Faculty of Computer Science Institute of Software and Multimedia Technology, Software Technology Group

WS2020/21 – Model-driven Software Development in Technical Spaces

Reference Attribute Grammars with JastAdd

Professor: Prof. Dr. Uwe Aßmann
Tutor: Johannes Mey johannes.mey@tu-dresden.de

1 JastAdd

In this exercise, Reference Attribute Grammars (RAGs) and JastAdd, a Java implementation for RAGs [1] are introduced.

JastAdd is a tool that generates a Java implementation of a given RAG comprising of a context-free grammar and a set of attribute definitions. In addition to common RAG features like synthesized and inherited attributes, JastAdd supports some features that are used in this exercise:

- Compiler/DSL features (lexer and parser generation)
- Rewrites
- Special attribute types (e.g., collection attributes)

A comprehensive overview of all features supported by JastAdd can be found in the reference manual[1].

In this exercise, a simple domain-specific language (DSL) for mathematical expressions is defined and evaluated. JastAdd supports automatic, on-demand rewriting of syntax trees of a grammar, which later is used to optimize mathematical expressions.

1.1 Task 1: Extension of Expression Language

Develop a RAG specification for a small expression language, that can evaluate simple expressions. An initial specification comprising grammar (for addition, multiplication, constant numbers and variables) and some attributes are given. The first task is to extend the specification to support unary and binary minus, binary division and pretty-printing. To achieve this, the following subtasks are needed:

- Unpack and run the gradle project. **Read the README.**
- Extend the DSL enabling new nonterminal types. Use `DivExp`, `MinusExp` and `UnaryMinusExp` as their names.
  - Extend the grammar specification (`expressions.ast`)

[1] `http://jastadd.org/web/documentation/reference-manual.php`

  - Extend the lexical analysis specification (`expressions.flex`)
  - Extend the parser specification (`expressions.parser`)
  - Extend the existing pretty-printing attribute `syn String ASTNode.print();` to support the new nonterminals.

- Implement new attributes.
  - A name resolution attribute `syn Def Var.getDef();`
  - An evaluation attribute `syn float Exp.eval();`

1.2 Task 2: Rewrite to Optimize

Given a generated tree, perform two optimization: constant folding and multiplicative annhilator detection.

In the first case, expressions comprising only constant values are rewritten to the constant value they (always) evaluate to, e.g., `(3 + 4)` is rewritten to `7`. Write the following attribute and rewrite:

- `syn boolean Exp.isConstantSubtree();` to find constant subtrees.
- `rewrite Exp { when ( isConstantSubtree() ) to Number { ...} }`

The second case will rewrite multiplications with zero, i.e., it will rewrite `((4 + a) * 0)` to `0`. To complete the second task, the following attributes and rewrites are to be written:

- `syn boolean MulExp.isMulWithZero();`
- `rewrite MulExp { when ( isMulWithZero() ) to Number { ...} }`

Finally, test the result with a small example checking the AST before and after optimization (using the attribute `Print` from Task 1.1).

1.3 Task 3: Code Analysis

The list of variables may contain names that are not used in the expression. Write a collection attribute that finds these variable definitions.

- `coll java.util.Set<Def> Root.unusedDefs() [new java.util.HashSet<Def>()] with add root Root;`

1.4 Submission of the results

All files in `src/main/` are to be submitted.

1.5 Additional JastAdd Links

- JastAdd: `http://jastadd.org`
- Source: `https://bitbucket.org/jastadd/jastadd2`
- Manual: `http://jastadd.org/web/documentation/reference-manual.php`

References

[1] Görel Hedin. Reference attributed grammars. *Informatica (Slovenia)*, 24(3), 2000. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.108.8792&rep=rep1&type=pdf.

2 Solutions

See https://git-st.inf.tu-dresden.de/jastadd/most-rag-exercises

2.1 Task 1: Extension of Expression Language

Install and understand JastAdd

- Just do it.

Extend the grammar enabling new nonterminal types

- Extend the grammar specification (`expressions.ast`)

```
UnaryMinusExp:UnaryExp ;
DivExp:BinExp ;
MinusExp:BinExp ;
```

- Extend the lexical analysis specification (`expressions.flex`)

```
"/"           { return sym(Terminals.DIV); }
```

- Extend the parser specification (`expressions.parser`)

```
LP exp.a PLUS exp.b RP              {: return new AddExp(a, b); :}
| LP exp.a MULT exp.b RP            {: return new MulExp(a, b); :}
| LP exp.a MINUS exp.b RP           {: return new MinusExp(a, b); :}
| LP exp.a DIV exp.b RP             {: return new DivExp(a, b); :}
| MINUS exp                         {: return new UnaryMinusExp(exp); :}
...
```

- Extend the existing pretty-printing attribute `syn String ASTNode.print();` to support the new nonterminals.

```
eq UnaryMinusExp.print() = "-" + getExp().print();
eq DivExp.print() = "(" + getA().print() + " / " + getB().print() + ")";
eq MinusExp.print() = "(" + getA().print() + " - " + getB().print() +
    ")";
```

- Implement a name resolution attribute `syn Def Var.getDef();`

```
syn Def Var.getDef();
eq Var.getDef() = getDef(getName());

inh Def Var.getDef(String name);
eq Root.getExp().getDef(String name) {
  for (Def def : getDefList()) {
    if (def.getName().equals(name)) {
      return def;
    }
  }
  return null;
}
```

- Implement an evaluation attribute `syn float Exp.eval();`

```
syn float Exp.eval();
eq AddExp.eval() = getA().eval() + getB().eval();
eq MulExp.eval() = getA().eval() * getB().eval();
eq Number.eval() = getValue();
eq Var.eval() = getDef().getValue();
eq UnaryMinusExp.eval() = -getExp().eval();
eq DivExp.eval() = getA().eval() / getB().eval();
eq MinusExp.eval() = getA().eval() - getB().eval();
```

2.2 Task 2: Rewrite to optimize

Implement rewrites for both cases

- Implement the two rewrites and their helper attributes:

```
syn boolean Exp.isConstantSubtree();
eq Exp.isConstantSubtree() = isConstant();
eq Number.isConstantSubtree() = false;

syn boolean Exp.isConstant();
eq BinExp.isConstant() = getA().isConstant() && getB().isConstant();
eq Number.isConstant() = true;
eq Var.isConstant() = false;
eq UnaryMinusExp.isConstant() = getExp().isConstant();

rewrite Exp {
  when ( isConstantSubtree() )
  to Number {
    return new Number(eval());
  }
}

syn boolean MulExp.isMulWithZero();
eq MulExp.isMulWithZero() = (getA().isConstant() && getA().eval() == 0f)
                          || (getB().isConstant() && getB().eval() == 0f);

rewrite MulExp {
  when ( isMulWithZero() )
  to Number {
    return new Number(0);
  }
}
```

2.3 Task 3: Code Analysis

- Implement an attribute that finds unused definitions:

```
coll java.util.Set<Def> Root.unusedDefs() [new java.util.HashSet<Def>()]
    with add root Root;
Def contributes this
  when !isUsed()
  to Root.unusedDefs();

inh boolean Def.isUsed();
```

```
eq Root.getDef(int i).isUsed() = getExp().uses(getDef(i));

syn boolean Exp.uses(Def def);
eq BinExp.uses(Def def) = getA().uses(def) || getB().uses(def);
eq Number.uses(Def def) = false;
eq Var.uses(Def def) = getName().equals(def.getName());
eq UnaryMinusExp.uses(Def def) = getExp().uses(def);
```