Project 1

Overview, Guidelines, and JavaCC Tutorial

Project 1

- **Due**: Monday Feb 18 (2 weeks+1 weekend)
- **Groups**: 2-3 People
- **Languages**: Java, JavaCC (like yacc)
- **Goal**: Familiarize yourself with manipulating RA
- Project 1 update posted on of Jan 31
  - Revisions to group submission instructions.

Project 1: Grading

- Test cases are provided
  - `edu.buffalo.cse.sql.test.*`
- Tests for both parts are provided
  - Part 1: 60 pts (55 from provided tests)
  - Part 2: 40 pts (35 from provided tests)

Part 1
A Relational Algebra Interpreter

- **Given**: Data File(s), Relational Algebra Tree
- **Compute**: Evaluate the RA expression
- How? Use Naive In-Memory Algorithms
  - *Selection/Projection/Union*: Basic Pipelined Impl.
  - *File Scan*: Java IO (or NIO for better perf.)
  - *Join*: Nested-Loop (or others for better perf.)
  - *Aggregates*: Hash-table grouping

Part 1

Keep in mind that in Project 3,

performance will be important

So starting now can't hurt.

Part 1

1. Implement relational operators.
   - The iterator interface is the simplest way.
   - init(), readNext(), hasMore(), close()
2. Implement a translator to assemble your relational operators from the RA plan.
3. Implement an evaluator for arithmetic expressions.

Projection Example

- ProjectionOperator(ColNames, Schema, Source)

```
this.OutCols = ColNames.map(Schema.getIndex);
this.Source = Source;
```

- hasNext()

```
return this.Source.hasNext();
```

- readNext()

```
Datum[] next = this.Source.readNext();
return this.OutCols.map(next.get);
```

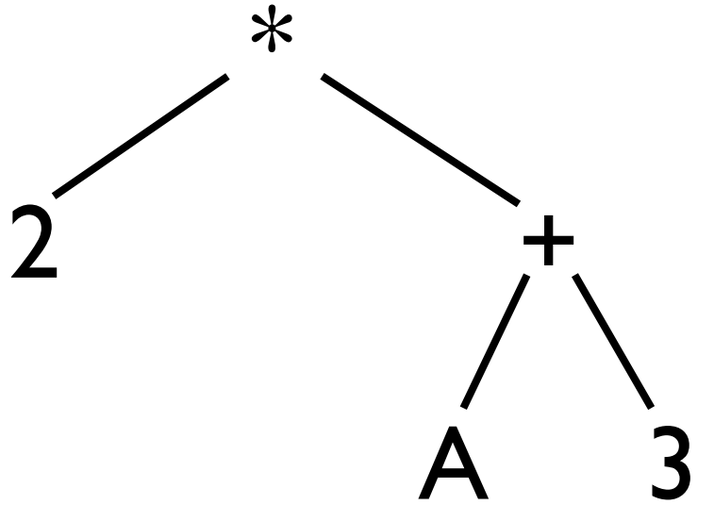
Evaluating Arithmetic

Bindings

A → 5

B → 2

C → ‘Foo’




```
eval(node,bindings)
  switch(node.type)
    case ‘*’:
      return eval(node.l)
             * eval(node.r)
case ‘+’: …
case const:
  return node.value
case var:
  return bindings.get(
           node.var)
```

