

COMS 4115 – Programming Languages and Translators

FINAL REPORT


NING YU
NY2186@COLUMBIA.EDU

MARKUSH DESCRIPTION
LANGUAGE FOR
CHEMICAL PATENTS

I. INTRODUCTION

1. Background

Close to 30 percent of all patents are chemical patents, of which roughly 10 percent claim IP rights over novel chemical structures using the Markush notation[1-5]. An example of an approved patent with claims on a series of 4-anilino-3-quinolinecarbonitriles-based chemical structures for the treatment of chronic myelogenous Leukemia (CML) is shown below in Figure 1.


US007417148B2

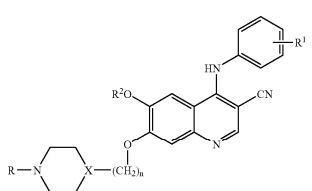
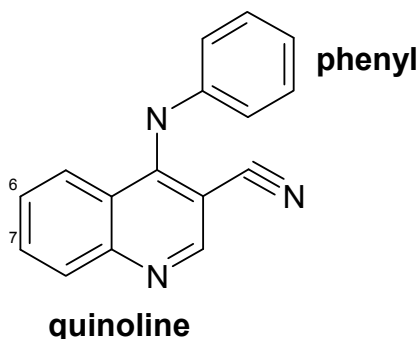
<p>(12) United States Patent Boschelli et al.</p> <p>(54) 4-ANILINO-3-QUINOLINECARBONITRILES FOR THE TREATMENT OF CHRONIC MYELOGENOUS LEUKEMIA (CML)</p> <p>(75) Inventors: Frank Boschelli, New City, NY (US); Kim T. Arndt, Towaco, NJ (US); Jennifer M. Golas, Hewitt, NJ (US)</p> <p>(73) Assignee: Wyeth, Madison, NJ (US)</p> <p>(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 446 days.</p> <p>(21) Appl. No.: 10/980,097</p> <p>(22) Filed: Nov. 3, 2004</p> <p>(65) Prior Publication Data US 2005/0101780 A1 May 12, 2005</p> <p>Related U.S. Application Data</p> <p>(60) Provisional application No. 60/517,819, filed on Nov. 6, 2003.</p> <p>(51) Int. Cl. <i>C07D 215/38</i> (2006.01) <i>A61K 31/47</i> (2006.01)</p> <p>(52) U.S. Cl. 546/159; 546/157; 546/153; 514/313; 514/253.06</p> <p>(58) Field of Classification Search 514/313, 514/253.06; 546/157, 159, 153, 53 See application file for complete search history.</p> <p>(56) References Cited</p> <p style="text-align: center;">U.S. PATENT DOCUMENTS</p> <p>6,002,008 A 12/1999 Wissner et al. 6,780,996 B2 8/2004 Boschelli et al.</p> <p style="text-align: center;">FOREIGN PATENT DOCUMENTS</p> <p>WO 03/093241 A1 11/2003 WO WO 2004/075898 A1 9/2004</p> <p style="text-align: center;">OTHER PUBLICATIONS</p> <p>Boschelli, J Med Chem, 2001, vol. 44, pp. 822-833.* Boschelli, J Med Chem, 2001, vol. 44 pp. 3965-3977.* Registry compound No. 220127-57-1, Mar. 3, 1999.*</p>	<p>(10) Patent No.: US 7,417,148 B2</p> <p>(45) Date of Patent: Aug. 26, 2008</p> <p>Berger, D., et al.; Substituted 4-Anilino-7-phenyl-3-quinolinecarbonitriles as Src Kinase Inhibitors; Bioorg. Med. Chem. Lett. 12:2989-2992 (2002). Boschelli, D.H.; et al.; J. Med. Chem. 44:3965-3911 (2001). Boschelli, D.H.; et al.; J. Med. Chem. 47:1599-1601 (2004). Boschelli, D.H.; et al.; J. Med. Chem. 44:822-833 (2001). Boschelli, D.H.; et al.; Bioorganic & Medicinal Chemistry Letters 13:3797-3800 (2003). Golas, J.M.; et al.; Cancer Research 63:375-381 (Jan. 15, 2003). Ye, F.; 221st National Meeting of the American Chemical Society, San Diego (Apr. 2001). Huy, et al.; "Requirement of Src kinases Lyn, Hck and Fgr for BCR-ABL1-induced B-lymphoblastic leukemia but not chronic myeloid leukemia", Nature Genetics, vol. 36, (2004) pp. 453-461.</p> <p>* cited by examiner</p> <p><i>Primary Examiner</i>—D. Margaret Seaman <i>(74) Attorney, Agent, or Firm</i>—Stephen E. Johnson</p> <p>(57) ABSTRACT</p> <p>Compounds of the formula:</p> <div style="text-align: center;"></div> <p>wherein: n is an integer from 1-3; X is N, CH, provided that when X is N, n is 2 or 3; R is alkyl of 1 to 3 carbon atoms; R¹ is 2,4-diCl, 5-OMe; 2,4-diCl; 3,4,5-tri-OMe; 2-Cl, 5-OMe; 2-Me, 5-OMe; 2,4-di-Me; 2,4-diMe-5-OMe, 2,4-diCl, 5-OEt; R² is alkyl of 1 to 2 carbon atoms, and pharmaceutically acceptable salts thereof.</p> <p style="text-align: center;">12 Claims, No Drawings</p>
--	---

Figure 1 Example of a pharmaceutical patent

In this example, the patent is on the therapeutic potential of the chemical structures bearing a 4-anilino-3-quinolinecarbonitrile core structure, depicted in Figure 2, with varying substituents at the phenyl ring and the 6 and 7 positions of the quinoline ring. The rules for substituents around the core structure are elaborated in prose below the core structure. These rules include, for example, the alkyl chain represented by (CH₂)_n can be of length of 1, 2 or 3; the group represented by X can be either N or CH, provided that when X is N the alkyl chain next to it is of length of 2 or 3 but not 1. The purpose of allowing patent applicant to use the Markush notation to make claims on similar structures is because of the well-known fact that similar structures often share similar properties and,

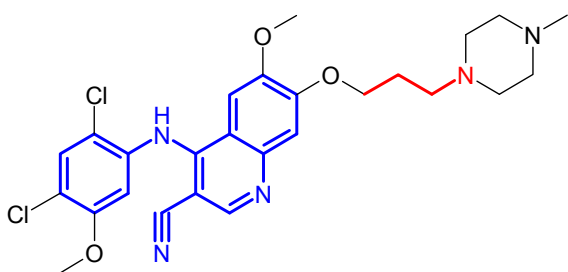
in the medical field, similar pharmacological profiles. Given that the developer of a therapy often only has resources to progress one candidate out of a pool of similar structures for clinical testing, allowing the patent to cover the whole series of structures gives the applicant some protection against its competitors of attempting to develop a variant into a marketable product. On the other hand, patent examiners must balance the interests of patent applicants with those of the public. For example, if an application is so general that it covers any unrelated structures, its patentability must be challenged.



SMILES: N#CC1=C(NC2=CC=CC=C2)C2=CC=CC=C2N=C1

Figure 2 Structure of the 4-anilino-3-quinolinecarbonitrile core

Given a patent, a so-called substructure search can be used to check for a possible infringement[6]. Two examples with similar structures are shown in Figure 3. Both contain the 4-anilino-3-quinolinecarbonitrile substructure depicted in Figure 2 (highlighted in blue), but structure A is covered by the patent whereas B is not. In fact, A is bosutinib, the clinical candidate for treating breast cancer currently in Phase III studies[7]. The reason is because the patent claims that when X is N the neighboring alkyl chain must be 2 or 3 carbons in length (highlighted in red). Note that the 4-anilino-3-quinolinecarbonitrile substructure in the example is laid out differently from that in Figure 2. However, substructure searching algorithms should be able to detect the core structure by traversing the graph, which is a problem typically handled by a Chemistry software toolkit and is not in the scope of the proposed language.

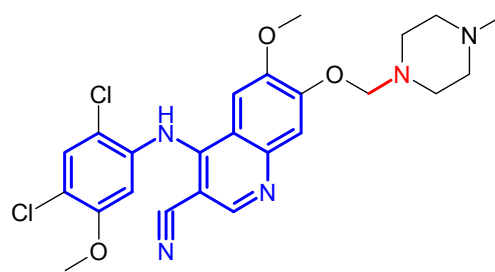


A

(Covered by US 7,417,148)

SMILES:

CN1CCN(CC1)CCCOC2=C(C=C3C(=C2)N=CC(=C3NC4=CC(=C(C=C4Cl)Cl)OC)C#N)O
C



B

(Not covered by US 7,417,148)

SMILES:

COC1=CC(NC2=C(C=NC3=CC(OCN4CCN(C)CC4)=C(OC)C=C23)C#N)=C(Cl)C=C1Cl

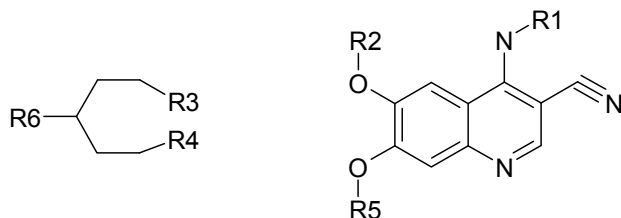
Figure 3 Two examples demonstrating the coverage of patent US 7,417,148

2. Language Proposal

The current system used to describe claims in chemical inventions has many limitations. While the core in a Markush claim provides an unequivocal definition, the variations in the substituents have to be elaborated using an imprecise human language. Thus, when a possible infringement arises where a structure is suspected to be covered by an existing patent, legal professionals have to first determine the core of the patent is a substructure in the structure in question. If so, they need to then match each substituent of the structure against the variations claimed by the patent to determine if the entire structure is covered. Since variations can only be interpreted by human experts, this process is often tedious and error-prone. Similarly, when a patent application containing a Markush claim is reviewed, the patent examiner that is charged with determining the patentability often has a hard time manually searching the patent database and published literature for possible prior arts. For these reasons, the USPTO observed that searches of Markush claims “often consume a disproportionate amount of Office resources as compared to other types of claims[8].”

The Markush Description Language (MDL) proposed here is meant to make claims on variations of chemical structures machine-encodable in order to enable automatic searches. This language will borrow elements from the well-known Simplified Molecular Input Line Entry System (SMILES) [9] and its companion for chemical queries, SMARTS[10], of Daylight Chemical Information, Inc. SMILES and SMARTS are line notations for describing chemical structures as undirected graphs with weights denoting bond orders. Examples of SMILES for the 4-anilino-3-quinolinecarbonitrile core and for the two sample structures are displayed in Figure 2 and Figure 3 respectively. A web service exists that takes a SMILES string as input and generates a graphical depiction[11].

In this language, what we attempt to do is to separate the Markush structure into a fixed core component and a variable number of substituents. This is done by disconnecting the bonds linking a core and a substituent. Once a bond is disconnected, we cap the broken bond at the core end with a dummy R_n atom and the corresponding substituent end with a dummy Z_n atom where n is a unique integer assigned to the disconnected bond. The matching R and Z atoms allow us to reconnect a core to a substituent once it has been fully specified. The R and Z symbols are not part of SMILES/SMARTS and are reserved for this purpose only. For example, after the said bonds are disconnected, the core part of the Markush structure in US 7,417,148 will look like Figure 4.



SMILES: [R6]CCC([R3])CC[R4].[R1]NC1=C(C=NC2=CC(O[R5])=C(O[R2])C=C12)C#N

Figure 4 Core structured used in the Markush Description Language

II. LANGUAGE TUTORIAL

In this section, we will show some sample programs written in MDL that demonstrate the key features of the language.

1. Creating a molecule from atoms

```
/*
 * Main function is the entry point of the program
 */
int main() {
    /* Declare a Mol object */
    Mol mol;

    /*
     * Instantiate three Atom objects, two carbons and one oxygen
     * using chemical literals enclosed in triple quotes, and use
     * the - operator to connect atom 1 and 2 with a single bond
     * and atom 2 and 3 with a single bond, thereby creating the
     * ethanol molecule.
     */
    mol = '''C''' - '''C''' - '''O''';

    /*
     * Call the print global function passing the ethanol molecule
     * as a parameter to print the SMILES representation of it
     */
    print( mol );
}
```

Output:

```
CCO
```

This simple program illustrates the following key MDL features:

- We can use triple quote to enclose chemical literals. When there is a single atom inside the triple quote, MDL creates an **Atom** object, instead of a **Mol** object containing only one atom.
- The – operator is left associative. Thus, the bond between atom 1 and 2 is created before the bond between 2 and 3 is.

One limitation of the – operator is that it can only be used to specify chain-like structures instead of structures containing branches, such as isopropyl (see Figure 5):

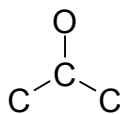


Figure 5 Structure of isopropyl

Such structures can only be specified as a whole SMILES CC(C)O to instantiate a **Mol** object directly as opposed to be constructed from individual atoms.

