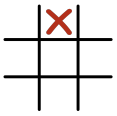
Programming 2 (SS 2023)
Saarland University
Faculty MI
Compiler Design Lab

Sample Solution 12
Expression Project & Compiler

Prof. Dr. Sebastian Hack
Pascal Lauer, M. Sc.
Marcel Ullrich, B. Sc.

The calendar indicates which script chapters you should study in conjuction with each lecture. The exercises are designed to enhance your understanding of the lecture material and prepare you for the mini-tests and the final exam. Additional exercises can be found at the end of each chapter in the script.
The difficulty of an exercise on the sheet is determined by the number of annotated 'X' and 'O' marks in the tic-tac-toe field, with four levels (1-4) increasing by one mark per level.

Exercise 12.1: Syntax-Directed Code Generation



The goal is to generate MIPS code for each of the following C0 codes. To achieve this, build the inference tree from the code generation rules and then write down the code sequence. Take $offs = \{x \mapsto 0, y \mapsto 4\}$ as the offset environment.

1. `x + 1` as an expression
2. `if (x > 0) x = 1; else x = y;` as a program
3. `y = (*x) + 1;` as a statement
4. `{int x; int y; y = 3; x = y;}` as a program

Solution

1. The derivation tree for the expression `x + 1`

$$\dfrac{\dfrac{}{[\text{CVar}]}\ offs\ x = 0}{codeL\ offs\ (\$t0 :: \$t1 :: rs)\ [\![x]\!]\ int = \texttt{\# c\_var}\ \texttt{addiu \$t0 \$sp 0}}$$

$$\dfrac{\dfrac{}{[\text{CVar}]}\ offs\ x = 0}{[\text{CLtoR}]\ codeR\ offs\ (\$t0 :: \$t1 :: rs)\ [\![x]\!]\ int = \texttt{\# c\_LtoR}\ \texttt{\# c\_var}\ \texttt{addiu \$t0 \$sp 0}\ \texttt{lw \$t0 (\$t0)}} \qquad \dfrac{}{[\text{CConst}]}\ codeR\ offs\ (\$t1 :: rs)\ [\![1]\!]\ int = \texttt{\# c\_const}\ \texttt{li \$t1 1}$$

$$\dfrac{\dfrac{\dfrac{}{[\text{CVar}]}\ offs\ x = 0}{[\text{CLtoR}]\ codeR\ offs\ (\$t0 :: \$t1 :: rs)\ [\![x]\!]\ int = \texttt{\# c\_LtoR}\ \texttt{\# c\_var}\ \texttt{addiu \$t0 \$sp 0}\ \texttt{lw \$t0 (\$t0)}} \qquad \dfrac{}{[\text{CConst}]}\ codeR\ offs\ (\$t1 :: rs)\ [\![1]\!]\ int = \texttt{\# c\_const}\ \texttt{li \$t1 1}}{[\text{CBinary}]}$$

$$codeR\ offs\ (\$t0 :: \$t1 :: rs)\ [\![x\ +\ 1]\!]\ int =$$
```
# c_Binary
# c_LtoR
# c_var
addiu $t0 $sp 0
lw $t0 ($t0)
# c_const
li $t1 1
add $t0 $t0 $t1
```

2. The derivation tree for the program `if (x > 0) x = 1; else x = y;`

[CIf] Subtree 1 Subtree 2 Subtree 3

CodeS offs ⟦`if (x > 0) x = 1; else x = y;`⟧ =

```
addiu $t0 $sp 0
lw $t0 $t0
li $t1 0
slt $t0 $t1 $t0

beqz $t0 F

addiu $t0 $sp 0
li $t1 1
sw $t1 ($t0)

b N
F:
addiu $t0 $sp 0
addiu $t1 $sp 4
lw $t1 $t1
sw $t1 ($t0)
N:
```

[CTerm] $\dfrac{}{\textit{CodeP offs } [\![\varepsilon]\!] = c_{Term}}$

[CSeq]

CodeP offs ⟦`if (x > 0) x = 1; else x = y;`⟧ =

```
addiu $t0 $sp 0
lw $t0 $t0
li $t1 0
slt $t0 $t1 $t0

beqz $t0 F

addiu $t0 $sp 0
li $t1 1
sw $t1 ($t0)

b N
F:
addiu $t0 $sp 0
addiu $t1 $sp 4
lw $t1 $t1
sw $t1 ($t0)
N:
```

Subtree 1

[CVar] $\dfrac{\textit{offs } x = 0}{\textit{CodeL offs } (\$t0 :: \$t1 :: rs)\ [\![x]\!]\ \texttt{int} =}$

```
# c_var
addiu $t0 $sp 0
```

[CLToR]

CodeR offs ($t0 :: $t1 :: *rs*) ⟦x⟧ int =

```
# c_LToR
addiu $t0 $sp 0
lw $t0 $t0
```

