

CS 131 Week 3 Worksheet Answers

(25 min) Let's Review Together

When reviewing for any class, one effective study method is the concept of [spaced repetition](#) which is when you review material at periodic, increasingly spaced intervals. For this exercise, you will spend some time reviewing past content and solidifying your understanding of it while prepping for the exam!

Part 1: Compiling Past Notes

First, take a few minutes to review the past 2 weeks of lecture and compile a list of key terms or concepts that you believe to be most important. Then, write a summary or a definition of those terms and concepts. Add examples where appropriate. Ideally, this should be a resource that you can reference later on, so make it organized and comprehensive (i.e. during the midterm). If you are working in a group (say in discussion), I would recommend splitting this task up in some way! For discussion today, feel free to make a loose/outline version of this, then flesh it out later.

Example 1: Person 1 does lecture 1 and 2, Person 2 does lecture 3, Person 3 and 4 split lecture 4

Part 2: Write Practice Questions

For the exercise, come up with 2-3 sample questions based on your list from above. Then, come up with answers for them. Here are some sample question styles that you can use. Feel free to try answering these too.

- **Conceptual:** An open-ended question that tests a person's understanding of a certain concept. Usually requires a person to reason beyond what was taught in lecture by extrapolating from basic ideas.

Example: What are the implications of OCaml not having side effects? Provide an example of behavior that is possible

The lack of side effects makes concurrency difficult in OCaml. For example, say there is a shared resource X that is used by two threads. In a language like C++, thread A could increment X (a side effect) and thread B could see the updated value. However, this is not possible in OCaml. Concurrent reads are allowed but concurrent updates cannot occur.

- **Compare and Contrast:** Similar to a conceptual question but more focused in scope. Usually worth fewer points on an exam.

Example: What is the difference between a left and right recursive rule in a context-free grammar?

```
term: term '*' factor      { /* left-recursive */ }
assignment: lval '=' assignment { /* right-recursive */ }
```

See:

<https://stackoverflow.com/questions/32123003/difference-between-left-right-recursive-left-right-most-derivation-precedence>

- **Knowledge Check:** The simplest kind of question. This question is generally closed and used to verify that a person understands a basic concept. These review questions are great for initial studying when you are first reviewing a concept!

Example: True or false? The presence of blind-alley rules in a grammar G alters the language defined by G.

False. Blind Alleys waste time (potentially infinitely) because a parser using a blind alley rule will never derive a list of terminals. However, the language defined by G does not change.

- **Technical:** As the name implies, this tests your technical ability to parse a grammar, write OCaml, etc.

Example: If f has type $a \rightarrow (b \rightarrow c) \rightarrow d$, and $f\ x\ y$ typechecks, what is the type of x ? y ? $f\ x$?
 x is a , y is $b \rightarrow c$, $f\ x$ is $(b \rightarrow c) \rightarrow d$

- **Your turn: Write 2-3 sample questions of any style.**

Part 3: Learning Together

If you are working in a group, share your questions with each other and attempt to answer your groupmates' questions. Work together! Guide your peers on how to work through your questions if they struggle. Explaining and teaching concepts is a great way to solidify your own understanding.

(10 mins) Railroad Diagrams

Eggert once referred to these as diagrams "in the style of Webber" 🤔 so remember that if you don't read the textbook.

[JSON](#) is a popular data-interchange format. It's also fairly simple to parse.

