Context Sensitivity of C++

It appears from Figure 7.8 that the C++ language is context-free. Every production rule has only a single nonterminal on the left side. This is in contrast to a context-sensitive grammar, which can have more than a single nonterminal on the left, as in Figure 7.3. Appearances are deceiving. Even though the grammar for this subset of C++, as well as the full standard C++ language, is context-free, the language itself has some context-sensitive aspects.

Consider the grammar in Figure 7.3. How do its rules of production guarantee that the number of c’s at the end of a string must equal the number of a’s at the beginning of the string? Rules 1 and 2 guarantee that for each a generated, exactly one C will be generated. Rule 3 lets the C commute to the right of B. Finally, Rule 5 lets you substitute c for C in the context of having a b to the left of C. The language could not be specified by a context-free grammar because it needs Rules 3 and 5 to get the C’s to the end of the string.

There are context-sensitive aspects of the C++ language that Figure 7.8 does not specify. For example, the definition of <parameter-list> allows any number of formal parameters, and the definition of <argument-expression-list> allows any number of actual parameters. You could write a C++ program containing a procedure with three formal parameters and a procedure call with two actual parameters that is derivable from <translation-unit> with the grammar in Figure 7.8. If you try to compile the program, however, the compiler will declare a syntax error.

The fact that the number of formal parameters must equal the number of actual parameters in C++ is similar to the fact that the number of a’s at the beginning of the string must equal the number of c’s at the end of the string in the language defined by the grammar in Figure 7.3. The only way to put that restriction in C++’s grammar would be to include many complicated, context-sensitive rules. It is easier for the compiler to parse the program with a context-free grammar and check for any violations after the parse—usually with the help of its symbol table—that the grammar cannot specify.

7.2 Finite State Machines

Finite state machines are another way to specify the syntax of a sentence in a language. In diagram form, a finite state machine is a finite set of states represented by circles called nodes and transitions between the states represented by arcs between the circles. Each arc begins at one state and ends at another, and contains an arrowhead at the ending state. Each arc is also labeled with a character from the terminal alphabet of the language.

One state of the finite state machine (FSM) is designated as the start state and at least one, possibly more, is designated a final state. On a diagram, the start state has an incoming arrow and a final state is indicated by a double circle.
An FSM to Parse an Identifier

Figure 7.10 shows an FSM that parses an identifier as defined by the grammar in Figure 7.1. The set of states is \{A, B, C\}. A is the start state, and B is the final state. There is a transition from A to B on a letter, from A to C on a digit, from B to B on a letter or a digit, and from C to C on a letter or a digit.

To use the FSM, imagine that the input string is written on a piece of paper tape. Start in the start state, and scan the characters on the input tape from left to right. Each time you scan the next character on the tape, make a transition to another state of the finite state machine. Use only the transition that is allowed by the arc corresponding to the character you have just scanned. After scanning all the input characters, if you are in a final state, the characters are a valid identifier. Otherwise they are not.

Example 7.4 To parse the string cab3, you would make the following transitions:

- Current state: A Input: cb3 Scan c and go to B.
- Current state: B Input: ab3 Scan a and go to B.
- Current state: B Input: b3 Scan b and go to B.
- Current state: B Input: 3 Scan 3 and go to B.
- Current state: B Input: Check for final state.

Because there is no more input and the last state is B, a final state, cab3 is a valid identifier.

You can also represent an FSM by its state transition table. Figure 7.11 is the state transition table for the FSM of Figure 7.10. The table lists the next state reached by the transition from a given current state on a given input symbol.

Simplified Finite State Machines

It is often convenient to simplify the diagram for an FSM by eliminating the state whose sole purpose is to provide transitions for illegal input characters. State C in this machine is such a state. If the first character is a digit, the string will not be a valid identifier, regardless of the following characters. State C acts like a failure state. Once you make a transition to C, you can never make a transition to another state, and you know the input string eventually will be declared invalid. Figure 7.12 shows the simplified FSM of Figure 7.10 without the failure state.
When you parse a string with this simplified machine, you will not be able to make a transition when you encounter an illegal character in the input string. There are two ways to detect an illegal sentence in a simplified FSM:

- You may run out of input, and not be in a final state.
- You may be in some state, and the next input character does not correspond to any of the transitions from that state.

