Lund University
Computer Science
Görel Hedin, Jesper Öqvist, Niklas Fors

Compiler Construction
EDA180
2014-09-07

Programming Assignment 2
LALR parsing and building ASTs

The goal of this assignment is to understand how to use LALR parsing for more complex grammars with expressions, and how to construct *Abstract Syntax Trees* (ASTs) during parsing. You will:

- Write a more complex context-free grammar and implement it using an LALR parser.
- Resolve ambiguities and experiment with LALR conflicts.
- Build an AST in the semantic actions of the parser.

In addition to JFlex, Beaver, and other tools you used in the previous assignment, you will use the JastAdd tool to define AST classes. For documentation of JastAdd, see `http://jastadd.org`.

You will define a language called *SimpliC* (Simplistic C), and generate a compiler that parses SimpliC into ASTs. To your help, there is a demonstration example CalcAST showing how to build ASTs for the Calc language.

As before, you should practice agile software development. Writing compilers can be very complex, and it is important that you break down your work in small steps with test cases, ensuring that each step compiles and works correctly before you go on with the next one.

▶ You must solve large parts of this assignment before going to the lab session. Tasks that you need to do in advance are marked with a black triangle, like this:

▶ **Print out your grammars and bring the printouts to the lab!**

▷ Additional tasks that need to be completed are marked with a hollow triangle, like this. You are strongly encouraged to start on these tasks before the lab as well.

1 The CalcAST Demo

The CalcAST demonstration example implements an extended version of the *Calc* language from assignment 1, introducing some new features; the language now allows multiple variable bindings in the `let` expression, and a new `ask user` expression has been added.

Here is an example of a Calc program:

```
let
    radius = ask user [1.0]
    pi = 3.14
in
    2.0 * pi * radius
end
```

The program uses the new `ask` expression to ask the user for a value for the `radius` variable. The default value, `1.0`, is given in the square brackets. The default value in the `ask` expression is optional.

This example program is named `example2.calc` and can be found in the `testfiles/ast` directory.

▶ Download the CalcAST example. Run the tests and see that they pass.

1.1 The abstract grammar

The *Context Free Grammar* (CFG) for the new Calc language is

$$
\begin{array}{lll}
\text{program} & \rightarrow & \text{exp <EOF>} \\
\text{exp} & \rightarrow & \text{factor} \;|\; (\text{ exp '*' factor }) \\
\text{factor} & \rightarrow & \text{let} \;|\; \text{numeral} \;|\; \text{id} \;|\; \text{ask} \\
\text{let} & \rightarrow & \text{'let' binding}^{+}\text{ 'in' exp 'end'} \\
\text{binding} & \rightarrow & \text{id '=' exp} \\
\text{numeral} & \rightarrow & \text{<NUMERAL>} \\
\text{id} & \rightarrow & \text{<ID>} \\
\text{ask} & \rightarrow & \text{'ask' 'user' ['[' exp ']']}
\end{array}
$$

To represent an AST in Java we will use a class hierarchy generated by JastAdd, and defined in an *abstract grammar*, in this case `calc.ast`. The abstract grammar is similar to the CFG:

```
Program ::= Expr;

abstract Expr;
Mul: Expr ::= Left:Expr Right:Expr;
Numeral: Expr ::= <NUMERAL:String>;
IdUse: Expr ::= <ID:String>;

Let: Expr ::= Binding* Expr;
Binding ::= IdDecl Expr;
IdDecl ::= <ID:String>;

Ask: Expr ::= [Default:Expr];
```

Each rule in the abstract grammar corresponds to one Java class. The name on the left-hand side specifies the name of the class. The keyword `abstract` makes the class abstract. The name after the colon (`:`) on the left-hand side specifies the superclass of the class, e.g., `Mul` is a direct subclass of `Expr`. The right-hand side defines the children (composition relationships in UML) and tokens (specified with `<>`). For example, the `Program` class has one child of type `Expr` and the name of the child is the same as the type. Children can also be named using colon (`:`), e.g., `Mul` has two children, one named `Left` and one named `Right`, both having the same type `Expr`. The type of a token is specified after the colon (`:`). The token `<TokenName>` is an abbreviation for `<TokenName:String>`. Kleene star (`*`) means iteration and square brackets (`[]`) means an optional child.

The nonterminal `exp` in the CFG has been modelled as an abstract class in the abstract grammar. The alternatives on the right-hand side have been modelled using subclassing. This is a common pattern when using object-orientation for representing ASTs: replace alternatives with subclassing.

