Lecture 12

Stack machines

Our focus in this course will now shift to more powerful computational models. We will begin with a new model of computation called the stack machine model. This model resembles the pushdown automata model, but unlike that model the stack machine model permits the use of multiple stacks. As it turns out, this makes a huge difference; whereas pushdown automata are highly limited computational models, stack machines are computationally universal. Informally speaking, this means that stack machines have the same language recognition power as a modern day programming language, or any other “reasonable” model of a general computer.

As you gain familiarity with the stack machine model, you will likely notice that it has a different character from the more limited models discussed previously in the course. In particular, through the use of multiple stacks one can mimic complex data structures, and transitions between states can mimic the sorts of control flow that is familiar in standard programming languages. Generally speaking, one has a great deal of freedom when defining a stack machine to make it operate in a desired way, just like an ordinary computer program.

12.1 Nondeterministic stack machines

We will mostly be concerned with a deterministic variant of the stack machine model in this course, but it is nevertheless convenient to begin by defining a nondeterministic variant of this model. There are two principal reasons for beginning with the nondeterministic version:

1. We have already studied the pushdown automata model, for which the nondeterministic variant is considered the default (and indeed we did not even cover the deterministic variant of this model), and the nondeterministic stack machine model is a conceptually simple variation on the pushdown automata model.
2. We will arrive at a formal definition for the deterministic stack machine model by placing restrictions on the nondeterministic version of the model. These restrictions will be natural, and they will also have a side-effect of making the model easier to work with, but they are not quite as simple as the analogous situation for finite automata, for instance.

In short, a nondeterministic stack machine (NSM) is just like a pushdown automaton, except that there are multiple stacks. The number of stacks is fixed for a particular machine, and when we wish to make explicit that the number of stacks is some positive integer $r$, we will refer to that machine as an $r$-NSM.

Here are a couple of additional points, which highlight differences between stack machines and pushdown automata that should be kept in mind as you consider the formal definition:

1. We will assume that a given NSM begins with its input string stored on its first stack (to be indexed by 0), and there will be a special bottom-of-the-stack symbol $\diamond$ at the bottom of this input stack underneath the input. This is just a simple way of allowing the machine access to its input without necessitating a special “read from the input” action that differs from an ordinary stack operation.

2. We will assume that every stack has the same stack alphabet, which must include the symbols in the input alphabet (because the input is assumed to be loaded into the first stack) along with the bottom-of-the-stack symbol $\diamond$, which may not be included in the input alphabet. We are free to include additional symbols in the stack alphabet if we choose. The computational power of the model would not actually change if we allowed each stack to have its own alphabet, but placing this restriction on the model keeps the definition and associated notation simpler.

Definition 12.1. An $r$-stack nondeterministic stack machine ($r$-NSM, for short) is an 8-tuple

$$M = (Q, \Sigma, \Delta, \delta, q_0, q_{acc}, q_{rej}),$$

where

1. $Q$ is a finite and nonempty set of states,
2. $\Sigma$ is an input alphabet, which may not include the bottom-of-the-stack symbol $\diamond$,
3. $\Delta$ is a stack alphabet, which must satisfy $\Sigma \cup \{\diamond\} \subseteq \Delta$,
4. $\delta$ is a transition function of the form

$$\delta : (Q \setminus \{q_{acc}, q_{rej}\}) \times \{\uparrow_0, \downarrow_0, \ldots, \uparrow_{r-1}, \downarrow_{r-1}\} \times \Delta \rightarrow \mathcal{P}(Q),$$

and
5. $q_0, q_{\text{acc}}, q_{\text{rej}} \in Q$ are the initial state, accept state, and reject state, respectively, which must satisfy $q_{\text{acc}} \neq q_{\text{rej}}$.

There are two operations a nondeterministic stack machine may perform: push operations and pop operations. The machine may also transition between states as it performs these operations. An interpretation of the transition function $\delta$, which specifies how these operations and transitions may be performed, is as follows:

1. If it is the case that $q \in \delta(p, \downarrow_k, a)$, then when the machine is in the state $p$, it may push the symbol $a$ onto stack $k$ and transition to state $q$.

2. If it is the case that $q \in \delta(p, \uparrow_k, a)$, then when the machine is in the state $p$ and the symbol $a$ is on the top of stack $k$, it may pop the symbol $a$ off of this stack and transition to state $q$.

We can describe NSMs using state diagrams in a similar way to finite automata and pushdown automata. As usual, states are represented by circles, and the inclusions $q \in \delta(p, \downarrow_k, a)$ and $q \in \delta(p, \uparrow_k, a)$ are indicated by arrows like this:

Stack machine computations begin as follows:

1. The input string $x \in \Sigma^*$ is assumed to be stored in stack 0. Specifically, the top symbol of stack 0 stores the first symbol of $x$, the second to top symbol of stack 0 contains the second symbol of $x$, and so on. At the bottom of stack 0, underneath all of the input symbols, is the bottom-of-the-stack symbol $\diamond$.