Figure 7.13 is the corresponding state transition table for Figure 7.12. The state transition table for a simplified machine has no entry for a missing transition. Note that this table has no entry under the digit column for the current state of A. The remaining machines in this chapter are written in simplified form.

### Nondeterministic Finite State Machines

When you parse a sentence using a grammar, frequently you must choose between several production rules for substitution in a derivation step. Similarly, nondeterministic finite state machines require you to decide between more than one transition when parsing the input string. Figure 7.14 is a nondeterministic FSM to parse a signed integer. It is nondeterministic because there is at least one state that has more than one transition from it on the same character. For example, state A has a transition to both B and C on a digit. There is also some nondeterminism at state B because, given that the next input character is a digit, a transition both to B and to C is possible.

**Example 7.5** You must make the following decisions to parse +203 with this nondeterministic FSM:

- Current state: A, Input: +203, Scan + and go to B.
- Current state: B, Input: 203, Scan 2 and go to B.
- Current state: B, Input: 03, Scan 0 and go to B.
- Current state: B, Input: 3, Scan 3 and go to C.
- Current state: C, Input: Check for final state.

Because there is no more input and you are in the final state C, you have proven that the input string +203 is a valid signed integer.

When parsing with rules of production, you run the risk of making an incorrect choice early in the parse. You may reach a dead end where no substitution will get your intermediate string of terminals and nonterminals closer to the given string. Just because you reach such a dead end does not necessarily mean that the string is invalid. All invalid strings will produce dead ends in an attempted parse. But even valid strings have the potential for producing dead ends if you make a wrong decision early in the derivation.
The same principle applies with nondeterministic finite state machines. With the machine of Figure 7.14, if you are in the start state, A, and the next input character is 7, you must choose between the transitions to B and to C. Suppose you choose the transition to C and then find that there is another input character to scan. Because there are no transitions from C, you have reached a dead end in your attempted parse. You must conclude, therefore, that either the input string was invalid or it was valid and you made an incorrect choice at an earlier point.

Figure 7.15 is the state transition table for the machine of Figure 7.14. The nondeterminism is evident from the multiple entries (B, C) in the digit column. They represent a choice that must be made when attempting a parse.

**Machines with Empty Transitions**

In the same way that it is convenient to incorporate the empty string into production rules, it is sometimes convenient to construct finite state machines with transitions on the empty string. Such transitions are called *empty transitions*. Figure 7.17 is an FSM that corresponds closely to the grammar in Figure 7.2 to parse a signed integer, and Figure 7.16 is its state transition table.

In Figure 7.17, F is the state after the first character, and M is the magnitude state analogous to the F and M nonterminals of the grammar. In the same way that a sign can be +, −, or neither, the transition from I to F can be on +, −, or ε.

![Figure 7.15](image1.png)

**Figure 7.15**
The state transition table for the FSM of Figure 7.14.

![Figure 7.16](image2.png)

**Figure 7.16**
The state transition table for the FSM of Figure 7.17.

![Figure 7.17](image3.png)

**Figure 7.17**
An FSM with an empty transition to parse a signed integer.

*Machines with empty transitions are considered nondeterministic.*

**Example 7.6** To parse 32 requires the following decisions:

- Current state: I, Input: 32
  - Scan ε and go to F.
- Current state: F, Input: 32
  - Scan 3 and go to M.
- Current state: M, Input: 2
  - Scan 2 and go to M.
- Current state: M, Input: Check for final state.

The transition from I to F on ε does not consume an input character. When you are in state I, you can do one of three things: (a) scan + and go to F, (b) scan − and go to F, or (c) scan nothing (that is, the empty string) and go to F.

Machines with empty transitions are always considered nondeterministic. In Example 7.6, the nondeterminism comes from the decision you must make when you are in state I and the next character is +. You must decide whether to go from I to F on + or from I to F on ε. These are different transitions because they leave you with different input strings, even though they are transitions to the same state.