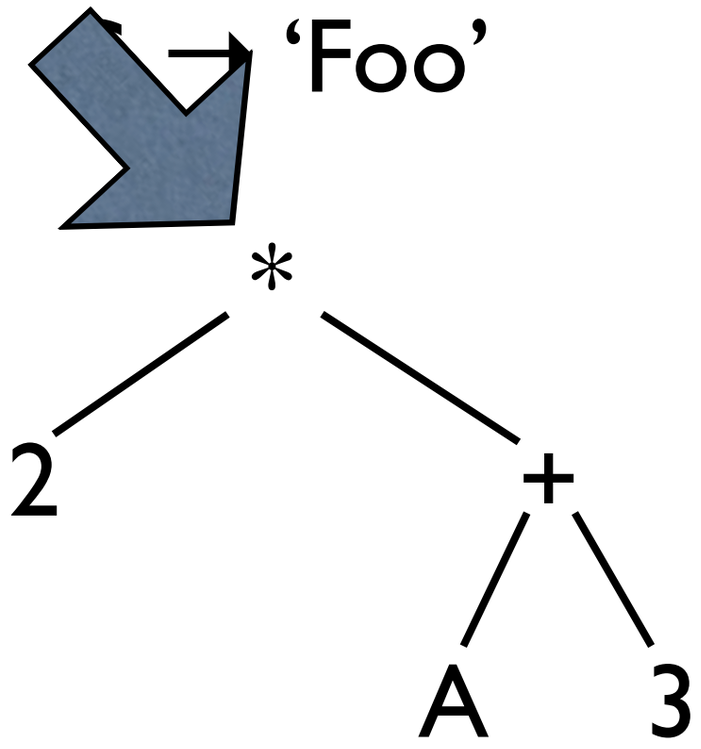
Evaluating Arithmetic

Bindings

A → 5

B → 2

→ ‘Foo’




```
eval(node,bindings)
  switch(node.type)
    case ‘*’:
      return eval(node.l)
                * eval(node.r)
case ‘+’: …
case const:
  return node.value
case var:
  return bindings.get(
              node.var)
```

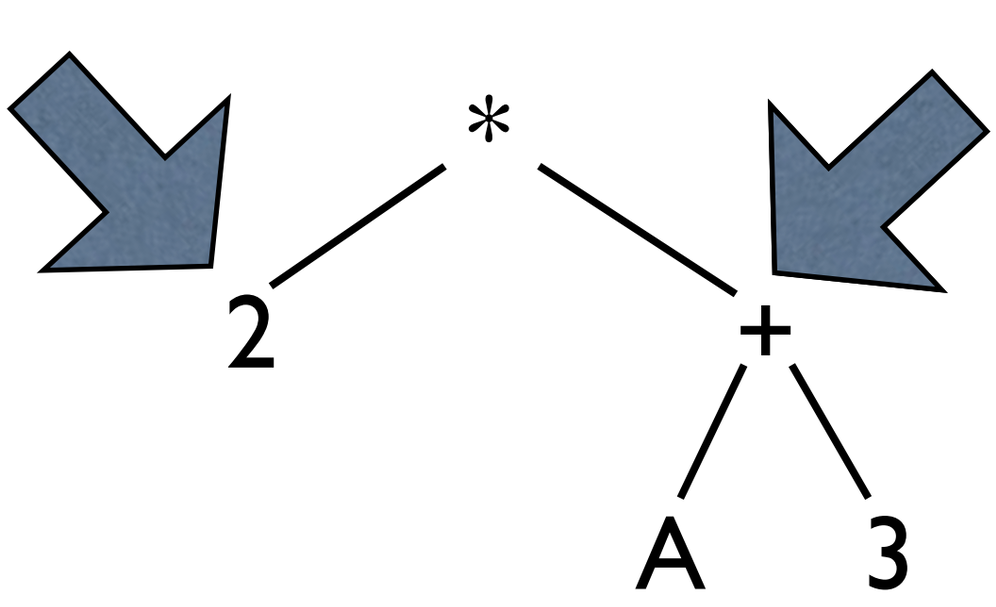
Evaluating Arithmetic

Bindings

A → 5

B → 2

C → ‘Foo’




```
eval(node,bindings)
  switch(node.type)
    case ‘*’:
      return eval(node.l)
           * eval(node.r)
case ‘+’: …
case const:
  return node.value
case var:
  return bindings.get(
           node.var)
```

