# 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.

Try to solve all parts of this assignment before going to the lab session. If you get stuck, use the Piazza forum to ask for help. If this does not help, you will have to ask at the lab session. Make notes about answers to questions in the tasks, and about things you would like to discuss at the lab session.

▶ Major tasks are marked with a black triangle, like this.

# 1 The CalcAST Demo

The CalcAST demonstration example implements an extended version of the *Calc* language from assignment 1, introducing some new features that will illustrate the use of lists and optionals. The language now allows a list of variable bindings in the `let` expression, and a new `ask user` expression with an optional default value 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 idea is that when this program is run, the `ask user` expression will ask the user to input a value, using the default value of `1.0`. An `ask user` expression without the optional default value would be written simply as `ask user`, without the square bracket clause.[1]

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

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

---

[1]Note, however, that we will not actually run Calc or SimpliC programs until assignments 5 and 6.

## 1.1 The abstract grammar

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

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

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];
```

Here, an abstract grammar rule

```
A: B ::= children;
```

corresponds to a Java class `A` which extends the superclass `B` and has a number of *children* (composition relationships in UML). The children can be of the following kinds:

```
C        A child of type C
C*       A list of children of type C
[C]      An optional child of type C
<T:S>    A token named T of type S
```

Children may be explicitly named, for example:

```
N:C      A child named N of type C
```

and if no explicit name is given, the type name is used for the child.

The nonterminal `exp` in the CFG has been modelled as an abstract class `Expr` in the abstract grammar. The alternatives on the right-hand side in the CFG have been modelled using subclassing in the abstract grammar. 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 an unambiguous way. Instead, all expressions (`Mul`, `Numeral`, `IdUse`) are direct subclasses of `Expr`.

Another difference is that the nonterminal `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 highly recommended to use (X*) instead of trying to model (+) as `X X*` in the abstract grammar. This is because the abstract grammar should be kept simple to make it easy to program semantic analysis later on. The requirement on there being at least one element can easily be checked either by the parsing rules, or by adding an attribute (see assignment 4).

## 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:
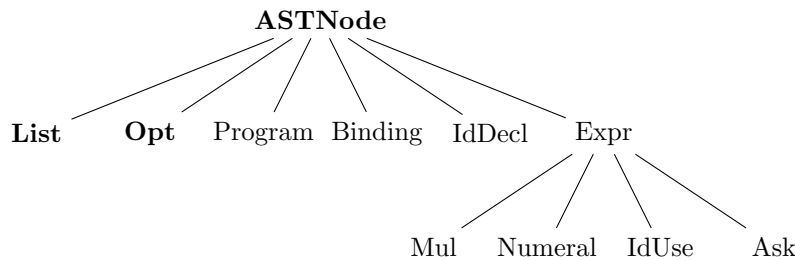


`ASTNode`, `Opt` and `List` are predefined implicit AST classes. They are generated by JastAdd without any mention in the `.ast` file. `Opt` and `List` are used to represent optional and list components in the abstract syntax tree. `ASTNode` serves as the common superclass of all AST nodes.

▶ 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[2] 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*. Suppose we have an abstract grammar class `P ::= A [B] C*`. A P node is well-formed if

- an ordinary child, like `A`, is represented by a node of type `A`.

- an optional child, like `[B]`, is represented by a node of type `Opt` which has zero or one children of type `B`.

- a list child, like `C*`, is represented by a node of type `List` which has zero or more children of type `C`.

- Each node must be a real object, i.e., not `null`.

---

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

- The node objects must form a tree, i.e., it is not allowed to reuse an node in several places in the same AST, or in different ASTs.

The example below shows a well-formed AST for `P ::= A [B] C*`
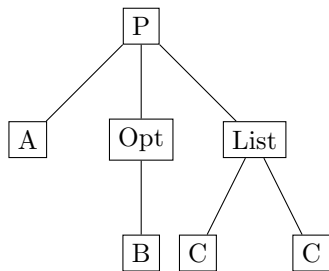


These conditions are not checked when generating the parser, and several of them are not even checked when the parser code is compiled. So take care in building the AST correctly, as you might otherwise run into runtime errors and problems that are difficult to debug.

▶ What would be examples of other well-formed ASTs for the `P ::= A [B] C*` class? What would be examples of ill-formed ASTs?


## 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. We have to instruct Beaver what kind of JastAdd AST node is built by the productions for each Beaver nonterminal. This will become the return type of the semantic action in the generated parser. We specify this using Beaver's `%typeof` directive, as the following code fragment illustrates.

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

Note that to the left in each clause, we have a nonterminal in the Beaver CFG, like `program`. To the right, we have a class from the JastAdd abstract grammar, like `Program`.


### 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 must be built.

Although Beaver does have some support for optionals and lists, it is easier to build the AST if we first rewrite the grammar to eliminate optionals and lists, using 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:

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

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`:

$$\text{let} \;\rightarrow\; \text{binding}^{+}$$

replacing the repetition with a new nonterminal `binding_list`:

$$\text{binding\_list} \rightarrow \text{binding} \mid \text{binding\_list binding}$$

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 printing an AST, using indentation to show the tree structure. This is useful when writing test cases and for debugging, to check that the AST is well-formed and correct.

The code is located in an "aspect file" `DumpTree.jrag` in the `src/jastadd` directory in the CalcAST project. A JastAdd *aspect* may contain methods of AST classes, but specified in a separate file. The JastAdd system *weaves* those methods into the generated AST classes. The file `DumpTree.jrag` contains a single aspect declaration:

```
1  aspect DumpTree {
2      public String ASTNode.dumpTree() {
3          ByteArrayOutputStream bytes = new ByteArrayOutputStream();
4          dumpTree(new PrintStream(bytes));
5          return bytes.toString();
6      }
7
8      public void ASTNode.dumpTree(PrintStream out) {
9          [...]
10     }
11
12     [...]
13 }
```

The methods in the aspect have an extra qualifier before the method name, e.g., `ASTNode.dumpTree`. The JastAdd code generator will weave the method into the generated `ASTNode` class. This way of writing a declaration in an aspect instead of in the class in which it belongs is called an *inter-type declaration*. Later in the course, we will look at other kinds of inter-type declarations you can write in an aspect.

The `compiler.jar` main class is `Compiler.java` and it will parse in a calc program and then print it using the `dumpTree` method.

▶ Try out `compiler.jar` on one of the test programs. For example:

```
./gradlew jar
java -jar compiler.jar testfiles/ast/example2.in
```

6

## 1.6  Testing framework

The CalcAST example uses a testing framework that is more advanced than the one used in assignment 1. The new test framework uses a JUnit feature called *parameterized tests*. With parameterized tests you don't need to write a test method for each test case: it is sufficient to add an input and expected output file to the `testfiles/ast` 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 `.in`-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.

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!

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

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


## 1.7  Build Script

The CalcAST Gradle script (`build.gradle`) contains some more elements and tasks compared to the script we used in assignment 1. To be able to run JastAdd, it uses the `jastadd` plugin and a new `generateJava` task:

```
compileJava.dependsOn 'generateJava', 'generateScanner', 'generateParser'
task generateJava(type: org.jastadd.JastAddTask) {
  description 'Generates the compiler AST classes from JastAdd specifications.'
  ...
}
```


# 2  Troubleshooting

Many things can go wrong when you generate a parser. Your grammar might be non-LALR in which case the parser generator will report conflicts. Some errors in the `.beaver` specification are not discovered until you compile the generated Java code, or until you run the generated parser. 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 experiment with the CalcAST example to provoke some common errors so that you understand them better.[3]


## 2.1  Provoke an LALR conflict in the parsing specification

▶ Add a conflict to the CalcAST parser specification by changing the production

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

to

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

---

[3]The examples in this section are tested on Beaver 0.9.11. If you are using another version, error handling might differ.

This grammar is not LALR, and there will be a conflict in the parsing table. In this case, it is a shift-reduce conflict.

▶ Rerun the build (`./gradlew test`) to see how Beaver handles this problem. Don't forget to do "refresh" in your project if you are running Eclipse.[4]

Note that the build goes through and the tests are run, even if there is a conflict. Scroll back to look at the Beaver output. Note that Beaver gives a *warning*, rather than an error. This is because it by default resolves shift-reduce conflicts by selecting shift. It goes on and generates a parser that will shift in the conflicting situation. In this case, shift is not the right way to resolve the conflict, and because we happen to have a good test suite, several tests fail.

***Important advice.*** When you get a conflict, try to figure out what the problem is, and change the grammar to resolve it. Even if Beaver happens to resolve the conflict in the right way, make sure to change the grammar so that you don't get any warnings and default resolutions by Beaver. Otherwise, it will be very difficult for you to sort out new conflicts later on.

To find out what the problem is, study the error message given by Beaver, study any test cases that fail, and think about your last change to the grammar. If you can construct an example program that illustrates the conflict, this will help you understand the problem.

In this particular case, the error message is:

```
Resolved Shift-Reduce conflict by selecting (MUL: SHIFT; goto 3) over
(MUL: REDUCE exp = exp.a MUL exp.b) using precedence
```

This means that if a `MUL` token appears after parsing the right hand side of

```
exp = exp.a MUL exp.b
```

both shift and reduce would fit the grammar, and Beaver will choose to shift.

We can try to understand what happens by constructing an example corresponding to that situation:

```
x * y * ...
```

The conflict occurs after `y`, when reading the second `MUL`. We can either reduce `x * y`, or shift the `MUL`. If we reduce `x * y`, a left-recursive tree will be built. If we shift the `MUL`, we will later have to reduce `y * ...`, and a right-recursive tree will be built.

▶ Undo the change and make sure all tests pass again.

## 2.2 Unintended empty productions

A common mistake is to use an extra | in a rule, which results in an extra empty production. This can easily happen when copying and pasting lines with production rules. As an example, consider the following rule:

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

When writing this rule, we might by mistake let the first production start with | as follows:

---

[4]When you run unix commands, Eclipse does not automatically know that files have been added, changed, or removed. When you do a "refresh" in Eclipse, it will scan the files in the project directory to look for changes.

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

But this means that before the first |, there is actually an empty production. If we do want an empty production, it is good practice to add a comment to make it more prominent, like this:

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

But in this case, we don't want the empty production, and running the grammar above through Beaver gives a shift-reduce conflict with the following message:

```
Resolved Shift-Reduce conflict by selecting (ID: SHIFT; goto 26) over
(ID: REDUCE binding_list = ) using precedence.
```

We see that it has found a shift-reduce conflict that would occur after parsing the right-hand side of the production

```
binding_list =
```

So there is obviously an empty production for `binding_list` in our grammar.

▶ Experiment with this problem by adding an extra | to the `binding_list` production in the `.beaver` file to get an empty production. Rebuild, and confirm that there is now a shift-reduce conflict.

## 2.3   Optional: try NeoBeaver for better error messages

At LTH, an experimental parser generator, NeoBeaver, is being developed with the intention of giving better error messages than Beaver. In case you experience that you don't understand the error messages Beaver gives, you are very welcome to try out NeoBeaver. We might use it in the next instance of the course, and are very happy to get feedback on it. However, although we have used NeoBeaver in extensive compiler projects, it is still experimental, and might contain bugs. So if you experience any problems with it, you are advised to switch back to Beaver.

You can use NeoBeaver as a drop-in replacement for Beaver, by changing one line in the `build.gradle` file. Change

