Universität Bremen

Praktische Informatik 3 WS 22/23

Christoph Lüth
Raphael Baass
Thomas Barkowsky
Tede von Knorre
Alexander Krug
Tarek Soliman

9. Übungsblatt

Ausgabe: 16.01.2023
Abgabe: 30.01.2023

Datenbanken sind der Kern der Informatik, sagen Datenbanker. Das mag etwas übertrieben sein, aber zumindest handelt es sich dabei um amtliche Systeme, die auf kellerfüllenden Mainframes laufen und in klimagekühlten Rechenzentren die Lagerbestände von Amazon, die Großkundendaten der Deutschen Bank oder die Steuerdaten der Bundesrepublik Deutschlands sicher verwalten.

Ein etablierter Standard für Datenbankanfragesprachen ist SQL, welchem wir uns in diesem Übungsblatt zuwenden wollen. Dabei wird Ihnen die eigentliche Datenbankfunktionalität zur Verfügung gestellt, Ihre Aufgabe ist es, einen Parser für Datenbankanfragen und ein Webinterface dazu entwickeln.

Vorbemerkungen

Parser für nicht-triviale Sprachen funktionieren meist in zwei Stufen:

1. In einer ersten Stufe wird aus einer Eingabe (meist eine Zeichenkette) ein Strom von *Tokens* erzeugt.
2. In einer zweiten Stufe wird dann aus einer Sequenz von Token eine Repräsentation der Zielsprache (*abstrake Syntax*) geparst.

In Haskell lassen sich Parser elegant und effizient mit Funktionen höherer Ordnung implementieren. Wir haben in `Parser.hs` eine Bücherei zur Implementierung von Parsern vorgegeben. Im Verzeichnis `examples/` der Vorlage gibt es einen Beispielparser für einfache Ausdrücke, wie wir sie in der letzten Vorlesung ausgewertet haben. Die Datei `README.md` in dem Verzeichnis erläutert die Grundlagen des Kombinatorparsings anhand des dortigen Beispiels. Bitte arbeiten Sie das `README` und die Quellen in dem Verzeichnis dort sorgfältig durch, sie sind die Grundlage für dieses Arbeitsblatt.

9.1 *Lexer* — *5 Punkte*

Der Lexer (oder Tokenizer) zerlegt den Eingabestrom in eine Sequenz von Token. Für unsere Sprache benötigen wir folgende Token:

- Schlüsselworte (`CREATE`, `DROP`, usw.) und Zeichen (`(`, `)`, `,` usw.) der Sprache;
- ganze Zahlen;
- Zeichenketten;
- Bezeichner: Tabellennamen (starten mit einem Großbuchstaben, gefolgt von beliebig vielen Buchstaben) und Attributnamen (starten mit einem Kleinbuchstaben, gefolgt von beliebig vielen Buchstaten);
- Datumsangaben (in der Form `dd-mm-yyyy`).

Tokens werden durch den Datentyp `Token` in `Frontend.Tokenizer` repräsentiert; die genaue Lexikalik findet sich im Anhang dieses Aufgabenblattes.

Implementieren Sie einen Lexer

```
lexer :: Parser Char [Token]
```

welcher den Eingabestrom in eine Sequenz von Token zerlegt. Dabei sollen Leerzeichen, Zeilenumbrüche und Zeilenvorschübe wegfallen (es sei denn, sie finden sich innerhalb von Zeichenketten). In der Datei `Frontend/Tokenizer.hs` finden Sie schon Parser für Schlüsselwörter und Zeichen, die sie zu dem Lexer vervollständigen können.

9.2 *Parser* *10 Punkte*

Unsere Anfragesprache (Arbeitsname “MiniSQL”) soll sechs Anfragen interpretieren können:

- Initialisierung (Erzeugung) neuer Tabellen, Schlüsselwort: `CREATE`
- Löschen von Tabellen, Schlüsselwort: `DROP`
- Daten in Tabellen in der Datenbank einfügen, Schlüsselwort: `INSERT`
- Tupel in vorhandenen Tabellen aktualisieren, Schlüsselwort: `UPDATE`
- Selektion aus Tabellen, Schlüsselwort: `SELECT`
- Speichern von Selektionen in neuen Tabellen, Schlüsselwort: `SAVE AS`

Die Anfragen werden durch den Datentyp `Query` in `src/Types.hs` repräsentiert.

