CS143
Compilers

Lecture 2

July 2, 2001

Recognition of Tokens

- Now that we know how to specify tokens, we can figure out how to recognize them.
- The basic idea is as follows:
  - First, we construct a regular expression for each token.
  - Then, we scan through the input, one character at a time, until the input is recognized as a string that matches one of the patterns described by the regular expressions.
  - Return the corresponding token, and repeat this process for the remaining input.

Transition Diagrams

- In attempting to recognize a token, we could try to match the input to each regular expression in turn until we are successful.
- This is woefully inefficient, as this forces us to pass through the input several times.
- We can instead use a *transition diagram* to effectively combine the regular expressions into a single regular expression that can recognize any valid token.
- A well-designed transition diagram minimizes the need to backtrack through the input.

Anatomy of Transition Diagrams

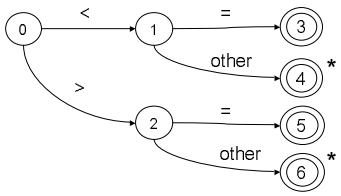
- A transition diagram consists of a set of *states* and a set of *transitions*.
- Each diagram contains a single state that is called the *start state*. There can be no transitions to the start state.
- Each diagram contains at least one state that is called a *final* or *accepting state*. There can be no transitions from an accepting state.

Using Transition Diagrams

- Transitions between states are guided by input symbols from a given alphabet.
- Initially, the scanner is in the start state. It then reads the first input symbol, and determines which transition to use.
- Input is read and transitions followed until an accepting state is reached. This signifies that a token has been recognized, and the string of input symbols read since the start state is the corresponding lexeme.
- Control returns to the start state to read the next token.

The following transition diagram can be used to recognize the relational operators <, <=, > and >=.

State 0 is the start state, and double circles denotes accepting states. The * next to a state indicates that the scanner should move back by one input symbol.



Implementing a Transition Diagram

- Once a transition diagram has been constructed, it is easy to write a program to implement it.
- Each state corresponds to a segment of code.
- If there are edges leaving the state, then this code must read the next input character and cause a branch to the code for the appropriate state.
- If a state is a final state, a token has been recognized.
- If a state is not a final state, and no transition to another state is available, then a lexical error has been detected.

Ad-hoc Scanners

- A lexical analyzer implemented from a transition diagram is called an *ad-hoc scanner*. This terminology stems from the manner in which the transition diagram is built from patterns for each token.
- It would be desirable to have a systematic method of creating this transition diagram, given a set of patterns that specify the lexemes for the tokens of the source language.
- Such a method would allow lexical analyzers to be created automatically.

Recognizers

- A *recognizer* for a language is a program that can determine whether a given input string belongs to the language.
- Our goal is to be able to systematically build a recognizer from any given regular expression.
- We will then show how such a recognizer can be used to create a lexical analyzer for a source language, provided that each token can be described using a regular expression.
- We will see that given regular expressions for each token, the recognizer can be generated automatically.

Finite Automata

- To convert a regular expression into a recognizer, we can construct a generalized transition diagram called a *finite automaton*.
- A *deterministic finite automaton* (DFA) is a transition diagram in which only one transition may be made from a state on a given input symbol.
- A *nondeterministic finite automaton* (NFA) allows more than one transition from a state on any given input symbol.
- Generally, using DFA's yields faster recognizers, but they are more difficult to design and are more complex.

Nondeterministic Finite Automata

- A nondeterministic finite automaton is a mathematical model consisting of:
  - A set of *states S*
  - A set of input symbols Σ
  - A transition function *move* that maps state-symbol pairs to sets of states
  - A state s_0 that is the *start* or *initial state*
  - A set of states *F* called *accepting* or *final states*

Transition Graphs

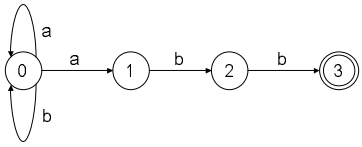
- An NFA can easily be represented by its *transition graph*. The transition graph is a generalized transition diagram that is built from the NFA as follows:
  - The states in the transition graph correspond to the states of the NFA, including the designation of the start state and accepting states.
  - The transitions in the transition graph represent the NFA's transition function, *move*. If *move* maps a state s_1 and input symbol *a* to state s_2, then the transition graph includes a transition from state s_1 to state s_2 on symbol *a*.