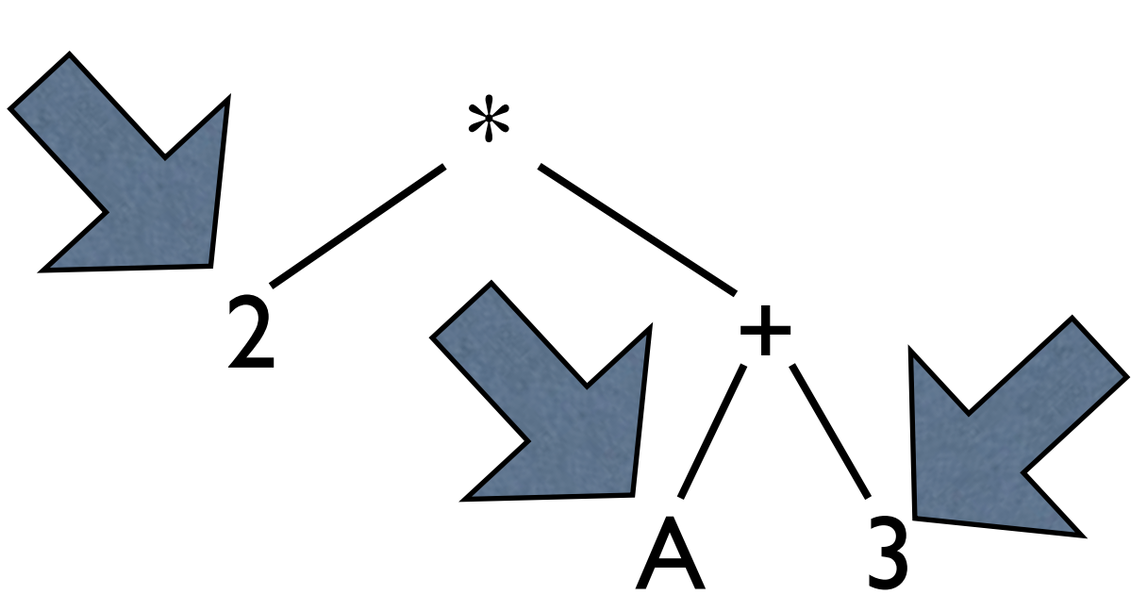
Evaluating Arithmetic

Bindings

A → 5

B → 2

C → ‘Foo’




```
eval(node,bindings)
  switch(node.type)
    case ‘*’:
      return eval(node.l)
             * eval(node.r)
case ‘+’: …
case const:
  return node.value
case var:
  return bindings.get(
           node.var)
```