Note that we are not interested in having `factor` as a class in the abstract grammar: this nonterminal is only used for disambiguating the CFG so that we can parse text into a tree in un anambiguous way. Instead, all expressions (`Mul`, `Numeral`, `IdUse`) are subclasses of `Expr`.

Another difference is that the production `id` in the CFG has been replaced with two classes `IdUse` and `IdDecl` in the abstract grammar. This separation is practical when implementing name analysis, where name uses are bound to name declarations. We will implement name analysis later on, in another assignment.

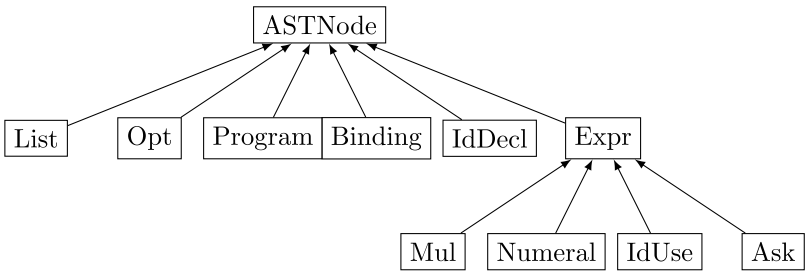
The JastAdd abstract grammar notation supports repetition of zero or more components (*), but not repetitions of one or more (+). It is fine to replace (+) in the context-free grammar with (*) in the abstract grammar, since a program lacking bindings in a let-expression will result in a syntax error. There is no need to model that there will be at least one binding in the abstract grammar, as that would just complicate the grammar.

1.2 Generated AST classes

The generated classes will contain accessor methods for the children and tokens, as the following simplified API illustrates for the classes `Mul` and `IdUse`.

```
public class Mul extends Expr {
    Expr getLeft();
    Expr getRight();
    void setLeft(Expr e);
    void setRight(Expr e);
}
public class IdUse extends Expr {
    String getID();
    void setID(String id);
}
```

The Java classes generated by JastAdd will have the following class hierarchy:



`ASTNode`, `Opt` and `List` are implicit, or predefined, AST classes. They are generated by JastAdd without any mention in the `.ast` file. `Opt` and `List` are used to represent optional and list components in the grammar.

► Look at the abstract grammar specification (`src/jastadd/calc.ast`) and make sure you understand it. Look at the corresponding generated classes in `src/gen/lang/ast`. The classes contain lots of generated methods, but you should be able to find, for example, `Mul`'s `getLeft()` and `getRight()` methods mentioned above. Familiarize yourself briefly with the JastAdd reference manual[1] so that you know where to look for details later.

1.3 Well-formed ASTs

When building a JastAdd AST, care must be taken to build it so that it is *well-formed* in the following sense:

- Each component of an AST node must be of the class stated in the abstract grammar, or an `Opt` node for an optional component, or a `List` node for a list component.
- An AST node for an optional/list component should be a child of the corresponding `Opt` or `List` node, and of the class stated in the grammar.
- Each component must be a real object, i.e., not `null`.
- The node objects must form a tree, i.e., it is not allowed to reuse an object in several places in the same AST, or in different ASTs.

These conditions are in general not checked when the parser code is compiled, so take care in building the AST correctly, as you might otherwise run into problems that are difficult to debug.

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

1.4 Building the AST

The abstract syntax tree is built by the parser, as specified in the `.beaver` file, using so called *semantic actions* of the production rules. Each nonterminal in the CFG is associated with a return type, i.e., the type of the ASTs built by the productions of that nonterminal. In Beaver, the return type is specified with Beaver's `%typeof` directive, as the following code fragment illustrates.

```
%typeof program = "Program";
%typeof exp = "Expr";
%typeof factor = "Expr";
%typeof let = "Let";
...
```

1.4.1 Semantic Actions

To build AST nodes, the productions in the parser specification need to be extended with semantic actions. Semantic actions are code snippets that are run during parsing. Consider the production for the `program` nonterminal:

```
program = exp.a {: return new Program(a); :} ;
```

Note the code inside the `{:` and `:}`. This is the semantic action code – ordinary Java code for constructing the AST node representing the parsed production. In this case, the semantic action simply creates a new node of type `Program` with an expression as its child. Note that `program` is a nonterminal in the parser specification, whereas `Program` is an AST node type, specified in the abstract grammar. The expression inside the semantic action must have the same type as specified in the `%typeof` directive.

In Beaver, a right-hand side component can be *named* using dot notation: in the example above, the nonterminal `exp` has been named `a`, by writing `exp.a`. The name `a` refers to the AST subtree constructed by the `exp` nonterminal, and can be used inside the semantic action to refer to that subtree. In this case, `a` is used in the semantic action as an argument to the `Program` constructor.