```
dependencies {
    ...
    beaver 'net.sf.beaver:beaver-cc:0.9.11'
}
```

to

```
dependencies {
    ...
    beaver 'org.extendj:nbfront:0.1.2'
}
```

*Note!* There are two lines in the build file that use Beaver dependencies. Make sure you change the right one (replace the line starting with `beaver`, not the one starting with `compile`).

NeoBeaver is actively developed by Jesper Öqvist. Any new releases will appear at Maven, and you find them by searching for `nbfront` at `http://search.maven.org`. Issues can be reported here: `https://bitbucket.org/joqvist/neobeaver`, and there is a minimal webpage for the tool here: `https://extendj.org/neobeaver.html`.

▶ Optional task: try NeoBeaver on the two errors in Sections 2.1 and 2.2.


## 2.4   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 compile-time or runtime 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. Note that the parser is generated without any errors, but when you compile the generated code, the Java compiler will complain about both incompatible types and missing symbols (declarations).

▶ In one of the semantic actions, remove the `return` keyword. Note that the parser is generated without any errors. Depending on what version of Beaver/NeoBeaver you are running, you might get a compile-time error when you run the Java compiler, or a run-time error when you run the parser.


## 2.5   Adding new terminals

When you add new terminals in the scanner specification, they need to be used by the parser specification, so that the parser generator generates the appropriate Java code for the tokens. Otherwise, you will get errors when you compile the generated scanner.

Therefore, it can be a good idea to add a dummy production as a first step, when you have added some new terminals in the scanner. This way, you don't have to immediately get a working parser, but can focus on first getting the generated scanner to compile, then to get the parser working.

Suppose you have added two new keywords, IF, and THEN in the scanner specification:

```
"IF"            { return sym(Terminals.IF); }
"THEN"          { return sym(Terminals.THEN); }
...
```

You will then have to declare the new terminals in the parser specification, and add a dummy production so that the terminals are reachable from the start symbol:

```
%terminals IF, THEN; // DECLARATION OF NEW TERMINALS
...
program = ...
          | IF THEN // DUMMY PRODUCTION FOR UNUSED TERMINALS
          ;
```

Now, you can check that the compiler builds and old tests pass, before starting to add the real productions that use the new terminals.

▶ As an experiment, try adding a new terminal in the scanner specification. Confirm that you get a compile-time error. Then update the parser specification to declare the new terminal and add a dummy production for it. Confirm that you can get the compiler to build again, and that all tests pass.

## 2.6  Cast exception when running the generated parser

If you leave out a semantic action in the parser specification, a default semantic action will be generated. The default action will return the result of parsing the last symbol in the production (or the last variable, in case the rule contains variables). This is sometimes what you want, but not always. Consider the following example:

```
factor =
        let
        | ask
        | numeral
        | id_use
        ;

let = LET binding_list IN exp END;
```

For `factor`, the default behavior is what we want. However, for the `let` rule, the last symbol is the `END` token. This is, of course, completely wrong—what we would like to return is a new `Let` subtree. Because of the way Beaver is implemented, the error will not show at compile-time, but instead at runtime, when running the parser. The generated parser will at some point cast an object to what it thinks is an expression, but since it is actually an `END` token, a class cast exception is thrown.

▶ Replace the `let` rule in the Beaver specification by the one shown above (without semantic action or variables). Confirm that you get a class cast exception when you run the compiler on `testfiles/ast/let.in`.

## 2.7  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.

▶ As an experiment, 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

The main task in this assignment is to implement a parser for the SimpliC language. You should do this by specifying an abstract grammar using the JastAdd system, a parser specification using the Beaver system, and a scanner specification using the JFlex system. The parser should produce a well-formed abstract syntax tree, that follows the abstract grammar. Furthermore, there should be automated tests showing that the parser produces the correct AST for a number of examples. There should also be some negative tests, showing that the parser detects parsing errors in erroneous input files.

The SimpliC language is a C-like language that uses only integer variables. A SimpliC program consists of a list of function declarations. Function declarations may not be nested, and all user-defined functions must return an integer result. There are also two built-in functions: `read` which reads an integer from standard in, and `print` which prints its argument to standard out. These are not declared in the program, but can be called from the program.

There are no global variables, only local variables and function parameters. Statements supported include assignments, `return` statements, function call statements, `if`, and `while` statements. Expressions supported include identifier uses, function calls, and common binary arithmetic expressions like +, -, *, /, % and the comparison expressions ==, !=, <=, <, >=, and >. Binary expressions should be represented as trees, where each node corresponds to an operator having its operands as left and right subtrees. The usual priority and associativity rules should apply. For example, "<" should have lower priority than + and be non-associative. The remainder operator, %, has the same priority and associativity as multiplication and division. Parentheses can be used in expressions. SimpliC also supports end-of-line comments.

## 3.1 SimpliC Code Sample

The example program below shows large parts of the SimpliC syntax. The `gcd1` and `gcd2` functions implement two different algorithms for finding the greatest common divisor (GCD) of two positive integers. The `main` function tests both algorithms 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 GCD algorithms and print the results.
// The results should be equal when both inputs are positive.
int main() {
  int a;
  int b;
  a = read();
  b = read();
  print(gcd1(a, b));
  print(gcd2(a, b));
  int diff = gcd1(a, b) - gcd2(a, b);
  print(diff);
  return 0;
}
```

