CMPSC380
Principles of Database Systems, Fall 2014

Final Project: Introduction to Compiler Design

Purpose: Implementation of Query Processing in a Relational Database Management System
Pledge:

Contents

List of Listings

1 Motivation

Relational database management systems (RDBMSs) are everywhere. The relational model is a model of data storage in which data is stored in *tuples*, or rows, which are grouped together to form *relations*, or tables [6,8,15]. The relational model is perhaps the most popular models of data storage currently in use, with Silberschatz, Korth, and Sudarshan calling it "[t]he primary data model for commercial data-processing applications" [15, page 39].

Users and other software typically interact with these systems through the use of a *query language*. A query language is a domain-specific declarative programming language that is used to express database queries, such as selecting data from the database; modifying the database's state by changing, inserting, or deleting records; and specifying the structure of the tables in the database.

Therefore, in order to facilitate users and application programs interaction with the database, it is necessary for an RDBMS to provide a query processing system with some method capable of executing query language statements. Essentially, this means that a compiler or an interpreter is a core feature of any RDBMS. These function similarly to compilers or interpreters which execute general-purpose programming languages.

A majority of modern relational database management systems, from the SQLite embedded database in every Android phone and iPhone [16] to the MySQL databases used in many web applications [1], implement the *Structured Query Language*, or SQL [15]. In order to learn more about how relational databases function, I have developed my own implementation, called DeeBee. DeeBee implements a limited subset of SQL chosen to be expressive enough to allow most relational database functionality.

DeeBee is released as open-source software under the MIT license. Current and past releases are available for download at `https://github.com/hawkw/deebee/releases`. The provided JAR file can be included as a library in projects which use the DeeBee API to connect to a DeeBee database; or it may be executed using the `java -jar` command to interact with a DeeBee database from the command line. Finally, DeeBee's ScalaDoc API documentation is available at `http://hawkw.github.io/deebee/api/index.html#deebee.package`.

2 Background

2.1 Database Query Processing

In the typical relational database management system, the processing of a SQL query takes the general form given in Figure 1. Queries are input into the system as strings. These strings are run through a query parser, which produces an intermediate representation of the query, in the form of a relational algebra expression. The relational algebra is run through a query optimizer, which uses metadata stored by the DBMS to determine the optimal execution plan for the query. These steps can be thought of as analogous to the compilation process of a general-purpose programming language.

Once the execution plan has been produced, it is evaluated against the data stored in the