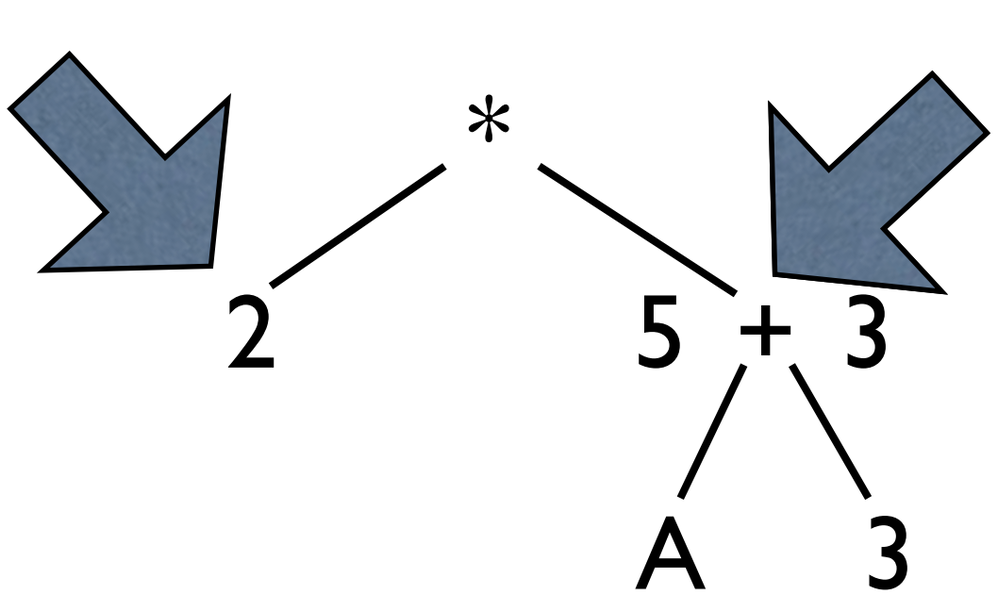
Evaluating Arithmetic

Bindings

A → 5

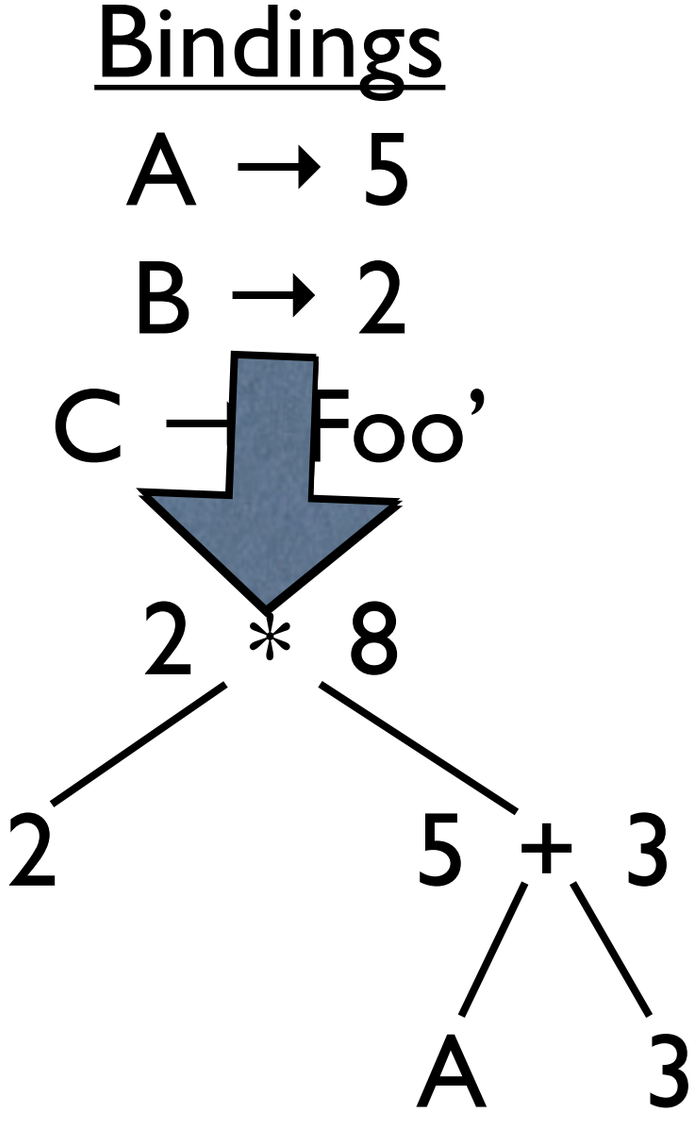
B → 2

C → ‘Foo’



