Lecture 13

Stack machine computations, languages, and functions

In this lecture we will discuss computations of stack machines at a formal level, and then we will define a few fundamental notions that we will carry with us for the rest of the course, including the notions of semidecidable and decidable languages and computable functions. The last part of the lecture will be devoted to further developing the stack machine model of computation, with a principle aim being to recognize that the deterministic stack machine model is, in an abstract mathematical sense, representative of the computational power of an ordinary computer.\(^1\)

13.1 Stack machine configurations and computations

In the previous lecture we defined the NSM model, and then we defined the DSM model by restricting the definition of NSMs in a way that guarantees that we always have just one possible transition out of each non-halting state at each instant. It remains for us to formally define acceptance and rejection for NSMs and DSMs; and although it would be routine to do this by mimicking the analogous definitions for PDAs, we will take a different approach that will allow us to define the important concept of computable functions more or less simultaneously. The approach is reminiscent of the definition for when a CFG generates a given string.

Let us begin by defining the set of configurations of a stack machine. Intuitively speaking, a configuration of a stack machine describes its current state and the contents of all of its stacks at a particular instant.

\(^1\)You might be inclined to think that ordinary computers are actually better modeled by finite automata than by stack machines, due to the fact that there is a finite (albeit large) amount of data storage available in the world. Keep in mind, however, that there is a difference between the physical world and the mathematical abstractions through which we aim to understand the world, and recognize that modeling an ordinary computer as a finite automaton is not very informative.
**Definition 13.1.** Let $M = (Q, \Sigma, \Delta, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ be an r-NSM. A configuration of $M$ is an element of the set $Q \times (\Delta^*)^r$.

The configuration $(p, x_0, \ldots, x_{r-1})$ of a stack machine $M$ indicates that the current state of $M$ is $p$, the contents of stack 0 is given by $x_0$ (with the left-most symbol of $x_0$ on the top of stack 0 and the rightmost symbol of $x_0$ on the bottom of stack 0), the contents of stack 1 is given by $x_1$, and so on.

Next, we will define a yields relation on pairs of configurations that represents the possibility that a given stack machine can transition from the first configuration to the second. Similar to the yields relation for context-free grammars, we will define two variants of this relation: the first represents the possibility to move from one configuration to another in a single step, and the second relation is the reflexive transitive closure of the first relation. In other words, the second relation is a starred version, representing the possibility to transition from one configuration to another through zero or more steps of the stack machine under consideration.

**Definition 13.2.** Let $M = (Q, \Sigma, \Delta, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ be an r-NSM. The yields relation defined by $M$ is the relation denoted $\vdash_M$ that consists of all of the following pairs of configurations:

1. For every pair of states $p \in Q \setminus \{q_{\text{acc}}, q_{\text{rej}}\}$ and $q \in Q$, every stack symbol $a \in \Delta$, and every stack index $k \in \{0, \ldots, r - 1\}$ for which $q \in \delta(p, \downarrow_k, a)$, the yields relation for $M$ includes the pair
   \[
   (p, x_0, \ldots, x_{r-1}) \vdash_M (q, x_0, \ldots, x_{k-1}, ax_k, x_{k+1}, \ldots, x_{r-1})
   \]  
   (13.1)
   for all strings $x_0, \ldots, x_{r-1} \in \Delta^*$.

2. For every pair of states $p \in Q \setminus \{q_{\text{acc}}, q_{\text{rej}}\}$ and $q \in Q$, every stack symbol $a \in \Delta$, and every stack index $k \in \{0, \ldots, r - 1\}$ for which $q \in \delta(p, \uparrow_k, a)$, the yields relation for $M$ includes the pair
   \[
   (p, x_0, \ldots, x_{k-1}, ax_k, x_{k+1}, \ldots, x_{r-1}) \vdash_M (q, x_0, \ldots, x_{r-1})
   \]  
   (13.2)
   for all strings $x_0, \ldots, x_{r-1} \in \Delta^*$.