2. All of the other stacks initially contain just the bottom-of-the-stack symbol $\diamond$.

3. The computation starts in the initial state $q_0$.

The computation of a stack machine continues (nondeterministically, in general) so long as the machine is in one of the states other than $q_{\text{acc}}$ and $q_{\text{rej}}$ and there are valid transitions that the machine may follow. Whenever one of the states $q_{\text{acc}}$ or $q_{\text{rej}}$ is reached, the computation stops (or halts, according to traditional parlance); the corresponding computation path is deemed as accepting or rejecting, accordingly. Computation paths that effectively terminate because they result in a situation in which there are no possible transitions that can be followed are also considered to be rejecting computation paths. It is important to note that there is also a possibility for computations to carry on indefinitely, and in such a situation we will refer to the machine running forever.
Based on the discussion above, it would be routine to formulate a definition for when an NSM $M$ accepts a given string $x \in \Sigma^*$, but we’ll postpone the formal definition of acceptance for now. In the next lecture we will discuss computations of NSMs somewhat more generally, and it will be most efficient for us to include the formal definition of acceptance of NSMs in this more general discussion.

**Example 12.2.** An example of a 3-NSM $M$, described by a state diagram, is illustrated in Figure 12.1. The operation of this machine is as follows. In its first phase (represented by states $p$, $p_0$, and $p_1$), $M$ pops 0 and 1 symbols off of the input stack (stack 0) and pushes the same symbol it pops onto stack 1. Assuming it encounters a # symbol in the input string, it transitions to $r$. The states $r$, $r_0$, and $r_1$ represent the second phase of the computation, in which 0 and 1 symbols are popped off of the input stack and onto stack 2 rather than stack 1. There will at some point be no legal transitions for the machine to make if the input contains two or more occurrences of the symbol #, or if it reaches the end of the input string without encountering a # symbol. If the input contains exactly one # symbol, then the computation will transition to $s$ after the entire input string has been popped off of the input stack. The final phase of the computation compares stacks 1 and 2, and allows the accept state $q_{acc}$ to be reached if and only if these two stacks contain the same string. The strings accepted by $M$ are therefore represented by the language

$$\{w\#w : w \in \{0,1\}^\ast\}.$$  (12.3)
The state $q_{\text{rej}}$ happens to be unreachable, and is not really needed at all—but it has been included in the diagram because the definition requires that this state exists.\(^1\)

The language (12.3) is not context-free, so with our very first example we have established that nondeterministic stack machines are computationally more powerful than pushdown automata. (We did not actually prove that nondeterministic stack machines are at least as powerful as pushdown automata, but this is nearly obvious: a 2-NSM can easily simulate a PDA.)

### 12.2 Deterministic stack machines

We will now define a deterministic variant of the stack machine model. Before doing this, it should be noted that this is not quite as simple as demanding that there should always exist exactly one possible transition that can be followed from every state at every instant during a computation of a given NSM. Of course we want this property to hold, but if we simply demand this property and nothing more, we run into a problem: it turns out that it is a undecidable problem to determine whether or not this property holds for a given NSM.

With that in mind, our definition for deterministic stack machines will be somewhat more restricted. What we will do is to insist that every non-halting state is either a push state or a pop state—so that only push transitions or pop transitions, but not both, may be followed from each state—and moreover we will always associate exactly one of the stacks with each state, so that all of the push transitions or pop transitions from that state are limited to the one stack associated with that state. Finally, if the state is a push state, there must be exactly one push transition leading from that state, and if the state is a pop state, then there must be exactly one pop transition leading from that state for every possible stack symbol.

**Definition 12.3.** An $r$-NSM

\[
M = (Q, \Sigma, \Delta, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})
\]

is an $r$-stack deterministic stack machine ($r$-DSM, for short) if, for every state $q \in Q \setminus \{q_{\text{acc}}, q_{\text{rej}}\}$, there exists an index $k \in \{0, \ldots, r - 1\}$ such that exactly one of the following two properties is satisfied:

1. (The state $q$ is a push state.) There exists a symbol $a \in \Delta$ such that

\[
|\delta(q, \downarrow_k, a)| = 1.
\]

\(^1\)There is no need for a reject state in general for nondeterministic stack machines, but this state is important in the deterministic case. It is included in the definition of NSMs so that deterministic stack machines can be more easily described as a special case of nondeterministic stack machines.
Moreover,
\[ \delta(q, \downarrow j, b) = \emptyset \quad (12.6) \]
for all \( j \in \{0, \ldots, r - 1\} \) and \( b \in \Delta \) satisfying \((j, b) \neq (k, a)\), as well as
\[ \delta(q, \uparrow j, b) = \emptyset \quad (12.7) \]
for all \( j \in \{0, \ldots, r - 1\} \) and \( b \in \Delta \).