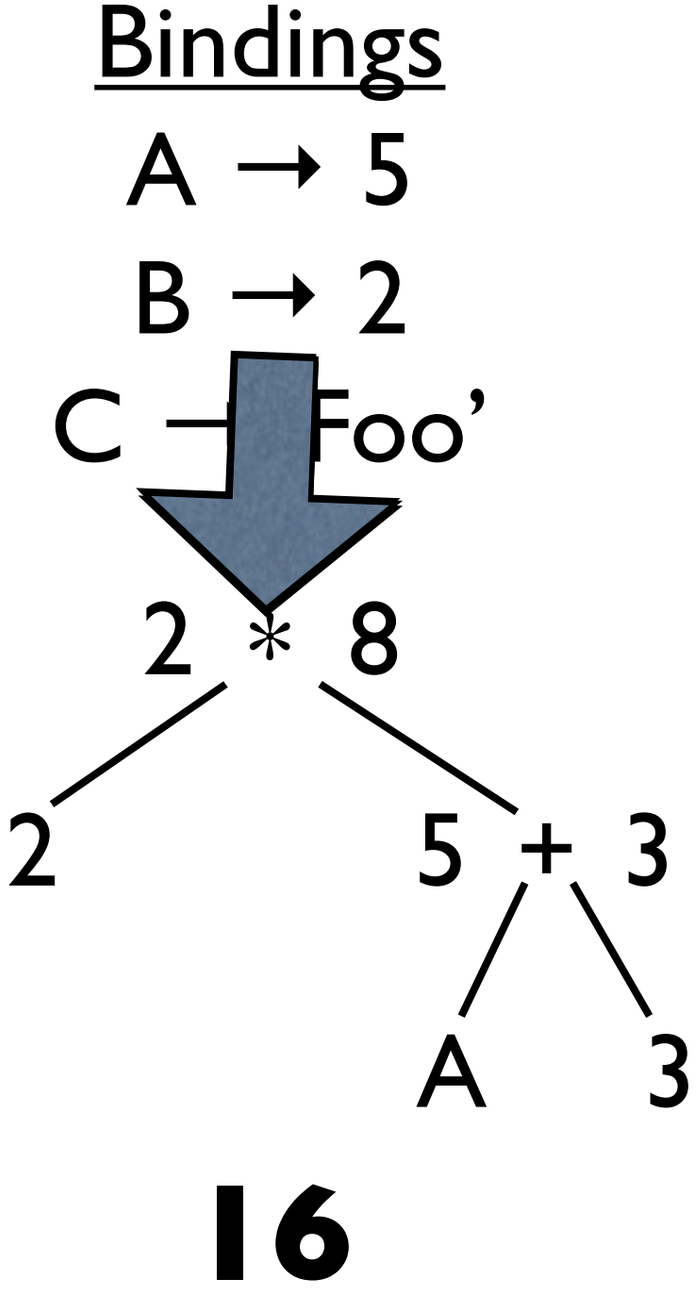

```
eval(node,bindings)
  switch(node.type)
    case ‘*’:
      return eval(node.l)
           * eval(node.r)
case ‘+’: …
case const:
  return node.value
case var:
  return bindings.get(
           node.var)
```

Evaluating Arithmetic




```
eval(node,bindings)
  switch(node.type)
    case ‘*’:
      return eval(node.l)
                * eval(node.r)
case ‘+’: …
case const:
  return node.value
case var:
  return bindings.get(
              node.var)
```

Evaluating Arithmetic




```
eval(node,bindings)
  switch(node.type)
    case ‘*’:
      return eval(node.l)
                * eval(node.r)
case ‘+’: …
case const:
  return node.value
case var:
  return bindings.get(
                node.var)
```

Part 2: Parsing SQL

- Design a **Grammar** for your language.
- Use a compiler compiler (e.g., Yacc, Bison, Javacc)
  - Takes a grammar as input
  - Generates "trees" for expressions in your language
- You specify how to interpret nodes in this tree.

Grammars

How do you specify a language?

```
A := BA | B | C
```

An **A** is a **B** followed by an **A**, or just a **B**, or just a **C**

B, C, BC, and BBBC are valid As

BCC is not valid, as although we can get ‘C’ as an A, no rule exists to let us prepend another ‘C’ before it.

CB is not valid, since we can only prepend ‘B’s to an ‘A’

Grammars

How do you specify a language?

```
A := BA | B | C
```

An **A** is a **B** followed by an **A**, or just a **B**, or just a **C**

B, C, BC, and BBBC are valid As

BCC is not valid, as although we can get ‘C’ as an A, no rule exists to let us prepend another ‘C’ before it.

