Language Processors Lab 6 (Week7)
Building Abstract Syntax Trees (ASTs) with jjtree

In this lab you will learn how to use jjtree to create Abstract Syntax Trees, and how to modify the generated classes to traverse the generated ASTs.

Downloading the files

Start a Unix shell window and move to your LanguageProcessors directory. Download the file `lab6.tar.gz` from Moodle (week7) or from `http://www.soi.city.ac.uk/~sbbc287/lab6.tar.gz`. Unzip and untar the file with the following commands:

```
gunzip lab6.tar.gz       // this will genenerate lab6.tar
tar -xvf lab6.tar
```

The last command will generate the `lab6` directory. It is possible that, while downloading, the file system automatically unzips, and even expands (untars), the file. If that's the case, you may have to skip one or both of the above commands, though make sure to copy the new directory.

Your new `lab6` directory should contain two directories: `exp`, and `tpl`. Double check that this is the case.

Load java and javacc by executing the command:

```
module add java javacc
```

Generating ASTs for expressions using jjtree

The Files

Move into the `exp` subdirectory, inside the `lab6` directory. **For example**, if you are in the `LanguageProcessors` directory, type:

```
cd lab6/exp
```

The contents of the `exp` directory are:

- `Exp.jj`: Defines the grammar, and parser, for arithmetical expressions.
- `Main.java`: Entry point of the program. It calls the parser for arithmetical expression.

Check the `Exp.jj` file. Note that it contains the grammar for arithmetical expressions used in the provious labs:

E → T (`+` T | `-` T)*
T → F (`*` F | `/` F)*
F → number | `(` E `)`

Note also that it contains no semantic actions and that all non-terminal definitions are of type `void` i.e. this file defines a pure parser.

Execute JavaCC, compile the generated java files and execute the program by typing the usual commands:

```
javacc Exp.jj          javac Main.java          java Main
```

Test the program with a few examples just to make sure that we are working with the correct parser.

Generating ASTs

The current parser does not generate ASTs. In the previous lab we did this by creating new classes that implemented the ASTs for arithmetical expressions and added semantics actions to the grammar with the respective constructor calls. In this lab we will use a program that can achieve the same result automatically: JJTree.

JJTree is a program, part of the JavaCC tool chain, that takes as input a .jjt file and generates a JavaCC file (`.jj`) together with a set of classes that implement the AST defined by the grammar in .jj. A brief (but comprehensive enough) introduction to jjtree can be found in `www.soi.city.ac.uk/~sbbc287/jjtree.html`.

We will start by working with the default settings of jjtree. First, we need to change the extension of our JavaCC file to `.jjt`. Type the following[1]:

```
mv Exp.jj Exp.jjt
```

Start a text editor and load Exp.jjt. Add the following to the beginning of the file:

```
options {
   MULTI=true;
}

PARSER_BEGIN(ExpParser)  //  first line old file - don't copy!
```

Now, invoke jjtree with the following command:

```
jjtree Exp.jjt
```

Type `ls` and double-check that jjtree has created a fresh Exp.jj file, and a class definition per non-terminal specified in the JJTree file, namely, ASTS, ASTE, ASTT and ASTF. All these classes inherit from Node and SimpleNode (part of the JJTree package).

In order to see JJTree in action, add the following line to `Main.java`:

```
 parser.S();  // Part of the original Main.java - don't add
((SimpleNode)parser.jjtree.rootNode()).dump("");
```

Now we are ready to call JavaCC, compile the resulting files and execute the new program:

```
javacc Exp.jj          javac Main.java              java Main
```

Type any arithmetical expression. What happens? Try more.
After parsing, `Main.java` prints the contents of the generated AST by invoking the method `dump` - that's the last line you added above!

