AY 2007-2008 – Facoltà di Ingegneria Informatica

Technologies for Information Systems

POLITECNICO DI MILANO

Parser Design and Implementation for PERLA query language

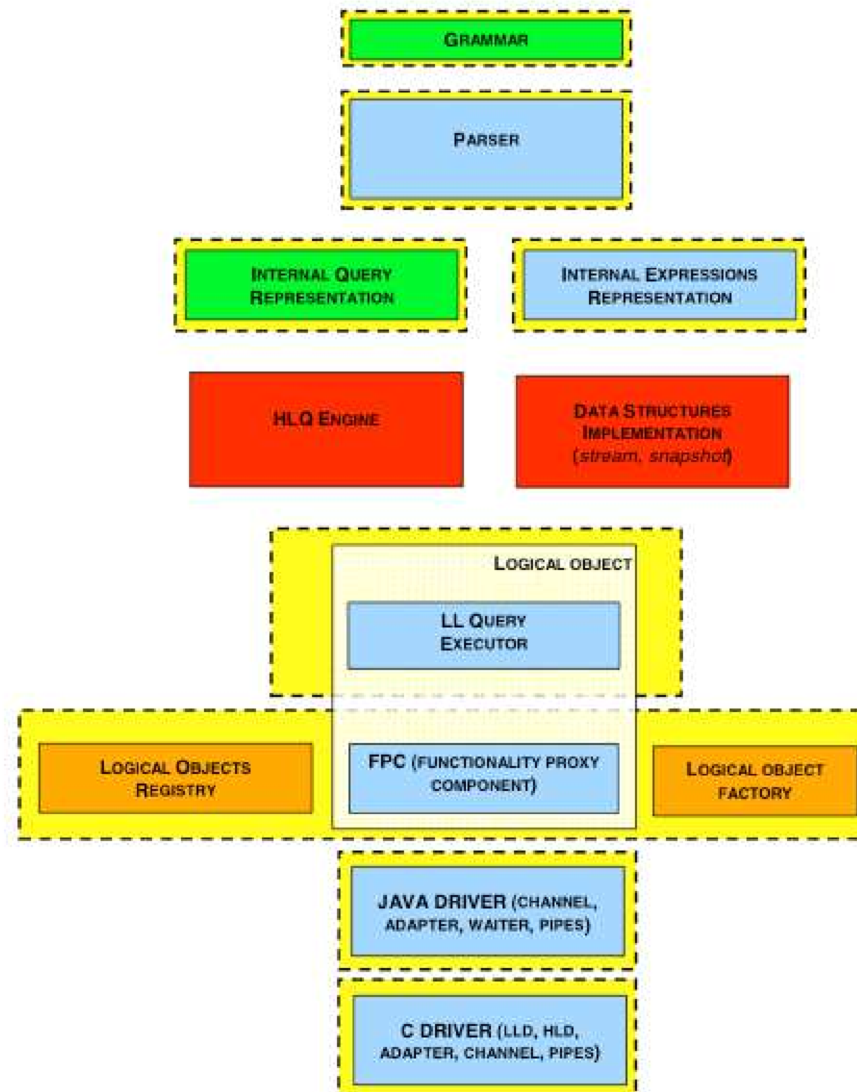
Prazzoli Pierpaolo

Rota Guido

Presentation Index

- Project Overview
- Project Goals
- Parser Model
- Context Classes
- Handler Classes
- Semantic Errors
- User Defined Constants
- Project Status and Conclusions

