Designing a grammar

Handout written by Maggie Johnson and revised by me.

Most students have the impression that grammars are useful only in programming languages and compilers. They may also have had some experience with them in theory/formal language courses (but most do not consider such experiences "useful").

In reality, grammars are a remarkably practical tool in certain situations. It's important to remember that a grammar captures a pattern in a precise and efficient form. Once we have a grammar representation of a set of patterns, we can do a lot with it. For example, we can devise automatic recognizers (parsers) which will tell us when a pattern has occurred. Or, we can perform an action when a particular pattern has been recognized.

So, in general, whenever one needs to recognize a pattern in some input, and then perform an action based on that recognition, a grammar can serve as a very efficient tool. As an example, any command interpreter that sits on top of an operating system (e.g., Unix, DOS, etc.) could easily be driven by a grammar.

Our purpose in this lecture is to help you build some skills in designing your own grammars. We will begin with some abstract examples, but we will end with some practical applications.

The Basic Process

To design a grammar, the first thing to do is to enumerate several examples of strings from the language that would be generated from the grammar. Once we have a set extensive enough to illustrate the embedded patterns, we do the following:

1) Recognize the pattern
2) Abstract it
3) Represent it as a grammar

Now, doesn't that sound easy? The first step is usually the easiest, if the set of enumerations is fairly comprehensive. The second step refers to taking a pattern found in step (1) and describing it in very precise terms. The third step takes the abstraction in step (2) and assigns terminal and non-terminal symbols to capture the elements of the abstraction. In the third step, we remember to use recursion for repetition of patterns and design new non-terminals to capture specific elements of patterns. Ideally, therefore each non-terminal serves a very specific role in terms of one or more patterns. These steps are done iteratively beginning at a high level, i.e., the start symbol and working down, one pattern at a time.

Let's apply this process to some abstract examples. Our alphabet is `{a, b}` for all these examples.

Example 1 Here is a partial set of strings from the language:

```
aba
abba
abbba
abbbba
...
```

Recognize the pattern and abstract it (i.e. describe it in precise terms):

Represent it as a grammar:

Example 2 Here is the set of enumerations:

a
b
aba
bab
aaa
bbb
aaaaa
bbbbb
abbba
babab
baaab
...

Recognize the pattern and abstract it (i.e. describe it in precise terms):

Represent it as a grammar:

Example 3 Here is the set of enumerations:

aa
aaab
aaaabb
aaaaabbb
aaaaaabbbb
...

Recognize the pattern and abstract it (i.e. describe it in precise terms):

Represent it as a grammar:

Example 4 Here is the set of enumerations:

∂
bb
aaa
abb
bab
abb
bbbb
bbaa
baba
abba
abab
...

Recognize the pattern and abstract it (i.e. describe it in precise terms):

Represent it as a grammar:

Example 5 Here is the set of enumerations (our alphabet has changed to {b, c, d, e, f}):

bccdef
bccdeeff
bccdeeefff
bbccccdeeff
bbccccdeeefff
bbbccccccdeeefff
...

Recognize the pattern and abstract it (i.e. describe it in precise terms):

Represent it as a grammar:

Abstract grammars like these are usually much more difficult to define than those needed in more practical applications. But being able to define abstract grammars is key to building the required skills. The process is more focused in these types of situations, making it easier to see the role that recursion and non-terminals play. Also, when dealing with grammars in formal languages we must

define grammars that not only generate a specific set of strings, but also exclude strings not in the set. We usually do not worry as much about exclusion in practical applications (like a compiler) because we have other techniques for dealing with these issues.

Let's move on and look at the application of our basic process in some practical examples.

Practical Applications

Example 6 Let's think of a favorite command-line interface and define a very simple set of commands for interfacing with the UNIX shell. Here are some examples of all the possible commands:

```
cd dir
cd /dir
cd dir1/subdir1
cd /dir1/subdir1
cd /dir1/subdir1/subdir2/subdir3....

mkdir dir
mkdir /dir
mkdir dir1/subdir1
mkdir /dir1/subdir1
mkdir /dir1/subdir1/subdir2/subdir3....

rmdir path

rm path1/file1
rm path1/file1, path2/file2, path3/file3
rm *

cp source_path dest_path
cp * dest_dir

mv source_path dest_path
mv * dest_dir
```

Let's say directory and file names may not include spaces to allow us to tokenize using whitespace as delimiters.