3. For every state $p \in Q \setminus \{q_{\text{acc}}, q_{\text{rej}}\}$, every stack symbol $a \in \Delta$, and every stack index $k \in \{0, \ldots, r - 1\}$ for which $q \in \delta(p, \uparrow_k, a)$, the yields relation for $M$ includes the pairs
   \[
   (p, x_0, \ldots, x_{k-1}, \varepsilon, x_{k+1}, \ldots, x_{r-1}) \vdash_M (q_{\text{rej}}, x_0, \ldots, x_{k-1}, \varepsilon, x_{k+1}, \ldots, x_{r-1})
   \]  
   (13.3)
   for all strings $x_0, \ldots, x_{k-1}, x_{k+1}, \ldots, x_{r-1} \in \Delta^*$.
The relation $\vdash^*_M$ is the symmetric, transitive closure of $\vdash_M$. In symbols, for every pair of configurations $(p, x_0, \ldots, x_{r-1})$ and $(q, y_0, \ldots, y_{r-1})$, it is the case that

$$ (p, x_0, \ldots, x_{r-1}) \vdash^*_M (q, y_0, \ldots, y_{r-1}) \quad (13.4) $$

if there exist a sequence of configurations $c_0, \ldots, c_t \in Q \times (\Delta^*)^r$, for some non-negative integer $t \in \mathbb{N}$, such that $c_0 = (p, x_0, \ldots, x_{r-1}), c_t = (q, y_0, \ldots, y_{r-1})$, and $c_j \vdash_M c_{j+1}$ for every $j \in \{0, \ldots, t - 1\}$.

**Remark 13.3.** The third category in the previous definition is a boundary case that we did not discuss in the previous lecture. It essentially says that it is possible to transition to the reject state by attempting to pop an empty stack. (When this happens, the empty stack remains empty.) By defining the yields relation in this way, we ensure that every non-halting configuration of a DSM has a well-defined next configuration, which is a rejecting configuration when a DSM attempts to pop an empty stack.

### 13.2 Languages and functions from DSMs

Having defined the yields relation $\vdash_M$ for a given stack machine $M$, we are now prepared to define a few important notions. Our primary focus will be on deterministic stack machines, but one can also consider analogous notions for non-deterministic stack machines.

**Acceptance, rejection, and running forever**

First we will formally define what it means for a DSM to accept or to reject a given input string, or to do neither.

**Definition 13.4.** Let $M = (Q, \Sigma, \Delta, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ be an $r$-DSM and let $w \in \Sigma^*$ be an input string.

1. $M$ accepts $w$ if there exist strings $x_0, \ldots, x_{r-1} \in \Delta^*$ such that

$$ (q_0, w\diamond, \diamond, \ldots, \diamond) \vdash^*_M (q_{\text{acc}}, x_0, \ldots, x_{r-1}). \quad (13.5) $$

2. $M$ rejects $w$ if there exist strings $x_0, \ldots, x_{r-1} \in \Delta^*$ such that

$$ (q_0, w\diamond, \diamond, \ldots, \diamond) \vdash^*_M (q_{\text{rej}}, x_0, \ldots, x_{r-1}). \quad (13.6) $$

3. $M$ runs forever on $w$ if $M$ neither accepts nor rejects $w$. 


It is important to recognize that running forever is a possibility for deterministic stack machines; but you are surely familiar with the same possibility for ordinary computer programs, so it will not likely come as a surprise that this might happen for a stack machine. For the first two possibilities in the previous definition (\( M \) accepts \( w \) and \( M \) rejects \( w \)), the strings \( x_0, \ldots, x_{r-1} \) represent whatever happens to be on the stacks of \( M \) when it reaches the accept or reject state. We don’t place any restrictions on what these stacks might contain, all that matters for the sake of this definition is that either the state \( q_{\text{acc}} \) or the state \( q_{\text{rej}} \) was reached.

At this point we can be precise about the definition of the language recognized by a DSM.

**Definition 13.5.** Let \( M = (Q, \Sigma, \Delta, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}) \) be an r-DSM. The language recognized by \( M \) is defined as

\[
L(M) = \{ w \in \Sigma^* : M \text{ accepts } w \}. \tag{13.7}
\]

**Semidecidable languages**

We now define the class of semidecidable languages to be the class of languages for which there exists a DSM that recognizes that language.

**Definition 13.6.** Let \( \Sigma \) be an alphabet and let \( A \subseteq \Sigma^* \) be a language. The language \( A \) is **semidecidable** if there exists a DSM \( M \) such that \( A = L(M) \).

There are several alternative names that people often use in place of semidecidable, including Turing recognizable, partially decidable, and recursively enumerable (or r.e. for short).

The name semidecidable reflects the fact that if \( A = L(M) \) for some DSM \( M \), and \( w \in A \), then running \( M \) on \( w \) will necessarily lead to acceptance; but if \( w \notin A \), then \( M \) might either reject or run forever on input \( w \). That is, \( M \) does not really decide whether a string \( w \) is in \( A \) or not, it only “semidecides”; for if \( w \notin A \), you might not ever really know this as a result of running \( M \) on \( w \).