2. Counting number of atoms

```
int main() {
    Mol aspirin;
    int c_count;
    int o_count;

    aspirin = "'c1ccc(c(c1)OC(=O)C)C(=O)O'";

    c_count = 0;
    o_count = 1;
    for (Atom atom : getAtoms(aspirin)) {
        /* Is atom a carbon? */
        if (getWeight(atom) == 12) {
            c_count = c_count + 1;
        }
        else {
            /* Is atom an oxygen? */
            if (getWeight(atom) == 16) {
                o_count = o_count + 1;
            }
        }
    }

    print( "The input molecule with a structure of " );
    print( aspirin );
    print( " has " );
    print( c_count );
    print( " carbon atoms and " );
    print( o_count );
    print( " oxygen atoms " );
}
}
```

Output:

```
The input molecule with a structure of c1ccc(c(c1)OC(=O)C)C(=O)O has 9
carbon atoms and 5 oxygen atoms
```

This example shows that we can use an iterator-based for loop to iterate over all the atoms in a molecule and do some calculations.

3. Calculating molecular weight

```

int global;

/*
 * Function to compute molecular weight of an input molecule
 */
int computeWeight(Mol mol) {
    int weight;
    weight = 0;

    /*
     * getAtoms(Mol) is a library function that returns a list/tuple of
     * atoms in the mol
     */
    for (Atom atom : getAtoms(mol)) {
        weight = weight + getWeight(atom);
    }
    return weight;
}

/*
 * Main function is the entry point of the program
 */
int main() {
    Mol a;

    /* Set Mol a to the aspirin structure */
    a = '''CC(=O)Oc1ccccc1C(=O)O''';

    global = computeWeight( a );

    print( global );
}

```

Output:

172

In this example, we create a reusable MDL function, `computeWeight`, which takes a **Mol** as input and returns the molecular weight computed as the sum of the weights of all the constituent atoms.

Within this function, we invoke two global library functions `getAtoms` and `getWeight`. The former returns a tuple of atoms to be consumed by the iterator variable defined by the **for** loop and allows for an atom-by-atom traversal of the whole molecule as described in Section III; the latter returns the weight of the input atom according to its element as detailed in Section III.

We also demonstrate that we can assign the return value of `computeWeight` to a global variable and print the value of the variable afterwards. Global variables can be used to share data among multiple functions of a MDL program.

4. Depth-first Traversal of a molecule structure

```
/*
```



```

* Function to traverse a molecule structure starting at a given
* atom through a recursive depth-first search
*/
int dfs(Atom v) {
    Atom atom;

    /* Print the unique index of the atom in the enclosing molecule */
    print(getIdx(v) + " ");

    /* Mark the atom as visited */
    setMarked(v);

    /* Iterate over all the bonds incident to v */
    for (Bond bond : getBonds(v)) {

        /* If the bond has not been discovered, explore it */
        if (!isMarked(bond)) {
            setMarked(bond);

            /*
            * If not, retrieve the atom on the opposite
            * end of the bond
            */
            atom = getNbr(bond, v);

            /*
            * If atom has not been discovered, explore that
            * atom recursively
            */
            if (!isMarked(atom)) {
                dfs(atom);
            }
            /*
            * If it has been discovered, the bond is a back edge.
            * But we do not do anything here.
            */
        }
    }

    /* Return a value to match the declared type of the function */
    return 0;
}

int main() {
    Mol aspirin;
    Atom atom;
    int index;
    int idx1;
    int idx2;

    aspirin = "'c1cccc(c(c1)OC(=O)C)C(=O)O'";

    print("Connection table for aspirin is: \n");

    /*
    * Print a list of the bonds (edges) of the structure as
    * (u, v) where u and v are atom indices, ignoring the

```

```

    * detail of the bonds (e.g. whether single or double)
    */
index = 0;
for (Atom it1 : getAtoms(aspirin)) {
    /* Save the first atom as the starting point of DFS */
    if (index == 0) {
        atom = it1;
    }
    index = index + 1;

    idx1 = getIdx(it1);
    for (Atom it2 : getAtoms(it1)) {
        idx2 = getIdx(it2);
        if (idx1 < idx2) {
            print("(" + idx1 + ", " + idx2 + ") ");
        }
    }
}
print("\n\n");

/* Now perform DFS, starting with the picked atom */
print("DFS produces: \n");
dfs(atom);
print("\n");
}

```

Output:

```

Connection table for aspirin is:
(0, 5) (0, 1) (1, 2) (2, 3) (3, 4) (3, 10) (4, 5) (4, 6) (6, 7) (7, 8) (7,
9) (10, 11) (10, 12)

DFS produces:
0 5 4 3 2 1 10 11 12 6 7 8 9

```

This example demonstrates how one can write an MDL program to perform depth-first search on the aspirin structure (Figure 6). As the output shows, starting at atom 0, the carbon on the lower right corner, DFS follows 0 5 4 3 2 1 to hit a dead end, before backtracking to explore the branches of atoms 10, 11, 12 and atoms 6, 7, 8, and 9.

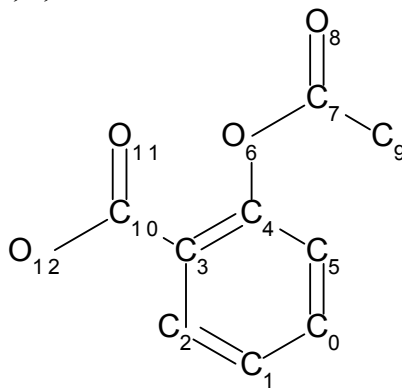


Figure 6 Structure of aspirin

The main MDL language features demonstrated in this example are:

- In both functions main and dfs, the global getIdx function is called on an atom to obtain the unique index of the atom in the molecule structure. This index is assigned to the atom when the molecule object is instantiated and stays fixed throughout the lifetime of the molecule object.
- When the getBonds global function is called on an atom, it returns a tuple of **Bond** objects of bonds incident to the input atom. This gives us a way to explore the neighbor list of atoms.
- When the getNbr global function is called on a bond and an atom, it returns the atom on the other end of the bond opposite to the input atom.
- For both bonds and atoms, we can use the setMarked/isMarked functions to mark/unmark them and test for marking status.
- The getAtoms global function is overloaded for both **Mol** and **Atom** objects. In the former case, it returns a **Tuple** of atoms contained by the molecule. In the latter case, it returns a **Tuple** of atoms in the neighbor list of the input atom, which can also be retrieved by two successive calls to getBonds and getNbr. But that is less convenient than the form involving only one call.

5. Performing Markush Analysis on Patent Claims

```
/*
 * Function to compute molecular weight of an input molecule
 */
int computeWeight(Mol mol) {
    int weight;
    weight = 0;

    /*
     * getAtoms(Mol) is a library function that returns a list/tuple of
     * atoms in the mol
     */
    for (Atom atom : getAtoms(mol)) {
        weight = weight + getWeight(atom);
    }
    return weight;
}

/*
 * Function to create an ethanol molecule
 */
Mol createEthanol() {
    /*
     * The - operator between two chemistry objects (Mol or Atom)
     * connects the two objects. In this particular case, we connect the
     * first two atoms and the second two atoms with two single bonds,
     * which creates an ethanol molecule.
     */
    return '''C''' - '''C''' - '''O''';
}

/*
```

```

* Main function is the entry point of the program
*/
int main() {
    int count;

    /* Declare a, c and d as Mol objects */
    Mol [] a;
    Mol c;
    Mol d;

    /* Declare b and markush as Mol Tuple objects */
    Mol [] b;
    Mol [] markush;

    /* Assign a to half of the aspirin structure*/
    a = [''OC(=O)C([R1])=C[R2]'''];

    /*
    * Assign b:
    * b[0] is the other half of the aspirin structure;
    * b[1] is an unrelated structure
    */
    b = [''[R2]C=CC=C([R1])OC(=O)C'', ''[R2]CCCC[R1]'''];

    /*
    * The + operator between a Mol[] and a Mol combines each
    * element in the LHS with the RHS Mol and applies the
    * PLUS operation. The PLUS operator glues each [R] atom
    * with the corresponding [R] atom to make a new bond.
    * The result is a new mol tuple where the first element
    * is aspirin and the second is an unrelated structure.
    */
    markush = a + b;

    /* Compute the molecular weight of each element in markush */
    print("The Markush structure is as follows: \n");

    count = 0;
    for (Mol mol : markush) {
        count = count + 1;
        print("Alternative #" + count + ": structure is ");
        print(mol);
        print("\tmolecular weight is " + computeWeight(mol) + "\n");
    }
    print("\n");

    /*
    * Assign c to ethanol -- this is a tedious way to
    * instantiate molecules
    */
    c = createEthanol();

    /* Does markush cover c? */
    if (markush covers c) {
        print("The markush covers ");
    }
    else {

```

```

        /* We should answer no */
        print("The markush does not cover ");
    }
    print(c);
    print("\n");

    /* Assign d to another representation of the aspirin structure */
    d = '''c1ccc(C(=O)O)c(OC(=O)C)c1''';

    /* Does markush cover d? */
    if (markush covers d) {
        /* We should answer yes */
        print("The markush covers ");
    }
    else {
        print("The markush does not cover ");
    }
    print(d);
    print("\n");
}

```

Output:

```

The Markush structure is as follows:
Alternative #1: structure is c1(ccccc1C(=O)O)OC(=O)C  molecular weight is
172
Alternative #2: structure is C1CCCC=C1C(=O)O  molecular weight is 116

The markush does not cover CCO
The markush covers c1ccc(c(c1)OC(=O)C)C(=O)O

```

Note the SMILES above can be examined at the following hyperlinks.

- c1(ccccc1C(=O)O)OC(=O)C <http://www.daylight.com/dayhttp/smi2gif?output=png&colorscheme=COW&linewidth=10&smiles=c1%28ccccc1C%28%3DO%29O%29OC%28%3DO%29C>
- C1CCCC=C1C(=O)O <http://www.daylight.com/dayhttp/smi2gif?output=png&colorscheme=COW&linewidth=10&smiles=C1CCCC%3DC1C%28%3DO%29O>
- CCO <http://www.daylight.com/dayhttp/smi2gif?output=png&colorscheme=COW&linewidth=10&smiles=CCO>
- c1ccc(c(c1)OC(=O)C)C(=O)O <http://www.daylight.com/dayhttp/smi2gif?output=png&colorscheme=COW&linewidth=10&smiles=c1ccc%28c%28c1%29OC%28%3DO%29C%29C%28%3DO%29O>

This example builds up from Examples 1 and 3, and demonstrates the following key features:

- In the line that reads “markush = a + b” where both a and b evaluate to **Mol Tuples** that contains R-groups in the chemical literals, the + operators effectively glues the corresponding R-groups separated in different structures and produces a **Tuple** of resultant molecules. This effectively simulates the process of combining a core in a patent with a set of possible variations at one of the substituents in order to enumerate the chemical space covered by the patent claims. It is worth noting that the + operator does not guarantee that all the R-groups in the core will be matched by the substituent, which is the case when there are more than one substituent such as in the motivating example presented in Section I. In

such cases, the resultant intermediate molecules will then be joined with other substituents successfully to fully exercise the patent claims.

- The **covers** operator takes a **Mol Tuple** as the Markush claim on the left hand side and a query **Mol** on the right hand side, and answers the question whether the query molecule falls into the proprietary chemical space claimed by the Markush.

III. LANGUAGE REFERENCE MANUAL

The Markush Description Language for Chemical Patents (MDLCP or MDL for short) is motivated by the desire to accurately express Markush-based claims on chemical structures frequently used in chemical patents. In the current system, these claims are recited in a human language, which complicates their interpretation and makes prior art searches and patentability determination very difficult. MDL allows patent workers to specify variations in chemical structures using a programming language to facilitate computerized processing of such patents.

In order to accomplish this goal, MDL must solve two key questions: one, how to express Markush-based chemical claims as special types of objects in a general-purpose programming language; and two, what kind of operations should be allowed on these objects and how should they be implemented.

The first question is addressed by borrowing elements from the well-known Simplified Molecular Input Line Entry System (SMILES) and its companion for chemical queries, SMARTS, of Daylight Chemical Information, Inc. SMILES and SMARTS provide natural ways to describe chemical structures using letters, digits and common symbols, which make the expressed structures human readable too. Additionally, MDL follows the hydrogen-suppressed convention of the SMILES system, allowing hydrogen atoms to be implied instead of specified.

On the second question, two key operations have been identified and assigned to the “+” and “-” operators respectively. The “+” operator handles merging, namely combining all matching [R] and [Z] groups in the two operands and fusing them to create a new molecule. The “-” operator, on the other hand, creates a single bond between the two operands to form a new molecule. This designation, while may seem unintuitive, is consistent with the SMILES language which uses “-” to denote a single bond, “=” a double bond, and “#” a triple bond. Finally, instead of supporting these operations natively, the MDL cross-compiles a MDL program into source code in a target language, where the low-level Chemistry-related operations are dispatched to a library named OEChem provided by [OpenEye Scientific Software](#) to do the actual processing. At the present, the target language is chosen to be Java.

1. Lexical Conventions

1.1 Comments

The characters `/*` introduces a comment, which terminates with the characters `*/`.

1.2 Identifiers

An identifier contains a sequence of alphanumeric characters, which starts with a character. The character `_` is also allowed, provided that it is not the first character. Identifiers in MDL are case sensitive.

