Implementation of Lexical Analysis

Outline

• Specifying lexical structure using regular expressions

• Finite automata
  - Deterministic Finite Automata (DFAs)
  - Non-deterministic Finite Automata (NFAs)

• Implementation of regular expressions
  \[ \text{RegExp} \Rightarrow \text{NFA} \Rightarrow \text{DFA} \Rightarrow \text{Tables} \]

Notation

• For convenience, we use a variation (allow user-defined abbreviations) in regular expression notation

  • Union: \[ A + B \equiv A \mid B \]
  • Option: \[ A + \varepsilon \equiv A? \]
  • Range: \[ 'a'+'b'+\ldots+'z' \equiv [a-z] \]
  • Excluded range:
    \[ \text{complement of } [a-z] \equiv [^a-z] \]

Regular Expressions in Lexical Specification

• 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
  • We will adapt regular expressions to this goal
Regular Expressions \Rightarrow \text{Lexical Spec. (1)}

1. Select a set of tokens
   - Integer, Keyword, Identifier, OpenPar, ...

2. Write a regular expression (pattern) for the lexemes of each token
   - Integer = digit +
   - Keyword = 'if' + 'else' + ...
   - Identifier = letter (letter + digit)*
   - OpenPar = '('
   - ...

Regular Expressions \Rightarrow \text{Lexical Spec. (2)}

3. Construct \( R \), matching all lexemes for all tokens
   \[
   R = \text{Keyword} + \text{Identifier} + \text{Integer} + ...
   \]
   \[= R_1 + R_2 + R_3 + ...
   \]

Facts: If \( s \in L(R) \) then \( s \) is a lexeme
- Furthermore \( s \in L(R_i) \) for some “i”
- This “i” determines the token that is reported

Regular Expressions \Rightarrow \text{Lexical Spec. (3)}

4. Let input be \( x_1...x_n \)
   - \( (x_1 ... x_n \text{ are characters}) \)
   - For \( 1 \leq i \leq n \) check
     \[ x_1...x_i \in L(R) ? \]

5. It must be that
   \[ x_1...x_i \in L(R_j) \text{ for some } j \]
   (if there is a choice, pick a smallest such \( j \))

6. Remove \( x_1...x_i \) from input and go to previous step

Regular Expressions \Rightarrow \text{Lexical Spec. (2)}

3. Construct \( R \), matching all lexemes for all tokens
   \[
   R = \text{Keyword} + \text{Identifier} + \text{Integer} + ...
   \]
   \[= R_1 + R_2 + R_3 + ...
   \]

Facts: If \( s \in L(R) \) then \( s \) is a lexeme
- Furthermore \( s \in L(R_i) \) for some “i”
- This “i” determines the token that is reported

How to Handle Spaces and Comments?

1. We could create a token \text{Whitespace}
   \[
   \text{Whitespace} = (\ ' + \text{"\n} + \text{"\t} )^+
   \]
   - We could also add comments in there
   - An input \text{" \t\n 5555 \"} is transformed into \text{Whitespace Integer Whitespace}

2. Lexer skips spaces (preferred)
   - Modify step 5 from before as follows:
     It must be that \( x_k ... x_i \in L(R_j) \) for some \( j \) such that \( x_1 ... x_{k-1} \in L(\text{Whitespace}) \)
   - Parser is not bothered with spaces
Ambiguities (1)

- There are ambiguities in the algorithm

- How much input is used? What if
  - $x_1...x_i \in L(R)$ and also
  - $x_1...x_K \in L(R)$

  - Rule: Pick the longest possible substring
  - The "maximal munch"

Ambiguities (2)

- Which token is used? What if
  - $x_1...x_i \in L(R_j)$ and also
  - $x_1...x_i \in L(R_k)$

  - Rule: use rule listed first ($j$ if $j < k$)

  - Example:
    - $R_1 = \text{Keyword}$ and $R_2 = \text{Identifier}$
    - "if" matches both
    - Treats "if" as a keyword not an identifier

Error Handling

- What if
  - No rule matches a prefix of input?

- Problem: Can't just get stuck ...

- Solution:
  - Write a rule matching all "bad" strings
  - Put it last

  - Lexer tools allow the writing of:
    - $R = R_1 + ... + R_n + \text{Error}$
      - Token Error matches if nothing else matches

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 (next)
  - Require only single pass over the input
  - Few operations per character (table lookup)
Regular Languages & Finite Automata

Basic formal language theory result:

Regular expressions and finite automata both define the class of regular languages.