Acceptance of Strings

- An NFA *accepts* a string *x* if there exists a path in the NFA's transition graph from the start state to an accepting state, such that the concatenation of the input symbols along this path yields the string *x*.
- The *language defined by an NFA* is the set of input strings that it accepts.
- If the NFA is built from a regular expression, then the language that it defines is the language denoted by that regular expression.

Example

This NFA recognizes the language denoted by the regular expression (a|b)*abb.

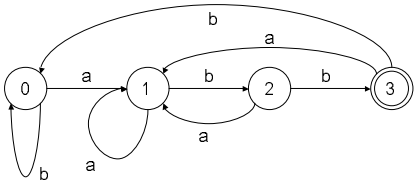


Deterministic Finite Automata

- A deterministic finite automaton is an NFA with the following restrictions:
  - No state has a transition on the empty string ε (known as an ε–transition)
  - For each state s and input symbol a, there is at most one transition from s on a.
- As a result of these restrictions, it is very easy to determine whether a DFA accepts a given string, i.e. whether a string matches the regular expression corresponding to the DFA.

Example

This DFA also recognizes the language denoted by the regular expression (a|b)*abb.



From a Regular Expression to an NFA

Our primary goal, building an efficient recognizer for a regular expression, can be accomplished by building an NFA that accepts exactly those strings in the language denoted by the regular expression, and then using one of two approaches:

- converting the NFA to an equivalent DFA, which can easily be simulated
- simulating the NFA directly.

We will examine the trade-offs involved in both approaches.

Construction of an NFA from a Regular Expression

An NFA can be built from a regular expression over an alphabet Σ by using the expression's syntactic structure as a guide for the construction:

- First, we show how to build an NFA that accepts the empty string ε, and each of the input symbols in Σ.
- We then use induction to handle all other regular expressions over Σ. Given NFA built from two regular expressions *r* and *s*, we show how to build NFA representing *r*|*s*, *rs*, and *r**.

Thompson's Construction

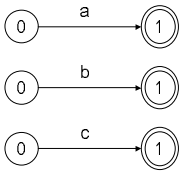
- Let r and s be regular expressions over Σ. For any regular expression r over S, let $N(r)$ be an NFA that accepts the strings in $L(r)$.
  - If r is the empty string ε, then $N(r)$ consists of an initial state i, a final state f, and a single transition from i to f on ε.
  - If r is a symbol a from Σ, then $N(r)$ contains the states i and f, and a transition from i to f on a.
  - $N(r|s)$ contains the states i and f, along with ε-transitions from i to the initial states of $N(r)$ and $N(s)$, and ε-transitions from the final states of $N(r)$ and $N(s)$ to f.

Thompson's Construction, cont'd

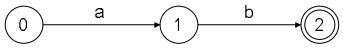
- $N(rs)$ is built as follows: The initial state of $N(rs)$ is the initial state of $N(r)$, and the final state is the final state of $N(s)$. The final state of $N(r)$ is merged with the initial state of $N(s)$ to form a new state, whose transitions are the union of the transitions to the final state of $N(r)$ and the transitions from the initial state of $N(s)$.
- $N(r^*)$ is built using $N(r)$, an initial state i, and a final state f. Four e-transitions are added: from i to f, from the final state of $N(r)$ to its initial state, from i to the initial state of $N(r)$, and from the final state of $N(r)$ to f.

Example

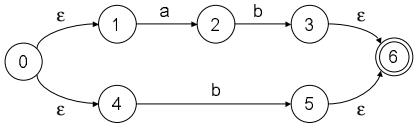
We now show how an NFA for the regular expression (ab|b)*c can be built using Thompson's construction. We begin with the following basic NFA's that recognize each of our alphabet symbols, a, b, and c:



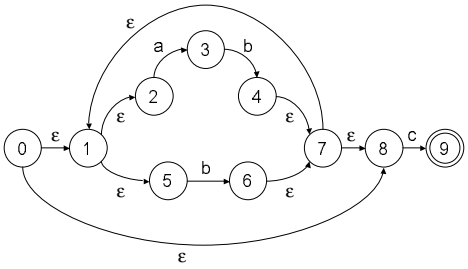
Using these basic NFA's, we build an NFA for the regular expression ab.



Using the above NFA, we then build an NFA for (ab|b).



We complete the construction by implementing the Kleene closure operator *, and then concatenating c.