1.3 Keywords

The following identifiers are reserved for use as keywords and may not be used otherwise:

Atom	Bond	Mol	covers
else	for	if	int
return	void	while	String
boolean			

1.4 Special Characters

<code>[]</code>	Brackets are used to enclose tuples, e.g. <code>[''[Z2]C=CC=C([Z1])OC(=O)C''', ''[Z2]CCCC[Z1]'']</code> is a tuple of two chemical literals.
<code>()</code>	Parentheses are used to indicate precedence in expressions and to enclose formal arguments to functions.
<code>{}</code>	Braces are used to enclose the body of a function or a list of statements.
<code>=</code>	Assignment operator.
<code>,</code>	Commas are used to separate elements in tuples and formal/actual arguments in parentheses.
<code>;</code>	Semicolons are used to terminate statements.
<code>'''</code>	Triple quotes are used to enclosed chemical literals such as Atoms, Bonds and Molecules. For example, <code>'''C'''</code> means a carbon Atom object, <code>'''='''</code> means a double bond object, etc.
<code>"</code>	Double quotes are used to enclose string literals.

2. Types

- **int:** The int data type is a 32-bit signed integer. It has a minimum value of -2,147,483,648 and a maximum value of 2,147,483,647 (inclusive).
- **Boolean:** The boolean type represents the logical values of **true** or **false**, which are the only possible **boolean** literals in MDL.

- **Atom:** The Atom type represents a chemical atom object. It is internally implemented with the [OEAtomBase](#) class of the OEChem library. Examples of literals that can be used to instantiate an Atom object are shown in the table below:

<code>'''C'''</code>	A carbon atom
<code>'''N'''</code>	A nitrogen atom
<code>'''O'''</code>	An oxygen atom
<code>'''F'''</code>	A fluorine atom
<code>'''[Cl-]'''</code>	A chlorine anion
<code>'''[R1]'''</code>	A dummy R atom capping a position numbered 1 of a core
<code>'''[Z2]'''</code>	A dummy Z atom capping a position numbered 2 of a substituent

- **Bond:** The Bond type represents a chemical bond object. It is internally implemented with the [OEBondBase](#) class of the OEChem library. Examples of literals that can be used to instantiate a Bond object are shown in the table below:

<code>'''-'''</code>	A single bond
<code>'''='''</code>	A double bond
<code>'''#'''</code>	A triple bond
<code>''':''''</code>	An aromatic bond
<code>'''/''''</code>	An "up" single bond (next to a double bond for cis-trans specification)
<code>'''\''''</code>	A "down" single bond (next to a double bond for cis-trans specification)

- **Mol:** The Mol type represents a chemical molecule object. It is internally implemented with the [OEMolBase](#) class of the OEChem library. Examples of literals that can be used to instantiate a Mol object are shown in the table below:

<code>'''CCO'''</code>	An ethanol molecule
<code>'''O=C=O'''</code>	A carbon dioxide molecule
<code>'''C#N'''</code>	A hydrogen cyanide molecule
<code>'''[Na+][Cl-]'''</code>	A sodium chloride molecule
<code>'''CC(=O)O'''</code>	An acetic acid molecule
<code>'''c1ccccc1'''</code>	A benzene molecule

Note that specification of hydrogens atoms is not required for chemical literals that evaluate to a Mol object, since MDL uses the so called implicit hydrogen notation which allows hydrogen counts for each non-hydrogen atom to be derived based on a set of valence rules and the bond structure.

- **String:** The String type represents a sequence of ASCII characters. String literals start with a double quote and are terminated by the first non-escaped double quote. Non-printable characters may be embedded in a string by escaping them with a `\` prefix. The following is a list of escape sequences supported by MDL:

<code>\b</code>	backspace
<code>\n</code>	newline
<code>\r</code>	carriage return

<code>\t</code>	tab
<code>\'</code>	single quote
<code>\"</code>	double quote
<code>\\</code>	backslash

- **Tuple:** A tuple is an immutable homogeneous list of ints, booleans, Mols, Atoms, Bonds, or Strings.

3. Expressions

3.1 Primary Expressions

- Identifier
- Integer literal
- Boolean literal
- String literal
- Chemical literal
- Parenthesized expression
- Function invocation

3.2 Unary Operators

3.2.1 *! expr*

This operator applies only to expressions that evaluate to a **boolean**. The result is **false** if *expr* evaluates to **true**, and vice versa.

3.3 Multiplicative Operators

3.3.1 *expr * expr*

This operator applies only when both LHS and RHS evaluate to an **int**, and the result is their product.

3.3.2 *expr / expr*

This operator applies only when both LHS and RHS evaluate to an **int**, and the result is the quotient rounded to the nearest integer.

3.4 Additive Operators

3.4.1 *expr + expr*

- When both LHS and RHS evaluate to an **int**, the result is their sum.
- If either side evaluates to a **String**, the result is obtained by converting the other side to **String** and concatenating the two strings together. The rules for converting other types to **String** are as follows:

- Integers are converted to a sequence of digits according to their value, with negative integers having a negative sign in front of them.
- Booleans are converted to the strings of **true** or **false** according to their value.
- Mols are converted to a SMILES representation of their structure.
- No other conversions are performed.
- Otherwise
 - If both operands evaluate to a **Mol**, the result is a **Mol** with the matching [Rn] and [Zn] atoms merged.
 - If at least one operand evaluate to a **Mol[]**, the result is a **Mol[]** formed by exhaustively combining the elements from the LHS and RHS and merging the matching [Rn] and [Zn] atoms.
- No other type combinations are allowed.

3.4.2 *expr - expr*

- When both LHS and RHS evaluate to an **int**, the result is their difference.
- If both operands evaluate to a **Mol** or an **Atom**, the result is a combined new **Mol** with a single bond between the last atom of the LHS and the first atom of the RHS.
- No other type combinations are allowed.

3.5 Relational operators

- `expr < expr`
- `expr > expr`
- `expr <= expr`
- `expr >= expr`

All four operators above only apply if both operands evaluate to an **int**. They yield a **boolean** value representing the result of the evaluation.

3.6 Equality operators

- `expr == expr`
- `expr != expr`

Both operators above only apply if both operands evaluate to an **int**. They yield a **boolean** value representing the result of the evaluation.

3.7 Covers expression

The **covers** keyword is an operator that takes a **Mol** or **Mol[]** object on the LHS and a **Mol** object on the RHS and returns **true** if the RHS is found to be covered by the Markush claims recited by the LHS after a substructure search.

3.8 Assignment expression

An assignment expression has the following form:

$$\text{identifier} = \text{expr}$$

which can only be valid if one of the following is true:

- If identifier is of type int, expr must evaluate to an int
- If identifier is of type Atom, expr must evaluate to an Atom
- If identifier is of type Bond, expr must evaluate to a Bond
- If identifier is of type Mol, expr must evaluate to either a Mol or an Atom
- If identifier is of type tuple, expr must evaluate to a tuple of compatible objects

The assignment operator groups right-to-left and returns the value that is stored in the identifier, allowing for example `id1 = id2 = expr` to set the value of `expr` to both `id1` and `id2`.

4. Declarations

Declarations are used to specify types for

- Global variables outside function definitions
- Local variables within function definitions

Declarations have the following form:

```
type identifier;
```

where type is one of the items described above in Section 3.

5. Statements

5.1 Expression statement

Expression statements have the following form:

```
expr;
```

The most frequently used expressions in expression statements are assignments and function calls.

5.2 Compound statement

Compound statements are used to group several statements together. They have the following form:

```
{ expr1; expr2; ...; }
```

5.3 Conditional statement

Conditional statements have two possible forms:

```
if ( expr ) stmt1  
if ( expr ) stmt1 else stmt2
```

The expression and statements are evaluated lazily as follows: if `expr` evaluates to **true**, then `stmt1` is executed; otherwise, in the first form nothing further is done and in the second form `stmt2` is executed. Following the convention, in case there are nested **if-else** statements, every **else** is matched with the last encountered elseless **if**.

5.4 While statement

While statements have the following form:

```
while ( expr ) stmt
```

If `expr` evaluates to **true**, the `stmt` is executed. After every execution, `expr` is re-evaluated and, if it still evaluates to **true**, `stmt` is re-executed. As soon as `expr` evaluates to **false**, the **while** statement terminates.

5.5 For statement

For statements have the following form:

```
for ( type identifier : expr ) stmt
```

This construct mimics the iterator-based for loops found in many modern programming languages such as Java and Python. For a for statement to be valid, `expr` must evaluate to a tuple. A for statement is evaluated as follows:

- First the intended type of the identifier is checked against the value of the expression. For example, if `expr` evaluates to `Mol[]`, `type` must be `Mol`; if `expr` evaluates to `Atom[]`, `type` must be `Atom` or `Mol`.
- A local temporary variable named by the identifier is declared as described in Section 5
- The identifier is assigned to the value of every element in the tuple returned by evaluation of the expression in succession as done in Section 4.8.
- After every assignment, `stmt` is executed once. In total, it is executed exactly the length-of-the-tuple times during execution of the for statement.

5.6 Return statement

Return statements have the following form:

```
return expr;
```

For a return statement to be valid, `expr` must evaluate to the same type as that defined by the enclosing function. If so, the value that `expr` evaluates to is assigned to the caller of the function.

6. Function Definitions

Function definitions have the following form:

```
fdecl:  
  type identifier ( formals_list ) { vdecl_list stmt_list }
```

where,

```
formals_list:  
    type identifier  
    formal_list, type identifier
```

and where `vdecl_list` is a list of variable declarations as described in Section 5 and `stmt_list` is a list of statements as described in Section 6.

An MDL program consists of a series of global variable declarations and function definitions.

7. Scope and Names

7.1 Static Scoping

MDL follows the rules of static scoping, meaning that the context in which an identifier is resolved is solely determined from the static program structure, instead of from the runtime call stack.

7.2 Lexical Scope

- Variables declared outside function definitions are global variables, which are visible in every function and persist up through the end of the program.
- Variables declared in the formal argument list or within the body of a function are local variables, which are only visible in the function in which they are defined and persist up through the end of the function call.
- A variable declared in a for loop is local to the body of the for statement only and persists up through the end of the statement execution.
- A function becomes visible to any code that follows it immediately after its declaration, which means recursive calls are supported.

7.3 Scope Conflict

Re-declaration of any variable under the same name or any function with the same signature is not allowed.

8. Global Library Functions

8.1 `print`

This function takes one parameter of the following type and prints the textual representation of the parameter as explained below:

- **int**: prints the integer value
- **Mol**: prints the SMILES of the molecule
- **String**: prints the string with escape sequences replaced by appropriate control characters
- **boolean**: prints the string of **true** or **false** according to the boolean value

8.2 getWeight

This function takes an **Atom** object and returns the atomic weight of the atom.

8.3 getAtoms

This function takes one parameter of the following type and returns a **Tuple** of **Atom** objects as explained below:

- **Mol**: returns all the atoms contained by the **Mol**
- **Atom**: returns all the atoms in the neighbor list of the **Atom**

8.4 getBonds

This function takes one parameter of the following type and returns a **Tuple** of **Bond** objects as explained below:

- **Mol**: returns all the bonds contained by the **Mol**
- **Atom**: returns all the bonds incident to the **Atom**

8.5 getIdx

This function takes an **Atom** object and returns an integer representing the index of the **Atom** in the **Mol**. Throughout the lifetime of the **Mol** object containing the atom, the indexes of atoms are guaranteed to stay fixed.

8.5 getNbr

This function takes a **Bond** object and an **Atom** object on one end of the bond and returns the atom that is on the other end of the bond.

8.4 setMarked

This function takes one parameter of the following type and a **boolean** and generates a side effect on the first parameter as explained below:

- **Atom**: if the second parameter evaluates to **true** sets the atom as marked, and vice versa
- **Bond**: if the second parameter evaluates to **true** sets the bond as marked, and vice versa

8.4 isMarked

This function takes one parameter of either **Atom** or **Bond** type and returns a **boolean** indicating whether or not the **Atom** or **Bond** is marked.

IV. PROJECT PLAN

1. Development process

I started by creating a very simple initial version by modifying MicroC. This version contained a scanner, a parser, an AST, a pretty printer and a command-line driver. Most of the data types and language constructs were supported in that version. After the midterm, I started looking into static and semantic analysis. Initially I thought about defining an intermediate Semantic Abstract Syntax Tree to encapsulate the MDL language and using a separate module to translate from an SAST to an AST based on the Java language. After some trial and error I aborted the effort and decided to combine Semantic Analysis and Java Translation into one module. Once that was done, I was able to quickly put together the Java dependencies to support the global library functions such as print, getAtoms, getBonds, etc. Towards the end, I also thought about writing a MDL pretty printer similar to the dump function of Python's AST module. However, this was not undertaken due to time constraints.