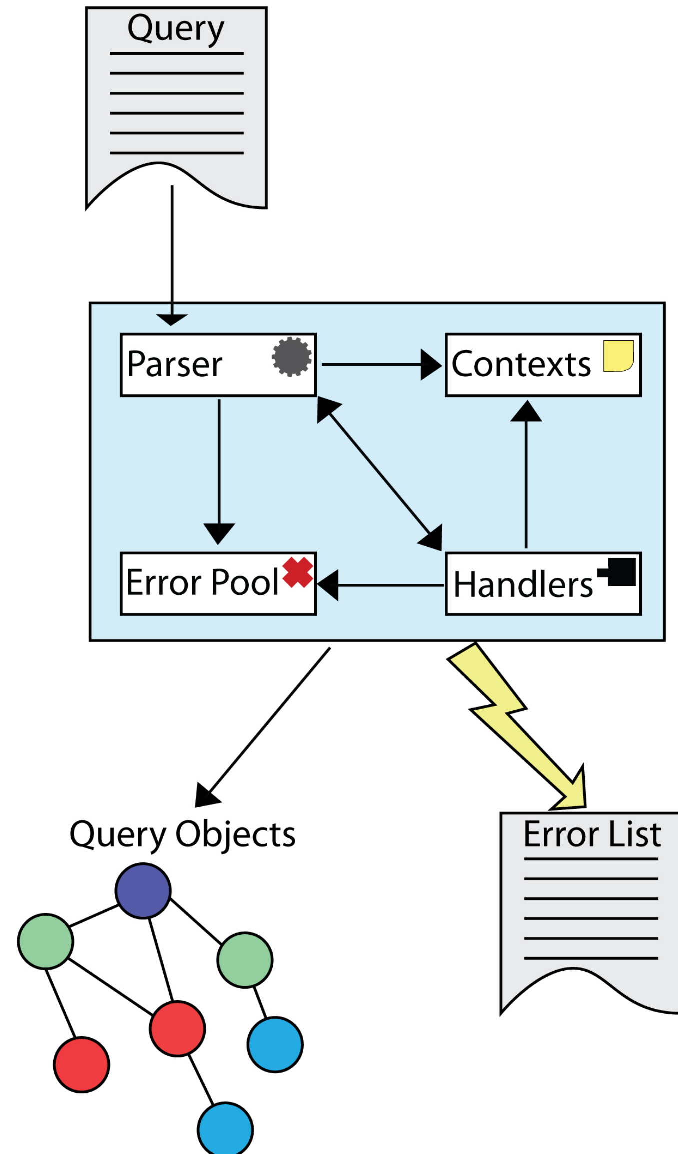
Project Overview



Project Goals

- Extend the existing parser to generate the query objects structure
- Keep the original JavaCC grammar as clean as possible
- Use a modular, extensible and simple solution
- Error checking and reporting (semantic and syntactic)

Parser Model



Inputs

Internal Structure

Outputs

Context Classes

- Static classes used to store objects parameters for objects created in nested productions

- Mostly used to assign production results and to build objects lists

- Reduced parameter passing in the productions and reduced code in the grammar

- Expressions:
  - non static classes since many ExpressionContext are active at the same time
  - stack structure to keep track of the current working ExpressionContext

Context Classes - Example (1/2)

Assignments to context attributes

```
void LowSelectionStatement(ExpressionType parExpressionType) :
{}
{
    LLContext.selectClause = SelectClause(parExpressionType)
    [
        GroupByClause()
    ]
    [
        LLContext.havingClause = HavingClause(parExpressionType)
    ]
    [
        LLContext.upToClause = UpToClause()
    ]
```

List elements created and stored using the context

```
void FieldDefinition() :
{ Token tokenId; FieldType type; Constant defaultValue = null; }
{
    tokenId = Identifier() /* DataStructureField */
    type = FieldType()
    [
        <KEYWORD_DEFAULT>
        defaultValue = SignedConstant()
    ]
    { StatementContext.addField(tokenId, type, defaultValue); }
}
```

Context Classes - Example (2/2)

POLITECNICO DI MILANO

Implementation of StatementContext.*addField*, which creates <Token, Field> elements used to fill DataStructure field lists

```java
/**
 * Aggiunta di un Field a tokenDataStructureFieldList
 * @param tokenName Token contenente di dati del Field
 * @param type Tipo del Field
 * @param defaultValue Costante di default del Field
 */
public static void addField(Token tokenName, FieldType type, Constant defaultValue) {

    if(tokenDataStructureFieldList == null) {

        tokenDataStructureFieldList = new ArrayList<PairValue<Token, Field>>();
    }

    PairValue<Token, Field> value = new PairValue<Token, Field>();
    Field field = new Field();
    field.setDefaultValue(defaultValue);
    field.setName(tokenName.image);
    field.setType(type);

    value.setFirst(tokenName);
    value.setSecond(field);

    tokenDataStructureFieldList.add(value);
}
```