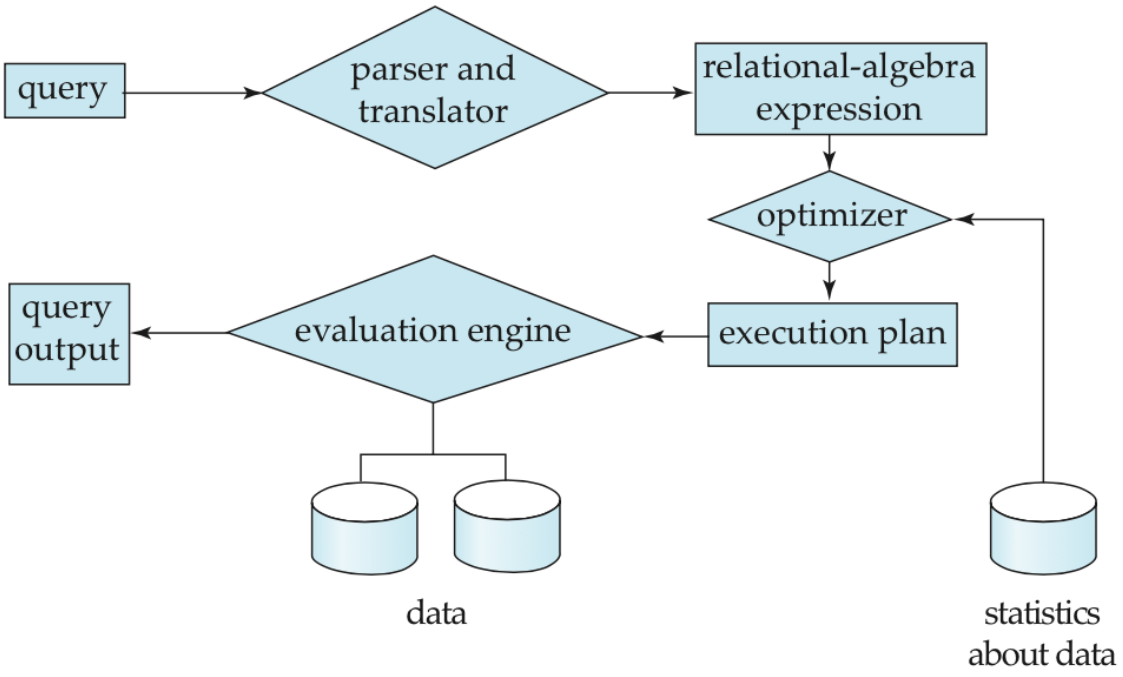

Figure 1: Steps in query processing [15].

database. If the query is one which accesses data, then output may be returned to the querying program; otherwise, there may be changes to the database's internal state [6, 8, 15].

DeeBee's query processing system uses a simplified version of this general flow. For ease-of-implementation reasons, DeeBee interprets queries, rather than compiling and optimizing them. Furthermore, DeeBee's internal query representation takes the form of an abstract-syntax tree (AST) rather than a relational algebra expression. This is due to the fact that DeeBee does not implement the entire relational algebra, and some queries which DeeBee is capable of processing cannot be expressed in this manner.

2.2 The Structured Query Language

SQL consists of two primary components, the *Data Definition Language* or DDL, which is used to define the structures in which data is stored, and the *Data Manipulation Language*, or DML, which is used for statements which access and modify the data stored in the database. The set of DDL statements which are run in order to create a database is referred to as the database's `schema`. A schema typically consists of a series of `CREATE TABLE` statements, which define each relation in the database by enumerating the names, data types, and constraints of each column in a table, as well as other statements, such as `CREATE TRIGGER`, which can be used to assign limited behaviors to various tables, and to create relationships between tables. Once a database has been created, it may be manipulated using DML statements such as `INSERT` to add data, `DELETE` to remove data, and `SELECT` to access data.

2.2.1 DDL Grammar

DeeBee's data definition language consists of the `CREATE TABLE` and `DROP TABLE` statements. Since DeeBee does not implement a number of advanced database features, such as the creation of triggers, the SQL constructs corresponding to those features have been removed. A Baukus-Naur Form for the DeeBee data definition language is given in listing 1. Do note that in DeeBee SQL, as in many SQL dialects, keywords are not case sensitive; however, SQL keywords are conventionally typeset upper case.

⟨*DDL-statement*⟩ ::= ⟨*create-statement*⟩
  | ⟨*drop-statement*⟩

⟨*create-statement*⟩ ::= '`CREATE`' '`TABLE`' ⟨*name*⟩ '`(`' ⟨*members*⟩ '`)`' '`;`'

⟨*members*⟩ ::= ⟨*schema-item*⟩ '`,`' ⟨*column-list*⟩
  | ⟨*schema-item*⟩

⟨*schema-item*⟩ ::= ⟨*attribute*⟩
  | ⟨*referential-constraint*⟩

⟨*attribute*⟩ ::= ⟨*name*⟩ ⟨*data-type*⟩ ⟨*constraint-list*⟩

⟨*constraint-list*⟩ ::= ⟨*attr-constraint*⟩ '`,`' ⟨*constraint-list*⟩
  | ⟨*attr-constraint*⟩

⟨*attr-constraint*⟩ ::= '`NOT`' '`NULL`' | '`UNIQUE`' | '`PRIMARY`' '`KEY`'

⟨*data-type*⟩ ::= '`INT`' | '`INTEGER`' |'`CHAR`' | '`VARCHAR`' | '`NUMERIC`' | '`DECIMAL`'