2. Development environment and programming style

The code was developed using OcaIDE (depicted in Figure 7) on Eclipse 3.5 (Galileo). The build system was ocamlbuild, the default suggested by OcaIDE. This combination of choices has the following distinct advantages:

- Syntax highlighting improves code readability
- Function definitions can be looked up by holding the Ctrl key while clicking on a function reference
- Pop-up text after hovering the mouse over OCaml library functions makes it easy to view the documentation
- Content assist does a good job at reducing mistypes
- Code cleaning/reformatting ensures a consistent appearance
- It was easy to manage change sets and experimental code with the seamless subversion integration
- Built-in Camlp4 syntax checker provides instantaneous feedback on mismatched parentheses and quotes
- Ocamlbuild in combination with Eclipse's Problems view makes it easier to understand compilation errors and improves the speed of error correction

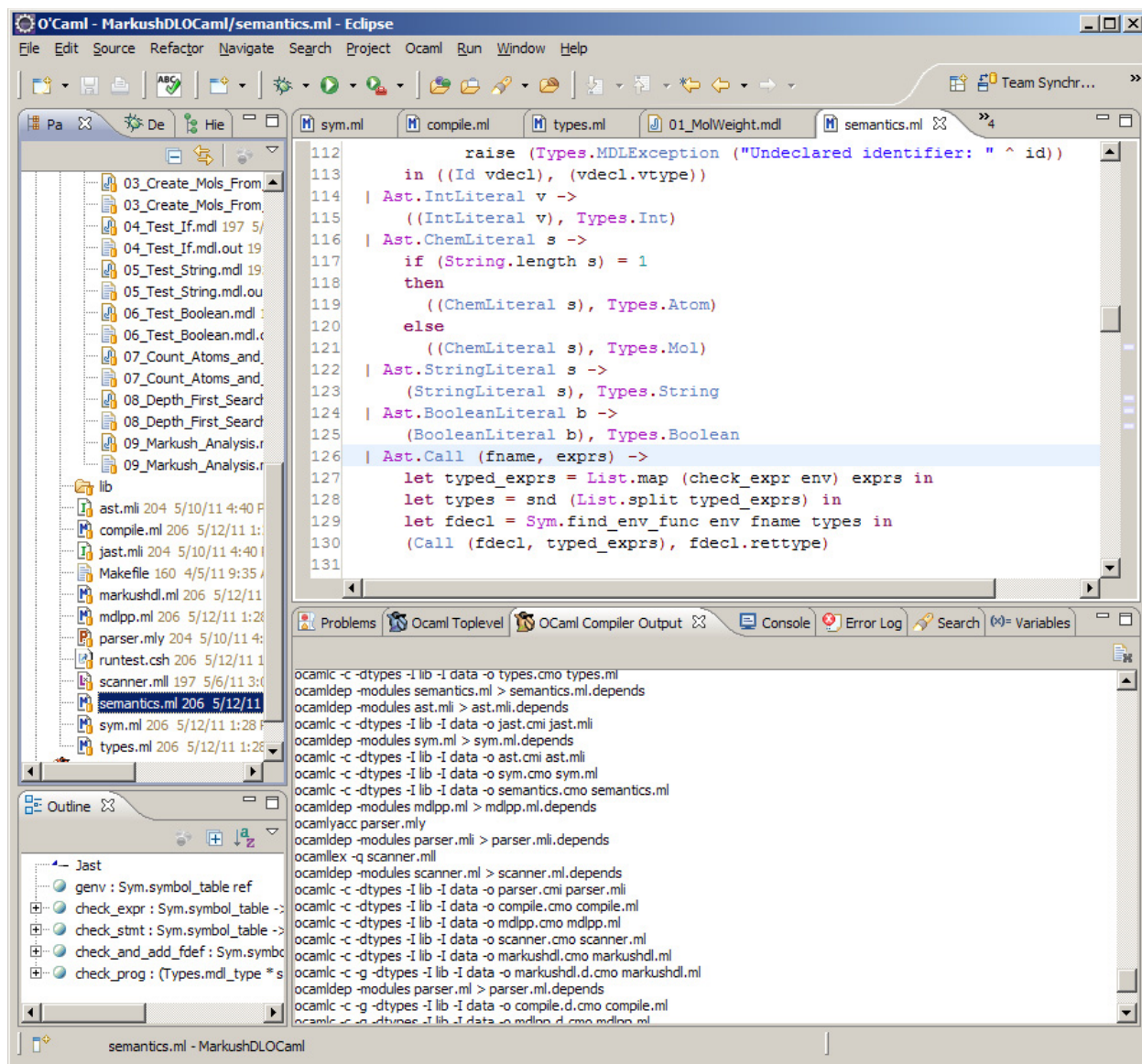


Figure 7 OcaIDE plug-in on Eclipse 3.5

3. Project timeline as shown by SVN log

r206 | yun03 | 2011-05-12 13:28:36 -0400 (Thu, 12 May 2011) | 3 lines

- * Final clean-up of the code
- * Removed outdated comments and dead code
- * Fixed a small error introduced to test 01

r205 | yun03 | 2011-05-12 13:14:26 -0400 (Thu, 12 May 2011) | 1 line

r204 | yun03 | 2011-05-10 16:40:54 -0400 (Tue, 10 May 2011) | 2 lines


```
* Added support for elseless if statements
* Updated the test suite
-----
r203 | yun03 | 2011-05-09 16:50:44 -0400 (Mon, 09 May 2011) | 1 line

* Cleaned up and reformatted code
-----
r202 | yun03 | 2011-05-09 16:42:44 -0400 (Mon, 09 May 2011) | 2 lines

* Updated the test suite
-----
r201 | yun03 | 2011-05-09 16:36:44 -0400 (Mon, 09 May 2011) | 4 lines

* Added several additional APIs for Atom/Bond structure traversal
* Added a test for DFS
* Added a test for Patent Markush Analysis
-----
r200 | yun03 | 2011-05-09 10:17:14 -0400 (Mon, 09 May 2011) | 1 line

* Added the ability to define recursive functions
-----
r199 | yun03 | 2011-05-07 21:39:32 -0400 (Sat, 07 May 2011) | 1 line

* Beautified semantics and jast
-----
r198 | yun03 | 2011-05-06 15:11:51 -0400 (Fri, 06 May 2011) | 1 line

* Revised RxnUtils.covers method in response to the newly introduced boolean
type
-----
r197 | yun03 | 2011-05-06 15:01:47 -0400 (Fri, 06 May 2011) | 2 lines

* Introduced a new type boolean
* Updated regression tests
-----
r196 | yun03 | 2011-05-06 13:21:23 -0400 (Fri, 06 May 2011) | 1 line

* Added code for two-step joining of fragments
-----
r195 | yun03 | 2011-05-06 12:50:32 -0400 (Fri, 06 May 2011) | 1 line

* Created Java JUnit test suite for RxnUtils
-----
r194 | yun03 | 2011-05-06 12:25:14 -0400 (Fri, 06 May 2011) | 1 line

* Updated the regression tests
-----
r193 | yun03 | 2011-05-06 12:24:48 -0400 (Fri, 06 May 2011) | 2 lines

* Added the String type
* Updated the regression tests
-----
r192 | yun03 | 2011-05-06 10:04:22 -0400 (Fri, 06 May 2011) | 1 line
```

r191 | yun03 | 2011-05-06 09:56:24 -0400 (Fri, 06 May 2011) | 1 line

* Created regression test suite

r190 | yun03 | 2011-05-06 09:55:36 -0400 (Fri, 06 May 2011) | 1 line

* Added the covers() and getAtomicWeight(OEAtomBase) methods in RxnUtils

r189 | yun03 | 2011-05-06 09:01:07 -0400 (Fri, 06 May 2011) | 1 line

* Added ant script for building Java project

r188 | yun03 | 2011-05-06 09:00:28 -0400 (Fri, 06 May 2011) | 1 line

* Added code in RxnUtils.java for joining fragments

r187 | yun03 | 2011-05-05 11:59:54 -0400 (Thu, 05 May 2011) | 2 lines

* Completed wholesale changes

* We now directly translates an AST to a JAST

r186 | yun03 | 2011-05-05 09:13:48 -0400 (Thu, 05 May 2011) | 1 line

* Saving the codebase before making wholesale changes

r185 | yun03 | 2011-05-04 15:44:52 -0400 (Wed, 04 May 2011) | 2 lines

* Added Java AST

* Moved Types.string_of_type to mdlpp.ml

r184 | yun03 | 2011-05-04 10:08:34 -0400 (Wed, 04 May 2011) | 1 line

* Added support for the + operator on two Mol objects

r183 | yun03 | 2011-04-28 15:26:50 -0400 (Thu, 28 Apr 2011) | 1 line

* Beautified the compile.ml file

r182 | yun03 | 2011-04-25 16:19:13 -0400 (Mon, 25 Apr 2011) | 1 line

* Added check for types involved in iterated-based for loops

r181 | yun03 | 2011-04-25 10:18:40 -0400 (Mon, 25 Apr 2011) | 1 line

* Completed the Java printer to generate Java code from a Semantic AST

r180 | yun03 | 2011-04-22 17:19:55 -0400 (Fri, 22 Apr 2011) | 1 line

* Fixed code for semantic type checking of lists and for loops

r179 | yun03 | 2011-04-22 16:48:29 -0400 (Fri, 22 Apr 2011) | 2 lines

* Added static type checking for expressions

* Only thing to be done for semantic analysis is the return type of Tuples

r178 | yun03 | 2011-04-18 16:24:53 -0400 (Mon, 18 Apr 2011) | 3 lines

* Added function map into symbol table
* Added semantics checks for various expressions

r177 | yun03 | 2011-04-12 16:54:47 -0400 (Tue, 12 Apr 2011) | 1 line

* Added function map to the symbol table to track all function declarations in scope

r176 | yun03 | 2011-04-07 16:53:07 -0400 (Thu, 07 Apr 2011) | 1 line

* Cleaned up debugging code

r175 | yun03 | 2011-04-07 14:40:28 -0400 (Thu, 07 Apr 2011) | 1 line

* First version of symbol lookup in semantics analysis

r174 | yun03 | 2011-04-07 12:39:31 -0400 (Thu, 07 Apr 2011) | 1 line

* Added sym.ml to store and represent symbols

r173 | yun03 | 2011-04-07 12:05:48 -0400 (Thu, 07 Apr 2011) | 1 line

* Changed the type MolTuple to Tuple of mdl_type

r172 | yun03 | 2011-04-07 11:25:01 -0400 (Thu, 07 Apr 2011) | 1 line

r171 | yun03 | 2011-04-07 11:17:56 -0400 (Thu, 07 Apr 2011) | 1 line

r170 | yun03 | 2011-04-07 11:15:14 -0400 (Thu, 07 Apr 2011) | 1 line

r169 | yun03 | 2011-04-07 11:14:40 -0400 (Thu, 07 Apr 2011) | 1 line

r168 | yun03 | 2011-04-06 15:54:19 -0400 (Wed, 06 Apr 2011) | 2 lines

* Split out fundamental types and operators into types.mli
* Created sast.mli to represent the Semantic AST

