CS 434 Meeting 13 — 3/16/06

Anouncements

1. Homework assignment 2 is due.
2. Phase 2.1 due.

YACC - a final word

1. Last time, we looked at many aspects of YACC and Lex, but we skipped on important detail — the y.output file.
   - The y.output file contains
     - a summary of the productions of the grammar, and
     - a description of the pushdown automaton used to perform parsing.
   - The descriptions of the states in the y.output file look like:

```
state 1
        date : MONTH . day ',' year  (3)

        NUMBER  shift 6
        .  error

        day  goto 7
```

   - Each such state description starts with a list of productions annotated with embedded periods. These productions indicate what rules it believes may become candidates for applying to reduce the handle in a future step.

     The period indicates the parser's progress toward recognizing an instance of each production.
     - Everything to the left of the period must appear on the top of the parser's stack when it reaches this state.
     - It is still looking for everything after the period.
   - After the list of productions there is a list that indicates how the parser should behave based on the next input symbol.
     - It either says "shift" (meaning shift the token onto the stack) and includes the state the parser should move into, or
     - It says reduce (and specifies what rule to use to match the handle).
     - A period is used to indicate what to do for input tokens not otherwised mentioned
   - Finally, there is a description of how to behave if a non-terminal is shifted onto the stack after reducing a suffix of the stack.
2. Not all grammars can be used to build deterministic bottom-up parsers.
   - If Yacc cannot build a deterministic parser for a grammar, it will concisely report the error in the terminal window and provide more complete details in the y.output file.
   - There are two problems that may be reported:
     - A reduce/reduce conflict means that in some state and with certain next token values, Yacc has realized it should reduce but it does not know how to decide which production to use when reducing.
     - A shift/reduce conflict means that in some state Yacc has realized it may not be able to decide whether to shift or to reduce.

How Lex Works

1. Let's try to use all the good things you learned about regular languages and finite state machines to try to figure out how a program like Lex works. (We will save figuring Yacc out for after break).
2. Lex starts with a collection of regular expressions.
   - With a bit of luck, you remember that the set of languages described by regular expressions is identical to the set of languages that can be recognized by finite state automata (and the set described by regular grammars).

- With this in mind, a reasonable strategy for building a program that recognizes strings that match regular expressions is to build the FSA that corresponds to the regular expressions in a Lex input file.

3. First, recognize that there are a number of things we can do to reduce the apparent complexity of the language of regular expressions before we even begin.
   - For example, Lex allows expressions of the form α^+. Any such expression can be rewritten as $\alpha(\alpha^*)$. Accordingly, if we figure out how to handle concatenation and *, we don't have to worry about +. We can just rewrite any expression that uses + before we start trying to build an FSA.
   - Through similar tricks we can get the set of regular expressions we have to worry about down to single symbols from our alphabet, concatenation, alternation (|), and the Kleene star (*).
4. Next, recall the structure of a deterministic finite state machine.
   (a) A finite set of states, π.
   (b) An input alphabet, Σ.
   (c) A transition function $\delta : \pi \text{ x } \Sigma \rightarrow \pi$.
   (d) A subset F of π called the set of final states.
   (e) An element π_0 of π called the initial state.
5. Although what I have shown above is the standard way to define a FSA, it is worth noting that there is one oddly unmathematical aspect of this definition. The transition function is undefined on many elements in its domain. When it is helpful, we can eliminate this oddity by adding an error state with the idea that in any case where δ would have been undefined we will define its result to be the error state.
6. While you are at it, recall (or at least note) that we can explain the behavior of a deterministic finite state machine by defining a function that extends δ to strings over the input alphabet. In particular, we can define $\Delta : \pi \text{ x } \Sigma^* \rightarrow \pi$ recursively as
   - $\Delta(\pi, \epsilon) = \pi$
   - $\Delta(\pi, \gamma x) = \delta(\Delta(\pi, \gamma), x)$

   and then state that the language accepted by the machine is

$$\{\gamma \in \Sigma^* \mid \Delta(\pi_0, \gamma) \in F\}$$

7. Building a FSA to match a single symbol x is quite easy.
   - It will have two states (call them 0 and 1).
   - The initial state will be 0 and 1 will be the only final state.
   - The transition function will be defined as $\delta(0, x) = 1$.
8. Building an FSA to match an RE of the form $\alpha\beta$ (i.e. concatenation) is a bit trickier.
   - We would like to do the job recursively. That is, start by building FSAs for α and β independently.
   - Two issues complicate connecting the two machines.
     - The machine for α may have many final states.
     - It may not be clear when we should take a transition from one of these final states to the initial state for β.
       * Suppose that α is a^* and β is $aaab^*$.
9. We can simplify this issue by giving up one (very key) property of the automaton we have been building, its deterministic behavior. That is, if we allow ourselves to build a non-deterministic finite state machine instead of deterministic one, the construction becomes much easier.
   - A non-deterministic FSA may include ϵ-transitions.
   - We can connect all the final states in the machine for α with ϵ-transitions to the initial state in the machine for β.
10. So, now consider how to define a non-deterministic finite state machine:
    (a) A finite set of states, π.
    (b) An input alphabet, Σ.

(c) A transition function $\delta : \pi$ x $(\Sigma + \epsilon) \rightarrow 2^\pi$.

(d) A subset F of π called the set of final states.

(e) An element π_0 of π called the initial state.

11. As we did for FSA, we would like to define an extension of δ to a function Δ that handles transitions on strings rather than single symbols from Σ.

    - Unlike the Δ for an FSA, the Δ for a NFSA must accept a set of states and a string as its input:

    $$\Delta : 2^\pi \text{ x } (\Sigma + \epsilon)^* \rightarrow 2^\pi$$

    - The first step in defining Δ is to take care of the ϵ productions by defining the *closure* of a set of states:

        We define the closure of a set of states, *closure(P)* to be the smallest set P' such that:

        - $P' \supseteq P$.
        - if $\pi_i \in P'$ and $\pi_j \in \delta(\pi_i, \epsilon)$ then $\pi_j \in P'$.

    - Given this definition, we can say
        - $\Delta(P, \epsilon) = \text{closure}(P)$
        - $\Delta(P, \gamma x) = \text{closure}(\delta(\Delta(P, \gamma), x))$.
    - Given this definition, we can define the language defined by a NFSA to be the set of strings for which the set of state obtained by applying Δ to the initial state of the machine and the input produce a set of states that includes at least one of the machine's final states.

12. Working with NFSAs also makes it easy to construct a machine for a RE of the form α^*.

    - We add a state that is both an initial and final state.
    - We add epsilon transitions from this state to the initial state for α.
    - We add epsilon transitions from the final states for α's machine to this new initial state.

13. Now we can build a complete NFSA for each RE provided in a Lex input file. This leaves us with two problems.

    - We really need a deterministic machine
    - We really want one BIG machine rather than a separate machine for each RE.

14. To be continued.... ?????