

Problem Set 3

Semester 1, 2012/13

Due: October 7, 23:59

Marks: 16

Submission: In IVLE, in the cs4212 workbin, you will find a folder called “Homework submissions”. In that folder, you will find **4 new subfolders: PS3P01, ..., PS3P04**. The last two digits of the folder name indicate the solution that is to be submitted into that folder: the solution to *Question 1* into **PS3P01**, and so on (that is, you need to submit 4 separate solutions to 4 problems). A solution should consist of a *single text file* that can be loaded into the SWI-Prolog environment and executed. You should provide as much supplementary information about your solution as you can, *in the form of Prolog comments*.

The authors’ *names and matriculation numbers should appear clearly in a comment* at the top of the file. Teams may have at most 2 members. Each team should have a single submission for each problem; there is *no need for multiple submissions of the same solution* from each of the contributors. Please remember that for team submissions, each team member receives 75% of the marks awarded to the solution.

Problem 1 [2 marks, submit to PS3P01]

Implement a `do...while` statement in the toy language of `compiler_non_scoped_while_lang.pro`.

Problem 2 [4 marks, submit to PS3P02]

The objective of this problem is to implement a *computed goto* statement in the toy language of `compiler_non_scoped_while_lang.pro`. To make this work, we devise a new kind of label, with the following syntax:

```
indexlabel(N) :: Statement
```

Here, `N` is a positive integer, ranging between 0–255, and `Statement` is a (possibly compound) statement. The computed `goto` statement then has the syntax

```
goto Expr
```

where `Expr` is an expression expected to evaluate to a positive integer between 0–255 (it’s acceptable if the generated program crashes when `Expr` is out of range, or a matching

`indexlabel` does not exist). Thus the following should be a correct program in your compiler.

```
                a = 1 ; % may be changed to 2
                goto 2*a ;
indexlabel(2) :: b = 10 ;
                goto exit ;
indexlabel(4) :: b = 20 ;
exit ::         a = 0
```

Hints: Create a table of labels similar to the one that is created for “switch-case” statements. When translating the `goto Expr` statement, simply assume that the table exists (pick one, global name for it), and perform an indirect jump to the target contained in `table[val]`, where `val` is the value of `Expr`. In translating a `indexlabel(N)`, generate a label, and store the label name in an associative array (store the associative array in one of the attributes). At the end of compiling the program, create the label table from the accumulated associative array.

Problem 3 [6 marks, submit to PS3P03]

Modern compilers often optimize their generated code by performing loop unrolling. That is, they transform a `while` loop of the form:

```
while ( B ) do {
    S
}
```

into a code fragment of the form:

```
if ( B ) then {
    do {
        S ;
        if ( B ) then {
            S ;
            if ( B ) then {
                S ;
                .....
                if ( B ) then {
                    S ;
                } ..... } while ( B )
            }
        }
    }
}
```

} N copies of S

The following translation scheme can be applied to code of this form:

```
Compiled_B
jump_if_not_true L_WhileExit
L_WhileTop:
    Compiled_S
    Compiled_B
    jump_if_not_true L_WhileExit
    Compiled_S
    ... ..
    Compiled_B
    jump_if_true L_WhileTop
L_WhileExit:
```

} N copies of S

This code, when translated into assembly language, has the advantage of performing much fewer jumps than the code obtained from the original translation scheme. This reduces branch penalty at the expense of code size (however since memory is very inexpensive nowadays, an increase in code size is much more acceptable than an increase in execution time).

Modify the compiler in `compiler_scoped_while_lang.pro` so that it unrolls every while loop, by replicating its body N times. Store the parameter N in a new attribute, `unroll_count`, which you can set in the top-level compile predicate. The initial value of

your new attribute should be provided as an extra argument to your top-level predicate.