CB is not valid, since we can only prepend ‘B’s to an ‘A’

Grammars

How do you specify a language?

```
A := BA | B | C
```

An **A** is a **B** followed by an **A**, or just a **B**, or just a **C**

B, C, BC, and BBBC are valid As

BCC is not valid, as although we can get ‘C’ as an A, no rule exists to let us prepend another ‘C’ before it.

CB is not valid, since we can only prepend ‘B’s to an ‘A’

Grammars

How do you specify a language?

```
A := BA | B | C
```

An **A** is a **B** followed by an **A**, or just a **B**, or just a **C**

B, C, BC, and BBBC are valid As

BCC is not valid, as although we can get ‘C’ as an A, no rule exists to let us prepend another ‘C’ before it.

CB is not valid, since we can only prepend ‘B’s to an ‘A’

Grammars

How do you specify a language?

```
A := BA | B | C
```

An **A** is a **B** followed by an **A**, or just a **B**, or just a **C**

B, C, BC, and BBBC are valid As

BCC is not valid, as although we can get ‘C’ as an A, no rule exists to let us prepend another ‘C’ before it.

CB is not valid, since we can only prepend ‘B’s to an ‘A’

Grammars

How do you specify a language?

```
A := BA | B | C
```

An **A** is a **B** followed by an **A**, or just a **B**, or just a **C**

B, C, BC, and BBBC are valid As

BCC is not valid, as although we can get ‘C’ as an A, no rule exists to let us prepend another ‘C’ before it.

CB is not valid, since we can only prepend ‘B’s to an ‘A’

Grammars

How do you specify a language?

```
A := BA | B | C
B := ‘B’
C := ‘C’
```


Friday, February 1, 13

B, C, BC, and BBBC are valid As

BCC is not valid, as although we can get ‘C’ as an A, no rule exists to let us prepend another ‘C’ before it.

CB is not valid, since we can only prepend ‘B’s to an ‘A’

Grammars

How do you specify a language?

```
A := BA | B | C
B := 'B'
C := 'C'
```

Which of the following are valid **A**s?

'B' 'C' 'BC' 'BCC' 'BBBC' 'CB'

Friday, February 1, 13

B, C, BC, and BBBC are valid As
BCC is not valid, as although we can get 'C' as an A, no rule exists to let us prepend another 'C' before it.
CB is not valid, since we can only prepend 'B's to an 'A'

Group Exercise

Define a grammar for basic arithmetic using addition, multiplication, and negation?

Group Exercise

Define a grammar for basic arithmetic using addition, multiplication, and negation?

Hint: This problem is underspecified. What additional information do you need?

Group Exercise

Define a grammar for basic arithmetic using addition, multiplication, and negation?

Hint: This problem is underspecified. What additional information do you need?

Use integers, and assume <INT> is defined

Tokens

- Strings are complex!
- Tokenize strings first
  - Simple, regular expression matching on strings to make ‘tokens’.
    - e.g., `<INT> := /-?[0-9]+/`
    - An optional ‘-’, followed by ≥ 1 digits
- Build parse trees with tokens as leaves.

JavaCC

- Java Compiler Compiler (Part of Sun JDK)
  - Reads in a grammar file.
  - Generates a Java class that parses your language.
    - The Java class can be compiled as normal.
- JavaCC files have a .jj suffix.