Today we'll look at objects in JSON, which look like this:

```
{
  "latitude": 34.07099896148599,
```

```

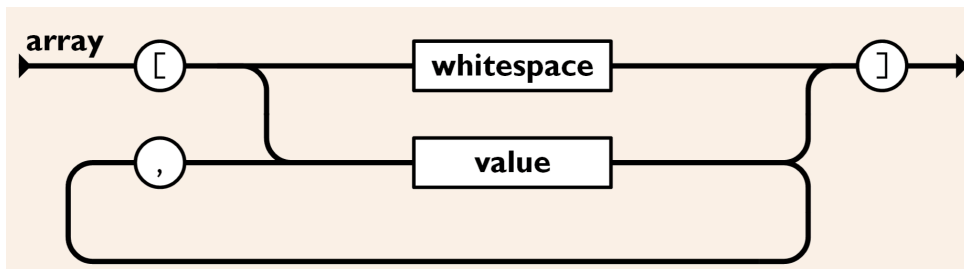
    "longitude": -118.44504522008893,
    "name": "Bruin Statue"
  }

  {
    "center": {
      "x": 5.0,
      "y": 0.0
    },
    "radius": 4.0
  }

  {
    "error": null,
    "response": [
      "878ec3ec",
      "c1a01689",
      "e956be4d",
      "1fc07907"
    ]
  }

```

As reference here is the railroad diagram for **array** from the [JSON website](#), which has a simpler grammar than **Object**.



Create a grammar and corresponding railroad diagram for **Object**. You can ignore whitespace and you can assume there are nonterminals **String** and **Value**. You can define more nonterminals if you want.

Remember objects can be empty: `{}`. Also remember that commas only go **between** attribute-value pairs.

```

Object -> '{' '}' | '{' Members '}'
Members -> Pair | Pair ',' Members

```

Pair -> String ':' Value

Past this point was not intended to be required in your answer, but for completeness,

Value -> String | Number | Object | Array | Null

Array -> '[' ']' | '[' Elements ']'

Elements -> Value | Value ',' Elements

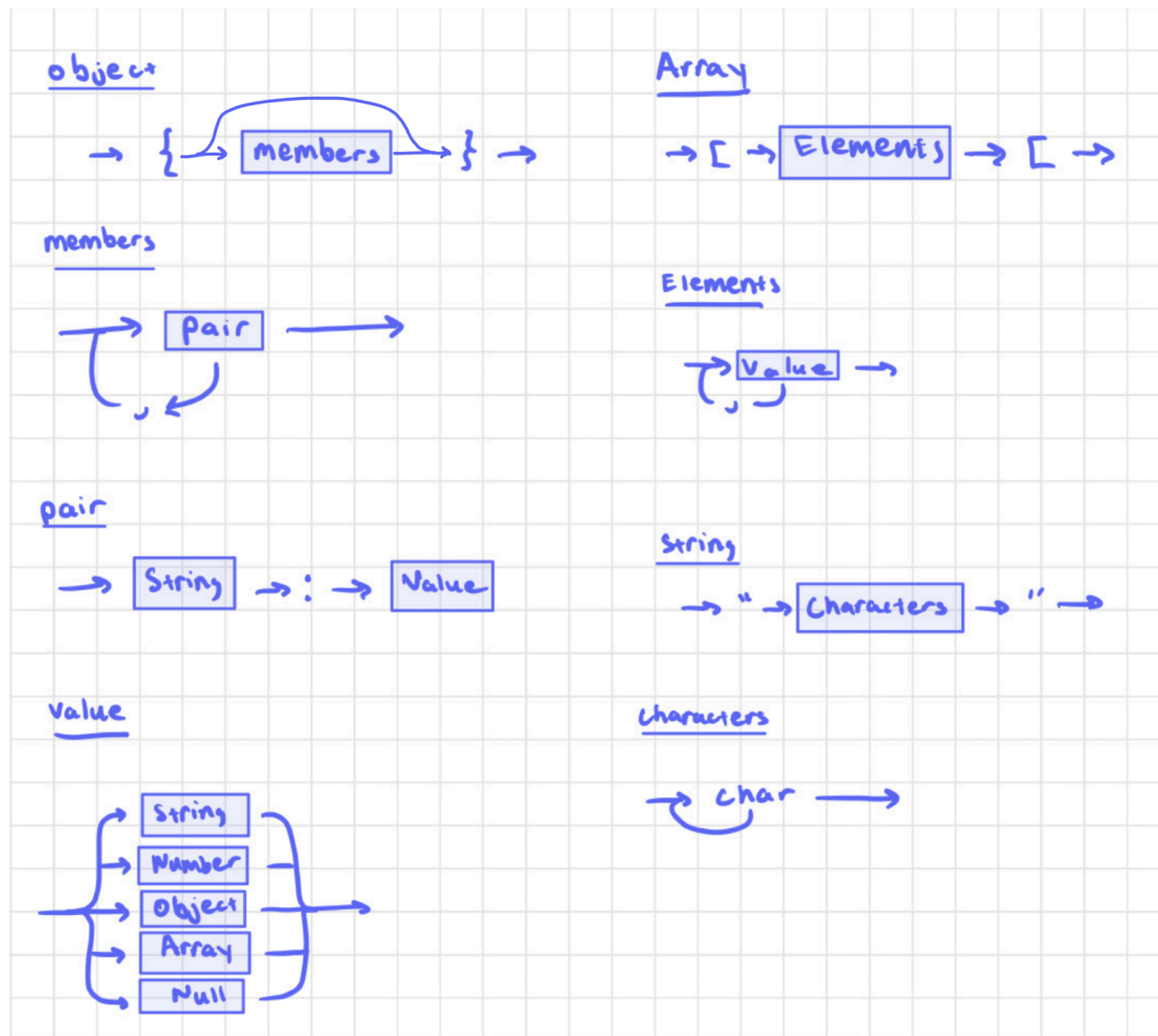
String -> '"' Characters '"'