⟨*referential-constraint*⟩ ::= '`FOREIGN`' '`KEY`' '`(`' ⟨*names-list*⟩ '`)`' '`REFERENCES`' ⟨*name*⟩ '`(`' ⟨*names-list*⟩ '`)`'

⟨*names-list*⟩ ::= ⟨*name*⟩ '`,`' ⟨*names-list*⟩
  | ⟨*name*⟩

⟨*delete-statement*⟩ ::= '`DROP`' '`TABLE`' ⟨*name*⟩ '`;`'

Listing 1: Grammar of DeeBee DDL statements

In this grammar, the DeeBee DDL is defined as consisting of either `CREATE TABLE` or `DROP TABLE` statements. A `CREATE TABLE` statement consists of the keyword `CREATE TABLE`, followed by the name of the table to be created, followed by a listing of the attributes and constraints placed on the table, surrounded by parentheses. Each attribute definition consists of a name, a type, and optional row constraints, such as `NOT NULL` or `UNIQUE`. Listing 2 provides an example of a `CREATE TABLE` statement. All valid SQL statements are terminated with the semicolon character.

A `DROP TABLE` statement simply consists of the keywords `DROP TABLE` followed by the name of

```
CREATE TABLE Writers (
    id                    INTEGER NOT NULL PRIMARY KEY,
    first_name           VARCHAR(15) NOT NULL,
    middle_name          VARCHAR(15),
    last_name            VARCHAR(15) NOT NULL,
    birth_date           VARCHAR(10) NOT NULL,
    death_date           VARCHAR(10),
    country_of_origin    VARCHAR(20) NOT NULL
);
```

Listing 2: Example SQL `CREATE TABLE` statement.

the table to be deleted. Listing 3 provides an example of such a statement.

```
DROP TABLE Writers;
```

Listing 3: Example SQL `DROP TABLE` statement.

2.2.2 DML Grammar

DeeBee's data manipulation language consists of `SELECT`, `INSERT`, and `DELETE` statements. `INSERT` statements add rows to a table, while `DELETE` statements remove rows, and `SELECT` statements access rows. Most SQL implementations also include a `UPDATE` statement that modifies the data stored in table rows, but this statement was excluded from DeeBee as it is essentially syntactic sugar for actions that can be accomplished through the use of `DELETE` and `INSERT` statements. Listing 4 presents a Baukus-Naur Form for the DeeBee DML.

An `INSERT` statement is used to add a row to a table in the database. It consists of the keywords `INSERT INTO` followed by the name of a table, followed by the keyword `VALUES` followed by a comma-separated list of values surrounded in parentheses. Listing 5 provides a sample of such a statement.

A `SELECT` statement returns rows contained in a table. Optionally, specific columns may be specified, referred to as *projections*. Additionally, the rows may be filtered according to a predicate, or a `LIMIT` clause may be used to select up to a certain number of rows. **??** gives a number of sample `SELECT` statements, demonstrating the multiple forms possible for this statement.

Many relational database management systems support `SELECT` statements with `JOIN` clauses, which allow the records of one or more tables to be combined. There are multiple forms of `JOIN` statement, such as inner join, cross join, and various forms of outer join, which allow records to be joined in different ways according to the rules of relational algebra. While these expressions are very useful in many database applications, DeeBee does not implement any form of `JOIN` statement due to the significant increase in complexity provided by the wide range of possible join expressions.