**Decidable languages**

Next, we define the class of decidable languages. As the following definition makes clear, a decidable language is one for which there exists a DSM that correctly answers whether or not a given string is in the language (and therefore never runs forever).
Definition 13.7. Let $\Sigma$ be an alphabet and let $A \subseteq \Sigma^*$ be a language. The language $A$ is decidable if there exists a DSM $M$ with these two properties:

1. $M$ accepts every string $w \in A$.
2. $M$ rejects every string $w \in \overline{A}$.

Example 13.8. The language $\{w#w : w \in \{0, 1\}^*\}$ is decidable, as the DSM for this language defined in Lecture 12 accepts every string in this language and rejects every string not in this language.

Analogous notions for NSMs

Although our main focus for the remainder of the course will be on deterministic stack machines, it is appropriate to take a moment to briefly discuss nondeterministic stack machines and analogous notions to those discussed above for DSMs.

We will begin by formally defining acceptance for NSMs as well as the language recognized by a given NSM. These definitions are actually exactly the same as for DSMs, as you might have expected. We will not, however, define rejection or running forever for NSMs.

Definition 13.9. Let $M = (Q, \Sigma, \Delta, \delta, q_0, q_{acc}, q_{rej})$ be an $r$-NSM. The NSM $M$ accepts the string $w \in \Sigma^*$ if there exist strings $x_0, \ldots, x_{r-1} \in \Delta^*$ such that

$$(q_0, w \diamond, \diamond, \ldots, \diamond) \vdash^r_M (q_{acc}, x_0, \ldots, x_{r-1}).$$

The language recognized by $M$ is defined as

$$L(M) = \{w \in \Sigma^* : M \text{ accepts } w\}.$$
the proof is that for a given NSM $N$, we can construct a DSM $M$ that effectively searches through all of the possible computations of $N$ on a given input string $w$ using a breadth-first search of the “computation tree” of $N$. This search might take a long time, this is not a problem because the theorem makes no claims about computational efficiency.

It is also possible to prove a variant of Theorem 13.10 that gives a characterization of decidable languages in terms of NSMs, but we’ll skip this in the interest of saving time. In short, a language $A$ is decidable if and only if there exists an NSM $M$ with two properties: the first is that $A = \mathcal{L}(M)$, and the second is that it is never possible for $M$ to make nondeterministic transitions that cause it to run forever. The idea behind the proof of this fact is exactly the same as the idea of the proof of Theorem 13.10.

**Computable functions**

Next we will define the class of computable functions. The motivation behind this concept is quite straightforward: we often want to compute functions, or consider computations that evaluate functions, as opposed to just deciding membership in languages. The concept of a computable function is also useful as a tool for studying decidability and semidecidability of languages.

It is easy to adapt the stack machine model so that it computes functions rather than just deciding membership in a language by (i) requiring that a given DSM $M$ that computes a function always accepts, and (ii) requiring that the output of the function is stored on the first stack (and all other stacks store the empty string) when the computation eventually enters the accept state. We can generalize this definition without difficulty to allow for different input and output alphabets, and to allow for functions having multiple input and output arguments.

**Definition 13.11.** Let $\Sigma$ and $\Gamma$ be alphabets, neither of which includes the bottom-of-the-stack symbol $\diamondsuit$. A function $f : \Sigma^* \to \Gamma^*$ is **computable** if there exists an $r$-DSM $M = (Q, \Sigma, \Delta, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ such that, for every string $x \in \Sigma^*$, it is the case that

\[
(q_0, x\diamondsuit, \diamondsuit, \ldots, \diamondsuit) \vdash^* M (q_{\text{acc}}, f(x)\diamondsuit, \diamondsuit, \ldots, \diamondsuit).
\]  

(13.10)

More generally, a function $g : (\Sigma^*)^n \to (\Gamma^*)^m$, for positive integers $n$ and $m$, is **computable** if there exists an $r$-DSM $M = (Q, \Sigma, \Delta, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ such that, for every choice of strings $x_1, \ldots, x_n \in \Sigma^*$, it is the case that

\[
(q_0, x_1\diamondsuit, \ldots, x_n\diamondsuit, \diamondsuit, \ldots, \diamondsuit) \vdash^* M (q_{\text{acc}}, y_1\diamondsuit, \ldots, y_m\diamondsuit, \diamondsuit, \ldots, \diamondsuit)
\]  

(13.11)

for $(y_1, \ldots, y_m) = g(x_1, \ldots, x_n)$. 

6
You could generalize the definition even further by allowing each of the arguments to have its own alphabet, but the definition above is sufficient for our needs.

