Aflevering 1 - Programmeringssprog

Exercise (1).

Extend the implementation of Unparser.scala (see exercise 10) and the "trace" mechanism (see exercise 9) with the new language features (variable declarations, variable uses, and blocks). Then explain briefly how the rules in the operational semantics for block expressions and val declarations (slide 41) relate to the implementation in Interpreter.scala. Finally, explain the output of running MiniScala with arguments -unparse -run -trace examples/ex21.s

• • •

Solution. Lets start by extending the unparse function, as stated we have three new language features that we need to cover. These are variables (`VarExp`), value declarations (`ValDecl`) and blocks (`BlockExp`). We will show the code for each in turn.

```
case VarExp(x) => s"${x}"
```

Variables are just letters, so we simply return the value `x` in `VarExp(x)`and convert it to a string.

```
case ValDecl(v_name, exp) => s"val ${vd} = " + unparse(exp) + "; "
```

Value declarations consist of a variable name `v_name`, and an expression `exp` which is assigned to the variable. The syntax required by the parser is of the form, `val var = exp;` Since `exp` can be a number of different expressions, we evaluate this recursively by calling `unparse(exp)`.

```
case BlockExp(lists, rightexp) =>
  val rightval = unparse(rightexp)
  def unparseList(list: List[ValDecl]): String = list match {
    // matches on any list(c1, ..., cn) if n>=1
    case c :: cs => unparse(c) + unparseList(cs)
    // handles the case where cs was the empty list
    case Nil => ""
  }
```

Block expressions contain two types; lists (made up of value declarations) and expressions. The expression is handled recursively, while the list is handled by a helper function `unparseList()`. Whenever the list has at least one element (`case c :: cs`), matches the expression, splitting it into its first element, followed by the remainder of the list. The first element is then unparsed, and the function is called on the remainder. When the remainder is the empty list, the second case is matched, and the function terminates.

Exercise (2)**.**

Write a function simplify(exp: Exp): Exp that simplifies MiniScala v2 expressions. (Performing such simplifications is common in optimizing compilers.) You might consider the simplifications on slide 33 plus any others you find useful and correct. (Hint: use pattern matching!) Write a few tests to check that your function works as intended.

• • •

Solution.

Exercise (3)**.**

Show how to rewrite VarEnv (slide 35) to use "object-oriented style" (with extend and lookup being methods in the classes, not using pattern matching, and not using the scala.collection standard library). Use a fresh .scala file for this exercise - you don't need to modify your implementation of the MiniScala interpreter to use the new definition of VarEnv. As always, it is a good idea to write a few tests to check that your code works as intended

• • •

Solution.