Additionally, most SQL implementations support aggregate functions, such as `SUM`, `COUNT`, and `AVERAGE` in their select statements. Rather than returning all rows matching the `SELECT` statement,

⟨*DML-statement*⟩ ::= ⟨*select-statement*⟩
| ⟨*insert-statement*⟩
| ⟨*delete-statement*⟩

⟨*insert-statement*⟩ ::= '`INSERT`' '`INTO`' ⟨*name*⟩ '`VALUES`' '`(`' ⟨*vals-list*⟩ '`)`' '`;`'

⟨*vals-list*⟩ ::= ⟨*literal*⟩ '`,`' ⟨*vals-list*⟩
| ⟨*literal*⟩

⟨*select-statement*⟩ ::= '`SELECT`' ⟨*projections*⟩ '`FROM`' ⟨*name*⟩ ⟨*where-clause*⟩ ⟨*limit-clause*⟩'`;`'

⟨*delete-statement*⟩ ::= '`DELETE`' '`FROM`' ⟨*name*⟩ ⟨*where-clause*⟩ '`;`'

⟨*projections*⟩ ::= ⟨*projection-list*⟩
| '`*`'

⟨*projection-list*⟩ ::= ⟨*name*⟩ '`,`' ⟨*projection-list*⟩
| ⟨*name*⟩

⟨*where-clause*⟩ ::= '`WHERE`' ⟨*predicate*⟩
| ε

⟨*limit-clause*⟩ ::= '`LIMIT`' ⟨*int-const*⟩
| ε

⟨*predicate*⟩ ::= ⟨*comparison*⟩ ⟨*combining-op*⟩ ⟨*comparison*⟩
| ⟨*comparison*⟩

⟨*combining-op*⟩ ::= '`AND`' |'`OR`'

⟨*comparison*⟩ ::= ⟨*expr*⟩ ⟨*comparison-op*⟩ ⟨*expr*⟩

⟨*comparison-op*⟩ ::= '`=`' |'`!=`' | '`<>`' | '`>=`' | '`<=`' | '`<`' | '`>`'

⟨*expr*⟩ ::= ⟨*literal*⟩ | ⟨*name*⟩

⟨*literal*⟩ ::= ⟨*int-const*⟩ | ⟨*double-const*⟩ | ⟨*string-const*⟩ | '`null`'

Listing 4: Grammar of DeeBee DML statements.

these aggregate functions reduce the result set to a single record, usually numeric. Although these aggregate functions would not be as difficult to implement as `JOIN` clauses, DeeBee does not currently support them as they were deemed to fall outside of the core SQL functionality necessary for DeeBee.

A `DELETE` statement removes rows from a table. By default, it will remove all rows, but the deleted rows may be filtered according to a predicate, or provided with a limit clause, as in the

```
INSERT INTO Writers VALUES (
        1,
        'Robert',
        'Anton',
        'Heinlein',
        '7/7/1907',
        '5/8/1988',
        'USA'
);
```

Listing 5: Sample SQL `INSERT` statement.

```
INSERT INTO Writers VALUES (
        1,
        'Robert',
        'Anton',
        'Heinlein',
        '7/7/1907',
        '5/8/1988',
        'USA'
);
```

Listing 6: Sample SQL `SELECT` statements.

`SELECT` statement. Listing 7 provides samples of the possible forms of this statement.

```
DELETE FROM Writers;
DELETE FROM Writers LIMIT 10;
DELETE FROM Writers WHERE country_of_origin != "USA";
DELETE FROM Writers WHERE id >= 8 LIMIT 3;
```

Listing 7: Sample `SQL DELETE` statements.

3 Implementation

3.1 The Scala Programming Language

DeeBee was implemented using the Scala programming language, an object-oriented functional programming language which runs on the Java virtual machine [12–14]. Scala was designed by Martin Odersky of the Programming Methods Laboratory at École Polytechnique Fédérale de

Lausanne with the intention of developing a highly scalable programming language, "in the sense that the same concepts can describe small as well as large parts" [13] and in the sense that Scala should be applicable to tasks of various sizes and complexities, and provide high performance at large scales [14].

Scala was inspired by criticisms of Java and by the recent popularity of functional programming languages such as Haskell. It aims to provide a syntax that is more expressive than that of Java but is still readily accessible to Java programmers. It is a statically-typed language and was developed with a focus on type safety, immutable data structures, and pure functions [12–14]. Because it compiles to Java bytecode and runs on the JVM, Scala is mutually cross-compatible with Java, meaning that Scala code can natively call Java methods and use Java libraries, and vise versa [14].