Although we did not refer to them as such, we already saw a few examples of computable functions in Lecture 12:

1. The function \( f : \{0, 1\}^* \rightarrow \{0, 1\}^* \) defined as \( f(x) = \varepsilon \) for every \( x \in \{0, 1\}^* \) is computable.
2. The function \( f : \{0, 1\}^* \rightarrow \{0, 1\}^* \times \{0, 1\}^* \) defined as \( f(x) = (x, x) \) for every \( x \in \{0, 1\}^* \) is computable.
3. The function \( f : \{0, 1\}^* \rightarrow \{0, 1\}^* \) defined as \( f(x) = x^R \) for every \( x \in \{0, 1\}^* \) is computable.

All three of these examples could easily be generalized to any alphabet \( \Sigma \) in place of \( \{0, 1\} \). We will see more examples later.

### Subroutines from computable functions

It is perhaps evident, but nevertheless worth mentioning, that computable functions can easily be turned into subroutines. In particular, suppose that we have a computable function

\[
f : (\Sigma^n)^* \rightarrow \Gamma^*.
\]  

(13.12)

For any choice of distinct stacks \( Y_1, \ldots, Y_n, Z \), we can easily transform a DSM that computes \( f \) into a stack machine subroutine for the instruction

\[
Z \leftarrow f(Y_1, \ldots, Y_n).
\]  

(13.13)

Our interpretation of such a subroutine is that \( Z \) is overwritten with the string \( f(y_1, \ldots, y_n) \), for \( y_1, \ldots, y_n \) being the strings stored by \( Y_1, \ldots, Y_n \), and that once the subroutine is finished the stacks \( Y_1, \ldots, Y_n \) will still store their original strings \( y_1, \ldots, y_n \).

Of course it is possible to generalize this sort of construction, but this one will be sufficient for our needs.

### 13.3 New computable functions from old

We will end this lecture by discussing a few ways in which computable functions can be combined or modified so that new computable functions are obtained. Using these methods is often an effective way to prove that certain functions are computable without explicitly constructing DSMs that compute them.

Let us begin with three propositions that state simple facts concerning computable functions that may be considered to be straightforward.
Proposition 13.12 (Tuples of computable functions are computable). Let $\Sigma$ and $\Gamma$ be alphabets, let $n$ and $m$ be positive integers, and let
\[
g_1 : (\Sigma^*)^n \rightarrow \Gamma^*
\]
\[
\vdots
\]
\[
g_m : (\Sigma^*)^n \rightarrow \Gamma^*
\]
be computable functions. The function $f : (\Sigma^*)^n \rightarrow (\Gamma^*)^m$ defined as
\[
f(x_1, \ldots, x_n) = (g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))
\]
for all $x_1, \ldots, x_n \in \Sigma^*$ is a computable function.

Proposition 13.13 (Projection functions are computable). Let $\Sigma$ be an alphabet, let $n$ be a positive integer, and let $k \in \{1, \ldots, n\}$. The function $\pi^n_k : (\Sigma^*)^n \rightarrow (\Sigma^*)^1$ defined as
\[
\pi^n_k(x_1, \ldots, x_n) = x_k
\]
is computable.

Proposition 13.14 (Compositions of computable functions are computable). Let $\Sigma$, $\Gamma$, and $\Lambda$ be alphabets, let $n$ and $m$ be positive integers, and let $f : (\Gamma^*)^m \rightarrow \Lambda^*$ and
\[
g_1 : (\Sigma^*)^n \rightarrow \Gamma^*
\]
\[
\vdots
\]
\[
g_m : (\Sigma^*)^n \rightarrow \Gamma^*
\]
be computable functions. The function $h : (\Sigma^*)^n \rightarrow \Lambda^*$ defined as
\[
h(x_1, \ldots, x_n) = f(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))
\]
for all $x_1, \ldots, x_n \in \Sigma^*$ is a computable function.

Finally, let us prove the following theorem, which will allow us to conclude that certain functions obtained by recursion from computable functions are also computable. Using this theorem effectively can sometimes greatly simplify the job of proving that a given function is computable.

Theorem 13.15 (Recursions on computable functions are computable). Let $\Sigma$ be an alphabet, let $g_a : \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$ be a computable function for each symbol $a \in \Sigma$, and let $w \in \Sigma^*$ be a string. The function $h : \Sigma^* \rightarrow \Sigma^*$ defined as
\[
h(\epsilon) = w
\]
\[
h(ax) = g_a(x, h(x)) \quad (\text{for every } a \in \Sigma \text{ and } x \in \Sigma^*)
\]
is computable.
Figure 13.1: This DSM computes the function \( f(x) = (x, h(x)) \), where \( h \) is obtained from \( \{g_a : a \in \Sigma\} \) and \( w \) by recursion, as defined in (13.19).