## 3.2 Implementing the parser

In implementing the parser, you should work in small steps: first implement a parser for a small subset of the language, then add a few constructs, and so on. Many things can go wrong, so this way you can easier relate any new problems to your latest changes.

▶ For each of the implementation steps below, this is how you should work: Write down a test program that you want to be able to parse. Then design the abstract grammar (using the JastAdd .ast format), and a corresponding concrete grammar in EBNF (the latter can be on paper). Try to make the EBNF as similar as possible to the .ast grammar. However, it must be unambiguous, for example dealing with priorities and associativity, which might call for adding more rules. Then rewrite the EBNF grammar to eliminate lists and optionals, and code it up in a .beaver specification. Get the parser to work, and automate the test. Run all your tests to make sure you have not broken anything that worked previously. Review your implementation to make sure you use good names for classes and components in the abstract grammar. If you get stuck, take a look at the CalcAST example for inspiration.

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.

It is recommended to use Git (or some other version control tool), and save a new version as soon as you have a working version of your project where all current tests pass. This way, you can easily back up to an earlier stable state in case you get into a too big mess. Remember to make the Git repository private, so only you and your lab partner can see it.

### 3.2.1   Step 1: Simplified function declarations

▶   As the first step, implement a parser for programs consisting of only a list of empty function declarations without parameters. For example:

```
int a() { }
int b() { }
```

### 3.2.2   Step 2: Variable declarations and uses

▶   As a next step, extend your parser to support variable declarations and uses, and also assignments and integer literals. Use the classes `IdDecl` and `IdUse` to distinguish between declarations and uses of identifiers in the abstract grammar. Here is an example test program:

```
int a() {
    int x;
    int y;
    x = 5;
    y = x;
}
```

### 3.2.3   Step 3: Expressions

▶   Expressions is usually the trickiest thing to get right in a parser. The abstract grammar should be simple, but the parser needs to deal with priorities and associativity. Extend your parser to support all the different kinds of expressions. First write down some simple test programs that will make it possible for you to check that you get everything right. Then try to make them work. It is a good idea to break this problem down into several subproblems. For example, start by making + and * work. Then add the other operators.

### 3.2.4   Step 4: Complete the language

▶   Complete the parser by adding new test cases and extending the specification. Remember to work in small steps: add one or a few language constructs at a time and make them work, before going on with the next step.

### 3.2.5   Step 5: Look over your test cases and specifications

▶   Look over your test cases and specifications. There should be test cases covering all parts of the syntax, and test cases covering tricky interactions between constructs (like priorities, etc.). There should also be negative test cases that check that the parser gives errors for syntactically incorrect input programs. The gcd program in section 3.1 should be one of the test cases. Make sure also that your abstract grammar is simple and well designed with good names for the different constructs. It is important to get everything correct and simple, as the following assignments will build on this one.