A JavaCC .jj File

`Header`

Define code to explicitly copy into the java file

`Grammar Rules`

Declare rules each (roughly) of the form:

```
A := B | C | D | …
```

`Tokenizer Rules`

Declare all token types and how to identify them

A JavaCC Header

JavaCC will insert code here

```
PARSER_BEGIN( [class] )
package ...
import ...
...
public class [class] {
  ...
}
...
PARSER_END( [class] )
```

A JavaCC Grammar Rule

```
Program Program() :
  { Program p = new Program();
    Statement s;}
  {
    ( s = Statement() <EOS>
              { p.addStatement(s); }
    )+ <EOF>
      { return p; }
  }
```

Each JavaCC rule is a function definition

A JavaCC Grammar Rule

Rule Name

```
Program Program() :
  { Program p = new Program();
    Statement s;}
  {
    ( s = Statement() <EOS>
          { p.addStatement(s); }
    )+ <EOF>
      { return p; }
  }
```

Each JavaCC rule is a function definition

A JavaCC Grammar Rule

```
Program Program() :
  { Program p = new Program();
    Statement s;}
  {
    ( s = Statement() <EOS>
                { p.addStatement(s); }
    )+ <EOF>
      { return p; }
  }
```

Rule Body

Each JavaCC rule is a function definition

A JavaCC Grammar Rule

Rule Type

```
Program Program() :
  { Program p = new Program();
    Statement s;}
  {
    ( s = Statement() <EOS>
          { p.addStatement(s); }
    )+ <EOF>
      { return p; }
  }
```

Each JavaCC rule is a function definition

A JavaCC Grammar Rule

```
Program Program() :
   { Program p = new Program();
   Statement s;}
{
   ( s = Statement() <EOS>
        { p.addStatement(s); }
   )+ <EOF>
     { return p; }
}
```

Rule Header

Each JavaCC rule is a function definition

A JavaCC Grammar Rule

```
Program Program(): Rule Arguments
  { Program p = new Program();
    Statement s;}
  {
    ( s = Statement() <EOS>
              { p.addStatement(s); }
    )+ <EOF>
      { return p; }
  }
```

Each JavaCC rule is a function definition

A JavaCC Grammar Rule

```
Program Program() :
  { Program p = new Program();
    Statement s;}
  {
    ( s = Statement() <EOS>
          { p.addStatement(s); }
    )+ <EOF>
      { return p; }
  }
```

Each JavaCC rule is a function definition

The Rule Body is a **Pattern**

A JavaCC Grammar Rule

Rule Type

Rule Name

Rule Arguments

```
Program Program() :
    { Program p = new Program();
      Statement s;}
{
    ( s = Statement() <EOS>
        { p.addStatement(s); }
    )+ <EOF>
      { return p; }
}
```

Rule Header

Rule Body

Each JavaCC rule is a function definition

The Rule Body is a **Pattern**

A JavaCC Rule Pattern

```
Pattern :=

    [var =] <TOKEN>
```

Meaning: Match a token of the indicated type optionally assign it to variable `var`

Note: `var` must be of the Java class Token
`var.image` is the *string* matched by the token's regex

A JavaCC Rule Pattern

EXAMPLE

```
IntBase :=

    i = <INT>
```

Meaning:

match an <INT> token
assign it to the variable `i`

Note:

You'll soon see how to use `i` to assign a 'meaning' to this token (i.e, an integer)

A JavaCC Rule Pattern

```
Pattern :=

    [var =] Rule(args)
```

Meaning: Invoke the indicated rule (with arguments)
The pattern matches if the rule finds a match
Optionally assign the matched value to `var`

Note: `var` must be of the type 'returned' by Rule()

A JavaCC Rule Pattern

```
Pattern :=

    Pattern1 Pattern2 Pattern3 …
```

Meaning: Match an input consisting of something matching pattern 1, followed by a match for pattern 2, followed by pattern 3, …

A JavaCC Rule Pattern

EXAMPLE

```
Program :=

    s = Statement() <EOS>
```

Meaning:

Find a match for the ‘Statement’ rule followed by an EOS token.
Assign the matched Statement to s

A JavaCC Rule Pattern

```
Pattern :=

    Pattern { java code }
```

Meaning:

If the Pattern is matched
Then Execute the Java Code

A JavaCC Rule Pattern

EXAMPLE

```
IntBase :=

    i = <INT> {return parseInt(i.image);}
```

Meaning: match an <INT> token and assign it to the variable `i`
Return the parsed integer as the value for this rule.