Proof. Consider the DSM \( M \) described in Figure 13.1. Our goal will be to prove that \( M \) computes the function \( f(x) = (x, h(x)) \). It is to be understood that \( X \) is stack 0 and \( Y \) is stack 1, so that \( X \) initially stores the input string while the pair \((X,Y)\) represents the output. The stacks \( Z \) and \( W \) are used as workspace. Let us note explicitly that we have represented the execution of multiple subroutines as single nodes to make the figure more compact and readable. All of the subroutines used have already been discussed, with the exception of the one labeled “\( Y \leftarrow w \).” As \( w \) is a fixed string, this subroutine is easily implemented by simply pushing the symbols of \( w \) onto \( Y \) one at a time.

We will prove by induction that \( M \) computes \( f \). The base case is that \( x = \epsilon \). An inspection of the DSM \( M \) reveals that it accepts with \( X \) containing \( \epsilon \), \( Y \) containing \( w \), and \( Z \) and \( W \) containing \( \epsilon \), which is consistent with the value \( f(\epsilon) = (\epsilon, h(\epsilon)) \).

Now suppose that \( a \in \Sigma \) is any symbol, and assume that for a given input string \( x \in \Sigma^* \), the DSM \( M \) correctly computes \( f(x) = (x, h(x)) \). For the input \( ax \), we combine this assumption with an examination of \( M \) to conclude that this DSM must reach the state labeled “\( \text{pop} \ Z' \)” with \( X \) storing \( x \), \( Y \) storing \( h(x) \), and \( Z \) storing \( a \) (as opposed to \( Z \) storing \( \epsilon \), as it would if the input had been \( x \) rather than \( ax \)). The loop is iterated one more time, and we find that \( X \) stores \( ax \), \( Y \) stores \( h(ax) = g_a(x, h(x)) \), and \( Z \) stores \( \epsilon \), from which it follows that \( M \) correctly computes \( f(ax) = (ax, h(ax)) \), as required.
Given that \( f(x) = (x, h(x)) \) is a computable function, it follows from Propositions 13.13 and 13.14 that \( h \) is computable.

**Example 13.16.** Let \( \Sigma = \{0, 1\} \) be the binary alphabet and define two functions, \( g_0, g_1 : (\Sigma^*)^2 \to \Sigma^* \) as follows:

\[
\begin{align*}
g_0(x, y) &= 1x \\
g_1(x, y) &= 0y
\end{align*}
\] (13.20)

for all \( x, y \in \Sigma^* \). These are easily shown to be computable functions. Now consider the function \( h : \Sigma^* \to \Sigma^* \) obtained by recursion as follows:

\[
h(\varepsilon) = 0 \\
h(ax) = g_a(x, h(x))
\]

\[
= \begin{cases} 
1x & \text{if } a = 0 \\
0h(x) & \text{if } a = 1.
\end{cases}
\] (13.21)

The first few values of \( h \) are as follows:

\[
h(\varepsilon) = 0, \quad h(0) = 1, \quad h(1) = 00, \quad h(00) = 10, \\
h(10) = 01, \quad h(01) = 11, \quad h(11) = 000, \quad h(000) = 100.
\] (13.22)

The function \( h \) is almost a function that increments with respect to lexicographic ordering, except that left and right are reversed. The function \( \text{inc} : \Sigma^* \to \Sigma^* \) defined as

\[
\text{inc}(x) = (h(x^R))^R
\] (13.23)

will actually increment with respect to lexicographic order—but because it is obtained from \( h \) and the reverse function by composition (two times), and both \( h \) and the reverse function are computable, it follows that \( \text{inc} \) is also computable.

It is possible to prove more general forms of Theorem 13.15. For example, if \( h : (\Sigma^*)^{n+1} \to \Sigma^* \) is defined as

\[
\begin{align*}
h(\varepsilon, y_1, \ldots, y_n) &= f(y_1, \ldots, y_n) \\
h(ax, y_1, \ldots, y_n) &= g_a(x, h(x, y_1, \ldots, y_n), y_1, \ldots, y_n)
\end{align*}
\] (13.24)

for every \( a \in \Sigma \) and \( x, y_1, \ldots, y_n \in \Sigma^* \), for computable functions \( f : (\Sigma^*)^n \to \Sigma^* \) and \( g_a : (\Sigma^*)^{n+2} \to \Sigma^* \) (for each \( a \in \Sigma \)), then \( h \) is computable.