Context-Free Grammars with Multi-Letter Symbols
Martha Kosa

The theoretical ideas that you are studying have been used to help develop the compilers for the programming languages that you use today. Compilers are complex systems. They take the code that you write in your favorite language (Java, C++, Python, etc.) and translate it to an alternate form that can be executed by the computer. They check to see that your code obeys the rules of its language as an early step. You know about rules; they are the basis for the grammar for a language. Most features (but not all) of modern programming languages are context-free.

We share below links to information about the specifications for three widely used programming languages: C++, Java, and Python (in alphabetical order for neutrality, even though JFLAP was written in Java ☺).

https://isocpp.org/std/the-standard provides information about the C++ language standard.

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf provides information about C++ Language Specification. The discussion starts at page 1161, and the grammar starts at page 1269.

http://docs.oracle.com/javase/specs/jls/se7/html/jls-2.html provides information about the Java language specification.

https://docs.python.org/3/reference/grammar.html provides information about the Python language specification.

Each of these languages has hundreds of grammar rules; they need to specify the rules for classes, methods, variable declarations, constants, assignment statements, loops, and decision statements. JFLAP was not designed to handle huge grammars, so we will explore simple subsets of the languages.

One of the first things you did in your beginning programming class was to learn how to use variables to store data. You had to determine the variable's type (integer, float, etc.) and give the variable a name. You also needed to initialize the variable's value before using the variable. You could declare as many variables as you needed. We will keep things simple. We will assume one types: **int**. We will allow variable names consisting of at least one lowercase letter chosen from the set {a,b}. We will assume that variables will be initialized to constants containing at least one decimal digit; to shorten our grammar, assume the digits are in the set {0,1,2}.

Since we are allowing multiple variable declarations separated by semicolons (C++/Java-style punctuation), we will need a recursive grammar rule. Since we are not required to

use variables in our programs (although we wouldn't be able to do a lot of interesting things), our grammar can generate the empty string λ.
as
Try It!

1. Start JFLAP 8.0.
2. Select ***Help > Preferences ...*** to begin.
3. If the **Multi-Char Default Mode** radio button is not selected, select it.
4. Click the **Done** button.
5. From the main JFLAP GUI, click the **Grammar** button.
6. Enter two rules: **S** λ T;**S** and **S** λ λ.
7. T will be responsible for generating an individual assignment statement. Its rule will have a declaration part preceding an assignment part. The declaration part begins with **int** and ends with a variable name. The assignment part begins with = and ends with an integer constant.
8. It may be helpful to introduce two new nonterminal symbols D and A. Enter the T rule.
9. D will be responsible for generating a declaration. A declaration must start with **int** and end with a variable name. It may be helpful to introduce a new nonterminal symbol V. Enter the D rule.
10. If your terminal alphabet T does not contain **int** as a symbol, you may need to edit your terminal symbols. You can select a terminal symbol and modify it by typing your desired character sequence. You may need to edit the righthand side of associated grammar rules to fix things.

Your grammar so far should look similar to the following:

1. Now we can do the assignment part. An assignment must start with a = and end with an integer constant. It may be helpful to introduce a new nonterminal symbol C. Enter the A rule.
2. What have we not done yet? We need V rule(s) and C rule(s). Recursion will be involved because the variable names can have arbitrary positive length and the constants can have arbitrary positive length. The only constant which should begin with a 0 is 0 itself. You can introduce new nonterminal symbols as necessary. Enter the V and C rules and any additional rules with your new nonterminal symbols.

Your grammar should look similar to the following:

1. Save your grammar with a descriptive file name.

Now we will do some parsing of valid and invalid strings.

Try It!

1. In the current version of JFLAP 8.0, ***Input > User Control Parse*** is disabled, so we will select ***Input > Brute Force Parse*** to practice with parsing.
2. Enter **int a = 0;** in the **Input** text field and click the **Set** button or press the **Enter** key.
3. Click the **Step** button until you see notification that the input string has been accepted. If you are impatient, you can click the **Complete** button.
4. How many current derivations are available at the end? Why?
5. You can change to **Derivation View** from **Brute Parse Table** in the combo box to produce the parse tree or derivation table via the choice of the tabbed pane.
6. Click the **Change** button and update the **Input** text field to contain **int a = 01;** and repeat the parsing process.
7. How many current derivations are available at the end? Why?
8. Click the **Change** button and update the **Input** text field to contain **int a = 10;** and repeat the parsing process.
9. Click the **Change** button and update the **Input** text field to contain **int ab = 10;** and repeat the parsing process.
10. Click the **Change** button and update the **Input** text field to contain **int a = 0; int ab = 10;** and click the **Set** button or press the **Enter** key.
11. How many sentential forms were generated before the string was accepted?
12. Try some other valid and invalid strings. You may exhaust the resources of JFLAP's brute force parser. Real compilers do not perform brute force parsing; they use much more efficient algorithms.

We have just gotten a taste of what is involved when parsing assignment statements. What about some other statements? Decisions are very important in computer programs. We use **if-else** statements to handle decision making. An **if-else** statement starts with the keyword **if**, followed by a condition enclosed in a set of parentheses (for simplicity, we are assuming the keyword **cond** for condition) and a statement list afterward; remember that in a running program, the condition is evaluated, and the statement list is executed only when the condition is true. Next follows a placeholder for an optional else part to be executed if the condition is false. This part is either just the keyword **fi** (not valid in C++/Java/Python) or the keyword **else** followed by a statement list and finally the keyword **fi**. With **fi**, we don't need curly braces for grouping. A statement list is either empty or a statement followed by a statement list. For simplicity, we will assume two types of statements: **if-else** and the keyword **stmt** immediately followed by a semicolon (we know that this is not valid in C++/Java/Python). This description allows for nesting of **if-else** statements. Decisions can be chained together as needed to solve a problem.

Try It!

1. Create a new grammar file in JFLAP (make sure **Multi-Char Default Mode** is selected in ***Help > Preferences***).

2. Enter and edit rules as needed to produce the grammar shown below.
3.
4. Now let's do some parsing.
5.
6. What is the simplest string generated by the grammar? It will just be an **if** followed by **(cond)** followed by an empty statement list followed by **fi**. Verify that the described string is valid. Remember that brute force parsing is what is currently available in JFLAP 8.0.
7. What is the next simplest string generated by the grammar? It will be a simple modification of the above string that includes the keyword **else**. Verify that the described string is valid.

You should see something like the following for the derivation of the described string.

1. Let's make some nonempty statement lists. Parse the string **if (cond) stmt; stmt; else stmt; fi**.
2. How many sentential forms were generated before the string was accepted?
3. Let's do some nesting. Parse the string **if (cond) if (cond) stmt; else stmt; fi fi**.
4. How many sentential forms were generated before the string was accepted?
5. Remove a semicolon from the parse string. What happens?

Congratulations! You have explored building some context-free grammars with symbols consisting of multiple letters and parsing with them. These multi-letter symbols can represent keywords in programming languages. Hopefully you have a better appreciation for the compiler writers whose work allows you to write code in high-level languages.