2. (The state \( q \) is a pop state.) For every symbol \( a \in \Delta \) it is the case that
\[ |\delta(q, \uparrow_k a)| = 1, \quad (12.8) \]
as well as
\[ \delta(q, \uparrow j, a) = \emptyset \quad (12.9) \]
for all \( j \in \{0, \ldots, r - 1\} \setminus \{k\} \). Moreover,
\[ \delta(q, \downarrow j, b) = \emptyset \quad (12.10) \]
for all \( j \in \{0, \ldots, r - 1\} \) and \( b \in \Delta \).

**State diagrams for DSMs**

Deterministic stack machines may be represented by state diagrams in a way that is somewhat different from general nondeterministic stack machines. As usual, states will be represented by nodes in a directed graph, directed edges (with labels) will represent transitions, and the accept and reject states are labeled as such. You will be able to immediately recognize that a state diagram represents a DSM in these notes from the fact that the nodes are square-shaped (with slightly rounded corners) rather than circle or oval shaped.

What is different from the sort of state diagram we discussed earlier in the lecture for NSMs is that the nodes themselves, rather than the transitions, will indicate the operation (push or pop) that are to be performed, as well as the stack associated with that operation. Each push state must have a single transition leading from it, with the label indicating which symbol is pushed and with the transition pointing to the next state. Each pop state must have one directed edge leading from it for each possible stack symbol, indicating to which state the computation is to transition (depending on the symbol popped). Finally, we will commonly assign names like \( X, Y, \) and \( Z \) to different stacks, rather than calling them \( stack 0, stack 1, \) and so on, as this makes for more natural, algorithmically focused descriptions of stack machines, where we view the stacks as being akin to variables in a computer program.

Figure 12.2 gives an example of a state diagram of a 3-DSM. Two additional comments on state diagrams for DSMs follow.
Figure 12.2: A 3-DSM for the language \( \{ w \# w : w \in \{0, 1\}^* \} \). The input stack is named \( X \) and the other two stacks are named \( Y \) and \( Z \).

1. Aside from the accept and reject states, we tend not to include the names of individual states in state diagrams. This is because the names we choose for the states are irrelevant to the functioning of a given machine, and omitting them makes for less cluttered diagrams. In rare cases in which it is important to include the name of a state in a state diagram, we will just write the state name above or beside its corresponding node.

2. Although every deterministic stack machine is assumed to have a reject state, we often do not bother to include it in state diagrams. Whenever there is a state with which a pop operation is associated, and one or more of the possible stack symbols does not appear on any transition leading out of this state, it is assumed that the “missing” transitions lead to the reject state.

For example, in Figure 12.2, there is no transition labeled \( \Diamond \) leading out of the initial state, so it is implicit that if \( \Diamond \) is popped off of \( X \) from this state, the machine enters the reject state.
Figure 12.3: An example of a state diagram describing a 1-DSM, whose sole stack is named $X$. This particular machine is not very interesting from a language-recognition viewpoint—it accepts every string—but it performs the useful task of erasing the contents of a stack. Here the stack alphabet is assumed to be $\Delta = \{0, 1, \diamond\}$, but the idea is easily extended to other stack alphabets.

**Subroutines**

Just like we often do with ordinary programming languages, we can define *subroutines* for stack machines. This can sometimes offer a major simplification to the descriptions of stack machines.

For example, consider the DSM whose state diagram is shown in Figure 12.3. Before discussing this machine, let us agree that whenever we say that a particular stack *stores* a string $x$, we mean that the bottom-of-the-stack marker $\diamond$ appears on the bottom of the stack, and the symbols of $x$ appear above this bottom-of-the-stack marker on the stack, with the leftmost symbol of $x$ on the top of the stack. (We will only use this terminology in the situation that the symbol $\diamond$ does not appear in $x$.) Using this terminology, the behavior of the DSM illustrated in Figure 12.3 is that if its computation begins with $X$ storing an arbitrary string $x \in \{0, 1\}^*$, then the computation always results in acceptance, with $X$ storing $\varepsilon$. In other words, the DSM erases the string stored by $X$ and halts.

Now, the simple process performed by this DSM might be useful as a subroutine inside of some more complicated DSM, and of course a simple modification allows us to choose any stack in place of $X$ that gets erased. Rather than replicating the description of the DSM from Figure 12.3 inside of this more complicated hypothetical DSM, we can simply use the sort of shorthand suggested by Figure 12.4.