Handler Classes

- Static classes used to create instances of query objects
- Handler methods are called from the grammar when objects can be created
- Parameters:
  - Read from the context
  - Passed by the production
- Semantic checks made in the Handler classes

Handler Classes - Example (1/2)

Handler method called to create an *ExecuteIfClause* object

```
ExecuteIfClause ExecuteIfClause() :
{ Node n; RefreshClause refresh = null; }
{
    <KEYWORD_EXECUTE>
    <KEYWORD_IF>
    n = Expression(ExpressionType.LOW_LEVEL_NO_AGGR_NO_PILOT)
    [
        refresh = RefreshClause()
    ]
    { return ClausesHandler.getExecuteIfClause(n, refresh); }
}
```

Optional parameters are set to null by default and re-set only if needed

Handler Classes - Example (2/2)

ClausesHandler.*getExecuteIfClause* implementation

```
/**
 * Costruisce e recupera la clausola EXECUTE IF
 * @param condition Condizione della clausola
 * @param refresh Clausola di refresh associata
 * @return La clausola ExecuteIfClause creata
 */
public static ExecuteIfClause getExecuteIfClause(Node condition, RefreshClause refresh) {

    ExecuteIfClause executeIf = new ExecuteIfClause();
    executeIf.setCondition(condition);

    if(refresh != null) {

        executeIf.setRefreshClause(refresh);
    } else {

        // crea la clausola di refresh di default
        executeIf.setRefreshClause(new RefreshNever());
    }

    return executeIf;
}
```

Semantic Errors

- Checked inside handlers with support of new classes:
  - *IdTracker*: keeps the associations between created objects and their aliases
  - *Helper*: used to access IdTracker functionality and to check for alias misuses

- Each error is created with a specific priority and stored inside *ErrorPool* class

- Errors are shown according to their priority

- Critical errors stop parser execution

Semantic Errors - Example

- A list of some detected semantic errors:
  - Same alias for different DataSources in FROM
  - Attempt to insert values in fields not declared in the CREATE statement
  - Attempt to use a table that has not been created
  - Different UserDefinedConstants with the same alias
  - UserDefinedConstants with different aliases but same class
  - Attempt to use a field name that belongs to different DataSources without qualifiers in the same query

User Defined Constants

Grammar changed to add the possibility to use *User Defined Constant* inside query expressions

```
ConstantUserDefined ConstantUserDefined() :
{ Token t; ConstantString cs; }
{
    <NEW>
    "("
        t = <IDENTIFIER>
        ","
        cs = ConstantString()
    ")"

    { return ExpressionsHandler.getConstantUserDefined(t, cs); }
}
```

```
Constant Constant() :
{ Constant c; }
{
    (
        c = ConstantNull()
        |
        c = ConstantBoolean()
        |
        c = ConstantString()
        |
        LOOKAHEAD(1)
        c = ConstantInteger()
        |
        c = ConstantFloat()
        |
        c = ConstantUserDefined()
    )
    { return c; }
}
```

XML descriptors used to create the binding between the alias and the class of a User Defined Type

User Defined Constants - Example

XML descriptor specifying two user defined constants

```xml
<UserDefinedConstants>
    <UserDefinedConstant enabled="true">
        <package>org.dei.expressions</package>
        <class>Point</class>
        <alias>Point</alias>
    </UserDefinedConstant>
    <UserDefinedConstant enabled="true">
        <package>org.dei.expressions</package>
        <class>ConstantVector</class>
        <alias>Vector</alias>
    </UserDefinedConstant>
</UserDefinedConstants>
```

Reflection is used to load UserDefinedConstants at runtime

Status:
The project is almost done, we just need to finish the error reporting.

Conclusion:

- We reached our goal to keep the grammar understandable
- A set of guidelines has been written to help developers in their work and to keep future implementations consistent with the current one
- The solution is easy to extend to new grammar constructs
  - Example: the implementation of a new statement, used to set logical objects values, has already been designed
- New error checking routines can use the current error reporting system