CS 61A Fall 2021

Regular Expressions, BNF, SQL

Discussion 12: November 17, 2021 Solutions

BNF

Backus-Naur Form (BNF) is a syntax for describing a context-free grammar. It was invented for describing the syntax of programming languages, and is still commonly used in documentation and language parsers. EBNF is a dialect of BNF which contains some convenient shorthands.

An EBNF grammar contains symbols and a set of recursive production rules. In 61A, we are using the Python Lark library to write EBNF grammars, which has a few specific rules for grammar writing.

There are two types of symbols: Non-terminal symbols can expand into non-terminals (including themselves) or terminals. In the Python Lark library, non-terminal symbols are always lowercase. Terminal symbols can be strings or regular expressions. In Lark, terminals are always uppercase.

Consider these two production rules:

```
numbers: INTEGER | numbers "," INTEGER
INTEGER: /-?\d+/
```

The symbol `numbers` is a non-terminal with a recursive production rule. It corresponds to either an `INTEGER` terminal or to the `numbers` symbol (itself) plus a comma plus an `INTEGER` terminal. The `INTEGER` terminal is defined using a regular expression which matches any number of digits with an optional - sign in front.

This grammar can describe strings like:

```
10
10,-11
10,-11,12
```

And so on, with any number of integers in front.

A grammar should also specify a start symbol, which corresponds to the whole expression being parsed (or the whole sentence, for a spoken language).

For the simple example of comma-separated numbers, the start symbol could just be the numbers terminal itself:

```
?start: numbers
numbers: numbers "," INTEGER | INTEGER
INTEGER: /-?\d+/
```

EBNF grammars can use these shorthand notations for specifying how many symbols to match:

EBNF Notation	Meaning	Pure BNF Equivalent
item*	Zero or more items	items: | items item
item+	One or more items	items: item | items item
[item] item?	Optional item	optitem: | item

Lark also includes a few handy features:

- You can specify tokens to complete ignore by using the ignore directive at the bottom of a grammar. For example, `%ignore /\s+/` ignores all whitespace (tabs/spaces/new lines).
- You can import pre-defined terminals for common types of data to match. For example, `%import common.NUMBER` imports a terminal that matches any integer or decimal number.

Q1: lambda BNF

We've written a simple BNF grammar to handle lambda expressions. The body of our lambda has to consist of a single expression, which can be a number, word, or another lambda expression.

```
?start: lambda_expression
lambda_expression:  "lambda " arguments ":" body
arguments: WORD ("," WORD)*
body: expression
?expression: value | lambda_expression
?value: WORD | NUMBER

%import common.WORD
%import common.NUMBER
%ignore /\s+/
```

For each of the given examples, draw the resulting tree created by this BNF.

```
lark> lambda x: 5
```

```
lambda_expression
  arguments  x
  body  5
```

```
lark> lambda x, y: x
```

```
lambda_expression
  arguments
    x
    y
  body  x
```

```
lark> lambda x: lambda y: x
```

```
lambda_expression
  arguments  x
  body
    lambda_expression
      arguments  y
      body  x
```

SQL

SQL is an example of a declarative programming language. Statements do not describe computations directly, but instead describe the desired result of some computation. It is the role of the query interpreter of the database system to plan and perform a computational process to produce such a result.

For this discussion, you can test out your code at sql.cs61a.org. The records table should already be loaded in.

Select Statements

We can use a **SELECT** statement to create tables. The following statement creates a table with a single row, with columns named "first" and "last":

```
sqlite> SELECT "Ben" AS first, "Bitdiddle" AS last;
Ben|Bitdiddle
```

Given two tables with the same number of columns, we can combine their rows into a larger table with UNION:

```
sqlite> SELECT "Ben" AS first, "Bitdiddle" AS last UNION
...> SELECT "Louis", "Reasoner";
Ben|Bitdiddle
Louis|Reasoner
```

We can SELECT specific values from an existing table using a FROM clause. This query creates a table with two columns, with a row for each row in the records table:

```
sqlite> SELECT name, division FROM records;
Alyssa P Hacker|Computer
...
Robert Cratchet|Accounting
```

The special syntax SELECT * will select all columns from a table. It's an easy way to print the contents of a table.

```
sqlite> SELECT * FROM records;
Alyssa P Hacker|Computer|Programmer|40000|Ben Bitdiddle
...
Robert Cratchet|Accounting|Scrivener|18000|Eben Scrooge
```

We can choose which columns to show in the first part of the SELECT, we can filter out rows using a WHERE clause, and sort the resulting rows with an ORDER BY clause. In general the syntax is:

```
SELECT [columns] FROM [tables]
WHERE [condition] ORDER BY [criteria];
```

