tr>
</tbody>
</table>
Strings over \{a, b\} containing even number of a’s and odd number of b’s.

\[ \Sigma^* \]

\[ \{E_a, E_b\} \]

\[ \{O_a, O_b\} \]

Example

- Alphabet \{0, 1\}
- What language does this recognize?

Nondeterministic Finite Automata

DFA

\[ \delta_{DFA} : Q \times \Sigma \rightarrow Q \]

NFA

\[ \delta_{NFA} : Q \times \Sigma \rightarrow \text{Pow}(Q) \]

\[ \delta_{NFA} \subseteq Q \times \Sigma \times Q \]

\[ (a, b)^* bb \]

NFA State Diagram

(Strings over \{a, b\} ending in bb)

\[ Q = \{q_0, q_1, q_2\} \]

\[ \Sigma = \{a, b\} \]

\[ F = \{q_2\} \]

\[ \delta \]

\[ \begin{array}{c|cc}
   \delta & a & b \\
   \hline
   q_0 & \{q_0\} & \{q_0, q_1\} \\
   q_1 & \phi & \{q_2\} \\
   q_2 & \phi & \phi \\
\end{array} \]
Introducing $\varepsilon$-transitions into NFA

$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow P(Q)$

- A $\varepsilon$-transition causes the machine to change its state non-deterministically, without consuming any input.

$L(DFAs) \subseteq L(NFAs)$

Deterministic and Nondeterministic 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
- Finite automata have finite memory
  - Need only to encode the current state

How do we associate a language with NFA?

- A DFA can take only one path through the state graph
- NFAs can choose
  - Whether to make $\varepsilon$-moves
  - Or one of the multiple transitions for a single input to take
    - Accept if there exists a accepting computation.
    - Reject if all computations are non-accepting.
Every DFA is an NFA. However, does non-determinism make NFAs strictly more expressive (powerful) than DFAs?

\[ L(\text{NFAs}) \supseteq L(\text{DFAs}) \]

- For type 0 languages (Turning Machines) and type 3 languages (regular languages), non-determinism does not add expressive power.
- For context-free languages and context-sensitive languages, non-determinism does enhance the expressive power.
NFA vs. DFA

- NFAs and DFAs recognize the same set of languages (regular languages).
- DFAs are faster to execute
  - There are no choices to consider.
- For a given language, NFA can be simpler than DFA
- DFA can be exponentially larger than NFA.

Reg. Expr. to Finite Automata

- High-level sketch

Regular Expressions to NFA (1)

- For each kind of regular expr., define an NFA
  - Notation: NFA for rexp M

Regular Expressions to NFA (2)

- For \( A \) and \( B \)
  - For \( A \) and \( B \)
- For \( A | B \)
  - For \( A | B \)
Regular Expressions to NFA (3)

- For $A^*$

\[ A^* \]

\[ \varepsilon \]

Thompson's Construction

Reg. Expr. $\rightarrow$ NFA conversion

- Consider the regular expression $(1|0)^*1$

\[ (1|0)^*1 \]

- The corresponding NFA is

NFA to DFA: The Trick

- Simulate the NFA, *in parallel*
  - Michael Rabin and Dana Scott's work
- Each state of DFA
  - a non-empty subset of states of the NFA
- Start state
  - the set of NFA states reachable through $\varepsilon$-moves from NFA start state
- Add a transition $S \rightarrow S'$ to DFA *iff*
  - $S'$ is the set of NFA states reachable from some state in $S$ after seeing the input 'a', considering $\varepsilon$-moves as well
**NFA to DFA: Remark**

- 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 subsets are there?
  - $2^N - 1$ = finitely many
- NFA $\rightarrow$ DFA conversion is at the heart of tools such as flex. In practice, flex-like tools trade off speed for space in the choice of NFA and DFA representations. *(DFA Minimization)*
  - Myhill-Nerode's work

**NFA $\rightarrow$ DFA Example**

**Implementation**

- A DFA can be implemented by a 2D table $T$
  - One dimension is “states”
  - Other dimension is “input symbol”
  - For every transition $S_i \rightarrow S_k$ 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

**Table Implementation of a DFA**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>T</td>
<td>U</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>T</td>
<td>U</td>
</tr>
</tbody>
</table>
Finite State Machine Minimization

Language over \( \{H, T\} \) : Strings with even number of \( H \)

Minimizing DFA: The Idea

- Two states \( s \) and \( t \) should be merged unless there is some way to tell them apart.
  - Initially, assume that all states are equivalent until proven otherwise.
- How can we tell if two states \( s \) and \( t \) are different?
  - If one is accepting and the other is not.
  - If, on input \( c \), \( s \rightarrow x \) and \( t \rightarrow y \), and we already know \( x \) is different from \( y \).

Minimizing DFA: The Algorithm

- Overall, it partitions the set of states.
- Initial partition:
  - \( \emptyset \) (Accepting States, Non-accepting states)
- Refinement:
  - In each pass, find a new partition for each state such that
    - \( s \) and \( t \) are in the same new partition if and only if states \( s \) and \( t \) were in the same old partition, and, on each input \( c \), states \( s \) and \( t \) go to states in the same partition.