1.4.2 Optional Components

For most productions, building the AST is straightforward as explained in the previous section. But if there are optional or list components in a production, an extra `Opt` or `List` node needs to be built.

Beaver has support for optional and list components, but to build the AST it is simpler to first rewrite the grammar to use explicit alternatives. Recall the new `Ask` construct which has an optional default expression. The context-free rule is:

$$\text{ask} \;\rightarrow\; \text{'ask' 'user' ['[' exp ']']}$$

and the abstract grammar rule is:

```
    Ask: Expr ::= [Default:Expr];
```

To implement this in Beaver, we first rewrite the context-free rule to eliminate the optional notation. We can do this by introducing a new nonterminal `opt_exp` with two alternatives: one non-empty and one empty:

$$\begin{array}{lcl} \text{ask} & \rightarrow & \text{'ask' 'user' opt\_exp} \\ \text{opt\_exp} & \rightarrow & \text{'[' exp ']'} \;\mid\; \epsilon \end{array}$$

The Beaver specification can now be written as follows:

```
ask =
    ASK USER opt_exp.d {: return new Ask(d); :};

opt_exp =
      LBRACKET exp.a RBRACKET  {: return new Opt(a); :}
    | /* epsilon */            {: return new Opt(); :}
    ;
```

We also need to specify the type of the new `opt_exp` nonterminal, in the beginning of the beaver file:

```
%typeof opt_exp = "Opt";
```

Note that the semantic action always builds an `Opt` node to match the optional expression, and that different `Opt` constructors are used depending on if the optional expression is present or missing.

1.4.3 List Components

In order to handle list components (repetition), we use a similar technique as for optionals, rewriting the parsing grammar to replace repetition with a new nonterminal with explicit alternatives.

Recall the abstract grammar rules for bindings in let-expressions:

```
    Let: Expr ::= Binding* Expr;
    Binding ::= IdDecl Expr;
```

To implement this in Beaver, we first rewrite the context-free grammar rule for `Let`:

$$\mathsf{let} \ \rightarrow \ \mathsf{binding}^{+}$$

replacing the repetition with a new nonterminal `binding_list`:

$$\mathsf{binding\_list} \ \rightarrow \ \mathsf{binding} \ \mid \ (\ \mathsf{binding\_list}\ \mathsf{binding}\ )$$

The Beaver specification for `binding_list` can be written as follows:

```
%typeof binding_list = "List";
...
binding_list =
      binding.a                    {: return new List().add(a); :}
    | binding_list.list binding.a {: return list.add(a); :}
    ;
```

▶ Look at the parser specification (`src/parser/parser.beaver`) and make sure you understand how the AST is built. Where are the semantic actions? How are optionals and lists handled?

1.5 The DumpTree Aspect

The CalcAST example contains code for translating an AST into a string showing the tree structure using indentation and the class names of the nodes. This is useful when writing test cases and for debugging, to see if the correct tree has been built. The code is located in an "aspect file" `DumpTree.jrag` in the `src/jastadd` directory in the CalcAST project. This file contains a single aspect declaration:

```
aspect DumpTree {
    public String ASTNode.dumpTree() {
        ByteArrayOutputStream bytes = new ByteArrayOutputStream();
        dumpTree(new PrintStream(bytes));
        return bytes.toString();
    }

    public void ASTNode.dumpTree(PrintStream out) {
        [...]
    }

    [...]
}
```

The declarations in that aspect behave just like ordinary Java class body declarations, except that they have an extra qualifier before the declaration's name (`ASTNode.`) and they have been removed from the class they are supposed to be in. The JastAdd code generator weaves these declarations into the generated AST classes. We will look at other types of aspect declarations later in the course.

▶ Look at the testcases in the `testfiles/ast` directory. The `.expected` files contain the expected output after parsing the corresponding `.calc`-file and printing the AST using the `dumpTree` methods. Make sure you understand what AST an `.expected` file corresponds to.

1.6 New testing framework

The CalcAST example uses an improved testing framework (in comparison with the CalcParse example). It uses a technique in JUnit called *parameterized tests*. With this technique, you don't need to write a test method for each test, but it is sufficient to add input and expected output files to the testfile directory.

In CalcAST, this testing technique is used for the AST tests to check that correct ASTs are built when parsing different test programs. The framework starts one test per `.calc`-file in the `testfiles/ast` directory. Each test file is parsed and the tree is printed to a corresponding `.out` file. If there exists a matching `.expected` file then the contents of the output is checked against the expected output. The file extensions used may be changed by editing e.g. the `IN_EXTENSION` constant in the `AbstractParameterizedTest` class.