For instance, the following statement lists all information about employees with the "Programmer" title.

```
sqlite> SELECT * FROM records WHERE title = "Programmer";
Alyssa P Hacker|Computer|Programmer|40000|Ben Bitdiddle
Cy D Fect|Computer|Programmer|35000|Ben Bitdiddle
```

The following statement lists the names and salaries of each employee under the accounting division, sorted in descending order by their salaries.

```
sqlite> SELECT name, salary FROM records
...> WHERE division = "Accounting" ORDER BY salary desc;
Eben Scrooge|75000
Robert Cratchet|18000
```

Note that all valid SQL statements must be terminated by a semicolon (;). Additionally, you can split up your statement over many lines and add as much whitespace as you want, much like Scheme. But keep in mind that having consistent indentation and line breaking does make your code a lot more readable to others (and your future self)!

Questions

Q2: SELECTs in BNF

Let's write a BNF grammar that describes `SELECT` statements in SQL. Your grammar should support the following:

- selecting one or more columns from a single table
- an optional `WHERE` clause
- any number of additional `AND` clauses if a `WHERE` clause is present
- the `WHERE` and `AND` clauses only need to support comparisons between column(s) and numbers

  The SQLite documentation actually uses BNF via railroad diagrams, which are a way of representing the grammar. Check out the diagram for a complete `SELECT` statement on the SQLite site here.

```
?start: select_statement
select_statement: "SELECT " columns "FROM" table ("WHERE" condition
    ("AND" condition)*)? ";"
columns: (WORD ",")* WORD
table: WORD
condition: expr COMPARATOR expr
?expr: WORD | NUMBER
COMPARATOR: "<" | ">" | "=" | ">=" | "<=" | "!="

%doctest
lark> SELECT name, age FROM cats
....> WHERE age > 3 AND lives > 5 AND tail = 1;
select_statement
  columns
    name
    age
  table  cats
  condition
    age
    >
    3
  condition
    lives
    >
    5
  condition
    tail
    =
    1
%end
%import common.WORD
%import common.NUMBER
%ignore /\s+/
```

SQL Queries

For the following questions, you will be referring to the **records** table:

Name	Division	Title	Salary	Supervisor
Alyssa P Hacker	Computer	Programmer	40000	Ben Bitdiddle
...	...	...	...	...

Q3: Oliver Employees

Write a query that outputs the names of employees that Oliver Warbucks directly supervises.

```
SELECT name FROM records WHERE supervisor = "Oliver Warbucks";
```

Q4: Self Supervisor

Write a query that outputs all information about employees that supervise themselves.

```
SELECT * FROM records WHERE name = supervisor;
```

Q5: Rich Employees

Write a query that outputs the names of all employees with salary greater than 50,000 in alphabetical order.

```
SELECT name FROM records WHERE salary > 50000 ORDER BY name;
```

Regular Expressions

Q6: Email Domain Validator

Create a regular expression that makes sure a given string `email` is a valid email address and that its domain name is in the provided list of `domains`.

An email address is valid if it contains letters, number, or underscores, followed by an @ symbol, then a domain.

All domains will have a 3 letter extension following the period.

Hint: For this problem, you will have to make a regex pattern based on the inputs `domains`. A for loop can help with that.

Extra: There is a particularly elegant solution that utilizes join and replace instead of a for loop.

Note: The skeleton code is just a suggestion; feel free to use your own structure if you prefer.

```
import re
def email_validator(email, domains):
    """
    >>> email_validator("oski@berkeley.edu", ["berkeley.edu", "gmail
    .com"])
    True
    >>> email_validator("oski@gmail.com", ["berkeley.edu", "gmail.
    com"])
    True
    >>> email_validator("oski@berkeley.com", ["berkeley.edu", "gmail
    .com"])
    False
    >>> email_validator("oski@berkeley.edu", ["yahoo.com"])
    False
    >>> email_validator("xX123_iii_OSKI_iii_123Xx@berkeley.edu", ["
    berkeley.edu", "gmail.com"])
    True
    >>> email_validator("oski@oski@berkeley.edu", ["berkeley.edu", "
    gmail.com"])
    False
    >>> email_validator("oski@berkeleysedu", ["berkeley.edu", "gmail
    .com"])
    False
    """
    pattern = r"^\w+@("
    for domain in domains:
        if domain == domains[-1]:
            pattern += domain[:-4] + r"\." + domain[-3:] + r")$"
        else:
            pattern += domain[:-4] + r"\." + domain[-3:] + "|"
    return bool(re.search(pattern, email))
    # Alternate, elegant solution
    domains_list = "|".join([domain.replace(".", "\.") for domain in
     domains])
    return bool(re.search(rf"^\w+@({domains_list})$", email))
```