r167 | yun03 | 2011-04-06 11:39:24 -0400 (Wed, 06 Apr 2011) | 1 line

r166 | yun03 | 2011-04-05 16:45:40 -0400 (Tue, 05 Apr 2011) | 1 line

r165 | yun03 | 2011-04-05 16:44:51 -0400 (Tue, 05 Apr 2011) | 1 line

* Declared mdl_type as a type in the AST

r164 | yun03 | 2011-04-05 16:08:28 -0400 (Tue, 05 Apr 2011) | 1 line

r163 | yun03 | 2011-04-05 16:02:43 -0400 (Tue, 05 Apr 2011) | 1 line

r162 | yun03 | 2011-04-05 12:24:55 -0400 (Tue, 05 Apr 2011) | 1 line

r161 | yun03 | 2011-04-05 12:22:57 -0400 (Tue, 05 Apr 2011) | 1 line

r160 | yun03 | 2011-04-05 09:35:06 -0400 (Tue, 05 Apr 2011) | 1 line

Broken ast.ml into ast.mli and mdlpp.ml

r159 | yun03 | 2011-04-01 12:48:44 -0400 (Fri, 01 Apr 2011) | 1 line

r158 | yun03 | 2011-04-01 09:17:48 -0400 (Fri, 01 Apr 2011) | 1 line

r157 | yun03 | 2011-03-31 17:37:34 -0400 (Thu, 31 Mar 2011) | 1 line

r156 | yun03 | 2011-03-31 17:15:41 -0400 (Thu, 31 Mar 2011) | 1 line

r155 | yun03 | 2011-03-31 14:11:45 -0400 (Thu, 31 Mar 2011) | 1 line

r154 | yun03 | 2011-03-31 11:54:43 -0400 (Thu, 31 Mar 2011) | 1 line

r153 | yun03 | 2011-03-31 11:37:13 -0400 (Thu, 31 Mar 2011) | 1 line

Added createEthanol to sample program

r152 | yun03 | 2011-03-31 10:48:05 -0400 (Thu, 31 Mar 2011) | 1 line

Comments added to the sample program

r151 | yun03 | 2011-03-31 10:14:24 -0400 (Thu, 31 Mar 2011) | 1 line

Removed the boolean type

r150 | yun03 | 2011-03-31 10:09:33 -0400 (Thu, 31 Mar 2011) | 1 line

r149 | yun03 | 2011-03-30 22:28:54 -0400 (Wed, 30 Mar 2011) | 1 line

r148 | yun03 | 2011-03-30 20:47:55 -0400 (Wed, 30 Mar 2011) | 1 line

r147 | yun03 | 2011-03-30 16:53:58 -0400 (Wed, 30 Mar 2011) | 1 line

r146 | yun03 | 2011-03-30 16:50:55 -0400 (Wed, 30 Mar 2011) | 1 line

r145 | yun03 | 2011-03-30 16:38:44 -0400 (Wed, 30 Mar 2011) | 1 line

r144 | yun03 | 2011-03-30 16:26:49 -0400 (Wed, 30 Mar 2011) | 1 line

r143 | yun03 | 2011-03-30 16:16:30 -0400 (Wed, 30 Mar 2011) | 1 line

r142 | yun03 | 2011-03-30 11:59:01 -0400 (Wed, 30 Mar 2011) | 1 line

r141 | yun03 | 2011-03-30 09:51:45 -0400 (Wed, 30 Mar 2011) | 1 line

r140 | yun03 | 2011-03-29 17:11:28 -0400 (Tue, 29 Mar 2011) | 1 line

r139 | yun03 | 2011-03-29 16:43:09 -0400 (Tue, 29 Mar 2011) | 1 line

r138 | yun03 | 2011-03-29 16:15:33 -0400 (Tue, 29 Mar 2011) | 1 line

Accepts C1(=0)-C1

r137 | yun03 | 2011-03-29 15:45:10 -0400 (Tue, 29 Mar 2011) | 1 line

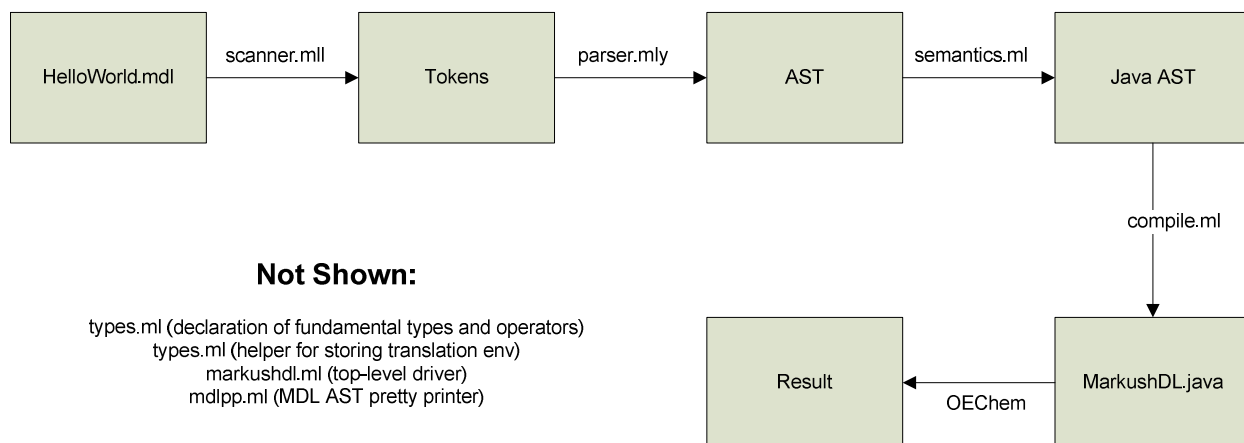
r136 | yun03 | 2011-03-29 15:37:38 -0400 (Tue, 29 Mar 2011) | 1 line

r135 | yun03 | 2011-03-29 15:35:12 -0400 (Tue, 29 Mar 2011) | 1 line

r133 | yun03 | 2011-03-29 15:27:51 -0400 (Tue, 29 Mar 2011) | 1 line

V. ARCHITECTURAL DESIGN

1. Major components



Note: mdlpp.ml was not fully implemented due to time constraints.

VI. TEST PLAN

1. Java code generation

The following examples demonstrate how a MDL program is translated into Java where the OEChem library is used to perform the chemistry operations.

Listing 1 Translating a molecule structure represented as chemical literal into the target language

MDL:

```

int global;

/*
 * Function to compute molecular weight of an input molecule
 */
int computeWeight(Mol mol) {
    int weight;
    weight = 0;

    /*
     * getAtoms(Mol) is a library function that returns a list/tuple of
     * atoms in the mol
     */
    for (Atom atom : getAtoms(mol)) {
        weight = weight + getWeight(atom);
    }
    return weight;
}

/*
 * Mainfunction is the entry point of the program
 */
int main() {
    Mol a;

    /* Set Mol a to the aspirin structure */
    a = '''CC(=O)Oc1ccccc1C(=O)O''';

    global = computeWeight( a );

    print( global );
}

```

Java:

```

import openeye.oechem.*;

public class MarkushDL {
    /*
     * Global variables in MDL are translated into static members of
     * the MarkushDL class with the default package-level visibility
     */
    static int global;

    public static void main(String[] args) {

        OEGraphMol a = null;

        /*
         * Note that the MDL assignment expression
         *     a = '''CC(=O)Oc1ccccc1C(=O)O''';
         * is translated into the following using Java's anonymous
         * inner class construct
         */
    }
}

```

```

a = new Object() {
    public OEGraphMol evalChemLiteral() {
        OEGraphMol temp = new OEGraphMol();
        oechem.OEParseSmiles(temp, "CC(=O)Oc1cccc1C(=O)O");
        return temp;
    }
}.evalChemLiteral();
global = computeWeight(a);
System.out.print(global);
}

public static int computeWeight(OEGraphMol mol) {

    int weight;
    weight = 0;
    for (OEAtomBase atom : mol.GetAtoms()) {
        weight = weight + RxnUtils.getAtomicWeight(atom);
    }

    return weight;
}
}

```

Listing 2 Translating the – operator that connects atom objects with single bonds to create a new molecule

MDL:

```

/*
 * Main function is the entry point of the program
 */
int main() {
    /* Declare a Mol object */
    Mol mol;

    /*
     * Instantiate three Atom objects, two carbons and one oxygen
     * using chemical literals enclosed in triple quotes, and use
     * the - operator to connect atom 1 and 2 with a single bond
     * and atom 2 and 3 with a single bond, thereby creating the
     * ethanol molecule.
     */
    mol = '''C''' - '''C''' - '''O''';

    /*
     * Call the print global function passing the ethanol molecule
     * as a parameter to print the SMILES representation of it
     */
    print( mol );
}

```

Java:

```
import openeye.oechem.*;

public class MarkushDL {

    public static void main(String[] args) {

        OEGraphMol mol = null;

        /*
         * Note in the following translation of the MDL expression
         *      mol = 'C' - 'C' - 'O'
         * we first parse atom 1 and 2 into molecules, and then
         * cast both into SMILES and concatenate the two strings
         * with the Java + operator. This creates the ethane molecule.
         * We then apply the same process to add the oxygen to
         * ethane, thereby creating ethanol as the end product
         */
        mol = new Object() {
            public OEGraphMol evalChemLiteral() {
                OEGraphMol temp = new OEGraphMol();
                oechem.OEParseSmiles(temp,
                    oechem.OECreateSmiString(new Object() {
                        public OEGraphMol evalChemLiteral() {
                            OEGraphMol temp = new OEGraphMol();
                            oechem.OEParseSmiles(temp,
                                oechem.OECreateSmiString(new Object() {
                                    public OEGraphMol evalChemLiteral() {
                                        OEGraphMol temp = new OEGraphMol();
                                        oechem.OEParseSmiles(temp, "C");
                                        return temp;
                                    }
                                }).evalChemLiteral()) +
                                oechem.OECreateSmiString(new Object() {
                                    public OEGraphMol evalChemLiteral() {
                                        OEGraphMol temp = new OEGraphMol();
                                        oechem.OEParseSmiles(temp, "C");
                                        return temp;
                                    }
                                }).evalChemLiteral()));
                            return temp;
                        }
                    }).evalChemLiteral()) +
                    oechem.OECreateSmiString(new Object() {
                        public OEGraphMol evalChemLiteral() {
                            OEGraphMol temp = new OEGraphMol();
                            oechem.OEParseSmiles(temp, "O");
                            return temp;
                        }
                    }).evalChemLiteral()));
                return temp;
            }
        }).evalChemLiteral();

        System.out.print(oechem.OECreateSmiString(mol, OESMILESFlag.DEFAULT
```

```
    & ~OESMILESFlag.Canonical));  
  }  
}
```

2. Test suite

I used the following suite of tests for my project. A simple C-shell script was used to perform automatic testing. The test code and script can be found in the Appendix.

Listing 3 List of tests used in the project

No.	Title
1	Calculating molecular weight
2	Tuples
3	Create a molecule from atoms
4	Test if statements
5	Test strings
6	Test boolean
7	Count atoms and bonds
8	Depth-first search of aspirin
9	Simple Markush analysis

VII. LESSONS LEARNED

- OcaIDE has a feature under Ocaml -> Format with camlp4. Do not use that feature – it generates an excessive amount of parentheses around all your concatenated strings which totally kills readability.
- The Ocaml -> Format feature formats the code to a clean and consistent appearance, but it only works if the file compiles with no errors.
- When programming in OCaml, it is important to follow a top-down approach to ensure you write the driver first and then the stub, since I found that if you write the stub first it is not adequately checked by the compiler.
- Sometimes when you need to write pattern matching code to process a large number of possibilities, such as when writing the expression checker, OCaml will not allow you to test the code with an incomplete set of cases or it will generate an annoying warning. I found it was useful to use the following trick to write code for one case and test it without getting the warnings.

```
let rec check_expr =  
  function  
  | Case0 ->  
    (* code for case0 *)  
  
  | Case1 | Case2 | Case3 | Case4  
  | Case5 | Case6 | Case7 ->
```

```
raise (Exception "Not implemented")
```

VIII. APPENDIX

scanner.mll

```
(* Scanner for Markush Description Language *)
```

```
{ open Parser }
```

```
let letter = ['a'-'z' 'A'-'Z']
```

```
let digit = ['0'-'9']
```

```
let num = digit+ | digit* '.' digit+
```

```
rule token = parse
```

```
| "/*" { comment lexbuf }
```

```
| [' ' '\t' '\r' '\n'] { token lexbuf }
```

```
(* 1 *)
```

```
| '(' { LPAREN }
```

```
| ')' { RPAREN }
```

```
| '{' { LBRACE }
```

```
| '}' { RBRACE }
```

```
| '[' { LBRACK }
```

```
| ']' { RBRACK }
```

```
(* 2 *)
```

```
| ':' { COLON }
```

```
| ';' { SEMICOLON }
```

```
| ',' { COMMA }
```

```
(* 3 *)
```

```
| '+' { PLUS }
```

```
| '-' { MINUS }
```

```
| '*' { TIMES }
```

```
| '/' { DIVIDE }
```

```
(* 4 *)
```

```
| "==" { EQ }
```

```
| "!=" { NEQ }
```

```
| '<' { LT }
```

```
| "<=" { LEQ }
```

```
| '>' { GT }
```

```
| ">=" { GEQ }
```

```
| '!' { NOT }
```

```
| '=' { ASSIGN }
```

```
(* 5 *)
```

```
| "int" { INT }
```

```
| "Mol" { MOL }
```

```
| "Atom" { ATOM }
```

```
| "Bond" { BOND }
```

```
| "String" { STRING }
```

```
| "boolean" { BOOLEAN }
```

```
| "true" { TRUE }
```

```
| "false" { FALSE }
```

```
(* 6 *)
```

```

| "if"           { IF }
| "else"        { ELSE }
| "for"         { FOR }
| "return"      { RETURN }
| "while"       { WHILE }
| "covers"     { COVERS }

| digit+ as s   { INT_LITERAL(int_of_string(s)) }
| letter (letter | digit | '_' ) * as identi { ID(identi) }
| eof          { EOF }

| "'''"        { chem_type "" lexbuf }

| ""'"        { string_type "" lexbuf }

(* comment *)
and comment = parse
  "*/" { token lexbuf }
| _ { comment lexbuf }

and chem_type str = parse
  "'''" { CHEM_LITERAL(str) }
| "\n" { let pos = lexbuf.Lexing.lex_curr_p in
          raise (Failure("Unclosed string literal, found beginning of"
            ^ " a new line without closure of string; "
            ^ " in line #" ^ (string_of_int pos.Lexing.pos_lnum))) }
| _ { chem_type (str^(Lexing.lexeme lexbuf)) lexbuf }

and string_type str = parse
  ""'" { STRING_LITERAL(str) }
| "\n" { let pos = lexbuf.Lexing.lex_curr_p in
          raise (Failure("Unclosed string literal, found beginning of"
            ^ " a new line without closure of string; "
            ^ " in line #" ^ (string_of_int pos.Lexing.pos_lnum))) }
| _ { string_type (str^(Lexing.lexeme lexbuf)) lexbuf }

```

parser.mly

```

/* Parser for Markush Description Language */

%{ open Ast %}

%token LBRACK RBRACK LPAREN RPAREN LBRACE RBRACE
%token COLON SEMICOLON COMMA
%token PLUS MINUS TIMES DIVIDE
%token EQ NEQ LT LEQ GT GEQ NOT ASSIGN
%token INT MOL ATOM BOND STRING BOOLEAN TRUE FALSE
%token IF NOELSE ELSE FOR RETURN WHILE COVERS
%token <string> ID
%token EOF
%token <int> INT_LITERAL
%token <string> CHEM_LITERAL
%token <string> STRING_LITERAL