Note: Remember, i.image is the string matched by the token

A JavaCC Rule Pattern

`Pattern :=`

`Pattern[op]` where `[op]` is `?`, `+` or `*`

Optional/Repeating pattern. Match…

Meaning:

? : at most once

+ : at least once (greedy)

* : any number of times (greedy)

Note: Code literals and variable assignments in the pattern are executed once for each match

A JavaCC Rule Pattern

EXAMPLE

```
Program :=

    (Statement() <EOS>)+ <EOF>
```

Meaning: Match one or more Statements as long as each statement is followed by an EOS Token
After all statements, there should be an EOF

Note: Parenthesis may be used to group patterns together

A JavaCC Rule Pattern

```
Pattern :=

    Pattern1 | Pattern2
```

Meaning: Match either of the two patterns

Note: The generated parser looks only at the next N *tokens* before committing to this decision (default N=1)
N is known as the **lookahead** parameter

A JavaCC Grammar Rule

Match one or more statements.
For each statement matched, assign it to `s`
then call `p.addStatement(s)`

```
Program Program() :
   { Program p = new Program();
     Statement s;}
{
   ( s = Statement() <EOS>
          { p.addStatement(s); }
   )+ <EOF>
     { return p; }
}
```

If successful, the 'value' of this rule is the `Program p`

A JavaCC Grammar Rule

A statement is a SELECT

```
Statement Statement() :
   { Statement s; PlanNode q; Table t; }
   {
     q = Select()  { return new Statement(q); }
   | t = Table()   { return new Statement(t); }
}
```

or a CREATE TABLE

Use a wrapper class to combine them into one type

A JavaCC Grammar Rule

Tokens are of type `Token`

```
int IntBase() :
   { Token t; }
   { t = <DECIMAL> { return Integer.parseInt(t.image); } }
```

t.image is the string matched by the token t

A JavaCC Tokenizer Rule

```
SKIP : { regex | regex | … }
```

Don’t generate a token for expressions that match a regex

```
TOKEN [IGNORE_CASE] : {
    <TOKEN1 : regex >
  | <TOKEN2 : regex >
  | …
}
```

Define the regex to generate each token type

A JavaCC Tokenizer Rule

```
SKIP : { " " }
SKIP : { "\n" | "\r" | "\r\n" }
```

Ignore whitespace
(unless it's the 2nd+ character in a token)

A JavaCC Tokenizer Rule

The string literal ‘SELECT’

an optional leading ‘-’

```
< SELECT  : "SELECT" >
| < DECIMAL : ("-")?(["0"-"9"])+ >
```

?,+,* defined as in rules (standard regex)

[Character Class]
Any character who’s ASCII code is between (inclusive) ‘0’ and ‘9’

A JavaCC Tokenizer Rule

~ inverts a class

Any character that is NOT a single quote.

OR defined as before

```
| < STRING  : "'" ((~["'"] | ("\\'") | ("\\\\"))*) "'" >
| < ID      : (["A"-"Z","_"])(["A"-"Z","0"-"9","_"])* >
```

Any character between A-Z, or an underscore

Invoking JavaCC Parsers

Instantiate your parser with a `java.io.InputStream`

```
public static List<List<Datum[]>> execFile(File fname)
  throws SqlException, IOException, ParseException
{
  Sql parser = new Sql(new FileInputStream(fname));
  Program p = parser.Program();
  return Runtime.execFile(p);
}
```

Each grammar rule defines a class method

Call the class method for the ‘root’ of your grammar.

Testing

- SQL examples in `sql/test`
- Use `edu.buffalo.cse.sql.test.*` to test both RA and SQL execution
  - Successful tests = guaranteed ≥ 90% grade
- Make sure to create your own test cases!