1 Domain-specific language

The general domain-specific language consists of two parts: A `query` and a `quantifier`. The query is used to find the matching values and variables, with the quantifier deciding which of the matches to actually use. In the current version there is only one quantifier: `all`, which simply uses everything matched by the query. Further quantifiers can be registered by developers. See the Java class `QuantifierResolverManager` for that purpose.

The query has the following grammar:

```
query ::= term ['.'  term].*
term ::= '*' | 'this' | 'recurse(' term ')' | '(' term ['|' term].* ')' |
identifier
```

A query must therefore consist of at least one term. The semantics are described below. For this, we introduce the following symbols and functions:

M	Set of previously matched values
VAR	Set of all variables
VAL	Set of all values
$\texttt{name}(v)$	Name of variable v
$\texttt{typeof}(v)$	Type of value v
$v.f$	Value of field f of value v or null if there is no such field
$\texttt{fields}(v)$	All the fields of value $vwithnon-nullvalues$

- `identifier`: The semantics of an identifier depend on where within the query it is located. If it is the first element in the query (i.e. the mandatory term), it must be either a variable's name or a type name. The resulting set M' is then the following:

  $$M' = \{v \mid v \in \text{VAR}, \texttt{name}(v) = \text{identifier}\} \cup \{v \mid v \in \text{VAL}, \texttt{typeof}(v) = \text{identifier}\}$$

  In type names, the dots in fully qualified names must be replaced by `/`: For example, `java/util/String` instead of `java.util.String`.
  If it is not the first element in the query, the identifier must stand for a field name. It will then match the field with that name for all existing matches. The resulting set M' is then the following:

  $$M' = \{v.\text{identifier} \mid v \in M, v.\text{identifier} \neq \textbf{null}\}$$

- `'*'`: `*` matches everything. If the first element in the query, it matches all variables and values that currently exist, with the resulting set $M' = \text{VAR} \cup \text{VAL}$. If it is not the first element, it matches all fields of all matched values, with the resulting set M' as follows:

  $$M' = \bigcup_{v \in M} \{v.f \mid f \in \texttt{fields}(v)\}$$

- `'this'`: `this` does not change anything, and simply matches the existing matches again. Therefore the resulting set is $M' = M$.
- `'recurse(' term ')'`: The recurse function recurses on the term. The the resulting set of matches M' can be defined as follows, where term(M) is defined as matching the term term to the set M:

$$M' = M \cup \text{term}(M) \cup \text{term}(\text{term}(M)) \cup \cdots = \sum_{i \in \mathbb{N}} \text{term}(M)^i$$

- `'(' term [ '|' term].* ')'`: The `|` operator applies the given list of terms separately and merges the results. The the resulting set of matches M' can be defined as follows:

$$M' = \sum_{i \in \{1,\ldots,n\}} \text{term}_i(M)$$

1.1 Extensions

The above domain-specific language is extended at multiple points. In those cases, the matches M from a `query` and `quantifier` component are taken and somehow further processed.

1.1.1 Ordering constraints

For an ordering constraint, an additional `order` is applied to the result M. This order has the following syntax:

```
order ::= query ['<' query]+
```

The semantics are as follows: The queries are resolved on each $v \in M$ separately as outlined above, using the `all` quantifier. For each $v \in M$ we then receive n sets $M_{v1}, \ldots, M_{vn}$, where n is the number of queries. All the values $v' \in M_{vi}$ will be ordered horizontally to the left of all values $v'' \in M_{vj}$ for any $i \in \{1, \ldots, n\}, j \in \{2, \ldots, n\} \wedge j > i$. This however only applies for values v' and v'' which are located in the same level of the hierarchy.

1.1.2 String formats

For a string format, an additional `format` is applied to the result M. Based on this format, the String to display the nodes in M with are determined. This format has the following syntax:

```
format ::= [part]+
part ::= '{' query '}' | identifier
```

The semantics are then as follows: The format string is applied separately to each value $v \in M$. The identifier parts are left as is, the `query` parts are resolved on that value $v \in M$ as outlined above, using the `all` quantifier. The resulting string for that

query is then defined as follows, where M' is the result of the query applied to v: If the set M' is empty, the empty string is returned. If the result is non-empty, the value of a random element from that set is returned.

1.1.3 Blacklist

For a blacklist, an additional `comparison` is applied to the result M. Based on this comparison, the values and variables to blacklist are determined. This comparison has the following syntax:

```
comparison ::= part [',' part].*
part ::= query [comparator value]?
comparator ::= '==' | '!='
value ::= 'null' | 'primitive' | '*' | identifier
```

The semantics are as follows: The comparison is applied separately to each value $v \in M$, resulting in some set $M' = \bigcup_{v \in M} \text{comparison}(v)$. All values and variables $v' \in M'$ are then blacklisted, meaning that they are not displayed in the graph on screen.
Understanding the semantics of the comparison is also important. A comparison consists of potentially multiple parts. Each part consists of a query and an optional comparison. Each part is applied independently: $M_v = \bigcup_{p \in \text{parts}} p(v)$ for each $v \in M$.
Each single part then has the following semantics:

- `query`: If the part consists only of a query, it is applied in the usual way with the `all` quantifier as explained earlier.
- `query == value`: If the part consists of a query compared with the `==` comparator, where the query is applied to some value v, the result will be M_v as follows:

  $$M_v = \left\{ v' \mid v' \in \texttt{query}(v), \texttt{equal}(v', \text{value}) \right\}$$

  The semantics of the `equal` function depend on the value to compare to:

  - `'*'`: Always returns `true`.
  - `'null'`: Returns `true` iff the value represents a null object.
  - `'primitive'`: Returns `true` iff the value represents a primitive object. A primitive object can be a usual Java primitive, a String or a primitive boxing object such as `Integer`.
  - `identifier`: Compares the current value to the identifier, returns whether they match.
- `query != value`: Performs the opposite of the `==` operator.

1.1.4 Abstractions

Like the blacklist, abstractions have an additional `comparison` with the same syntax and semantics. They additionally have an integer `count`. This count specifies the number

of objects which must at least match the comparison for one value, in order for an abstraction to occur. The comparison is applied to each $v \in M$ separately. Let $M_v = \texttt{comparison}(v)$. Then the abstraction is applied to create a single value from M_v iff $|M_v| \geq \texttt{count}$.