```

```

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left COVERS
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE
%right NOT

%start program
%type < Ast.program> program

%%

program:
| /* nothing */ { [], [] }
| program vdecl { ($2 :: fst $1), snd $1 }
| program fdecl { fst $1, ($2 :: snd $1) }

fdecl:
| types ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
  { { rettype = $1;
    fname = $2;
    formals = $4; (* a list of tuples *)
    locals = List.rev $7; (* a list of tuples *)
    body = List.rev $8 (* a list of statements *) } }

types:
| INT { Types.Int }
| MOL { Types.Mol }
| ATOM { Types.Atom }
| BOND { Types.Bond }
  | STRING { Types.String }
  | BOOLEAN { Types.Boolean }
  | INT LBRACK RBRACK { Types.Tuple(Types.Int) }
| MOL LBRACK RBRACK { Types.Tuple(Types.Mol) }
  | STRING LBRACK RBRACK { Types.Tuple(Types.String) }
  | BOOLEAN LBRACK RBRACK { Types.Tuple(Types.Boolean) }

formals_opt:
| /* nothing */ { [] }
| formal_list { List.rev $1 }

formal_list:
| types ID { [($1 , $2)] }
| formal_list COMMA types ID { ($3, $4) :: $1 }

vdecl_list:
| /*nothing*/ { [] }
| vdecl_list vdecl { $2::$1 }

vdecl:
| types ID SEMICOLON { ($1, $2) }

```

```

stmt_list:
| /* nothing */ { [] }
| stmt_list stmt { $2::$1 }

stmt:
  /* Expression statement */
| expr SEMICOLON { Expr($1) }

  /* Compound statement */
| LBRACE stmt_list RBRACE { Block(List.rev $2) }

  /* Conditional statements */
| IF LPAREN expr RPAREN stmt %prec NOELSE { IfThen($3, $5) }
| IF LPAREN expr RPAREN stmt ELSE stmt { IfThenElse($3, $5, $7) }

  /* While statement */
| WHILE LPAREN expr RPAREN stmt { While($3, $5) }

  /* For statement */
| FOR LPAREN types ID COLON expr RPAREN stmt { For($3, $4, $6, $8) }

  /* Return statement */
| RETURN expr SEMICOLON { Return($2) }

expr:
  /* Primary expressions */
| ID { Id($1) }
| INT_LITERAL { IntLiteral($1) }
| chem_expr { ChemLiteral($1) }
  | STRING_LITERAL { StringLiteral($1) }
  | TRUE { BooleanLiteral(true) }
  | FALSE { BooleanLiteral(false) }
| LPAREN expr RPAREN { $2 }
| ID LPAREN actuals_opt RPAREN { Call($1, $3) }

  /* Unary operators */
| NOT expr { Unop(Types.Not, $2) }

  /* Binary operators */
| expr PLUS expr { Binop($1, Types.Add, $3) }
| expr MINUS expr { Binop($1, Types.Sub, $3) }
| expr TIMES expr { Binop($1, Types.Mult, $3) }
| expr DIVIDE expr { Binop($1, Types.Div, $3) }
| expr EQ expr { Binop($1, Types.Equal, $3) }
| expr NEQ expr { Binop($1, Types.Neq, $3) }
| expr LT expr { Binop($1, Types.Less, $3) }
| expr LEQ expr { Binop($1, Types.Leq, $3) }
| expr GT expr { Binop($1, Types.Greater, $3) }
| expr GEQ expr { Binop($1, Types.Geq, $3) }
  | expr COVERS expr { Covers($1, $3) }

  /* Assignment operator */
| ID ASSIGN expr { Assign($1, $3) }

  /* List operator */
| LBRACK actuals_opt RBRACK { List($2) }

```

```
actuals_opt:
  /* nothing */ { [] }
  | actuals_list { List.rev $1 }

actuals_list:
  | expr { [$1] }
  | actuals_list COMMA expr { $3 :: $1 }

chem_expr:
  | CHEM_LITERAL { $1 }
```

ast.mli

(* AST for Markush Description Language *)

```
type expr =
  (* primary expressions *)
  | Id of string
  | IntLiteral of int
  | ChemLiteral of string
  | StringLiteral of string
  | BooleanLiteral of bool
  | Call of string * expr list

  (* Unary operators *)
  | Unop of Types.op * expr

  (* Binary operators *)
  | Binop of expr * Types.op * expr
  | Covers of expr*expr

  (* Assignment operator *)
  | Assign of string*expr

  (* List operator *)
  | List of expr list

type stmt =
  | Block of stmt list
  | Expr of expr
  | For of Types.mdl_type*string*expr*stmt
  | While of expr*stmt
  | IfThenElse of expr*stmt*stmt
  | IfThen of expr*stmt
  | Return of expr

type func_decl = {
  rettype: Types.mdl_type;
  fname : string;
  formals : (Types.mdl_type*string) list;
  locals : (Types.mdl_type*string) list;
  body : stmt list;
```

```
}  
  
type program = (Types.mdl_type*string) list * func_decl list
```

types.ml

```
(* Fundamental types and operators for Markush Description Language *)
```

```
type mdl_type =  
  | Int  
  | Mol  
  | Atom  
  | Bond  
  | String  
  | Boolean  
  | Tuple of mdl_type  
  
type op =  
  | Add  
  | Sub  
  | Mult  
  | Div  
  | Equal  
  | Neq  
  | Less  
  | Leq  
  | Greater  
  | Geq  
  | Not  
  
exception MDLException of string  
  
let rec string_of_type =  
  function  
  | Int -> "int"  
  | Mol -> "Mol"  
  | Atom -> "Atom"  
  | Bond -> "Bond"  
  | String -> "String"  
  | Boolean -> "Boolean"  
  | Tuple(t) -> (string_of_type t) ^ " []"  
  
let rec type_of_tuple =  
  function  
  | Int | Mol | Atom | Bond | String | Boolean | Tuple(Tuple _) ->  
    raise (MDLException "Type being check not a Tuple")  
  | Tuple(Int) -> Int  
  | Tuple(Mol) -> Mol  
  | Tuple(Atom) -> Atom  
  | Tuple(Bond) -> Bond  
  | Tuple(String) -> String  
  | Tuple(Boolean) -> Boolean  
  
let string_of_operator =
```

```
function
| Add -> "+"
| Sub -> "-"
| Mult -> "*"
| Div -> "/"
| Equal -> "=="
| Neq -> "!="
| Less -> "<"
| Leq -> "<="
| Greater -> ">"
| Geq -> ">="
| Not -> "!"
```

jast.mli

```
(* Java AST for cross compilation from MDL to Java *)
```

```
type scope = Global | Method | Formal | ForLoop
```

```
type var_decl = {
  vtype : Types.mdl_type;
  vname : string;
  vscope : scope;
}
```

```
type expr_detail =
  (* primary expressions *)
  | Id of var_decl
  | IntLiteral of int
  | ChemLiteral of string
  | StringLiteral of string
  | BooleanLiteral of bool
  | Call of method_decl * typed_expr list

  (* Unary operators *)
  | Unop of Types.op * typed_expr

  (* Binary operators *)
  | Binop of typed_expr * Types.op * typed_expr
  | Covers of typed_expr * typed_expr

  (* Assignment operator *)
  | Assign of var_decl * typed_expr

  (* List operator *)
  | List of typed_expr list
```

```
and typed_expr = expr_detail * Types.mdl_type
```

```
and stmt =
  | Block of stmt list
  | Expr of typed_expr
  | For of var_decl * typed_expr * stmt
```

```

| While of typed_expr * stmt
  | IfThenElse of typed_expr * stmt * stmt
  | IfThen of typed_expr * stmt
  | Return of typed_expr

and method_decl = {
  rettype: Types.mdl_type;
  fname : string;
  formals : var_decl list;
  locals : var_decl list;
  body : stmt list;
}

type java_class =
  | JClass of var_decl list * method_decl list

```

sym.ml

```

(* type checking exceptions *)
exception Symbol_not_found of string

exception Symbol_redefinition of string * Types.mdl_type

exception Type_mismatch of Types.mdl_type * Types.mdl_type

(* symbol table *)
type symbol_table =
  { parent : symbol_table option;
    context : Jast.scope;
    mutable scope : Jast.var_decl list;
    mutable funcs : Jast.method_decl list
  }

let string_of_scope = function
  | Jast.Global -> "*GLOBAL*"
  | Jast.Method -> "*METHOD*"
  | Jast.Formal -> "*Formal*"
  | Jast.ForLoop -> "*FORLOOP*"

let dump env =
  print_string ("context=" ^ ((string_of_scope env.context) ^ "\n"));
  List.iter
    (fun symbol ->
      Printf.printf "{sname=%s, st=%s}\n"
        symbol.Jast.vname
          (Types.string_of_type symbol.Jast.vtype))
    env.scope;
  List.iter
    (fun symbol ->
      Printf.printf "{fname=%s, ft=%s}\n"
        symbol.Jast.fname
          (Types.string_of_type
            symbol.Jast.rettype))

```

```

env.funcs

(* add symbol to current environment *)
let add_sym env (t, n) =
  if not (List.exists (fun sy -> sy.Jast.vname = n) env.scope)
  then
    {
      parent = env.parent;
      context = env.context;
      scope = { Jast.vname = n; Jast.vtype = t; Jast.vscope =
env.context; } :: env.scope;
      funcs = env.funcs;
    }
  else raise (Symbol_redefinition (n, t))

let same_sig (l1 : Jast.var_decl list) (l2 : Types.mdl_type list) =
  List.fold_left2
    (fun b v1 t2 -> b && (v1.Jast.vtype = t2))
    true l1 l2

let add_func env (fdecl : Jast.method_decl) =
  try
    let signiture = List.map (fun v -> v.Jast.vtype)
fdecl.Jast.formals
    and sameName =
      List.find (fun f -> f.Jast.fname = fdecl.Jast.fname)
env.funcs in
    if (same_sig sameName.Jast.formals signiture)
    then
      raise(Symbol_redefinition (fdecl.Jast.fname,
fdecl.Jast.rettype))
    else
      {
        parent = env.parent;
        context = env.context;
        scope = env.scope;
        funcs = fdecl :: env.funcs;
      }
  with Not_found ->
    {
      parent = env.parent;
      context = env.context;
      scope = env.scope;
      funcs = fdecl :: env.funcs;
    }

let rec find_env_sym_rec env n =
  try
    (* look in current scope ignore (dump env); *)
    (env, (List.find (fun sy -> sy.Jast.vname = n) env.scope))
  with
  | Not_found ->
    (match env.parent with
     | (* in top scope - symbol not found *) None ->
       raise (Symbol_not_found n)
     | (* search parent scope *) Some parent_env ->

```

```

                                find_env_sym_rec parent_env n)

(* find a type for a given symbol - search current and then parent      *)
(* environments                                                         *)
let find_env_sym env n =
  (* print_string ("\nlooking for variable ***" ^ (n ^ "****\n")); *)
  snd (find_env_sym_rec env n)

let rec find_env_func_rec env fname (types : Types.mdl_type list) =
  try
    (* look in current scope ignore (dump env); *)
    (List.find
      (fun f ->
         f.Jast.fname = fname &&
         (same_sig f.Jast.formals types))
      env.funcs)
  with
  | Not_found ->
    (match env.parent with
     | (* in top scope - symbol not found *) None ->
       raise (Types.MDLException ("Undeclared
function: " ^ fname ^ "(" ^
                                (String.concat ","
                                (List.map Types.string_of_type types)) ^
                                ")"))
     | (* search parent scope *) Some parent_env ->
       find_env_func_rec parent_env fname types)

let find_env_func env fname (types : Types.mdl_type list) =
  (* print_string ("\nlooking for function ###" ^ (fname ^ "()\n")); *)
  find_env_func_rec env fname types

let symbol_names env = List.map (fun symbol -> symbol.Jast.vname) env.scope

(* create a new subordinate scope - keep reference to parent scope *)
let newscope c env = { parent = Some env; context = c; scope = []; funcs =
[]; }

```

semantics.ml

```
open Jast
```

```
(* global environment that sticks around for type checking of all modules *)
```

```
let genv =
  ref
  {
    Sym.parent = None;
    Sym.context = Global;
    Sym.scope = [];
    Sym.funcs =
      [
        { rettype = Types.Int;
          fname = "print";
```

```

vscope = Formal });
    formals = [{ vtype = Types.Int; vname = "";
    locals = [];
    body = []};
    { rettype = Types.Int;
      fname = "print";
      formals = [{ vtype = Types.Mol; vname = "";
vscope = Formal;}}];
    locals = [];
    body = []};
    { rettype = Types.Int;
      fname = "print";
      formals = [{ vtype = Types.String; vname = "";
vscope = Formal;}}];
    locals = [];
    body = []};
    { rettype = Types.Int;
      fname = "print";
      formals = [{ vtype = Types.Boolean; vname = "";
vscope = Formal;}}];
    locals = [];
    body = []};
    { rettype = Types.Int;
      fname = "getWeight";
      formals = [{ vtype = Types.Atom; vname = "";
vscope = Formal;}}];
    locals = [];
    body = []};
    { rettype = Types.Tuple(Types.Atom);
      fname = "getAtoms";
      formals = [{ vtype = Types.Mol; vname = "";
vscope = Formal;}}];
    locals = [];
    body = []};
    { rettype = Types.Tuple(Types.Atom);
      fname = "getAtoms";
      formals = [{ vtype = Types.Atom; vname = "";
vscope = Formal;}}];
    locals = [];
    body = []};
    { rettype = Types.Tuple(Types.Bond);
      fname = "getBonds";
      formals = [{ vtype = Types.Atom; vname = "";
vscope = Formal;}}];
    locals = [];
    body = []};
    { rettype = Types.Int;
      fname = "getIdx";
      formals = [{ vtype = Types.Atom; vname = "";
vscope = Formal;}}];
    locals = [];
    body = []};
    { rettype = Types.Atom;
      fname = "getNbr";
      formals = [{ vtype = Types.Bond; vname = "";
vscope = Formal;}}];