A key concept in the Scala design is the idea of an 'embedded domain-specific language (DSL)'. Essentially, this concept suggests that Scala's syntax should be modifiable to the extent that code for specific tasks be expressed with its' own syntax within Scala. These DSLs are still Scala code and can still be compiled by the Scala compiler, but their syntax differs based on the task they are intended for [7, 9, 14]. The Scala parsing library and the ScalaTest testing framework both provide exmaples of embedded DSLs.

Scala was chosen as the ideal language for DeeBee's implementation due to the expressiveness of its syntax, which allows complex systems to be implemented in few lines of code; its performance at scale; the existence of powerful libraries for text parsing and concurrent programming using the actors model; and the cross-platform capabilities of the JVM.

3.2 Query Parsing

DeeBee processes SQL queries by parsing the input query strings and generating an abstract syntax tree (AST) which represents the query. The AST is then interpreted against a context (typically a database or table) to determine the actions necessary to carry out the query. The query is then conducted using the API of the target relations to carry out the necessary operations.

DeeBee's query parser was implemented using the Scala standard library's combinator parsing [11] package. Combinator parsing is a method of parser implementation, popular in functional programming languages such as Scala and Haskell, which provides an alternative to parser generators such as `yacc` or `bison`.

In the parser generator approach, the grammar of a language is described using a syntax similar to Baukus-Naur Form (BNF), and then the parser generator is invoked on that grammar description. The parser generator uses the grammar description to produce the source code for a parser for that language[1]. This parser can be included in a project and used to parse text input.

While the use of these tools is often much less demanding than implementing a parser by hand, the use of parser generators has a number of disadvantages. The generated source code is frequently highly complex and, since it is generated automatically, it may be poorly documented and difficult to understand. This can make debugging and maintaining the generated code very difficult.

[1] In the case of `bison`, these are LALR(1) parsers implemented in C

3.2.1 Combinator Parsing

Combinator parsing provides a method of implementing recursive descent parsers using a BNF-like domain-specific language, but rather than using that grammar description to generate parsing code, the grammar description is the source code of the parser, and can be inspected, maintained, and modified transparently [7, 9, 11].

In this approach, a parser is defined as a function which takes as input a string and either accepts or rejects that string according to some rule. A parser combinator is a higher-order function which takes as parameters two parsers, and produces a new parser that combines the two input parsers. Parser combinators exist which combine the input parsers according to various rules, such as sequencing, alternation, optionality, or repetition. The language specification is written by repeatedly combining primitive parsers for tokens such as keywords, constants, and identifiers into more complex parsers for constructs such as expressions. At runtime, the parser then attempts to parse some input against the start symbol parser, which is composed of multiple smaller parsers, recursively descending through the grammar by calling each component parser. [2, 5, 11, 17]

For example, in the Scala parser combinators library, the `~` parser combinator is the sequencing parser combinator. If `a` and `b` are parser functions, then the expression `a ~b` would create a parser which accepts a string containing a substring accepted by `a` followed by a substring accepted by `b`. Similarly, `|` is the alternation combinator, so the expression `a | b` specifies a parser that accepts either a string accepted by `a` or `b`. The `.?` combinator can be applied to a single parser in order to make that parser optional (one or none), while the `.*` combinator will accept any number of repetitions of the input parser. Finally, the `^^` combinator creates a parser which applies a partial function to the parse result; this functions similarly to the "rules" specified in a `yacc` or `bison` grammar.

Multiple parser combinators can be combined into complex expressions. For example, if `a`, `b`. and `c` are all parser functions, the expression

```
def d = a ~ b | c.*
```

defines `d` as a parser that accepts a string which contains a substring matching `a` followed by a substring matching either `b` or any number of repetitions of `c`.