Example DFA
Refinement of State Partitions

- \{ \{ q_0, q_7 \}, \{ q_1, q_2, q_3, q_4, q_5, q_6 \} \}
- \{ \{ q_0 \}, \{ q_7 \}, \{ q_1, q_2, q_3, q_4, q_5, q_6 \} \}
  - On any transition
- \{ \{ q_0 \}, \{ q_7 \}, \{ q_1, q_2, q_3, q_4, q_5, q_6 \} \}
- \{ \{ q_0 \}, \{ q_7 \}, \{ q_1, q_4 \}, \{ q_2, q_3, q_5, q_6 \} \}
  - On "a" transition
- \{ \{ q_0 \}, \{ q_7 \}, \{ q_1, q_4 \}, \{ q_2, q_5 \}, \{ q_3, q_6 \} \}
  - On "b" transition

Example DFA showing equivalent states

Example Minimum DFA

Summary

- Lexer creates tokens out of a text stream.
- Tokens are defined using regular expressions.
- Regular expressions can be mapped to Non-deterministic Finite Automata (NFA)
- NFA is transformed to a DFA
  - By removing non-determinism, and
  - By minimizing states.
- Executing a DFA is straightforward.
- Common scanner generator tools
  - lex, flex in C
  - jflex in Java
What’s Next?

- Program (character stream)
- Lexical Analyzer (Scanner)
- Syntax Analyzer (Parser)
- Token Stream
- Parse Tree
- Intermediate Code Generator
- Intermediate Representation
- Intermediate Code Optimizer
- Optimized Intermediate Representation
- Code Generator
- Assembly code

Animating Lexical Analysis

Lexical Analyzer in Action

for ID("var1") eq_op Num(10) ID("var1") leq_op
Lexical Analyzer in Action

- Partition input program text into subsequence of characters corresponding to tokens
- Attach the corresponding attributes to the tokens
- Eliminate white space and comments

Animating NFA construction

(-|ε) (0|1|2|3|4|5|6|7|8|9)+ (.·(0|1|2|3|4|5|6|7|8|9)*)?

(-|ε)

Animating Token Recognition

String Matching

17
1. Closure

- The closure of a state is the set of states that can be reached from that state without consuming any of the input.
  - Closure(S) is the smallest set T such that
    \[ T = S \cup \bigcup_{s \in S} \text{edge}(s, \epsilon) \]

- Algorithm

  \[
  \begin{align*}
  T & \leftarrow S \\
  \text{repeat} \\
  T' & \leftarrow T \\
  T & \leftarrow T \cup \bigcup_{s \in S} \text{edge}(s, \epsilon) \\
  \text{until} \\
  T & = T'
  \end{align*}
  \]
\( S = \{1\} \)
\( T = \{1\} \)
\( T' = \{\} \)

1. \( T \leftarrow S \)
2. \( \text{repeat} \)
   \( T' \leftarrow T \)
   \( T \leftarrow T' \cup \bigcup_{s \in S} \text{edge}(s, s) \)
3. \( \text{until} \quad T = T' \)

\( S = \{1\} \)
\( T = \{1, 2\} \)
\( T' = \{1\} \)

1. \( T \leftarrow S \)
2. \( \text{repeat} \)
   \( T' \leftarrow T \)
   \( T \leftarrow T' \cup \bigcup_{s \in S} \text{edge}(s, s) \)
3. \( \text{until} \quad T = T' \)
\[ S = \{1\} \]
\[ T = \{1, 2\} \]
\[ T' = \{1, 2\} \]

\[ T \leftarrow S \]
\[ \text{repeat} \]
\[ T' \leftarrow T \]
\[ T \leftarrow T' \bigcup \bigcup_{s, (s, e)} \]
\[ \text{until} \quad T = T' \]
Question: What is closure(3)?

\[ S = \{3\} \]

\[ T = ??? \]

Question: What is closure(3)?

\[ S = \{3\} \]

\[ T = \{2, 3, 4, 8\} \]

2. DFAedge

- Given a symbol and a state, what states can you reach?
What is DFAedge({1}, 3)?

d = {1}

closure({1}) = {1, 2}

DFAedge({1}, 3) = {}

What is DFAedge({1}, 3)?

d = {1}

closure({1}) = {1, 2}

DFAedge({1}, 3) = {}

What is DFAedge({1}, 3)?

d = {1, 2}

DFAedge({1}, 3) = {3}

What is DFAedge({1}, 3)?

d = {1, 2}

closure({1}) = {1, 2}

DFAedge({1}, 3) = {3}
What is $\text{DFAedge}({1}, 3)$?

$d = \{1, 2\}$
$closure(3) = \{2, 3, 4, 8\}$

$\text{DFAedge}({1}, 3) = \{2, 3, 4, 8\}$

Question: What is $\text{DFAedge}({3}, .)$?

$d = \{3\}$

$\text{DFAedge}({3}, .) = \{5, 6, 8\}$
closure(1) =\{1, 2\}

DFAedge(\{1,2\}, \rightarrow) =\{2\}

Closure(\{2\}) =\{2\}

DFAedge(\{1,2\}, 0..9) =\{3\}
Closure({3}) = {2, 3, 4, 8}

DFAEdge({2, 3, 4, 8}, 0..9) = {3}
Closure({3}) = {2, 3, 4, 8}

DFAEdge({2, 3, 4, 8}, .) = {5}
Closure({5}) = {5, 6, 8}
What is the minimal DFA for the following?