Characters -> <nothing> | Char Characters

Char -> 'any Unicode character except " or \ or escape sequence'

Number -> 'integer or floating-point number'

Null -> 'null'



(10 mins) Ambiguous Grammars

Consider the following grammar.

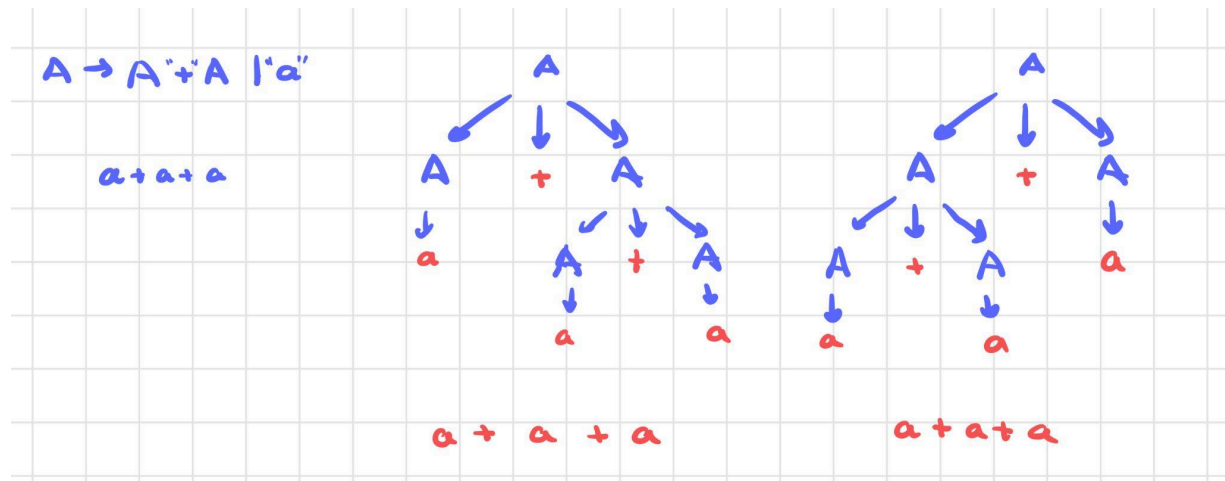
Start Symbol: A

$A \rightarrow A + A \mid "a"$

Part 1: Detecting Ambiguity

Come up with a string that has multiple derivations using this grammar. Draw the parse tree for at least 2 of the possible derivations to show the ambiguity.

$a + a + a$



Part 2: Fixing Ambiguity

How would you fix this? Rewrite rule A so that the ambiguity no longer exists.

