Announcements

- WA1 and PA0 are due Today
- WA2 and PA1 out today :-)  
- If you are not very, very busy right now, get started now

Outline

- Last time: Specifying lexical structure using regular expressions
- Today: How to recognize strings matching regular expressions using finite automata.
- We will see deterministic finite automata (DFAs) and non-deterministic finite automata (NFAs)
- High-level story: RegEx -> NFA -> DFA -> Tables

Regular Expressions in Lexical Specifications

- Last lecture: How to specify the predicate $s \in L(R)$
- But yes/no answer is not enough!
- We really want to partition input into tokens
- We adapt regular expressions for this goal

Regular Expressions to Lexical Specifications (1)

- Step 1: Write a regular expression for the lexemes of each token
  - Integer constant: $\text{digit}^+$
  - Identifier: $\text{letter} (\text{letter} + \text{digit})^*$
  - Lambda: ‘lambda’
  - ...

Regular Expressions to Lexical Specifications (2)

- Step 2: Construct $R$, matching lexemes for all tokens
  - $R = \text{Integer constant} + \text{Identifier} + \text{Lambda} + ...$
Regular Expressions to Lexical Specifications (3)

- Let the input be characters $x_1 \ldots x_n$
- **Step 3:** For each $1 \leq i \leq n$ check $x_1 \ldots x_i \in L(R)$ for some $j$
- Then, remove $x_1 \ldots x_j$ from input and repeat

Ambiguities I

- There are ambiguities in this algorithm. Where?
- How much input is used? What if $x_1 \ldots x_i \in L(R)$ and $x_1 \ldots x_j \in L(R)$?
- **Example:** identifier = letter (letter + digit)*, if = 'i' 'f'
- **Rule:** Pick longest possible string in $L(R)$
- This is known as "maximal munch"

Ambiguities II

- What if two rules match with the same number of characters?
  - $x_1 \ldots x_i \in L(R_1)$ and $x_1 \ldots x_i \in L(R_2)$?
- **Example:** "if"
- **Rule:** Use rule listed first
- This is how "if" is matched as a keyword, not identifier

Error Handling

- What if no rule matches a prefix of the input?
- **Solution 1:** Get stuck ⇒ Unacceptable
- **Better Solution:** Write a rule matching all "bad" strings
- **Question:** What kind of rule and where to place it?

Where are we?

- We now know how we can partition input string into tokens assuming we can decide if a string is in the language described by a regular expression.
- **Next:** How to decide if $s \in L(R)$

Finite Automata

- Regular Expressions ⇔ Specification
- Finite Automata ⇔ Implementation
- A finite automata formally consists of:
  - An input alphabet $\Sigma$
  - A set of states $S$
  - A start state $n$
  - A set of accepting states $F \subseteq S$
  - A set of transitions state $\rightarrow$ input state
Finite Automata

- Transition $S_1 \rightarrow^\alpha S_2$
- This means: In state $S_1$ and input character $\alpha$, go to state $S_2$
- If end of input and in accepting state $\Rightarrow$ accept
- Otherwise $\Rightarrow$ reject

Finite Automata as State Graphs

- It is much easier to imagine finite automata visually:
  - A state:
  - The start state:
  - An accepting state:
  - A transition:

A Simple Example

- Here is an automaton that only accepts the string "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 automata recognize?

Epsilon Moves

- Another kind of transition: $\varepsilon$-moves
  - Machine can move from state $A$ to $B$ without reading any input
Deterministic and Nondeterministic Automata

- Deterministic Finite Automata (DFA)
  - At most one transition per input on any state
  - No $\varepsilon$ moves
- Nondeterministic Finite Automate (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have $\varepsilon$-moves

Execution of Finite Automata

- A DFA can only take one path through the state graph that is completely determined by the input
- NFAs can choose:
  - Whether to make $\varepsilon$ moves
  - Which one of multiple transitions for a single input to take

Acceptance of NFAs

- This means: A NFA can get into multiple states at the same time
- Consider again the alphabet $\Sigma = \{0, 1\}$ and the language of all strings ending in at least two 0s.
- Consider input $1 \ 0 \ 0$

- Rule: NFA accepts if it can get to a final state

NFAs vs. DFAs

- Fundamental Result: NFAs and DFAs recognize the same set of languages (regular languages)
- DFAs are faster to execute, since there are no choices to consider
- But NFAs can be much simpler for the same language
- Result: DFAs can be exponentially larger than NFA recognizing same language

Regular Expressions to Finite Automata

- High-Level Sketch:
  - Lexical Specification
  - Regular Expressions
  - NFA
  - DFA
  - Implementation of DFA
- $\Rightarrow$ Lexer

Regular Expressions to NFA (1)

- For each kind of regular expression, define an NFA and combine
- Will use the following notation: NFA for regular expression $M$:

  $\varepsilon$

- Base cases:
  - For $\varepsilon$:
  - For input $a$:
Regular Expressions to NFA (2)

- For $AB$:

- For $A + B$:

Example of Regular Expression to NFA conversion

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

NFA to DFA: The Trick

- Insight: Simulate the NFA
- At any given time, the NFA is in a set of states
- States in the DFA ⇒ all (reachable) subsets of states in the NFA
- Start State: the set of states reachable through $\varepsilon$ moves from the NFA start state
- Add transition $A \rightarrow a B$ to DFA iff:
  - $B$ is in the set of states reachable from any state in $A$ after seeing input $a$, considering $\varepsilon$ moves as well

NFA to DFA: Example

Recall our friendly NFA for $(1 + 0)^*1$:

NFA to DFA: How many states?

- We need a state in the DFA for each set of states the NFA can be in
- How many different states?
- If there are $N$ states, the NFA must be in some subset of those $N$ states
- How many subsets of $N$ states? $2^N$
A DFA can be implemented by a 2D table $T$
- One dimension is "states"
- Other dimension is "input symbols"
- For every transition $A \rightarrow c B$, define $T[A,c] = B$

DFA "execution": If in state $A$ and input $c$, read $T[A,c] = B$ and skip to state $B$

Very efficient

Writing regular expressions as NFAs and converting them to DFAs is exactly what flex does

In fact, if you open the auto-generated flex file `lex.yy.c`, you will see these tables emitted

But, these DFAs can be huge

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