When adding new tests it is usually too tedious to hand-write each `.expected` file. Instead, you can add just the input file, run the tests to get the `.out` file, and then rename the `.out` file to use the `.expected` extension. Of course, it is important to inspect the `.out` file before renaming it, to make sure it really is correct!

Tip: You can use the new Ant target `update-expected-output` to copy all `.out` files to the corresponding `.expected` file:

```
ant update-expected-output
```

▶ Add a new AST test case in the CalcAST project.

1.7 Updated Build Script

The CalcAST build script contains a new `taskdef` element:

```
<taskdef classname="jastadd.JastAddTask"
         name="jastadd"
         classpath="${lib}/jastadd2.jar" />
```

This defines a new *Ant* task named `jastadd`. The task is used by the new `ast-gen` target to invoke the JastAdd program and generate the AST classes.

The `taskdef` requires the `jastadd2.jar` file to be in the `lib` directory. This jar file can be downloaded from `http://jastadd.org`, but you don't need to do that for CalcAST since it already contains the jar file, together with all other required jar libraries.

2 Debugging experiments

There are many things that can go wrong when you generate a parser. Your grammar might be non-LALR in which case the parser generator will report shift-reduce or reduce-reduce conflicts. In some cases you might write something incorrect in the .beaver specification which is not discovered until you compile the generated Java code. It is very likely that you will run into problems like these, and it will be much easier to deal with them if you have encountered them before in a controlled setting. So in this section we will provoke some common errors so you understand them better.

2.1 Provoke a shift-reduce conflict in expressions

▶ Add a shift-reduce conflict to the beaver parsing specification by changing the production

$$\mathsf{exp} \rightarrow \mathsf{exp}\ \text{'*'}\ \mathsf{factor}$$

to

$$\mathsf{exp} \rightarrow \mathsf{exp}\ \text{'*'}\ \mathsf{exp}$$

Rerun the parser generator (`build gen`). Don't forget to do refresh if you are running in Eclipse.

- What does the parser generator say?
- Why is this a conflict?
- Try running the tests. Do they pass?
- Can you give an example program that could be parsed in two different ways with the new grammar?
- What happens when these programs are parsed if *shift* is chosen in the conflict? What happens when *reduce* is chosen?
- Why was there not a conflict in the original version of the grammar?

Undo the change and make sure all tests pass again.

2.2 Introduce a shift-reduce conflict through a subtle error

▶ Add an empty production to the `binding_list` production in the `.beaver` file by changing

```
binding_list =
   binding.a      {: return new List().add(a); :}
   | binding_list.a binding.b {: return a.add(b); :}
   ;
```

to

```
binding_list =
   | binding.a      {: return new List().add(a); :}
   | binding_list.a binding.b {: return a.add(b); :}
   ;
```

This is something that could easily result from a copy paste, but it has the effect of adding an empty production (ϵ) to the `binding_list` rule. So we now have three productions rather than two. If we really wanted this empty production, it would be a good idea to add a comment to make this more explicit, like this:

```
binding_list =
     /* epsilon */
   | binding.a      {: return new List().add(a); :}
   | binding_list.a binding.b {: return a.add(b); :}
   ;
```

But of course, in this case, this extra empty production was just a mistake.

Rerun the parser generator on the grammar. What does the parser generator say? What does the error message mean? The parser automatically selects shift, which happens to be what we want it to do. But if you get warnings like this, you should always analyze them carefully and change the specification to resolve the conflict explicitly and thereby remove the warnings. Otherwise it will be difficult to see new warnings if you change the grammar later on.

Undo the change and make sure all tests pass again.

2.3 Errors in the generated Java code

The code you write in Beaver's semantic actions is Java code. But Beaver does not analyze it, so you will not see compile-time errors in it until you compile the generated code. It can be a bit tricky to understand how those errors correspond to the code you originally wrote in the semantic actions.

Try a couple of common mistakes:

▶ Remove one of the `%typeof` directives in the `.beaver` file. What happens when you generate the parser? What happens when you try to compile the code? If you are in Eclipse, what happens if you try to run the tests? Look at the compile-time error. Can you figure out how it relates to the error you inserted?

▶ In one of the semantic actions, remove the `return` keyword. What happens when you generate the parser? Look at the compile-time error. Can you figure out how it relates to the error you inserted?

2.4 Add print statements

As always, inserting print statements is a primitive but useful way to help with debugging. The semantic actions are just Java code inserted into the generated parser at different places. If you add more code than building ASTs, it will of course be run too.