Wir implementieren für jeden dieser Anfragen einen Parser, die wir dann am Ende leicht zusammensetzen können. Die Parser haben alle die Funktionalität `Parser Token Query`, d.h. sie konsumieren eine Sequenz von Token und liefern eine Anfrage. Zu jedem dieser Kommandos geben wir die Syntax in EBNF-Form[1] sowie eine Beipielanfrage.

1. Tabelle erzeugen: `createTable :: Parser Token Query`

   ```
   createTable ::= 'CREATE' 'TABLE' tableName createAttributes ';'

   createAttributes ::= '(' createAttribute {',' createAttribute} ')'
   createAttribute ::= attributeName attributeType [attributeConstraint]
   ```

   Beispielanfrage:

   ```
   CREATE TABLE Persons (
       personID int NOT_NULL,
       lastName string,
       firstName string,
       address string UNIQUE,
       city string NOT_NULL
   );
   ```

2. Tabelle löschen: `dropTable :: Parser Token Query`

   ```
   dropTable ::= 'DROP' 'TABLE' tableName ';'
   ```

   Beispielanfrage:

   ```
   DROP TABLE Persons;
   ```

3. Einfügen in Tabelle: `insertIntoTable :: Parser Token Query`

   ```
   insertIntoTable ::= 'INSERT' 'INTO' tableName insertIntoColumns 'VALUES' insertValues ';'

   insertIntoColumns ::= '(' attributeNames ')'
   attributeNames ::= attributeName {',' attributeName}
   insertValues ::= '(' values ')'
   values ::= value {',' value}
   value ::= int | double | string | boolean | date
   ```

   Beispielanfrage:

[1] Extended Backus-Naur-Form, siehe `https://de.wikipedia.org/wiki/Erweiterte_Backus-Naur-Form`

```
INSERT INTO Customers
  (customerName, contactName, address, city, postalCode, country)
VALUES
  ('Cardinal', 'Tom_Erichsen', 'Skagen_21', 'Stavanger', '4006', 'Norway');
```

4. Update von Tabellen: `parseUpdate :: Parser Token Query`

```
updateTable ::= 'UPDATE' tableName 'SET' equalToValues 'WHERE' predicates1 ';'

equalToValues ::= equalToValue {',' equalToValue}
predicates1 ::= predicates2 {'OR' predicates2}
predicates2 ::= predicate {'AND' predicate}
predicate ::= equalToValue     | notEqualToValue
            | biggerThanValue  | biggerEqualsThanValue
            | smallerThanValue | smallerEqualsThanValue
equalToValue ::= attributeName '=' value
notEqualToValue ::= attributeName ('<>' | '!=') value
biggerThanValue ::= attributeName '>' value
biggerEqualsThanValue ::= attributeName '>=' value
smallerThanValue ::= attributeName '<' value
smallerEqualsThanValue ::= attributeName '<=' value
```

Beispielanfrage:

```
UPDATE Customers
  SET contactName = 'Alfred_Schmidt', city = 'Frankfurt'
  WHERE customerID  = 1
  AND   contactName = 'Something';
```

5. Selektion aus einer Tabelle: `select :: Parser Token Query`

```
selectTable ::= 'SELECT' ['DISTINCT'] ('*' | attributeNames)
                    'FROM' tableName ['WHERE' predicates1] ';'
```

Beispielanfragen:

```
SELECT * FROM Table1;

SELECT customerName, city, country FROM Customers;

SELECT DISTINCT country FROM Customers;

SELECT DISTINCT country FROM Customers
  WHERE name = 'Zula_Batsukh'
  OR   age <= '34'
  AND  birthDate = 1995-01-01;
```

6. Speichern einer Anfrage: `saveTable :: Parser Token Query`

```
saveTable ::= 'SAVE' 'AS' tableName 'FROM' selectTable
```

Beispielanfrage:

```
SAVE AS Table2 FROM
  SELECT DISTINCT country FROM Customers
    WHERE name = 'Zula_Batsukh'
    OR   age <= '34'
    AND  birthDate = 1995-01-01;
```