Thus, we are going to use:

• Regular expressions for specification
• Finite automata for implementation
  (automatic generation of lexical analyzers)

Finite Automata

A finite automaton is a recognizer for the strings of a regular language.

A finite automaton consists of
- A finite input alphabet $\Sigma$
- A set of states $S$
- A start state $n$
- A set of accepting states $F \subseteq S$
- A set of transitions $\text{state} \rightarrow \text{input state}$

Finite Automata State Graphs

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

Finite Automata

• Transition
  $s_1 \rightarrow^a s_2$

• Is read
  In state $s_1$ on input “a” go to state $s_2$

• If end of input (or no transition possible)
  - If in accepting state $\Rightarrow$ accept
  - Otherwise $\Rightarrow$ reject
A Simple Example
• A finite automaton that accepts only “1”

Another Simple Example
• A finite automaton accepting any number of 1’s followed by a single 0
• Alphabet: {0,1}

And Another Example
• Alphabet {0,1}
• What language does this recognize?

And Another Example
• Alphabet still { 0, 1 }
• The operation of the automaton is not completely defined by the input
  - On input “11” the automaton could be in either state
Epsilon Moves

- Another kind of transition: $\varepsilon$-moves

- Machine can move from state A to state B without reading input

Deterministic and Non-Deterministic Automata

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

- Non-deterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have $\varepsilon$-moves

- Finite automata have finite memory
  - Enough to only encode the current state

Execution of Finite Automata

- A DFA can take only one path through the state graph
  - Completely determined by input

- NFAs can choose
  - Whether to make $\varepsilon$-moves
  - Which of multiple transitions for a single input to take

Acceptance of NFAs

- An NFA can get into multiple states

- Input: 1 0 1

- Rule: NFA accepts an input if it can get in a final state
NFA vs. DFA (1)

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

- DFAs are easier to implement
  - There are no choices to consider

NFA vs. DFA (2)

- For a given language the NFA can be simpler than the DFA

  ![NFA Diagram](image1)

  ![DFA Diagram](image2)

- DFA can be exponentially larger than NFA

Regular Expressions to Finite Automata

- High-level sketch

  ![Diagram](image3)

Regular Expressions to NFA (1)

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

  ![Diagram](image4)

- For ε

  ![Diagram](image5)

- For input a

  ![Diagram](image6)
Regular Expressions to NFA (2)

• For $AB$

\[
\begin{array}{c}
A \\
\epsilon \\
B
\end{array}
\]

• For $A + B$

\[
\begin{array}{c}
A \\
\epsilon \\
B \\
\epsilon \\
A \\
\epsilon
\end{array}
\]

Regular Expressions to NFA (3)

• For $A^*$

\[
\begin{array}{c}
A \\
\epsilon
\end{array}
\]

Example of Regular Expression $\rightarrow$ NFA conversion

• Consider the regular expression $(1+0)^*1$

• The NFA is

\[
\begin{array}{c}
A \\
\epsilon \\
B \\
\epsilon \\
C \\
1 \\
D \\
0 \\
E \\
\epsilon \\
F \\
\epsilon \\
G \\
\epsilon \\
H \\
\epsilon \\
I \\
1 \\
J
\end{array}
\]

NFA to DFA. The Trick

• Simulate the NFA
• Each state of DFA
  - a non-empty subset of states of the NFA
• Start state
  - the set of NFA states reachable through $\epsilon$-moves from NFA start state
• Add a transition $S \rightarrow^a S'$ to DFA iff
  - $S'$ is the set of NFA states reachable from any state in $S$ after seeing the input $a$
    - considering $\epsilon$-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

**Implementation**

- A DFA can be implemented by a 2D table T
  - One dimension is "states"
  - Other dimension is "input symbols"
  - For every transition \(S_i \rightarrow a S_k\) 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
Implementation (Cont.)

- NFA → DFA conversion is at the heart of tools such as lex, ML-Lex or flex
- But, DFAs can be huge
- In practice, lex/ML-Lex/flex-like tools trade off speed for space in the choice of NFA and DFA representations

Theory vs. Practice

Two differences:

- DFAs recognize lexemes. A lexer must return a type of acceptance (token type) rather than simply an accept/reject indication.
- DFAs consume the complete string and accept or reject it. A lexer must find the end of the lexeme in the input stream and then find the next one, etc.