More explicitly, the diagram on the left-hand side of Figure 12.4 suggests a small part of a hypothetical DSM, where the DSM from Figure 12.3 appears inside of the dashed box. Note that we have not included the accept state in the
Lecture 12

Figure 12.4: The diagrams on the left and right describe equivalent portions of a larger DSM; the contents of the dotted rectangle in the left-hand diagram is viewed as a *subroutine* that is represented by a single rectangle labeled “$X \leftarrow \epsilon$” in the right-hand diagram.

dashed box because, rather than accepting, we wish for control to be passed to the state labeled “another state” as the erasing process completes. We also do not have that the “pop $X$” state is the initial state any longer, because rather than starting at this state, we have that control passes to this state from the state labeled “some state.” (There could, in fact, be multiple transitions from multiple states leading to the “pop $X$” state in the hypothetical DSM we’re considering.) In the diagram on the right-hand side of Figure 12.4 we have replaced the dashed box with a single rectangle labeled “$X \leftarrow \epsilon$.” This is just a label that we’ve chosen, but of course it is a fitting label in this case. The rectangle labeled “$X \leftarrow \epsilon$” looks like a state, and we can think of it as being like a state with which a more complicated operation than push or pop is associated—but the reality is that it is just a short-hand for the contents of the dashed box on the left-hand side diagram.

The same general pattern can be replicated for just about any choice of a DSM. That is, if we have a DSM that we would like to use as a subroutine, we can always do this as follows:

1. Let the original start state of the DSM be the state to which some transition points.
2. Remove the accept state, modifying transitions to this removed accept state so that they point to some other state elsewhere in the larger DSM to which control is to pass once the subroutine is complete.
Figure 12.5: An example of a state diagram describing a 3-DSM (with stacks named $X$, $Y$, and $Z$). This machine performs the task of copying the contents of one stack to another: $X$ is copied to $Y$. The stack $Z$ is used as workspace to perform this operation.

Naturally, one must be careful when defining and using subroutines like this, as computations could easily become corrupted if subroutines modify stacks that are being used for other purposes elsewhere in a computation. (The same thing can, of course, be said concerning subroutines in ordinary computer programs.)

Another example of a subroutine is illustrated in Figure 12.5. This stack machine copies the contents of one stack to another, using a third stack as an auxiliary (or workspace) stack to accomplish this task. Specifically, under the assumption that a stack $X$ stores a string $x \in \{0, 1\}^*$ and stacks $Y$ and $Z$ store the empty string, the illustrated 3-DSM will always lead to acceptance—and when it does accept, the stacks $X$ and $Y$ will both store the string $x$, while $Z$ will revert to its initial configuration in which it stores the empty string. This action can be summarized as follows:

\[
\begin{array}{c c}
X \text{ stores } x & X \text{ stores } x \\
Y \text{ stores } \epsilon & Y \text{ stores } x \\
Z \text{ stores } \epsilon & Z \text{ stores } \epsilon
\end{array}
\]
If we wish to use this DSM as a subroutine in a more complicated DSM, we could again represent the entire DSM (minus the accept state) by a single rectangle, just like we did in Figure 12.4. A fitting label in this case is “Y ← X.”

One more example of a DSM that is useful as a subroutine is pictured in Figure 12.6. Notice that in this state diagram we have made use of the two previous subroutines to make the figure simpler. After each new subroutine is defined, we’re naturally free to use it to describe new DSMs. The DSM in the figure reverses the string stored by X. It uses a workspace stack Y to accomplish this task—but in fact it also uses a workspace stack Z, which is hidden inside the subroutine labeled “Y ← X.” In summary, it performs this transformation:

\[
\begin{align*}
X & \text{ stores } x \\
Y & \text{ stores } \varepsilon \\
Z & \text{ stores } \varepsilon
\end{align*}
\quad \rightarrow \quad
\begin{align*}
X & \text{ stores } x^R \\
Y & \text{ stores } \varepsilon \\
Z & \text{ stores } \varepsilon
\end{align*}
\]  

(12.12)

Hereafter we will tend not to list explicitly all of the workspace stacks used by our DSMs. So long as we are careful to always revert our workspace stacks back to their initial configuration, in which they store the empty string, we can just imagine that each occurrence of a subroutine in a given state diagram has its own workspace stacks associated with it that we never use in other subroutine occurrences or elsewhere in the larger DSM under consideration. We also won’t worry about using an excessive number of stacks; there are strategies for limiting the number of stacks required to perform computations (and in fact just two stacks are always enough, as we will see later), but for the purposes of this course there will be no compelling reasons to focus on this issue.