Lexical Analysis

CS 4447 / CS 9545 -- Stephen M. Watt
University of Western Ontario

Reading

- Aho, Lam, Sethi, Ullman: Chapter 3

Lex, JFlex

- We will soon see the scanner generating tools Lex and JFlex
- How do they work?

Finite Automata Theory Hinted at by JFlex

- If you look at the output of JFlex, you see

```
obelix[16]%  jflex CScanner.lex
Reading "CScanner.lex"
Constructing NFA : 461 states in NFA
Converting NFA to DFA :
.....................................................
.....................................................
.....................................................
.....................................................
237 states before minimization, 212 states in minimized DFA
Writing code to "CScanner.java"
obelix[17]%
```

- What does this mean?

Regular Expressions, RegEx(Σ)

- Regular expressions on alphabet Σ = {a,b,c,…}
- Empty expression, ε
- Element of Σ, a
- Concatenation, αβ α, β ∈ RegEx(Σ)
- Union, α|β
- Kleene Closure, α*
- Parentheses, (α)

Extra Regular Expression Syntax

- These are just shorthands and can be defined in terms of the previous:

  α+ α? α{n} α{n-m} α{n-}

  ! α (*not* α)

  ~ α (*upto* α)

- Context is useful in making scanners:

  ^ $ <<EOF>>

  α/β (α, *but only if followed by* β)

  Start conditions

One Way of Looking at Regular Languages

- Input alphabet Σ, variable set V
- Set of regular definitions
  d[1] → r[1]
  d[2] → r[2]
  …
  d[n] → r[n]
- Each d[i] is a distinct element of V
- Each r[i] is a regular expression on
  Σ U { d[1], d[2], …, d[i-1]}

Building an NFA from a Regular Language

- $a \in \Sigma$ becomes a two state NFA.
  "a" gives a transition from the initial state to an accepting final state.

- αβ becomes NFA for α, followed by an ε rule to the NFA for β. The final states of the NFA for α are no longer final, and have ε rules to the initial state of the NFA for β.

- α|β becomes a fork of ε rules to the NFA α and the NFA for β.

- α* becomes the NFA for α with an ε rule back to the beginning. The initial state of the NFA becomes an accepting state, and all the final states have an ε rule back to it.

Building an NFA for a Lex Specification

- Treat the Lex specification as a big union:

  Rule1 | Rule2 | Rule3 | … | RuleN

Simulating an NFA (1)

- **`ε-closure(S)`**, the set of states that can be reached from the set of states S with ε transitions.
- **`move(S,a)`**, the set of states that can be reached from the set S with input “a”

Simulating an NFA (2)

```
S := ε-closure({s0})
a := nextchar();
while (a ≠ EOF) {
  S := ε-closure(move(S,a));
  a := nextchar();
}
return S ∩ F ≠ { }
```

NFA vs DFA: Space-Time Tradeoffs

- Regular expression *r*, size |*r*|
- Input string *x*, length |*x*|

- NFA Space $O(|r|)$, Time $O(|r| * |x|)$
- DFA Space $O(2^{|r|})$, Time $O(|x|)$

NFA to a DFA: the Subset Construction

- Given N, an NFA, construct D, a DFA accepting the same language.
- The states of D will correspond to *sets of states* of N.

The Subset Construction

```
DStates := { ε-closure({s0}) };       // unmarked
while there is unmarked T in DStates {
   mark T;
   for each input symbol a in Σ {
      U := ε-closure(move(T,a));
      if U not in DStates {
         DStates := DStates union {U};// unmarked
      }
      DTransitions[T, a] := U;
   }
}
```

DFA Minimization

- Each regular language is recognized by a minimal DFA, unique up to state names.

- Use the idea of a string *w* distinguishing two states *s* and *t* :

  Starting in *s* and reading *w* ends in accepting state, while starting in *t* and reading *w* ends in non-accepting state, or vice versa.

Basic Idea

- Work with partitions of states
- Start with coarse partition: accepting states, and non-accepting states.
- Split a partition if there is an input symbol that gives transitions to different partitions.

- E.g. The partition P1 is split because input “b” gives transitions from P1 →P1 and P1 →P2

  P1 = {S1, S2}, P2 = {S3,S4}

  (S1,a) → S1, (S2,a) → S1

  (S1,b) → S1, (S2,b) → S3

DFA Minimization

1. Construct initial partition Π of two groups: accepting states F, and non-accepting S-F
2. Form Πnew by splitting each group where there is an input symbol that gives transitions to distinct groups
3. If Πnew = Π, let Πfinal = Π and goto step 4. Otherwise repeat step 2 with Π := Πnew
4. Choose one state in each group of Πfinal as its representative.
   These are the states of the minimal DFA.
   Form transitions based on transitions between groups.
   The new start state is the representative of the group containing the old start state.
   The new final states are the representatives of the groups containing old final states.
5. Remove dead states (non-accepting that have no transitions to other states), and unreachable states.