Listing 8 provides a brief example of the usage of parser combinators, selected from an early version of the DeeBee SQL parser. Lines 1 through 7 of **??** define `query` as a parser which produces `Node`s (the DeeBee base type for an AST node), and then enumerates that `query` accepts a string parsed by the `createTable`, `dropTable`, `select`, `delete`, or `insert` parsers. Lines 9 through 11 define `dropTable` as a parser which produces `DropStmt` AST nodes, and then states that the `dropTable` parser accepts the keyword "drop" followed by the keyword "table" followed by an identifier, extracts the parsed identifier through pattern matching, and returns a new `DropStmt` node containing that identifier. The relative similarity of this source code to the Baukus-Naur Form for the DeeBee DDL grammar given in listing 1 highlights the power of combinator parsing.

```
def query: Parser[Node] = (
  createTable
    | dropTable
    | select
    | delete
    | insert
  ) <~ ";"

def dropTable: Parser[DropStmt] = "drop" ~> "table" ~> identifier ^^{
  case i => DropStmt(i)
}
```

Listing 8: Combinator parsing in Scala.

3.2.2 Packrat Parsing

A major disadvantage of top-down parsers, such as the recursive descent parsers created through the use of parser combinators, is that they cannot parse left-recursive grammars[2]. Scala's parsing library offers a solution to this problem by providing *packrat parsing* functionality.

Packrat parsing, first described by Bryan Ford in his masters' thesis [3] is a technique that both improves performance of backtracking parsers (such as recursive descent parsers), guaranteeing linear parse time, and allows the parsing of grammars containing left recursion, as well as any LL(k) or LR(k) language. Packrat parsing works by memoizing the results of each invocation of a parsing function at a given location in the input text, and returning that cached result when the parsing function is called again for that location, significantly improving performance [3, 4, 10]. Left recursion is accepted by attempting to 'grow' the cached input by attempting to reapply the production to the input, and checking if the position of the new parse result is greater than or less than the previous position. If the positions higher than the previously cached result, then the process is repeated, otherwise, the cached result is returned [3, 10].

In the Scala packrat parsing implementation, memoization is implemented through the use of *lazy evaluation*, an evaluation common in functional programming languages. Lazy evaluation refers to the practice of differing the evaluation of some expression until that expression's value is needed elsewhere, and by caching that result and reusing it every time the expression is evaluated. This means that in Scala, packrat parsing may be used simply by mixing in the `PackratParsers` trait and by replacing parsers written using `def`, indicating that they are defined as functions, with `lazy val`, which defines those parsers as lazily-evaluated values, instead. Therefore, we could rewrite the code example in Listing 8 as Listing 9.

The use of packrat parsing was invaluable in DeeBee's implementation, mostly in situations related to the parsing of nested predicate expressions in SQL `WHERE` clauses containing nested comparisons (using the `AND` and `OR` operators). These issues were eventually resolved by separating

[2]Those in which there exists a non-terminal symbol in which the left-most symbol in a production for that non-terminal evaluates to that symbol.

```
lazy val query: PackratParser[Node] = (
  createTable
    | dropTable
    | select
    | delete
    | insert
  ) <~ ";"

lazy val dropTable: PackratParser[DropStmt] = "drop" ~> "table" ~> identifier ^^{
  case i => DropStmt(i)
}
```

Listing 9: Packrat parsing in Scala.

the productions for 'leaf' predicates (those consisting of a comparison between an attribute and an expression) and predicates consisting of multiple predicate expressions, and using the longest-match parser combinator to back-track in order to parse the entire `WHERE` clause. Unlimited backtracking was guaranteed through the use of packrat parsing, allowing predicates to be nested to arbitrary depths.

3.3 Query Interpretation

Queries are interpreted against a table using the `Relation` API. DeeBee defines a core trait for all tables, `Relation`, which defines a number of 'primitive' operations on a table, such as filtering rows by a predicate, projecting a relation by selecting specific columns, and taking a subset number of rows. These in turn rely on two abstract methods, for accessing the table's attribute definitions and the set of the table's rows. These methods are implemented by each concrete class that implements `Relation`.