OK, so JJTree is working. Now we need to tell JJTree to generate the ASTs we want. Instead of the default classes we want ASTs made of classes that follow the structure used in the previous lab i.e. a class per binary expression and another for numerical constants. First, remove the generated files with:

```
rm  AST*
```

i.e. remove all files starting with AST. Make sure `Exp.jjt` is loaded in your text editor and follow the instructions below:

- Change the declaration of `S()` to:

```
void S()  #void :
```

[1] You can also do Save As and remove the old file

By adding `#void` we stop jjtree from creating a new node. Why? Because we don't want an `S` node.

- Change the specification of `E()` to:

```
void E() #void :
{  }
{
    T() (  "+" T() #PlusExp(2)
         | "-" T() #MinusExp(2)
        )*
}
```

Similarly, `#void` prevents jjtree from creating a new `E` node. Instead, we instruct jjtree to create `#PlusExp(2)` and `#MinusExp(2)` nodes using the last two nodes created by jjtree, that is, the ones matched by `T()`.

- Change the specification of `F()` to:

```
void T() #void:
{  }
{
   F() (  "*"  F() #TimesExp(2)
        | "/" F() #DivExp(2)
       )*
}
```

Explanation: Same as above

- And, finally, change the specification of `F()` to:

```
void F() #void :
{  }
{
     <NUM> #NumExp
  | "(" E() ")"
}
```

As above, `#void` prevents jjtree from creating a new `F` node. Instead, we instruct jjtree to create `#NumExp` after the token `<NUM>` is matched. Note, however, that we are not storing the numeric value of the number!! (we'll sort that out soon).
We don't need to create a new node for expression in parenteses.

We are ready to run our new parser. Execute each of the commands below:

```
jjtree Exp.jjt     javacc Exp.jj       javac Main.java        java Main
```

Type a few arithmetical expression. What happens? The program, after parsing, prints the structure of the generated AST. Make sure that the strcuture makes sense!!

Modifying the generated jjtree classes

The current version of the parser generates a well-formed AST. However, the new AST does not store - as with our previous versions - the value of integer constants (the ones matched against `<NUM>`).

Change the specification of `F()` to:

```
void F() #void :
{ Token t; }    //
{
     (t=<NUM>  { jjtThis.val = Integer.parseInt(t.image);})  #NumExp
  | "(" E() ")"
}
```

Note that we have introduced a `Token t` variable, and that the parsed integer value is being stored in data field `val` of generated class `ASTNumExp`. Therefore, we need to declare a new field `val` inside `ASTNumExp` class declaration. Furthermore, we would like to modify the `dump` method to print the value associated woth ASTNumExp.

Open file `ASTNumExp.java` and add the following (at the bottom of the class declaration):

```
    public int val;

    public void dump(String prefix) {
       System.out.print(toString(prefix));
       System.out.println(" "+val);
    }
 } // This must be the parenthesis that closes the class declaration
```

MAke sure you save the file!! Execute jjtree, javacc, javac and run the program¿ Do you see any changes?

Task: How would you implement the calculator? **Hint:** For tree traversal, use the method `jjtGetChild(int i)`, which returns a Node's child (also a Node) indexed by i (starts from 0).

Introducing the SIMPLE language

Go to the `tpl` subdirectory. There you will find four files: TPL.jjt, TPLTypes.java, TPL.java and ident.tpl. TPL.jjt defines the syntax of the SIMPLE language used in your coursework, together with the jjtree annotations used to create the respective AST. ident.tpl is a sample program written in the SIMPLE language.

Execute: `jjtree TPL.jjt`, `javacc TPL.jj` and javac `TPL.java`.

What happens? You will get a few compilation errors. That's because the generated AST classes are missing a few fields (set by TPL.jjt). In order to fix this you ned to declare:

```
    public int type;
    public String name;
```

in `ASTVarDeclaration.java`,

```
   public String name;
```

in `ASTReadStatement.java`, `ASTWriteStatement.java`, and `ASTId.java`, and:

```
   public int val;
```

in `ASTIntConst.java`. Recompile and run.