Programming Languages (H) Assessed Coursework

Holly Hewitt
2463548H
14/03/2022

Status Report

The code I have written can be run on windows from the FunExtended directory using the following command. If using MacOS replace the semicolon with a colon.

javac -cp “antlr.jar” -d bin/ -sourcepath src/ src/fun/<prog>.java

java -cp “antlr.jar;bin” fun/<prog> tests/<test>

<prog> : FunParse, FunCheck or FunRun

<test> : any .fun file within the tests directory

Assignment stage 1: syntactic analysis

Extension A: the for-command

I made 2 extensions to the fun.g4 file to allow for syntactic analysis of the for-command. I added a for-command to the fun grammar as well as the addition of the keywords ‘for’ and ‘to’ to the fun lexicon.

I used ‘expr’ for expressions e1 and e2 of the for command, however upon reflection, whilst this grammar is general enough to allow for the correct expressions, it could be stricter so that e1 and e2 are only of type int. As the ‘expr’ within the fun grammar allows for types that are not int.

Extension B: the switch-command

I made 3 extensions to the fun.g4 file to allow for syntactic analysis of the switch-command. I added a switch command to the fun grammar, added a switch expression to the fun grammar, included the keywords ‘switch’, ‘case’ and ‘default’ in the fun lexicon, and added a new range operation to the lexicon.

The switch command allows for any number of cases and exactly one default case. I included switch_expr within the command to allow for a new expression which can include a range operation.

Looking closer at the switch_expr grammar, I allowed the switch expression to be either a primitive expression or a range of primitive expressions. Again, upon reflection whilst this is general enough to allow for the correct syntax, for the range option of the switch expression, the use of e1=NUM op RANGE e2 =NUM could be more appropriate for the enforcement of correct syntax.

Assignment stage 2: contextual analysis

Both extensions for stage 2 can be found at the bottom of the FunCheckerVisitor class.

Extension A: the for-command

Simple type checking is required in the contextual analysis stage for the for-command. I implemented this within the visitFor(ForContext) method in the FunCheckerVisitor class.

The method does the following:

- If the ID of the variable is not already in the type table (i.e. An int variable with ID hasn't been declared prior to the for command) define the type of ID as type int.
- Using the checkType method, check that both expressions e1 and e2 are of type int. Outputting errors if required.
- Visit the sequence of commands within the for-command such that the commands also undergo required contextual analysis.

Extension B: the switch-command

A bit more was required for the contextual analysis of the switch-command, in addition to implementing the required type checking, I also implemented the necessary check for guard overlap. This was done using VisitSwitch(SwitchContext), visitSwitch_expr(Switch_exprContext) and a private helper method checkForGuardOverlap(Switch_exprContext).

VisitSwitch(SwitchContext) calls the other two methods, thus I will summarize what the 3 methods do by outlining the workflow of VisitSwitch(SwitchContext)

- If the switch statement includes cases, iterate through a list of the case switch expressions.
- Check that all case switch expressions are of the same type. The method visitSwitch_expr is called to aid this. Again as with the contextual analysis of the for-command I made use of the checkType method.
- Using the helper method checkForGuardOverlap, by the end of the iteration, all guard values should be added to a list. checkForGuardOverlap includes logic to handle expressions including range, ensuring that all values within the range are added to the list.
- Check for duplicates within the guard list, if there are, output an error, as duplicate guards infer overlap or repetition.
- Visit the commands under all cases and the default case so that they undergo required contextual analysis.

Assignment stage 3: code generation

Both extensions for stage 3 can be found at the bottom of the FunEncoderVisitor class.

Extension A: the for-command

The code generation for the for-command is implemented within the visitFor method.

The method does the following I have simplified the code by showing which instruction forward jumps will go to, however within the method the use of conditional addresses and patching was employed:

1. Store ID in the address table
2. LOAD e1
3. LOAD e2
4. CMPGT
5. JUMPT (11)
6. Execute seq_com
7. LOAD e1
8. INC
9. STORE e1
10. JUMP (2)
11. Return null

Extension B: the switch-command

The code generation for the switch-command is implemented within two methods: visitSwitch(SwitchContext) and visitSwitch_expr(Switch_exprContxt), as with the contextual analysis, the visitSwitch method handles most of the logic for this implementation.

Let expr represent the expression being evaluated, let e1 represent the first expression within each switch expression and if e2 exists then it represents the final value within a range operation. VisitSwitch does the following:

- For each switch_expr within the switch-command
    - If the switch expression does not include a range operation
        - If (expr=e1), execute the command sequence below the switch_expr and return null
    - If the switch expression includes a range operation
        - If ((not) expr < e1) and ((not) expr > e1), execute the command sequence below the switch_expr and return null
- If this part of the code is reached, none of the switch expressions match the expression being evaluated, therefore execute the command sequence below the default statement

Below, I will illustrate how I executed this with the use of forward jumps. I have simplified the code by showing which instruction forward jumps will go to and by explicitly showing comparisons instead, however within the method the use of conditional addresses and patching was employed. I have also simplified the load instructions to allow for increased readability:

1. For each switch_expr within the switch command
    a. Int i is the index of the switch_expr in the list of switch expressions
    b. If switch_expr doesn't include a range operation
        i. LOAD expr

        ii. LOAD switch_expr.e1
        iii. CMPEQ
        iv. JUMPT (3)
    c. Else
        i. LOAD expr
        ii. LOAD switch_expr.e1
        iii. CMPLT
        iv. JUMPT (1.c.ix)
        v. LOAD expr
        vi. LOAD switch_expr.e2
        vii. CMPGT
        viii. JUMPF (3)
        ix. continue
2. Jump to 5
3. Visit (Seq_com(i))
4. Return null
5. Visit default seq_com
6. Return null