Conversion of an NFA into a DFA

- While NFA's are easier to design for a given language than DFA's, they require more effort to simulate.
- Using an algorithm called *subset construction*, a given NFA can be converted into an equivalent DFA.
- Each state in the DFA represents the set of states in the NFA that can be reached using a given input string.
- The number of states in the DFA can be as large as the number of subsets of the set of the NFA's states, which is 2^n, where n is the number of states in the NFA.

Constructing a DFA from an NFA

- To convert an NFA *N* to an equivalent DFA *D*, we proceed by constructing a transition function *Dtran* for *D* so that *D* will simulate all possible moves *N* can make on a given input string.
- Each state in *D* represents a set of states from *N*. To keep track of sets of NFA states, we use the ε-*closure* operation, defined as follows:
  - Given a set of states *T* from *N*, ε-*closure*(*T*) is the set of states of *N* that can be reached from some state in *T* on ε-transitions.

E-closure

ε-*closure* can be implemented as follows:

```
push all states in T onto stack
ε-closure(T) = {T}
while stack is not empty
        pop t, the top element, off of stack
        for each state u such that move(t,ε)=u
                if u is not in ε-closure(T)
                        add u to ε-closure(T)
                        push u onto stack
                end
end
```


Subset Construction

```
ε-closure(s0) is the only state in Dstates, and is
      unmarked
while there is an unmarked state T in Dstates
    mark T
    for each input symbol a
        U = ε-closure(move(T,a))
        if U is not in Dstates
            add U as an unmarked state to Dstates
        Dtran[T,a] = U
    end
end
```

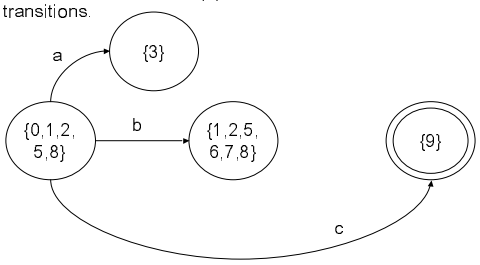

Notes on Subset Construction

In the preceding pseudo-code:

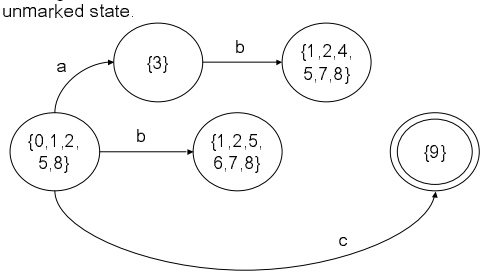
- *T* is a set of states from the NFA *N*. It corresponds to a state in the newly constructed DFA *D* for which outgoing transitions have not yet been assigned.
- Given the input symbol *a*, *U* is the set of all states from *N* that can be reached from some state in *T*.
- ε-*closure*(*T*) is a depth-first search through the graph whose vertices are the states of *N* and whose edges are the ε-transitions.

First, we create the initial state of our DFA, by computing the ε-*closure* of state 0.

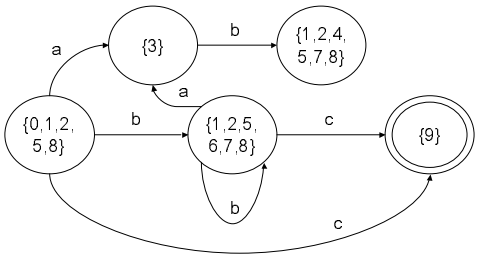
Then, we add the ε-*closure* of any states that can be reached from ε-*closure* (0), as well as the appropriate transitions.



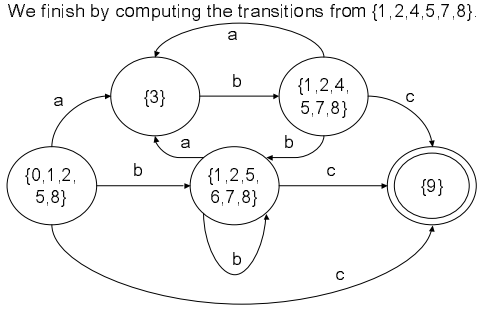
Next, we mark {3} and compute its going transitions, causing us to add the ε-*closure* of state 4 as a new unmarked state.



Next, we mark {1,2,5,6,7,8} and compute its outgoing transitions.



We finish by computing the transitions from {1,2,4,5,7,8}.