Relation is extended by two other traits, `Selectable` and `Modifyable`, which provide polymorphic functions for actually processing queries. These can be mixed in as needed to represent tables which support these functionalities.

Queries involving predicates, such as `SELECT` and `DELETE` statements with `WHERE` clauses, must go through an additional step of predicate construction, which converts SQL ASTs for `WHERE` clauses into Scala partial functions which take as a parameter a row from a table and return a Boolean. These functions are emitted by the AST node for `WHERE` clauses, using the target relation as a context for accessing the attributes corresponding to names in the SQL predicate. A similar process is performed for `INSERT` statements, which must be checked against the type and integrity constraints in the attribute corresponding to each value in the statement.

4 Conclusions

4.1 Further Research

While DeeBee currently implements the entirity of the planned subset of SQL, there are a number of additional features that could be implemented. As discussed in **??**, CSV storage is not as performant as the storage methods used by real RDBMSs, and I am considering implementing a more advanced storage system.

Furthermore, DeeBee SQL does not currently implement a large portion of SQL syntax, such as `JOIN`s, `UPDATE` statements, `VIEW`s, or aggregate functions such as `SUM` and `COUNT`. DeeBee also does not implement all of the data types available in other SQL databases, such as BLOBs (binary large objects) and CLOBs (character large objects). An implementation of the SQL `DATE` type based on `java.util.date` was prototyped but is currently not yet implemented.

Finally, DeeBee's connections API, used by other programs to interact with the system, could be expanded with support for non-blocking connections and additional options to configure the connected database, similar to those available in the Java DataBase Connection driver (JDBC). I have considered the implementing a JDBC compatible API for DeeBee, but am not currently certain if this is possible, due to the differences between DeeBee and other SQL databases.

5 References

[1] Dale Dougherty. LAMP: The Open Source Web Platform - O'Reilly Media, 2001. `http://www.onlamp.com/pub/a/onlamp/2001/01/25/lamp.html`.

[2] Jeroen Fokker. Functional parsers. In *Advanced Functional Programming*, pages 1–23. Springer, 1995.

[3] Bryan Ford. *Packrat parsing: a practical linear-time algorithm with backtracking*. PhD thesis, Massachusetts Institute of Technology, 2002.

[4] Bryan Ford. Packrat parsing: simple, powerful, lazy, linear time, functional pearl. In *ACM SIGPLAN Notices*, volume 37, pages 36–47. ACM, 2002.

[5] Richard A Frost, Rahmatullah Hafiz, and Paul Callaghan. Parser combinators for ambiguous left-recursive grammars. In *Practical Aspects of Declarative Languages*, pages 167–181. Springer, 2008.

[6] Hector Garcia-Molina, Jeffrey D Ullman, and Jennifer Widom. *Database System Implementation*, volume 654. Prentice Hall Upper Saddle River, NJ:, 2000.

[7] Debasish Ghosh. *DSLs in action*. Manning Publications Co., 2010.

[8] Jan L Harrington. *Relational database design and implementation: clearly explained*. Morgan Kaufmann, 2009.

[9] Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. Polymorphic embedding of DSLs. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*, pages 137–148. ACM, 2008.

[10] Manohar Jonnalagedda, Martin Odersky, and Tiark Rompf. Packrat Parsing in Scala. Technical report, Ecole Polytechnique Fédérale de Lausanne, 2009.

[11] Adriaan Moors, Frank Piessens, and Martin Odersky. Parser combinators in Scala. Technical report, Katholieke Universiteit Leuven, 2008.

[12] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. Technical report, 2004.

[13] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. The scala language specification, 2004.

[14] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Inc, 2008.

[15] A. Silberschatz, H. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill Education, 2010.

[16] SQLite. Famous Uses of SQLite, 2014. `https://www.sqlite.org/famous.html`.

[17] S Doaitse Swierstra. Combinator parsers: From toys to tools. *Electronic Notes in Theoretical Computer Science*, 41(1):38–59, 2001.