Note 1: pay attention to the way you implement local variables, they have to continue to work in the same way (the translation scheme given above is a bit sloppy with regard to the translation of local variables, you will have to fix it so that it works as expected). You should not incur extra memory storage for variables in the new translation scheme (i.e. the maximum stack space computed for the variable storage should not be different as compared to the case when loop unrolling is not performed).

Note 2: `break` and `continue` statements must continue to work as before.

Note 3: the replication of program fragment `S` may lead to duplication of target labels, which is not allowed at the assembly language level. There are two ways to avoid this. The naive solution will re-compile `S` from scratch every time a new copy of `Compiled_S` needs to be laid out. The efficient solution will extract a code template from `Compiled_S`, whereby the “local” label placeholders have not yet been bound. Then, instead of recompiling `S` multiple times, it will produce new instances of `Compiled_S` using `copy_term` (check the documentation to find out how it works), and lay out these new instances.

Problem 4 [4 marks, submit to PS3P04]

In this question, we consider new variants of the `break` and `continue` statements. They have the following format:

```
break N
```

and

```
continue N
```

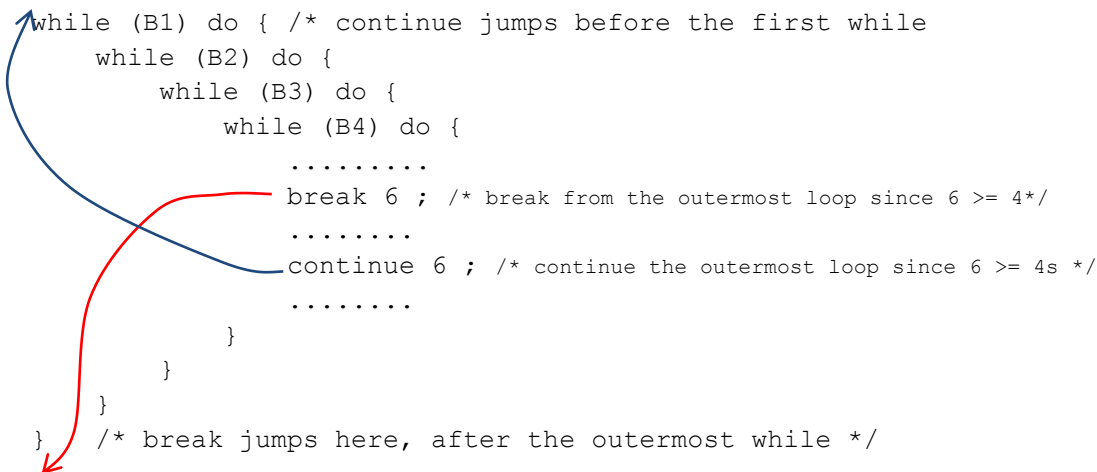
These statements would be useful when we need to alter the control flow of nested `while` loops, by allowing the `break/continue` statement to refer to the loop that is `N` levels of nesting away from the statement. Let us clarify the semantics of these statements by means of two examples.

Example 1

```
while (B1) do {
  while (B2) do {
    while (B3) do { /* continue will jump here, before the 3rd while */
      while (B4) do {
        .....
        break 2 ; /* break from the loop that is 2 levels of nesting away */
        .....
        continue 2 ; /* continue the loop that is 2 levels of nesting away */
        .....
      }
    } /* the break will jump here, after the second closing brace */
  }
}
```