Recognize the pattern and abstract it (i.e. describe it in precise terms):

Represent it as a grammar:

Example 7 Another case where parsing techniques can come in handy is when working with various standardized protocols, such as FTP, HTTP, or MIDI. For example, IMAP is a protocol that specifies the valid commands and responses that can be sent to a mail server. Whether you were writing the client or the server end of the connection, you would find that you have a lot of pattern matching to do in recognizing and reacting to the various data received from the other end.
For example consider these possible commands that can be sent to fetch information about the messages on an IMAP server:

```
1 FETCH 1 (FLAGS)
2 FETCH 1,3, 10 (BODY)
3 FETCH 1:100 (BODY[HEADER])
4 FETCH 1:5, 15, 25 (FLAGS BODY[TEXT])
5 FETCH 1:* (BODY[HEADER] FLAGS UID)
```

All IMAP commands must begin with a command identifier (usually a number). The fetch command has a variety of possibilities for how you specify which messages to fetch. The entries in the parenthesis indicate which part of the message to fetch (headers, body, flags, etc.)

Recognize the pattern and abstract it (i.e. describe it in precise terms):

Represent it as a grammar:

Example 8 In many cases, when an application requires a lot of tedious and repetitive code, it is faster and easier to design a special purpose language, and write a little compiler that translates, e.g., the special purpose language into C. For example, consider a menu-generation language that allows a user to specify the layout and items in a series of menus, as well as what procedures are called when a menu item is chosen. Such a description can be placed in a file, which a little compiler parses and translates to C. When we compile and run the C code, the menu is executed.
Here is a sample description file:

```
begin MainMenu
title "Welcome to my calendar!"
title "Main menu"

item "List today's events"
command "list"
action execute list-today

item "Add new event"
command "add"
action execute add-event

item "Delete event"
command "delete"
action execute delete-event

item "Quit"
command "quit"
action execute quit

end
```

This will result (after parsing, compiling, and then compiling the resulting C program) in the following menu:

```
Welcome to my calendar!
Main menu

1) List today's events
2) Add new event
3) Delete event
4) Quit
```

If the user presses 1, or enters the command "list", the function "list-today" is executed. The idea is the generated menu code takes care of numbering of commands, matching of numbers and mnemonics to the appropriate action, rejecting invalid input, and so on. When you need to make a change to your menus, you go back to the description file, make changes, and then parse the file and generate a new set of menu routines. Voila!

A menu description consists of:

- A name for the menu screen
- A title or titles
- A list of menu items each consisting of:
  item
  [command]

action

where “item” is the text string that appears on the menu, “command” is the mnemonic used to provide command-line access, “action” is the function to be performed.. Brackets indicate optional items.

- A terminator

Our job is to define a grammar for this description language. To design such a grammar, we need to understand the types of patterns we might encounter in the description file.

At the highest level, what can we expect to encounter in the file?

As we progress through the parts of the menu specification, what other patterns do we expect to find?

Define a grammar for this menu-generation language.

The next step once we have our grammar is to embed “actions” into the productions to generate the C-code that will place the menu on the screen and execute the right functions when menu items are chosen. Learning how to do this is coming up in the next lectures.

Inclusion versus exclusion

Thus far, we have been able to define very “tight” grammars, meaning the grammar defines not only the valid sentences/structures in a language, but also restricts the generation of invalid structures. We were able to do this because the applications were quite simple.

In designing grammars for programming languages or natural languages, it is very difficult to define tight grammars. It’s not so bad coming up with a grammar that generates all valid constructs, but when one starts modifying this grammar to restrict the generation of invalid constructs, the grammar can grow exponentially. Not to mention, a little change here and a little change there in order to <u>not</u> allow a particular invalid construct may all of a sudden disallow certain valid constructs.

When it comes down to this, it is far more important to have a grammar that generates all possible valid constructs. It is also important to remember that in a compiler, there are many opportunities to find errors in a source program during semantic analysis. It is simply not necessary to build a lot of special error-checking into the grammar itself. It's most important to keep the grammar small and comprehensible, and be certain it generates all valid constructs.

Bibliography

Levine, Mason and Brown: *Lex and YACC*, O'Reilly and Associates, 1992.