$A \rightarrow "a" + A \mid "a"$



(15 mins) Another Ambiguous Grammar

This was a midterm practice problem we got in winter 2024.

Find the ambiguity in this grammar:

```
syntax = syntax rule, {syntax rule};
syntax rule = meta identifier, '=', definitions list, ';';
definitions list = single definition, {'|', single definition};
single definition = primary, {'.', primary};
primary = optional sequence | repeated sequence | special sequence
         | grouped sequence | meta identifier | terminal string | empty;
empty = ;
optional sequence = '[', definitions list, ']';
repeated sequence = '{', definitions list, '}';
special sequence = '?', {character}, '?';
grouped sequence = '(', definitions list, ')';
terminal string = '"', character, {character}, '"'
                | "'", character, {character}, "'";
meta identifier = letter, {letter | decimal digit};
```

The problem is with definitions list and terminal string, where the delimiter “|” is not excluded from “character” inside terminal string. Consider an input $X = "A|B"$, it is not clear if this means X refers to the terminal string $A|B$, or X is either terminal string A or terminal string B . Real working grammars should define special symbols to exclude the delimiters.

- Remember C++ use the escape sequence `\`

(10 mins) Using Grammars

You'll learn enough from CS 131 to implement a simple calculator!

Here is a grammar for basic arithmetic expressions:

Start symbol: `Expr`

`Expr -> Num`

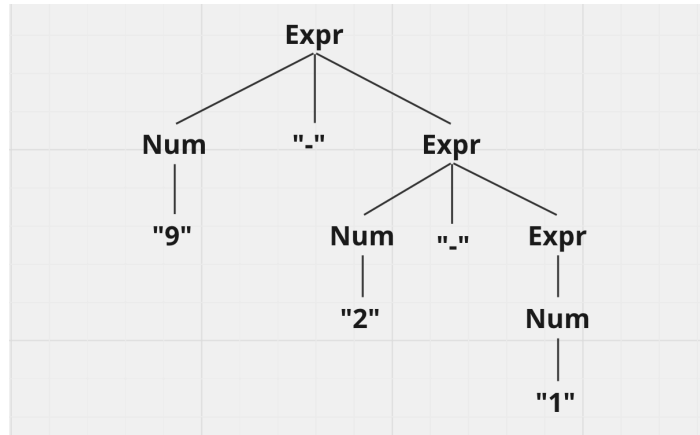
`| Num "+" Expr`

`| Num "-" Expr`

`Num -> "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"`

Draw the parse tree for the input `"9-2-1"`. Use paper if you want.

Solution:



Mathematically, what should 9 - 2 - 1 equal?

Solution: 6

What do you get if you "evaluate" the parse tree? That is, if each **Expr** with a "+"/"-" evaluates to the sum/difference of the values of its **Num** and **Expr** children, what does the root **Expr** evaluate to?

Solution: 8

How do you propose we fix this, so that the parse tree "evaluates" to the same thing as we mathematically expect? Multiple things could work.

(1) swap **Expr** and **Num** to have **Expr** "+"/"-" **Num**. Then the grammar will be left-recursive instead of right-recursive. Then the operators will be parsed as left-associative, which is the same way we treat them in math.

(2) use a different parsing algorithm altogether (e.g. shunting yard like from CS 32, Pratt parsing). See <https://matklad.github.io/2020/04/13/simple-but-powerful-pratt-parsing>.


(3) post-process the parse tree and rotate nodes to change associativity. (Apparently Haskell does something like this, since it allows user-defined operators with custom associativity/precedence at runtime: <https://news.ycombinator.com/item?id=40091238>)

Further Practice

Start Homework 2! Remember the hints document linked below.

We've had less time than ideal in discussions to share material with you, so please consider attempting some of Week 2's worksheet (particularly "More OCaml Practice"). Practicing OCaml and how to think functionally will more than pay off for HW2 and the midterm.

Homework 2 Hints

 W25 CS 131 Homework 2 Hints

LA Feedback

[LA Feedback Form W'25](#)