```

```

        { vtype = Types.Atom; vname = ""; vscope =
Formal;}}];
        locals = [];
        body = [];
        { rettype = Types.Boolean;
          fname = "setMarked";
          formals = [{ vtype = Types.Atom; vname = "";
vscope = Formal;}}];
        locals = [];
        body = [];
        { rettype = Types.Boolean;
          fname = "setMarked";
          formals = [{ vtype = Types.Bond; vname = "";
vscope = Formal;}}];
        locals = [];
        body = [];
        { rettype = Types.Boolean;
          fname = "isMarked";
          formals = [{ vtype = Types.Atom; vname = "";
vscope = Formal;}}];
        locals = [];
        body = [];
        { rettype = Types.Boolean;
          fname = "isMarked";
          formals = [{ vtype = Types.Bond; vname = "";
vscope = Formal;}}];
        locals = [];
        body = [];
        { rettype = Types.Mol;
          fname = "newBond";
          formals = [{ vtype = Types.Atom; vname = "";
vscope = Formal;}}];
        { vtype = Types.Atom; vname = ""; vscope =
Formal;}}];
        locals = [];
        body = [];
        { rettype = Types.Mol;
          fname = "newBond";
          formals = [{ vtype = Types.Mol; vname = "";
vscope = Formal;}}];
        { vtype = Types.Atom; vname = ""; vscope =
Formal;}}];
        locals = [];
        body = [];
        { rettype = Types.Tuple(Types.Mol);
          fname = "joinFragments";
          formals = [{ vtype = Types.Tuple(Types.Mol);
vname = ""; vscope = Formal;}}];
        { vtype = Types.Tuple(Types.Mol); vname
= ""; vscope = Formal;}}];
        locals = [];
        body = [];
];
}

let rec check_expr env =

```

```

function
(* primary expressions *)
| Ast.Id id ->
    let vdecl =
        try Sym.find_env_sym env id
        with
        | Not_found ->
            raise (Types.MDLException ("Undeclared
identifier: " ^ id))
    in ((Id vdecl), (vdecl.vtype))
| Ast.IntLiteral v ->
    ((IntLiteral v), Types.Int)
| Ast.ChemLiteral s ->
    if (String.length s) = 1
    then
        ((ChemLiteral s), Types.Atom)
    else
        ((ChemLiteral s), Types.Mol)
| Ast.StringLiteral s ->
    (StringLiteral s), Types.String
| Ast.BooleanLiteral b ->
    (BooleanLiteral b), Types.Boolean
| Ast.Call (fname, exprs) ->
    let typed_exprs = List.map (check_expr env) exprs in
    let types = snd (List.split typed_exprs) in
    let fdecl = Sym.find_env_func env fname types in
    (Call (fdecl, typed_exprs), fdecl.rettype)

(* Unary operators *)
| Ast.Unop (op, e) ->
    let typed_e = check_expr env e in
    let expr_type = snd typed_e in
    if expr_type = Types.Boolean
    then
        ((Unop (op, typed_e)), Types.Boolean)
    else
        raise (Types.MDLException ("Invalid expression type
for unary operator: " ^
                                (Types.string_of_type expr_type)))

(* Binary operators *)
| Ast.Binop (e1, op, e2) ->
    let te1 = check_expr env e1
    and te2 = check_expr env e2 in
    let typ1 = snd te1
    and typ2 = snd te2 in
    (
        match op with
        | Types.Sub ->
            if (((typ1 = Types.Mol) || (typ1 =
Types.Atom)) && (typ2 = Types.Atom))
            then
                check_expr env (Ast.Call
("newBond", [e1; e2]))
            else
                Binop (te1, op, te2), Types.Int

```

```

| Types.Add ->
  if (typ1 = Types.Int && typ2 = Types.Int)
  then
    Binop (te1, op, te2), Types.Int
  else if (typ1 = Types.String || typ2 =
Types.String)
  then
    Binop (te1, op, te2), Types.String
  else (
    match (typ1, typ2) with
    | (Types.Mol, Types.Mol) ->
      check_expr env
(Ast.Call ("joinFragments",
[Ast.List (e1 :: []); Ast.List (e2 :: [])]))
Types.Mol) ->
      check_expr env
(Ast.Call ("joinFragments",
Ast.List (e2 :: [])))
Types.Tuple(Types.Mol)) ->
      check_expr env
(Ast.Call ("joinFragments",
[Ast.List (e1 :: []); e2]))
Types.Tuple(Types.Mol)) ->
      check_expr env
(Ast.Call ("joinFragments",
e2]))
| _ ->
  raise
(Types.MDLError ("Unsupported types for + operator: " ^
(Types.string_of_type typ1) ^ " & " ^
(Types.string_of_type typ2)))
)
| Types.Mult | Types.Div ->
  (Binop (te1, op, te2), Types.Int)
| Types.Equal | Types.Neq | Types.Less
| Types.Leq | Types.Greater | Types.Geq ->
  (Binop (te1, op, te2), Types.Boolean)
| _ -> raise (Types.MDLError ("Unsupported types
for operator: " ^
(Types.string_of_type
typ1) ^ (Types.string_of_operator op) ^
(Types.string_of_type
typ2)))
)
| Ast.Covers (e1, e2) ->
  let te1 = check_expr env e1
  and te2 = check_expr env e2 in

```

```

    let typ1 = snd te1
    and typ2 = snd te2 in
    if (typ1 = Types.Tuple(Types.Mol) && typ2 = Types.Mol)
    then
        (Covers (te1, te2), Types.Boolean)
    else
        raise (Types.MDLException ("Unsupported types for
covers operator: " ^
                                (Types.string_of_type typ1) ^ " & "
                                ^ (Types.string_of_type typ2)))

    (* Assignment operator *)
    | Ast.Assign (id, e) ->
        let vdecl = Sym.find_env_sym env id
        and typed_e = check_expr env e in
        let etype = snd typed_e in
        if vdecl.vtype = etype
        then
            (Assign (vdecl, typed_e), etype)
        else
            raise (Types.MDLException ("Incompatible types for
variable " ^ id))

    (* List operator *)
    | Ast.List l ->
        let etype = snd (check_expr env (List.hd l)) in
        ((List (List.map (check_expr env) l)), Types.Tuple(etype))

let rec check_stmt env =
  function
  | Ast.Block stmts ->
      Block (List.map (check_stmt env) stmts)
  | Ast.Expr e ->
      Expr (check_expr env e)
  | Ast.Return e ->
      Return (check_expr env e)
  | Ast.While (e, s) ->
      let checked_e = check_expr env e in
      if ((snd checked_e) = Types.Boolean)
      then
          let checked_s = check_stmt env s in
          While (checked_e, checked_s)
      else
          raise (Types.MDLException ("While condition is not of
boolean type: "))
  | Ast.IfThenElse (e, s1, s2) ->
      let checked_e = check_expr env e in
      if ((snd checked_e) = Types.Boolean)
      then
          let checked_s1 = check_stmt env s1
          and checked_s2 = check_stmt env s2 in
          IfThenElse (checked_e, checked_s1, checked_s2)
      else
          raise (Types.MDLException ("If condition is not of
boolean type: "))
  | Ast.IfThen (e, s1) ->

```

```

let checked_e = check_expr env e in
if ((snd checked_e) = Types.Boolean)
then
  let checked_s1 = check_stmt env s1 in
  IfThen (checked_e, checked_s1)
else
  raise (Types.MDLException ("If condition is not of
boolean type: "))
| Ast.For (t, id, e, s) ->
  let checked_e = check_expr env e in
  let e_type = Types.type_of_tuple (snd checked_e) in
  if (t <> e_type)
  then
    raise (Sym.Type_mismatch (t, e_type))
  else
    let senv = Sym.newscope ForLoop env in
    let senv = Sym.add_sym senv (t, id) in
    let var = { vtype = t; vname = id; vscope =
senv.Sym.context } in
    let checked_s = check_stmt senv s in
    For (var, checked_e, checked_s)

(* check function definition and recurse down body *)
let check_and_add_fdef env fdef =
  let fenv = Sym.newscope Method env in
  let fenv = List.fold_left (Sym.add_sym) fenv fdef.Ast.formals in
  let fenv = List.fold_left (Sym.add_sym) fenv fdef.Ast.locals in
  let myformals = List.map (
    fun (t, n) -> { vtype = t; vname = n; vscope = Formal }
  )
    fdef.Ast.formals
  and mylocals = List.map (
    fun (t, n) -> { vtype = t; vname = n; vscope = Method }
  )
    fdef.Ast.locals in
  let self =
    {
      rettype = fdef.Ast.rettype;
      fname = fdef.Ast.fname;
      formals = myformals;
      locals = mylocals;
      body = [];
    }
  in
  let fenv = Sym.add_func fenv self in
  let ret_method =
    {
      rettype = fdef.Ast.rettype;
      fname = fdef.Ast.fname;
      formals = myformals;
      locals = mylocals;
      body = List.map (check_stmt fenv) fdef.Ast.body;
    }
  in
  Sym.add_func env ret_method

(* generate list of sast function definitions from ast function *)
(* definitions - this is the entry point for module *)

```

```

let rec check_prog ((vars : (Types.mdl_type * string) list), funcs) =
  let env = List.fold_left (Sym.add_sym) (!genv) vars in
  let funcs = (List.fold_left check_and_add_fdef env (List.rev
funcs)).Sym.funcs in
  JClass (env.Sym.scope, funcs)

```

compile.ml

```

(* Pretty-prints a Jast into Java code *)

let default_keyword = "public static"

let default_implicit_param = "null"

let rec string_of_expr ((expr : Jast.expr_detail), (etype : Types.mdl_type))
=
  match expr with
  | Jast.Id vdecl -> vdecl.Jast.vname
  | Jast.IntLiteral l -> string_of_int l
  | Jast.ChemLiteral s ->
    "new Object() {
      public OEGraphMol evalChemLiteral() {
        OEGraphMol temp = new OEGraphMol();
        oechem.OEParseSmiles(temp, \"
^ s ^ "\");
      return temp;
    }
  }.evalChemLiteral()"
  | Jast.StringLiteral s -> "\"" ^ s ^ "\""
  | Jast.BooleanLiteral b ->
    if b then "true" else "false"
  | Jast.Call (fdecl, te_list) ->
    (match fdecl.Jast.fname with
     | "getWeight" ->
       "RxnUtils.getAtomicWeight(" ^
       (String.concat "," (List.map
string_of_expr te_list)) ^
       ")")
     | "print" -> (
       let arg = List.hd te_list in
       let typ = snd arg in
       match typ with
       | Types.Int | Types.String |
Types.Boolean ->
         "System.out.print(" ^
(string_of_expr arg) ^ ")")
       | Types.Mol ->
         "System.out.print(oechem.OECreateSmiString(" ^
(string_of_expr arg) ^
", OESMILESFlag.DEFAULT
& ~OESMILESFlag.Canonical))"
     | Types.Atom | Types.Bond |

```

```

Types.Tuple(_) ->
    raise
    (Types.MDLException ("Unsupported type for print(): " ^
        (Types.string_of_type typ)))
    )
    | "getAtoms" ->
        (String.concat "," (List.map
string_of_expr te_list)) ^
        ".GetAtoms()"
    | "getBonds" ->
        (String.concat "," (List.map
string_of_expr te_list)) ^
        ".GetBonds()"
    | "getIdx" ->
        (String.concat "," (List.map
string_of_expr te_list)) ^
        ".GetIdx()"
    | "getNbr" ->
        "(" ^ (string_of_expr (List.hd te_list))
        (string_of_expr (List.nth te_list 1)) ^
        ")" ^ ".GetNbr(" ^
        (string_of_expr (List.nth te_list 1)) ^
        ")"
    | "setMarked" ->
        "(" ^ (string_of_expr (List.hd te_list))
        ".SetBoolData(\"marked\", true)"
    | "isMarked" ->
        "(" ^ (string_of_expr (List.hd te_list))
        ".GetBoolData(\"marked\")"
    | "newBond" ->
        let createSmi id =
"oechem.OECreateSmiString(" ^ id ^ ")" in
        "new Object() {
        public OEGraphMol evalChemLiteral() {
        OEGraphMol temp = new OEGraphMol();
        oechem.OEParseSmiles(temp, "
        ^ (String.concat "+"
            (List.map createSmi
                (List.map
string_of_expr te_list))) ^ ");
        return temp;
        }
        }.evalChemLiteral()"
    | "joinFragments" ->
        "RxnUtils.joinFragments" ^
        "(" ^
        (String.concat "," (List.map
string_of_expr te_list)) ^ ")"
    | _ ->
        fdecl.Jast.fname ^
        "(" ^
        ((String.concat "," (List.map
string_of_expr te_list)) ^ ")")
        | Jast.Unop (Types.Not, e) ->