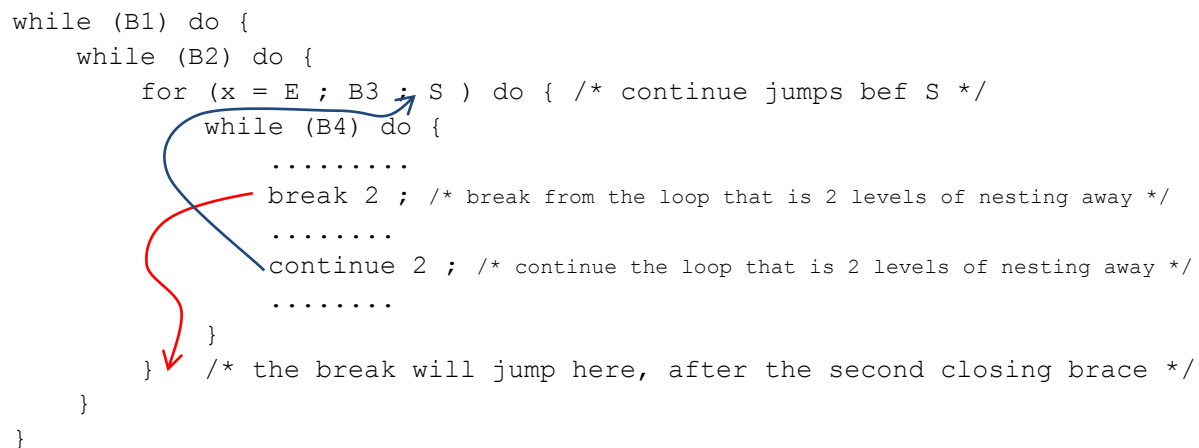
Example 2

```
while (B1) do { /* continue jumps before the first while
    while (B2) do {
        while (B3) do {
            while (B4) do {
                .....
                break 6 ; /* break from the outermost loop since 6 >= 4*/
                .....
                continue 6 ; /* continue the outermost loop since 6 >= 4s */
                .....
            }
        }
    }
} /* break jumps here, after the outermost while */
```



Example 3

```
while (B1) do {
    while (B2) do {
        for (x = E ; B3 ; S ) do { /* continue jumps bef S */
            while (B4) do {
                .....
                break 2 ; /* break from the loop that is 2 levels of nesting away */
                .....
                continue 2 ; /* continue the loop that is 2 levels of nesting away */
                .....
            }
        } /* the break will jump here, after the second closing brace */
    }
}
```



Example 1 showcases the most typical use of the new `break` and `continue` statements. With an argument of 2, they refer to the statement that is 2 levels of nesting away. In total, there are 4 `while` loops, and the selected loop is the third one from the top. The `break` statement jumps right outside this loop, as indicated by the red arrow. On the other hand, the `continue` statement jumps right before this loop, so that the loop test can be performed next. This behaviour is depicted by the blue arrow.

Example 2 showcases a limit case, when the `break/continue` argument is greater or equal to the number of levels of nesting of the `break/continue` statement. In this case, we shall consider that the numeric argument refers to the outermost loop, and thus the two statements will behave as if the numeric argument had the value 4.

Example 3 showcases the behaviour of the 2 statements in the case when the selected statement is a `for` loop, instead of a `while`. In this case, the `break` statement has the same behaviour, completely jumping outside of this loop. However, the `continue`

statement will transfer the control to the point right before the “increment” (third) statement appearing in the `for` loop’s round brackets. Executing this statement is likely to be essential for the termination of the loop.

It is probably obvious by now that whenever the numeric argument is 1, the `break/continue` statement will behave as if the argument was missing (i.e. as a regular `break/continue` statement, as implemented during the recitation session).

The objective of this problem is to add the `break N` and `continue N` statements to our toy language, and augment the compiler of `compiler_scoped_while_lang.pro` so that it generates code for them. In your implementation, you will have to change the type of the `break` and `continue` attributes from a simple label, to a pair `(label, N)`, where `N` is the number of levels till the destination. As you are about to leave each loop, first test if the `break/continue` attribute has a value different than `none`. If it does, check if the value of `N` is 1, and if it is, then this is the place to lay out the `break/continue` label. However, if `N` is greater than 1, change the value of the attribute to `(label, N-1)`, and pass the attribute back to the level above. Keep in mind that there is a chance that `N` may be larger than the number of levels of nesting. Thus, you will have to figure out a way to detect that you have reached the outermost loop, and lay out the `break/continue` label there, even though `N` may still be greater than 1.