Languages: syntax and parsing

Glenn Bruns
CSUMB

Write a JSON parser

```
{
   "1": {
      "location": "assign-ostep7",
      "weight": 20
   },
   "2": {
      "location": "assign-ostep8",
      "weight": 20
   },
   "3": {
      "location": "assign-msh3",
      "weight": 60
   }
}
```

Could you do it?

Someone asks you to build a JSON parser in Python.

- it takes as input some JSON
- it returns a Python data structure
- your code should report errors in the JSON

Lecture Objectives

When this lecture is finally over, you should be able to:

- Define the elements of a BNF grammar
- Write a simple BNF grammar
- Show whether a string can be derived from a grammar

Let's play the Backus-Naur Form game

BNF is a language for defining syntax

```
A ::= foo | bar
```

From this rule we can derive the strings "foo" and "bar"

```
Derive foo:

A → foo
```

Let's play some more

```
A ::= [ B ]
B ::= a | b | c
```

(start with A)

Derive [a]:

A → [B] → [**a**]

Derive [c]:

A → [B] → [**c**]

Let's play some more

```
A ::= "" | a A b
```

Derive "a b":

A →
a A b →
a b

Derive "a a b b":

A →
a A b →
a a A b b →
a a b b

Derive "a a b b b":

impossible!

Let's play some more

```
A ::= "" | a , A
```

Derive "":

A → ""

Derive "a ,":

A →

a , A →

a ,

Derive "a , a":

A →

a , A →

a , **a** , A →

a , **a** ,

Doesn't work!

Let's play some more

```
expr ::= expr + expr | 1 | x
```

Derive "1 + x":

```
expr →
expr + expr →
1 + expr →
1 + x
```

Derive "1 + 1":

```
expr →
expr + expr →
1 + expr →
1 + 1
```

Derive "x + 1 + x":

```
expr →
expr + expr →
expr + expr + expr →
x + expr + expr →
x + 1 + expr →
x + 1 + x
```

Let's play the BNF-writing game

Write a rule that can derive only "a", "(a)", and "((a))"

```
expr ::= a | ( a ) | ( ( a ) )
```

Let's play some more

Write a rule that can derive only "a", "(a)", "((a))", …

```
expr ::= a | ( expr )
```

Let's play some more

Write a rule that can derive a simple regular expression containing any number of these elements: ".", "[a]", and "*"

```
regex ::= "" | regex . | regex [ a ] | regex *
```

Rules of the game

```
expr ::= expr + expr | ( expr ) | num
num ::= digit | digit num
digit ::= 0 | 1 | 2 | 3
```

- non-terminals: symbols on the left of a rule
- terminals: the other symbols (like digit)
- On the right of each rules are one or more productions, separated by "|"

The rules to derive a string from some productions:

1. start with the first non-terminal (the start symbol)
2. replace it with one of its productions
3. keep replacing a non-terminal with one of its productions until only terminals remain

BNF Grammars

```
expr ::= expr + expr | ( expr ) | num
num ::= digit | digit num
digit ::= 0 | 1 | 2 | 3
```

(an example grammar)

A BNF Grammar consists of:

- some non-terminals (including the start non-terminal)
- some terminals
- one or more productions for each non-terminal

The language of a BNF Grammar:

- all the strings you can derive from the grammar

Summary

- The syntax of a language defines what it means for a phrase in the language to be "well formed"
- We will use BNF grammars to define language syntax