```

```

    "!" ^ "(" ^ (string_of_expr e) ^ ")"
  | Jast.Unop (_, e) ->
    raise (Types.MDLError "Unsupported unary operator")
  | Jast.Binop (e1, o, e2) ->
    (string_of_expr e1) ^ " " ^ (Types.string_of_operator o)
^ " " ^
    (string_of_expr e2)
  | Jast.Assign (vdecl, e) ->
    (vdecl.Jast.vname) ^ " = " ^ (string_of_expr e)
  | Jast.List l ->
    let typeDetect fstElem =
      match fstElem with
      | Types.Int _ -> "int"
      | Types.Mol -> "OEGraphMol"
      | Types.Atom -> "OEAtomBase"
      | Types.Bond -> "OEBondBase"
      | Types.String -> "String"
      | Types.Boolean -> "boolean"
      | Types.Tuple(t) -> raise (Types.MDLError
"Unsupported list type")
    in
      "new " ^ (typeDetect (snd (List.hd l))) ^ " [] {" ^
        (String.concat "," (List.map string_of_expr l)) ^ "}"
  | Jast.Covers (chem1, chem2) ->
    "RxnUtils.covers(" ^
      (string_of_expr chem1) ^ ", " ^ (string_of_expr chem2) ^
")"

let rec string_of_type =
  function
  | Types.Int -> "int "
  | Types.Mol -> "OEGraphMol "
  | Types.Atom -> "OEAtomBase "
  | Types.Bond -> "OEBondBase "
  | Types.String -> "String "
  | Types.Boolean -> "boolean "
  | Types.Tuple(t) -> (string_of_type t) ^ "[]"

let string_of_vdecl vdecl =
  match vdecl.Jast.vscope with
  | Jast.Global ->
    "static " ^ (string_of_type vdecl.Jast.vtype) ^
vdecl.Jast.vname ^ ";\n"
  | Jast.Method ->
    (match vdecl.Jast.vtype with
    | Types.Mol | Types.Atom | Types.Bond | Types.String
->
      (string_of_type vdecl.Jast.vtype) ^
vdecl.Jast.vname ^ " = null;\n"
    | _ ->
      (string_of_type vdecl.Jast.vtype) ^
vdecl.Jast.vname ^ ";\n")
  | Jast.Formal | Jast.ForLoop ->
    string_of_type vdecl.Jast.vtype ^ vdecl.Jast.vname

let rec string_of_stmt =

```

```

function
| Jast.Block stmts ->
    "{\n" ^ ((String.concat "" (List.map string_of_stmt stmts))
^ "}\n")
| Jast.Expr expr -> (string_of_expr expr) ^ ";\n"
| Jast.Return e -> "return " ^ ((string_of_expr e) ^ ";\n")
| Jast.While (e, s) ->
    "while (" ^ ((string_of_expr e) ^ (")\n" ^ (string_of_stmt
s)))
| Jast.IfThenElse (e, s1, s2) ->
    "if (" ^ (string_of_expr e) ^
")\n" ^ (string_of_stmt s1) ^ "else\n" ^ (string_of_stmt
s2)
| Jast.IfThen (e, s1) ->
    "if (" ^ (string_of_expr e) ^ ")\n" ^ (string_of_stmt
s1)
| Jast.For (vdecl, e, stmt) ->
    let loopType typ = match typ with
        | Types.Atom -> "OEAtomBase "
        | Types.Bond -> "OEBondBase "
        | Types.Mol -> "OEGraphMol "
        | Types.Int -> "int "
        | Types.String -> "String "
        | Types.Boolean -> "boolean "
        | Types.Tuple(t) ->
            raise (Types.MDLException ("Invalid loop
variable " ^
                                     (Types.string_of_type
typ)))
    in
    "for (" ^
    ((loopType vdecl.Jast.vtype) ^ vdecl.Jast.vname ^
    (" : " ^
    ((string_of_expr e) ^ (")\n" ^ ((string_of_stmt
stmt) ^ "\n")))))

let string_of_method_decl f =
    let mainDetect f =
        match f.Jast.fname with
        | "main" ->
            if (List.length f.Jast.formals) = 0
            then " void main(String [] args)"
            else
                raise (Types.MDLException ("Entry point of a
MDL program must have " ^
                                           "the signiture of int
main()"))
        | _ ->
            " " ^
            ((string_of_type f.Jast.rettype) ^
            (" " ^
            (f.Jast.fname ^
            ("(" ^
            ((String.concat ","
(List.map
string_of_vdecl f.Jast.formals))

```

```

^ ")")))))))
in
if (List.exists (fun fdecl -> f = fdecl) !(Semantics.genv).Sym.funcs)
then ""
else
  default_keyword ^
  (mainDetect f) ^
  "\n{\n\n" ^
  (String.concat "" (List.map string_of_vdecl f.Jast.locals)) ^
  (String.concat "" (List.map string_of_stmt f.Jast.body)) ^ "}\n"

let string_of_jclass =
  fun (Jast.JClass (vars, methods)) ->
    "\npublic class " ^ "MarkushDL" ^ "\n{\n" ^
    String.concat "\n" (List.map string_of_vdecl vars) ^
    "\n\n" ^
    String.concat "" (List.map string_of_method_decl methods) ^
    "}\n"

(* e.g., import lib.*; *)
let string_of_imports import_list = (String.concat ";\n" import_list) ^ ";\n"

let translate program =
  (string_of_imports [ "import openeye.ochem.*" ]) ^
  "\n" ^ (string_of_jclass program) ^ "\n"

```

markushdl.ml

```
type action = | Ast | Compile
```

```

let _ =
  let action =
    if (Array.length Sys.argv) > 1
    then List.assoc Sys.argv.(1) [ ("-a", Ast); ("-c", Compile) ]
    else Compile in
  let lexbuf = Lexing.from_channel stdin in
  let program = Parser.program Scanner.token lexbuf
  in
  match action with
  | Ast ->
    print_string (Mdlpp.string_of_program program)
  | Compile ->
    print_string (Compile.translate (Semantics.check_prog
program))

```

mdlpp.ml

```
open Ast
```

```
let string_of_program (vars, funcs) =
```

```
raise (Types.MDLException "Not implemented!")
```

RxnUtils.java

```
import java.util.ArrayList;
import java.util.List;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import openeye.oechem.OEAtomBase;
import openeye.oechem.OEExprOpts;
import openeye.oechem.OEGraphMol;
import openeye.oechem.OEHasAtomicNum;
import openeye.oechem.OEQMol;
import openeye.oechem.OESubSearch;
import openeye.oechem.oechem;

/**
 * @author yun03
 *
 */
public final class RxnUtils {

    static final String R_PATTERN = "\\[R\\d+\\]";

    public static OEGraphMol [] joinFragments(OEGraphMol [] lhs, OEGraphMol []
rhs) {
        ArrayList<OEGraphMol> result = new ArrayList<OEGraphMol>();
        for (OEGraphMol m1 : rhs) {
            ArrayList<Integer> rgs = new ArrayList<Integer>();
            for (OEAtomBase atom : m1.GetAtoms( new OEHasAtomicNum(0)))
            {
                rgs.add( atom.GetMapIdx() );
            }

            String s1 = oechem.OECreateSmiString(m1);
            s1 = replaceRZ(s1, R_PATTERN, rgs);
            for (OEGraphMol m2 : lhs) {
                String s2 = oechem.OECreateSmiString(m2);
                s2 = replaceRZ(s2, R_PATTERN, rgs);
                OEGraphMol mol = new OEGraphMol();
                oechem.OEParseSmiles(mol, s1+"."+s2);

                oechem.OEAssignMDLHydrogens( mol );
                oechem.OEFindRingAtomsAndBonds( mol );
                oechem.OEAssignAromaticFlags( mol );

                result.add(mol);
            }
        }
        return result.toArray(new OEGraphMol [0]);
    }
}
```

```

    static String replaceRZ(String smiWithRZ, String regex, List<Integer>
rgs) {
        Pattern p = Pattern.compile("^"+regex);
        Matcher m = p.matcher(smiWithRZ);
        if (m.find()) {
            String head = m.group();
            int end = m.end();
            String atom = smiWithRZ.substring(end, end+1);
            smiWithRZ = atom + "(" + head + ")" +
smiWithRZ.substring(end+1);
        }

        p = Pattern.compile(regex);
        m = p.matcher(smiWithRZ);
        StringBuffer sb = new StringBuffer();
        while (m.find()) {
            String num = smiWithRZ.substring(m.start()+2, m.end()-1);
            int n = Integer.parseInt(num);
            if (rgs.contains(n)) {
                m.appendReplacement(sb, "%"+(80+n));
            }
        }
        m.appendTail(sb);
        return sb.toString();
    }

    public static boolean covers(OEGraphMol[] markush, OEGraphMol test) {
        for (OEGraphMol mol : markush) {
            oechem.OESuppressHydrogens(mol);
            OEMol qmol = new OEMol(mol);
            qmol.BuildExpressions(OEExprOpts.DefaultAtoms,
OEExprOpts.DefaultBonds);
            OESubSearch sss = new OESubSearch();
            sss.Init(qmol);
            if (sss.SingleMatch(test)) {
                return true;
            }
        }
        return false;
    }

    /**
     * Note that only selected elements frequently seen in organic
molecules
     * are supported. Result is rounded to the nearest integer.
     *
     * @param atom
     * @return weight of the input atom
     */
    public static int getAtomicWeight(OEAtomBase atom) {
        switch (atom.GetAtomicNum()) {
            case 0: // R-group
                return 0;
            case 1: // Hydrogen
                return 1;
            case 6: // Carbon

```

```
        return 12;
    case 7: // Nitrogen
        return 14;
    case 8: // Oxygen
        return 16;
    case 9: // Fluorine
        return 19;
    case 11: // Sodium
        return 23;
    case 12: // Magnesium
        return 23;
    case 16: // Sulfur
        return 32;
    case 17: // Chlorine
        return 35;
    default:
        throw new RuntimeException("Unsupported atom: " + atom);
    }
}
}
```

RxnUtilsTest.java

```
import static org.junit.Assert.assertEquals;

import java.util.Arrays;

import openeye.oechem.OEGraphMol;
import openeye.oechem.oechem;

import org.junit.Test;

public class RxnUtilsTest {
    @Test
    public void test_replaceRZ() {
        assertEquals("C(%81)[R2]",
            RxnUtils.replaceRZ("[R1]C[R2]",
                RxnUtils.R_PATTERN, Arrays.asList(1)));

        assertEquals("C([R1])%82",
            RxnUtils.replaceRZ("[R1]C[R2]",
                RxnUtils.R_PATTERN, Arrays.asList(2)));
    }

    /**
     * Proves the RxnUtils.joinFrgs works for two-step joining of
     fragments
     */
    @Test
    public void test_joinFrgs() {
        OEGraphMol core = new OEGraphMol();
        oechem.OEParseSmiles(core, "[R1]c1cccc([R2])n1");
    }
}
```

```
OEGraphMol rg1 = new OEGraphMol();
oechem.OEParseSmiles(rg1, "[R1]N");

OEGraphMol rg2 = new OEGraphMol();
oechem.OEParseSmiles(rg2, "[R2]C(=O)O");

OEGraphMol [] mols = RxnUtils.joinFragments(new OEGraphMol[] {core},
      new OEGraphMol [] {rg1});
mols = RxnUtils.joinFragments(mols, new OEGraphMol [] {rg2});

    assertEquals("c1cc(nc(c1)N)C(=O)O",
oechem.OECreateIsoSmiString(mols[0]));
    }
}
}
```

runtests.csh

```
#!/bin/tcsh

foreach file (`ls -l data/*.mdl`)
    echo "\n\n\tTesting $file"
    ocamlrun _build/markushdl.d.byte < $file > ../java/src/MarkushDL.java
    cd ../java
    ant build > $file.out
    ant MarkushDL >> $file.out
    diff -b ../ocaml/$file.out $file.out
    rm $file.out
    cd ../ocaml
end
```

IX. BIBLIOGRAPHY

1. Markush, E.A. 1924, Pharma-Chemical Corporation: USA.
2. *Ex parte Markush*. 340 O.G. 839. 1924.
3. Simmons, E.S., *Markush structure searching over the years*. World Patent Information, 2003. **25**: p. 195-202.
4. Cielen, E., *Searching Markush formulae directed to medical applications*. World Patent Information, 2009. **31**: p. 178-183.
5. Barnard, J.M. and P.M. Wright, *Towards in-house searching of Markush structures from patents*. World Patent Information, 2009. **31**: p. 97-103.
6. Leach, A.R. and V.J. Gillet, *An Introduction to Chemoinformatics*. 2005, Dordrecht, The Netherlands: Springer.
7. Vultur, A., et al., *SKI-606 (bosutinib), a novel Src kinase inhibitor, suppresses migration and invasion of human breast cancer cells*. Mol Cancer Ther, 2008. **7**(5): p. 1185-94.
8. *Examination of Patent Applications That Include Claims Containing Alternative Language*, P.a.T. Office, Editor. 2007, Federal Register. p. 44992-45000.

9. Weininger, D., *SMILES, a chemical language and information system. 1. Introduction to methodology and encoding rules*. Journal of Chemical Information and Computer Sciences, 1988. **28**(1): p. 31-36.
10. *Daylight Theory Manual*, Daylight Chemical Information, Inc.: Aliso Viejo, CA.
11. *Interactive depiction of SMILES*. [cited 2011 February 8th]; Available from: <http://www.daylight.com/daycgi/depict>.