Process Supplement: BNF grammars

The input language for `sh61` command lines is described in terms of a **BNF grammar**, where BNF stands for Backus–Naur Form, or Backus Normal Form. BNF is a declarative notation for describing a **language**, meaning simply a set of strings. BNF notation is built from:

- **Terminals**, such as `"x"`, which must exactly match characters in the input.
- **Nonterminals** (or **symbols** for short), such as `lettera`, which represent sets of strings. One of the nonterminals is called the **root** or **start symbol** of the grammar. By convention, the root is the first nonterminal mentioned in the grammar.
- **Rules**, such as `lettera ::= "a"` or `word ::= letter word`, which define how nonterminals and strings relate.

A string is in the language if it can be obtained by following the rules of the grammar starting from a single instance of the root symbol.

It's useful to be able to read BNF grammars because they're very common in computer science and programming. Many specification documents use BNF to define languages, including programming languages, configuration files, and even network packet formats. Any context-free language can be described using a BNF grammar.

Example

This grammar that can recognize exactly two strings, namely `"cs61 is awesome"` and `"cs61 is terrible"`:

```
cs61 ::= "cs61 is " emotion
emotion ::= "awesome" | "terrible"
```

The vertical bar, `|`, is shorthand for alternate definitions of a nonterminal, so this defines the exact same language. (A given language may be defined by many grammars.)

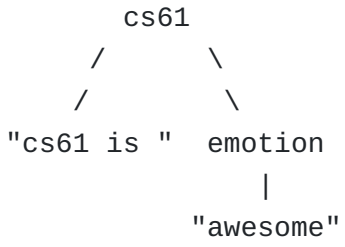
```
cs61 ::= "cs61 is " emotion
emotion ::= "awesome"
```

```
emotion ::= "terrible"
```

The language includes `"cs61 is awesome"` because:

- `"cs61 is awesome"` is the same as `"cs61 is " "awesome"` (two concatenated strings).
- The `emotion ::= "awesome"` rule means that `"cs61 is awesome"` can be written as `"cs61 is " emotion`.
- That, in turn, matches the single rule for the `cs61` start symbol.

We can visualize this derivation as a **parse tree** that shows how the string is formed. The start symbol is at the top; each tree branching point expands one nonterminal using a rule.



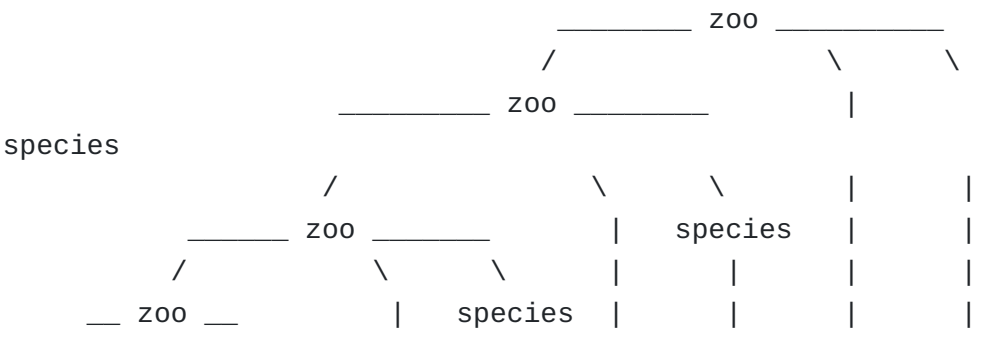
The language does not include `"cs121 is great"` or `"cs61 is meh"`; there is no way to build up either of those strings using the grammar.

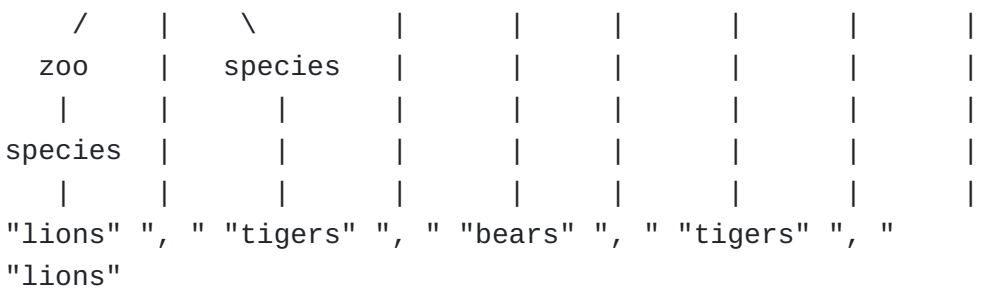
Recursive example

Grammars often contain recursive rules. This grammar recognizes one or more kinds of zoo animal, separated by commas:

```
zoo     ::= species
         |  zoo ", " species
species ::= "lions" | "tigers" | "bears"
```

The string `"lions, tigers, bears, tigers, lions"` is in the language because of the following parse tree:



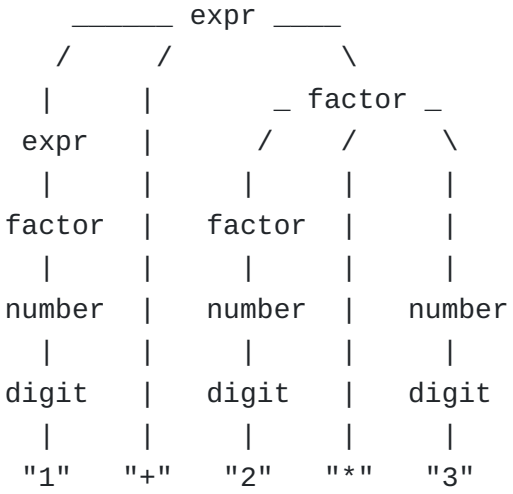


Precedence example

Grammars can define the precedence of infix operators. For example, the "PEMDAS rule" says that Multiplication has higher precedence than Addition, so that the expression "1+2*3" equals 7, not 9 ("1+(2*3)", not "(1+2)*3"). This grammar makes that precedence explicit:

```
expr   ::= factor
         |  expr "+" factor
factor ::= number
         |  factor "*" number
number ::= digit
         |  number digit
digit  ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" |
"8" | "9"
```

The grammar represents the string `1+2*3` as `1` plus `2*3`, because the `2*3` part is grouped together into a single nonterminal.



Exercises

> **EXERCISE.** Define a grammar for the **empty language**, which is a language containing no strings. (There is no valid sentence

in the empty language.)

```
empty ::=
```

This grammar has one nonterminal, the root symbol `empty`. The `::=` notation has nothing on the right, indicating that no string will match `empty`.

Hide solution

EXERCISE. Define a grammar for the **children language** (because children should be "not heard"). The empty string is the only valid sentence in this language.

```
children ::= ""
```

Note the difference between `children` and `empty`. The `empty` language contains no strings, not even the empty string. The `children` language contains one string, the empty string `""`.

Hide solution

EXERCISE. Define a grammar for the **letter language**. A letter is a lower-case Latin letter between `a` and `z`.

```
letter ::= "a" | "b" | "c" | "d" | ... | "z"
```

The `|` symbol indicates alternate choices for the rule's definition. It's actually shorthand for multiple rules for the same nonterminal:

```
letter ::= "a"
letter ::= "b"
letter ::= "c"
letter ::= "d"
...
letter ::= "z"
```

Hide solution

EXERCISE. Define a grammar for the **word language**. A word is a sequence of one or more letters. You may refer to the `letter` nonterminal defined above.

```
word ::= letter | word letter
```

This grammar features a recursive rule! Why is `"butt"` in this language? Well:

- `"b"` is a `word` (a sentence in the word language) because it is a `letter`. (This uses the first alternative, `word ::= letter`.)
- `"bu"` is a `word` because `"b"` is a word (see above) and `"u"` is a letter. (This uses this second alternative, `word ::= word letter`.)
- `"but"` is a `word` because `"bu"` is a word and `"t"` is a letter.
- `"butt"` is a `word` because `"but"` is a word and `"t"` is a letter.

Hide solution

EXERCISE. Define a **different** grammar for the word language.

```
word1 ::= letter | word1 letter

word2 ::= letter | letter word2

word3 ::= letter | word3 word3
```

Each of these grammars recognizes the same language: every string that matches the `word1` nonterminal matches `word2` and `word3`, and every string that doesn't match `word1` doesn't match `word2` or `word3` either.

It is possible to prove that these simple grammars recognize the same language, but in general, testing whether two context-free grammars recognize the same language is **undecidable**! There is **no** algorithm that can reliably report whether two grammars are effectively the same. If this blows your mind (and it should), take CS121.

Hide solution

EXERCISE. Define a grammar for the **parenthesis language**, where all strings in the parenthesis language consist of balanced pairs of left and right parentheses. Some examples:

Valid	Invalid
`()`	`())(`
`(())`	`(`
`()()`	`)(`
`((()())()())`	`))))))))))(`

An important question is, is the empty string a member of this language? Let's say yes. Then this works:

```
parens ::= "(" parens ")"
         |  parens parens
         |  ""
```

Note that the following grammar accepts no finite strings:

```
parens2 ::= "(" parens2 ")"
```

This is because this recursion has no base case. (You could argue that the infinite string consisting of an infinite number of `(`s, followed by an infinite number of `)`s, would match the grammar, but also you could argue that it would not, because infinity is weird.)

If the empty string is *not* a member of the language, then we need a different base case:

```
parens3 ::= "(" parens3 ")"
          |  parens3 parens3
          |  "()"
```

The balanced parenthesis language is context-free, but not regular. In practical terms, this means that no regular expression can recognize the language. Regular expressions are too limited to remember a nesting depth.

Hide solution

Shell grammar

This is the BNF grammar for `sh61`, as found in problem set 5.

```
commandline ::= list
          |  list ";"
          |  list "&"

list     ::=  conditional
          |   list ";" conditional
          |   list "&" conditional

conditional ::=  pipeline
          |   conditional "&&" pipeline
          |   conditional "||" pipeline

pipeline ::=  command
          |   pipeline "|" command

command  ::=  word
          |   redirection
          |   command word
          |   command redirection

redirection  ::=  redirectionop filename
redirectionop  ::=  "<"  |  ">"  |  "2>"
```

This grammar doesn't define the `word` nonterminal, and doesn't define where whitespace is allowed or required. Such looseness is not atypical in grammars; in the case of the shell, our parsing functions in `helpers.cc` implement the necessary rules. Shorthand such as `command ::= [word | redirection]...` is also common.

The recursive definitions specify that:

- A `list` is one or more `conditionals`, separated by `;` or `&`.
- A `conditional` is one or more `pipelines`, separated by `&&` or `||`.
- A `pipeline` is one or more `commands`, separated by `|`.