University of Glasgow | School of Computing Science

Programming Languages H

Familiarization with ANTLR and Fun

Notes:

Please keep in mind the following items.

- Download the antlr.jar from Moodle.
- Do **NOT** copy-paste the commands shown below, but rather type them yourselves; often there is a problem with the dash '–' when doing so.
- If using Windows, **replace** ':' with ';' in the commands below.

Familiarization with ANTLR: the Calc running example

You should see the following files in the `src/calc` directory. Study them to understand what they define.

- `Calc.g4` contains the grammar of Calc, expressed in ANTLR notation.
- `ExecVisitor.java` defines a depth-first, left-to-right traversal of the syntax tree of a Calc program. Each method defines what will be done at one kind of node, and returns the value of the corresponding expression (or returns 0 if the node is a command).
- `CalcRun.java` is the top-level driver program, defining a `main` method. It expects an argument that is a named Calc source file. `CalcRun` creates a lexer and uses it to translate the source code into a token stream. Then it creates a parser and calls the parser's `prog()` method, which in turn calls `com()`, `expr()`, etc. The result of `prog()` is the syntax tree of the input source code. Finally, a visitor is created and its `visit` method is called on the syntax tree in order to execute the Calc program.

To make ANTLR generate a lexer and parser for Calc, enter the following commands

(**download antlr.jar from Moodle**):

```
$ java -jar antlr.jar -no-listener -visitor src/ast/Calc.g4
$ javac -cp "antlr.jar" -d bin/ -sourcepath src/ src/calc/CalcRun.java
```

To run `CalcRun` with a selected source file:

```
$ java -cp "antlr.jar:bin" calc/CalcRun tests/test1.calc
16
56
72
```

These are numbers printed by the "`put`" commands in the source program.

Now try it with `tests/test2.calc`:

```
line 4:15 no viable alternative at character '/'
line 4:16 extraneous input '2' expecting EOL
```

The Calc parser prints an error message because this source program uses "`/`", but Calc has no such operator.

(*Note:* The generated parser's error messages are not very informative. However, they always include a line number and column number, such as "4:15", so you can locate the error exactly.)

Experiment by modifying the Calc grammar in `Calc.g4`. Try at least one of the following:

(a) **Add comments** (choosing your preferred syntax, such as "`/*…*/`" or "`//…`"). To do this, add a line in `Calc.g4` after the line defining `SPACE`, which defines the syntax of comments and then specifies `-> skip` to say that they should be ignored.

(b) **Add a "/" operator**. You will also need to modify `ExecVisitor.java`. Find the place where the behaviour of the other operators is defined and extend it.

(c) **Allow variable identifiers to consist of *one or more* letters** (instead of just a single letter). If you make this change in the grammar and then compile the calculator, you will find that variables are only distinguished by the first letter of their name. To make them work properly, you can go into `ExecVisitor.java` and replace `int[] store = new int[26];` by a `HashMap` that will be indexed by the full variable names. You will need to make corresponding changes in the `visitSet` and `visitId` methods, where the store is used.

Familiarization with the Fun compiler

Download `Fun.zip` from Moodle and unzip it. Study the following files:

- `Fun.g4` defines the grammar of Fun.
- `FunCheckerVisitor.java` implements a visitor that will traverse a parse tree and enforce Fun's *scope rules* and *type rules*.
- `FunEncoderVisitor.java` implements another visitor that will traverse a parse tree and generate *SVM object code*.
- `SVM.java` defines the representation of SVM instructions. It also contains a group of methods for emitting SVM instructions, i.e., placing them one by one in the code store; these methods are called by the Fun code generator. This class also contains a method `interpret()` that interprets the program in the code store.
- `FunRun.java` is the driver program. It first compiles a named Fun source file to SVM object code. The program also prints the AST and the SVM object code. Finally (if compilation was successful) the program interprets the object code. There are two other driver programs: `FunParse.java` just does syntactic analysis (*parsing*), and `FunCheck.java` does syntactic analysis and contextual analysis (*typechecking*).

To make ANTLR generate a lexer and parser for Fun, enter the following commands:

```
$ java -jar antlr.jar -no-listener -visitor src/ast/Fun.g4
$ javac -cp "antlr.jar" -d bin/ -sourcepath src/ src/fun/FunRun.java
```

You will find several Fun test programs in the directory `tests`. Run the driver program with a selected source file with the following command:

```
$ java -cp "antlr.jar:bin" fun/FunRun tests/assign.fun
```

To make the interpreter print each instruction as it is executed, in `FunRun.java`, change the static variable `tracing` from `false` to `true`.

Note: On the Moodle page you can find a link to Calc and Fun from command line, step by step, as screenshots. Both Calc and Fun have been tested, as described above, by using the antlr.jar that you can download from Moodle.