Optional task: Add some print statement, for example `System.out.println("Hello")`, in one of the semantic actions. Generate the parser and run the tests.

3 The SimpliC Language

Your task is to design an abstract grammar for the SimpliC language, described below, using the JastAdd system. You should also write a Beaver parser specification to parse the grammar, and the parser should produce a well-formed abstract syntax tree. There should be automated tests showing that the parser produces the right tree for a number of well chosen examples.

Expressions should be represented as trees, where each node corresponds to an operator having its operands as left and right subtrees. Take a look at the CalcAST example for inspiration.

The SimpliC language is a C-like language that uses only integer variables. There are two built-in functions, `read` which reads an integer from standard in, and `print` which prints its argument to standard out. Functions may not be nested, and all functions must return an integer result (i.e. all functions must have a `return` statement). The `if` and `while` statements are the only branching statements other than function calls. There are no global variables, only local variables and function parameters.

You should derive the context-free grammar for the language from the sample code below. Your grammar should cover at least the example code, and in addition it should support the common arithmetic operators `+`, `-`, `*`, `/`, `%` and the comparison operators `==`, `!=`, `<=`, `<`, `>=`, and `>`. The usual priority and associativity rules should apply. For example, "`<`" should have lower precedence than `+` and be non-associative. The remainder operator, `%`, has the same precedence and associativity as multiplication and division.

3.1 SimpliC Code Sample

The `gcd1` and `gcd2` functions implement two different algorithms for finding the greatest common divisor (GCD) of two positive integers. The example program tests both implementations with inputs provided by the user:

```
int gcd1(int a, int b) {
    while (a != b) {
        if (a > b) {
            a = a - b;
        } else {
            b = b - a;
        }
    }
    return a;
}

int gcd2(int a, int b) {
    if (b == 0) {
        return a;
    }
    return gcd2(b, a % b);
}

/* Test two different implementations of GCD algorithms
 * and print the result. The results should be equal
 * provided that both inputs are positive.
 */
int main() {
    int a;
    int b;
    a = read();
    a = read();
    print(gcd1(a, b));
    print(gcd2(a, b));
    return 0;
}
```

3.2 Tasks

Before you start implementing, you should design the grammars:

- ▶ Start by designing the abstract grammar, using the JastAdd `.ast` format. Make sure you select good names for your classes and components. Comment on nontrivial design decisions you make.
- ▶ Write down the context-free grammar, using high-level EBNF notation (not Beaver). Make it as simple and clear as you can, and as similar to the abstract grammar as you can. Make sure it covers the example program above.
- ▶ Write down a lower-level version grammar that will be suitable to implement in Beaver. I.e., try to eliminate ambiguities and eliminate repetition and optionals.
- ▶ ***Bring printouts of all these three grammar versions to the lab.***
- ▷ Implement a scanner, parser, and AST builder for the SimpliC language, with test cases showing that the correct ASTs are built. The parser will be the most difficult part to get right. It is a good idea to start with a minimal parser, but add the full .ast and .flex specifications, possibly skipping support for comments to start with. Then grow the parser gradually, adding test cases, until you can handle the full `gcd` program. To your help there is an example called MinimalAST that you can download and use as a starting point. MinimalAST is similar to CalcAST, but supports a very small language with just one token and one production.

There are many things that can go wrong when you implement a parser. It is a good idea to save versions of your project so you can back up to an earlier stable state in case you get into a too big mess.

4 Topics for Discussion

1. What nontrivial problems did you encounter when designing and implementing your abstract and parsing grammars? How did you solve them?
2. Binary expressions can be modelled in JastAdd in the two following ways:

   ```
   abstract Expr;
   Mul: Expr ::= Left:Expr Right:Expr;
   Div: Expr ::= Left:Expr Right:Expr;
   ```

   or

   ```
   abstract Expr;
   abstract BinExpr : Expr ::= Left:Expr Right:Expr;
   Mul: BinExpr;
   Div: BinExpr:
   ```

   What is the advantage of the second model?
3. Could the semicolons be removed from the SimpliC language? Would the grammar still be unambiguous?
4. In mathematics you may write $0 < x < 1$ but many programming languages (e.g. Java) disallow this. Do you know of programming languages that allow this? What is the semantics of the construct in these languages? Why does not Java allow it?

4.1 Optional Challenges

When you have completed the compulsory part of the assignment you may try one or more of these optional challenges:

- Add `break` and `continue` statements
- Add global variables
- Allow the print statement to take a string argument
- Support negative integer literals
- Make semicolons optional