PL Coursework Report

Name: Bin Zhang

Student ID: 2941833z

Stage 1: syntactic analysis

Extension A: repeat-until command

```
|   REPEAT COLON
      seq_com
    UNTIL expr DOT          # repeat
```

```
REPEAT  :  'repeat' ;
UNTIL   :  'until' ;
```

Similar to the format of while command, having a COLON symbol before internal commands, and end with DOT symbol. In addition, add syntax lexicon: REPEAT and UNTIL.

Extension B: switch command

```
|   SWITCH expr
      case_seq
      default_case DOT        # switch
```

```
SWITCH  :  'switch' ;
CASE    :  'case' ;
DEFAULT :  'default' ;
```

```
case_seq
    :   case_com*                 # caseseq
    ;

case_com
    :   CASE (NUM | TRUE | FALSE | range) COLON seq_com      # case
    ;

range
    :   NUM RANGE NUM            # rangeofint
    ;

default_case
    :   DEFAULT COLON seq_com    # defaultcase
    ;
```

```
RANGE   :  '..' ;
```

Added four new syntax modules, for representing cases and the integer range format, and still using DOT to indicate the end of switch command. In addition, add syntax lexicon: SWITCH, CASE, DEFAULT and RANGE.

Stage 2: contextual analysis

Extension A: repeat-until command

visitRepeat(FunParser.RepeatContext ctx):

Similar to inversion of the context analysis of the while command, firstly visit and check the internal commands and then check and evaluate the expression and check if its type is bool, finally return null just like other command.

Extension B: switch command

```
private static class Pair<T1, T2> {
    final T1 key;
    final T2 value;

    Pair(T1 key, T2 value) {
        this.key = key;
        this.value = value;
    }

    // Getter method
    public T1 getKey() { return key; }
    public T2 getValue() { return value; }
}

Type switchType;
private Set<Integer> intCases = new HashSet<>();
private Set<Boolean> boolCases = new HashSet<>();
private List<Pair<Integer, Integer>> rangeCases = new ArrayList<>();
```

Add a structure Pair for recording the start and end value for one integer range. Add switchType variable to record the type of the expression after switch word. Add intCases and boolCases to record all int or bool case value for one switch command and use HashSet to help check if there are duplicate case values. Add rangeCases to store all integer range for checking if overlapping occurred.

1. visitSwitch(FunParser.SwitchContext ctx):

Visit the whole switch command and all the created data structure above for reusing, and check the expression type.

2. visitCaseseq(FunParser.CaseseqContext ctx):

Visit all cases by two directions: 1. firstly, check type and duplicate for all cases for

one switch command 2. Visit all internal commands for cases one by one.

3. visitCase(FunParser.CaseContext ctx) and visitRangeofint(FunParser.RangeofintContext ctx):

Check the type matching and duplicate values for cases.

4. visitCasecom(FunParser.CaseContext ctx):

New method added just to visit all internal commands for one case.

5. visitDefaultcase(FunParser.DefaultcaseContext ctx):

Visit the default case for one switch command.

Note: Reason for adding visitCasecom() but not just using visitCase() is for dealing with the contextual analysis for nested switch commands, and in this way we can check the type and duplicate for all cases for one switch command before visit the internal commands to avoid confusing case values of different switches.

Stage 3: code generation

Extension A: repeat-until command

Similar to contextual analysis' logic, firstly, record the start address and generate the code for internal commands, then generate the code for expression and using a JUMPF to deal with the condition checking for the loop. Finally return null just like other command.

Extension B: switch command

```
List<Integer> caseJumpAddresses;
FunParser.SwitchContext ctxcpy;
```

Add a List for recording the JUMP instructions' address for all cases, and use it backfill the address of the end of switch to JUMP after generating all other code. And add an global var to clone the SwitchContext instance ctx.

1. visitSwitch(FunParser.SwitchContext ctx):

The top-level method for switch command code generation, update the address list created above first then generate code for each case and then default case, finally backfill the JUMP address and return null.

2. visitCaseseq(FunParser.CaseseqContext ctx):

For generating code for all cases on by one.

3. visitCase(FunParser.CaseContext ctx):

Generate all code for one case by checking if the case value is int, bool, or integer range. The logic is firstly generating the code for expression each time and then getting the case value and compare it to the expression value after switch word, then add a JUMPF (or JUMPT for range case) and wait for backfilling the next case's address to it later. Then generate the code for internal commands and add a JUMP and add its address to the address list caseJumpAddresses and wait for backfill the switch-ending address to it.

4. visitRangeofint(FunParser.RangeofintContext ctx):

Do not use this method because the code for range cases just be handled in visitCase(). So just return null for successfully compile.

5. visitDefaultcase(FunParser.DefaultcaseContext ctx):

Visit the internal commands and generate code for default case.

Note: My implementation for switch's code generation unable to deal with nested switch command and some special situation, but for most conventional situations, it can output the correct results.

Test file:

For repeat-until command: 3 fail tests for contextual analysis, 3 running tests

For switch command: 7 fail tests for contextual analysis, 5 running tests