Warm-up: extending Fun to allow multiple procedure/function parameters

In the Fun language, procedures and functions have either *no* parameters or *one* parameter. In this warm-up exercise, you will extend Fun so that procedures and functions can have *any* number of parameters. Formal parameters (in procedure and function *definitions*) and actual parameters (in procedure and function *calls*) will be separated by commas. The warm-up exercise is in three stages, corresponding to the three stages of the assessed exercise. Each depends on some of the lecture material. You might be able to work ahead of the lectures by studying the Fun compiler, but it's OK to take the warm-up one stage at a time.

Warm-up stage 1 (depends on the lectures in week 5)

Download the file `Fun-multiple.g4` from the Moodle page. It contains a new version of the grammar. Look at this file and compare it with `Fun.g4`. There is a new non-terminal, `formal_decl_seq`, which is defined to be a sequence of one or more `formal_decl`, separated by commas. The optional tag (`?`) has moved from the definition of `formal_decl` into the definitions of `proc_decl` and `func_decl`. This means that the case of no parameters will be handled as a special case, and the general case is a non-empty sequence of parameters. It would be nice for the general case to be a sequence, empty or non-empty, of parameters, but the problem is that the comma only appears when we have at least two parameters.

Replace the content of `Fun.g4` with the content from `Fun-multiple.g4`. After building the compiler, you can parse (syntax check) `tests/multiple.fun` by running `FunParse`.

Warm-up stage 2 (depends on the lectures in week 6)

The next step is to extend the contextual analysis phase, which is defined in `FunCheckerVisitor.java`. The file `Type.java` already defines the class `Type.Sequence`, which represents a sequence of types; this class is not used yet, but the idea is to use it to represent the parameter types of a procedure or function. The same file also defines `Type.EMPTY`, representing an empty sequence of types.

Make the following changes to `FunCheckerVisitor.java`.

- In the method `predefine`, which defines the types of Fun's built-in procedures and functions, change the parameter type of `read` from `Type.VOID` to `Type.EMPTY`. Change the parameter type of `write` to be a `Type.Sequence` containing just `Type.INT` (you will have to do a little programming to construct this).
- Change the definition of `MAINTYPE` so that the parameter type is `Type.EMPTY`.
- In the methods `visitProc` and `visitFunc`, in the third line, instead of calling `ctx.formal_decl()`, call `ctx.formal_decl_seq()`. This is necessary to match the new grammar. The result type of this call is `FunParser.formal_decl_seqContext`. If it is null, meaning that there are no parameters, then the variable `t` should be set to `Type.EMPTY` instead of `Type.VOID`.
- Because we have added `formal_decl_seq` to the grammar, with the label `formalseq`, we need to add a method `visitFormalseq`. If you look in the file `FunBaseVisitor.java` you can see what the method header should be. The method needs to visit every item in `ctx.formal_decl()`, which has type `List<FunParser.Formal_declContext>`. Visiting an item returns a `Type`. These values need to be collected into an `ArrayList` and used to construct a `Type.Sequence`, which is returned.
- The method `visitFormal` can be simplified because the result of `ctx.type()` is never `null`. This is because the optional clause in the grammar is now `formal_decl_seq`, and if we have one, then it must be a non-empty sequence of declarations.
- `visitProccall` and `visitFunccall` need to be modified because `ctx.actual_seq()` might return `null`. In this case we construct an empty sequence of types; otherwise we visit the result of `ctx.actual_seq()` to get the sequence of types.

- Replace `visitActual` by `visitActualseq`, which needs to visit every item in `ctx.expr()` and construct a `Type.Sequence` of their types.

Now you should be able to typecheck `tests/multiple.fun` by running `FunCheck`.

Warm-up stage 3 (depends on the lectures in week 7)

Finally, a few changes are necessary in `FunEncoderVisitor.java`.

- In `visitProc` and `visitFunc`, replace `FunParser.formal_declContext` by `FunParser.formal_decl_seqContext`, and replace `ctx.formal_decl()` by `ctx.formal_decl_seq()`.
- Define the method `visitFormalSeq`; it just has to visit everything in `ctx.formal_decl()`.
- `visitFormal` can be simplified in the same way as in `FunCheckerVisitor`.
- In `visitProccall` and `visitFunccall`, use `ctx.actual_seq()` instead of `ctx.actual()`, but it might return `null`, so test for this. If it is `null` then there is no need to call `visit(ctx.actual_seq())`.
- Similarly to `FunCheckerVisitor`, replace `visitActual` by `visitActualseq`, which needs to visit every item in `ctx.expr()`.

Now you should be able to compile and run `tests/multiple.fun` by running `FunRun`.