Simulation of an NFA

To determine whether an NFA N, with initial state s_0 and final states F, accepts an input string x, we proceed as follows:

S = ε-*closure*(s_0)
a = first character of x
while characters remain in x
    S = ε-*closure*(*move*(S,a))
    a = next character in x
end
if S and F are not disjoint
    N accepts x

Two-Stack Simulation of an NFA

- The preceding algorithm can be implemented efficiently using two stacks and a bit vector indexed by the states of the NFA.
- The bit vector is used to remember whether a state has been added to a stack, so we do not add it twice.
- One stack represents the set S, and the other stores ε-*closure*(S,a) while it is being computed. ε-*closure*(S,a) is completed, the roles of the stacks are interchanged.

Efficiency Analysis

- Each state has at most two outgoing transitions, and the stack can hold at most $|N|$ elements, where $|N|$ is the number of states in N. Therefore, computing the ε-*closure*(S,a) requires time proportional to $|N|$.
- The ε-*closure* is computed $|x|$ times, where $|x|$ is the length of x.
- Therefore, this implementation runs in time proportional to $|N||x|$.

Simulating a DFA

To determine whether a DFA D accepts a string x, we proceed as follows.

s = the start state of D
c = first character in x
while characters remain in x
  $s = move(s, c)$
  c = next character in x
end
if s is an accepting state of D,
  x is accepted

Time-space Tradeoffs

- Let r be a regular expression, and x be an input string. Let $|r|$ be the length of r, and $|x|$ be the length of x.
- To determine whether x belongs to $L(r)$ using an NFA, $O(|r|)$ space and $O(|r||x|)$ time is required.
- To perform the same task using a DFA, $O(2^{|r|})$ space is required, but only $O(|x|)$ time is needed.
- For this reason, when a DFA is used, transitions are usually computed only when they are needed for the particular input string. This approach, called "lazy transition evaluation," requires only $O(|r|)$ space, the same as for an NFA.

Design of a Lexical Analyzer Generator

- We are now ready to show how a lexical analyzer can be generated for a given source language.
- We assume that the tokens for the source language can be recognized using a given set of patterns, described by regular expressions.
- We will create a recognizer that will try to match some prefix of the text of a source program against the patterns. If more than one pattern matches a prefix, the recognizer must choose the longest lexeme matched. If two or more patterns match the longest lexeme, the first pattern listed is chosen. The recognizer then continues with the remainder of the input.

Pattern Matching based on NFA's

A lexical analzyer recognizing tokens using patterns $p_1, p_2, \ldots, p_n$ may be generated using an NFA as follows:

- For each pattern p_i construct an NFA $N(p_i)$.
- Construct an NFA N from the regular expression $p_1|p_2|\ldots|p_n$. Add a start state s_0 to N, and ε-transitions from s_0 to the initial states of each $N(p_i)$.
- Simulate N, but instead of accepting a lexeme as soon as an accepting state is reached, continue until there are no transitions on the current input symbol.
- Retract the input pointer to the most recent position at which an accepting state was reached, and recognize the corresponding token.

DFA for Lexical Analyzers

- A similar approach may be used with a DFA, built from the NFA described in the previous algorithm.
- As with an NFA, the DFA must be simulated until we reach a state from which there is no transition on the current symbol.
- Upon termination, we backtrack through the input until we reach a position at which the DFA entered an accepting state, and recognize the corresponding token.

The Lookahead Operator

- Often, a lexical analyzer must look beyond the end of a lexeme in order to ensure that the proper token is recognized.
- For this purpose, the lookahead operator / can be used to describe the pattern for such a token.
- The pattern r/s is interpreted as follows: a string in $L(r)$ is a lexeme for the given token only if it is followed by a string in $L(s)$.

Implementing the Lookahead Operator

- When converting a pattern that includes the lookahead operator into an NFA, we treat the / as if it were ε, so as not to look for / in the input.
- If we recognize a token corresponding to a pattern that includes /, the end of the lexeme is not the most recent position at which the NFA was in an accepting state.
- In this case, we must find the most recent position in the input at which the NFA had a transition on /.

Optimization of DFA-based Pattern Matchers

- Various algorithms have been developed to optimize pattern matchers using DFA. Some techniques are:
  - Creating a DFA directly from a regular expression, without building an NFA.
  - Reducing the number of states of a DFA
  - Building a more compact representation of the transition table of a DFA to save space.
- For more information about these methods, consult the text by Aho, Sethi and Ullman (the "Dragon book")