**Running a bottom-up parse machine**

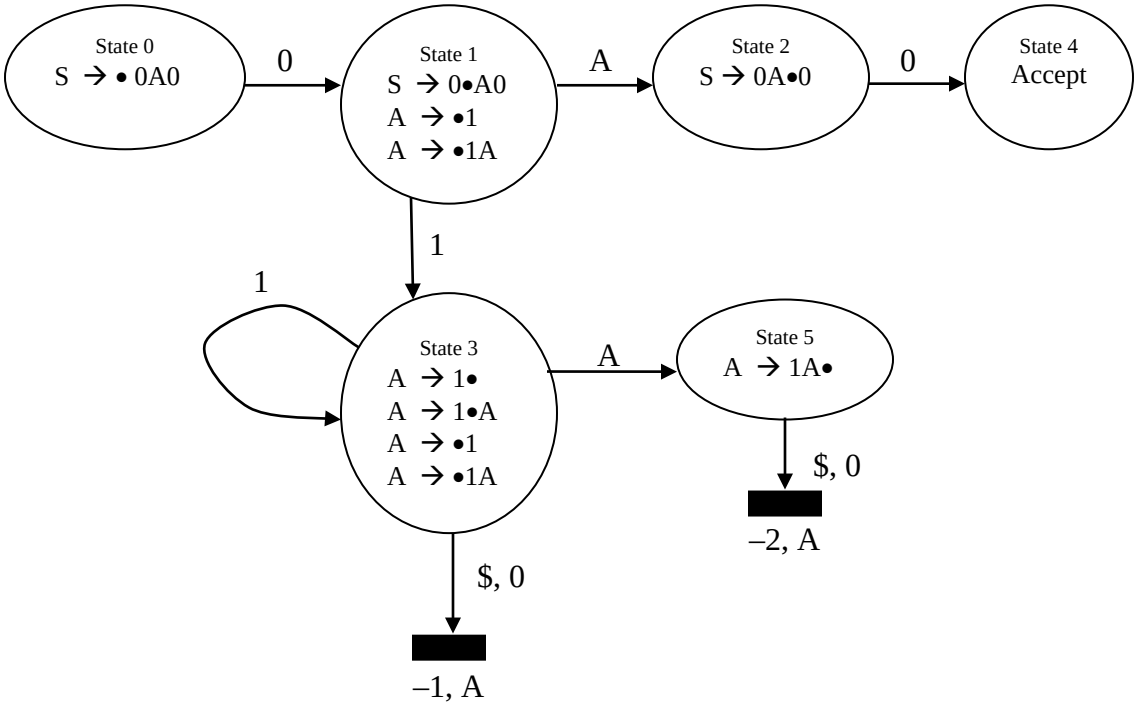
A programming language is essentially a restricted form of a context-free language. Like a context-free language, a programming language is defined by means of a grammar. And in place of a pushdown automaton, the classical approach is to create a program called a "parser" that scans input and looks up appropriate actions in a parse table. A more visual approach of depicting the actions of a parser can be seen in a "parse machine." In this handout, you will see how a parse machine works. After that, we will see how they are created.

Here is a simple example of a grammar that we can convert into a parse machine:

$$S \rightarrow 0A0$$
$$A \rightarrow 1 \mid 1A$$

Let's represent the end of input with a special symbol "$". The reason why we need this special symbol is that we probably need to take several actions upon reaching the end of the input. In the drawing of the parse machine below, the start state is 0 and the accept state is 4. Reading the transitions across the top row of states, these transitions, taking a "0", "A" and "0" from the input, correspond to the rule $S \rightarrow 0A0$, so the overall effect of the parse machine is visually intuitive. The "A" is resolved by states 3 and 5 that take care of the 1's that appear in the middle of the input string.

It is often helpful to understand the meaning of each state in the parse machine. I have included the items inside each of the states in the drawing of the parse machine. For example, state 0 represents the state in which we have not yet read the initial 0; state 1 represents that we have read the initial 0 but are now waiting to read the set of input symbols corresponding to "A" in the grammar; state 2 is the state where the "A" has been reduced and we are ready to read the final 0; and so on.



Or, we can represent this in the form of a parse table:

| From state | 0 | 1 | $ | S | A |
|---|---|---|---|---|---|
| 0 | 1 | | | | |
| 1 | | 3 | | | 2 |
| 2 | 4 | | | | |
| 3 | -1,A | 3 | -1,A | | 5 |
| 4 | | | accept | | |
| 5 | -2,A | | -2,A | | |

The black rectangles are not states, but instead represent *reduce blocks*. To "reduce" means that we need to backtrack from the current state. For example, there is a reduce block attached to state 3. The transition is labeled "$, 0", which means we have a reduce if we read a $ or a 0. The reduce block is annotated "–1, A" which means we backtrack one state, and we insert the grammar symbol A into the input at the current cursor position. *Notice that when we reduce, we don't consume any input.* The reduce block attached to state 5 is also taken on a $ or 0 input. In this case "–2, A" means we backtrack two states and insert A into the input.

**Step-by-Step Trace**

To start with, we need to have a parse stack to keep track of where we have been. This is not surprising because a pushdown automaton also maintains a stack.

Here is what happens when we run the parse machine on input 0110. The transitions are given below. In the state-stack column, the top of the stack is on the right. The number on top of the stack indicates the state we are currently in. In the input string column, the cursor (•) is included to show which symbol is being considered for taking transitions. For example, in the second row of data in the table, we are currently in state 1, the input character is "1" and the next state is 3. Each time we perform a goto operation, the cursor advances one symbol to the right.

The fourth and sixth transitions taken during the trace are reduce operations. Again, please note that when we reduce, we don't consume any input symbol. When we are in state 3 and we read a "0", this indicates that we are done with the grammar rule A → 1. And later, when we are in state 5 and read the same "0", this means that we are done with the grammar rule A → 1A. In other words, we are done reading all the 1's in the input, so next we should expect to read the final "0" from input.

Trace of running parse machine on input 0110:

| State Stack | Input string | Next state |
|---|---|---|
| 0 | • 0 1 1 0 | 1 |
| 0 1 | 0 • 1 1 0 | 3 |
| 0 1 3 | 0 1 • 1 0 | 3 |
| 0 1 3 3 | 0 1 1 • 0 | Need to backtrack: –1, A |
| 0 1 3 | 0 1 1 • A 0 | 5 |
| 0 1 3 5 | 0 1 1 A • 0 | Need to backtrack: –2, A |
| 0 1 | 0 1 1 A • A 0 | 2 |
| 0 1 2 | 0 1 1 A A • 0 | 4 (which is the accept state) |

While running the parse machine, a syntax error will result when an anticipated transition does not exist in the parse machine. For example, if the input string begins with a "1", in state 0 there is no transition for a "1", so the machine immediately halts and rejects the input.