Eine Anfrage ist jetzt eine von diesen Varianten:

```
query ::= createTable | dropTable | insertIntoTable | updateTable | selectTable | saveTable
```

realisiert durch die Funktion

```
query :: Parser Token Query
```

Hinweise:

1. Im Modul `Parser` werden weitere nützliche Hilfsfunktionen bereitgestellt, wie zum Beispiel

   ```
   sepBy :: Eq a⇒ Parser a b → a → Parser a [b]
   ```

   welches einen Parser mindestes einmal wiederholt, wobei die Wiederholungen durch das angegebene Token getrennt sind; bspw. erkennt `sepBy (satisfy isDigit) ","` durch Kommata getrennte Zahlen

2. Da wir mit einen LL-Parser[2] arbeiten, in dessen Grammatik die Linksrekursion[3] nicht eliminiert ist (wie es der Einfachheit halber in der Grammatik unserer Sprache der Fall ist), kann es vorkommen, dass mehrere Regeln einen identischen Präfix von Tokens haben. Hierbei ist die Reihenfolge bei der Alternative (<|>) entscheidend: es muss der längere Präfix zuerst geprüft werden. Beispielsweise erkennt `token "ab" <|> token "a"` die Zeichenkette `"ab"` als ein Token, aber `token "a" <|> token "ab"` erkennt bei `"ab"` nur ein Token Token `"a"` (und kann `"b"` nicht mehr konsumieren).

9.3 *Webinterface* *5 Punkte*

Um einen einfachen Zugriff auf unsere Datenbank zu ermöglichen, wollen wir ein Web-Interface implementieren. Den serverseitigen Teil gibt es schon: in `app/Main.hs` ist der Hauptteil des Servers, der Anfragen entgegen nimmt, parsiert und an die Datenbank weiterreicht. Ihre Aufgabe besteht darin, in `src/Website/Website.hs` die eigentliche Web-Schnittstelle zu implementieren:

```
website :: [T.Text] → Html
website xs = docTypeHtml (htmlPage xs)

htmlPage :: [T.Text] → Html
htmlPage xs = undefined
```

`xs` ist die dabei die Geschichte der bisherigen Anfragen (genauer gesagt, die Antworten darauf).

Die Webseite sollte oben die Geschichte (oder bspw. die letzten fünf) der Anfragen darstellen, und darunter ein Textfeld, in dem die Anfrage eingegeben werden kann, und ein Knopf, mit dem die Anfrage zum Server geschickt wird. Das entspricht technisch einem `POST` auf die Route `/new` (siehe den entsprechenden Teil in `app/Main.hs`).

Das Web-Interface bringen Sie mit dem Befehl `stack run ueb09` zum Laufen, danach können Sie sich lokal mit dem Server auf `localhost:8080` verbinden.

[2] https://de.wikipedia.org/wiki/LL-Parser
[3] https://en.wikipedia.org/wiki/Left_recursion#Indirect_left_recursion

Anhang: Lexikalik

Die Syntax der Tokens ist als EBNF wie folgt definiert:

```
token ::= keyword | symbol | double | int | string | tableName | attributeName

keyword ::= 'CREATE' | 'DROP' | ...
symbol ::= '=' | '<' | ...

int ::= ['-'] unsignedInt
double ::= int '.' unsignedInt
string ::= '\'' char* '\''
date ::= year '-' month '-' day
tableName ::= bigLetter {alphaNum}
attributeName ::= smallLetter {alphaNum}

unsignedInt ::= digit digit*
digit ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
year ::= digit digit digit digit
month ::= digit digit
day ::= digit digit

bigLetter = 'A' | 'B' | 'C' | 'D' | .., | 'Z'}
alphaNum ::= smallLetter | digit | bigLetter
smallLetter = 'a' | 'b' | 'c' | 'd' | ... | 'z'
```

Zur Erkennung von Buchstaben, Ziffern, Zahlen und Leerzeichen aller Art können Sie die Funktionen `isAlpha`, `isaAlphaNum`, `isDigit` und `isSpace` aus `Data.Char` vorteilhaft verwenden.

Die Tokens für Schlüsselwörter und Sonderzeichen sind schon vordefiniert.