[CConst] *CodeR offs* ($t1 :: *rs*) ⟦0⟧ int =

```
# c_Const
li $t1 0
```

[CBinary]

CodeR offs ($t0 :: $t1 :: *rs*) ⟦x>0⟧ int =

```
# c_Binary
addiu $t0 $sp 0
lw $t0 $t0
li $t1 0
slt $t0 $t1 $t0
```

Subtree 2

[CVar] $\dfrac{\textit{offs } x = 0}{\textit{CodeL offs } (\$t0 :: \$t1 :: rs)\ [\![x]\!]\ \texttt{int} =}$

```
# c_var
addiu $t0 $sp 0
```

[CConst] *CodeR offs* ($t1 :: *rs*) ⟦1⟧ int =

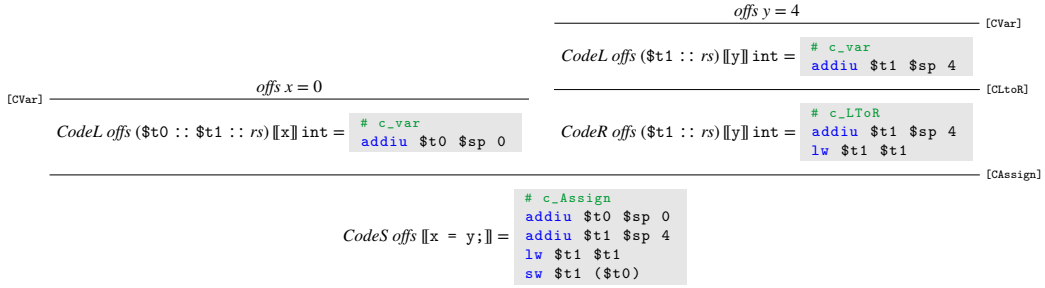
```
# c_Const
li $t1 1
```

[CAssign]

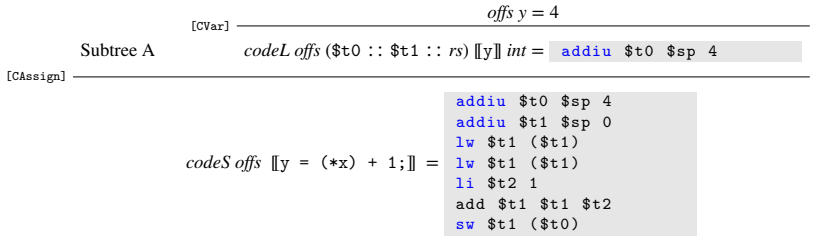
CodeS offs ⟦`x = 1;`⟧ =

```
# c_Assign
addiu $t0 $sp 0
li $t1 1
sw $t1 ($t0)
```

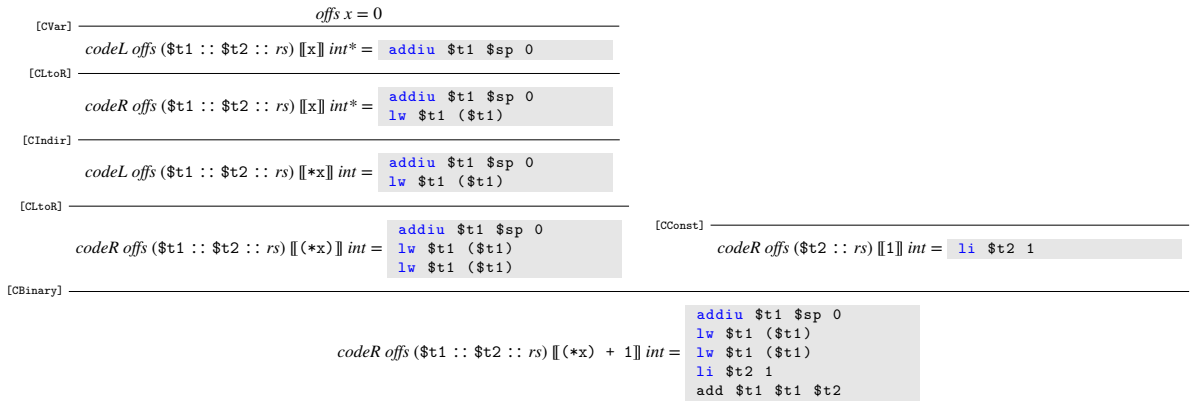
Subtree 3



3. The derivation tree for the statement `y = (*x) + 1;`



Subtree A:



4. The derivation tree for the program `{int x; int y; y = 3; x = y;}`

$$\dfrac{\dfrac{\text{Subtree 1} \quad \textit{CodeP offs'} \ [\![\texttt{x = y;}]\!] = \texttt{addiu \$t0 \$sp 8}\ \texttt{addiu \$t1 \$sp 12}\ \texttt{lw \$t1 (\$t1)}\ \texttt{sw \$t1 (\$t0)}}{[\text{CSeq}]} \quad \dfrac{\dfrac{\text{Subtree 2} \quad \textit{CodeP offs'} \ [\![\varepsilon]\!] = c_{Term}}{[\text{CTerm}]}}{[\text{CSeq}]}}{[\text{CSeq}]}$$

$$\dfrac{\textit{offs'} = \textit{offs}[x \mapsto 8, y \mapsto 12] \quad \textit{CodeP offs'} \ [\![\texttt{y = 3; x = y;}]\!] = \texttt{addiu \$t0 \$sp 12}\ \texttt{li \$t0 3}\ \texttt{sw \$t1 (\$t0)}\ \texttt{addiu \$t0 \$sp 8}\ \texttt{addiu \$t1 \$sp 12}\ \texttt{lw \$t1 (\$t1)}\ \texttt{sw \$t1 (\$t0)}}{[\text{CBlock}]}$$

$$\dfrac{\textit{CodeS offs} \ [\![\{\texttt{int x; int y; y = 3; x = y;}\}]\!] = \texttt{addiu \$t0 \$sp 12}\ \texttt{li \$t0 3}\ \texttt{sw \$t1 (\$t0)}\ \texttt{addiu \$t0 \$sp 8}\ \texttt{addiu \$t1 \$sp 12}\ \texttt{lw \$t1 (\$t1)}\ \texttt{sw \$t1 (\$t0)} \quad \dfrac{}{\textit{CodeP offs} \ [\![\varepsilon]\!] = c_{Term}}[\text{CTerm}]}{[\text{CSeq}]}$$

$$\textit{CodeP offs} \ [\![\{\texttt{int x; int y; y = 3; x = y;}\}]\!] = \texttt{addiu \$t0 \$sp 12}\ \texttt{li \$t0 3}\ \texttt{sw \$t1 (\$t0)}\ \texttt{addiu \$t0 \$sp 8}\ \texttt{addiu \$t1 \$sp 12}\ \texttt{lw \$t1 (\$t1)}\ \texttt{sw \$t1 (\$t0)}$$

Subtree 1:

$$\dfrac{\dfrac{\textit{offs'} \ y = 12}{[\text{CVar}]} \quad \dfrac{}{[\text{CConst}]}}{[\text{CAssign}]}$$

$$\textit{CodeL offs'} \ (\$t0 :: \$t1 :: rs) \ [\![\texttt{y}]\!] \ \texttt{int} = \texttt{addiu \$t0 \$sp 12} \qquad \textit{CodeR offs'} \ (\$t1 :: rs) \ [\![3]\!] \ \texttt{int} = \texttt{li \$t0 3}$$

$$\textit{CodeS offs'} \ [\![\texttt{y = 3;}]\!] = \texttt{addiu \$t0 \$sp 12}\ \texttt{li \$t0 3}\ \texttt{sw \$t1 (\$t0)}$$

Subtree 2:

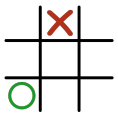
$$\dfrac{\dfrac{\dfrac{\textit{offs'} \ x = 8}{[\text{CVar}]} \quad \dfrac{\dfrac{\textit{offs'} \ y = 12}{[\text{CVar}]}}{[\text{CLtoR}]}}{[\text{CAssign}]}$$

$$\textit{CodeL offs'} \ (\$t0 :: \$t1 :: rs) \ [\![\texttt{x}]\!] \ \texttt{int} = \texttt{addiu \$t0 \$sp 8} \qquad \textit{CodeL offs'} \ (\$t1 :: rs) \ [\![\texttt{y}]\!] \ \texttt{int} = \texttt{addiu \$t1 \$sp 12}$$

$$\textit{CodeR offs'} \ (\$t1 :: rs) \ [\![\texttt{y}]\!] \ \texttt{int} = \texttt{addiu \$t1 \$sp 12}\ \texttt{lw \$t1 (\$t1)}$$

$$\textit{CodeS offs'} \ [\![\texttt{x = y;}]\!] = \texttt{addiu \$t0 \$sp 8}\ \texttt{addiu \$t1 \$sp 12}\ \texttt{lw \$t1 (\$t1)}\ \texttt{sw \$t1 (\$t0)}$$

Exercise 12.2: Code Generation



Generate MIPS code for the following C0 program. For that, draw an inference tree using the code generation derivation rules and give the MIPS code in the end. Use this offset function: $\textit{offs} = \{x \mapsto 0, y \mapsto 4, z \mapsto 8\}$.

```
while (1) x = *y + 2;
```

Solution

$$\dfrac{\dfrac{\dfrac{}{\textit{codeR offs} \ \$t0 :: \$t1 :: rs \ [\![1]\!] \ \texttt{int} = c_{const,1}}[\text{CConst}] \quad \dfrac{\dfrac{\textit{offs} \ x = 0}{\textit{codeL offs} \ \$t0 :: \$t1 :: \$t2 :: rs \ [\![x]\!] \ \texttt{int} = c_{var,1}}[\text{CVar}] \quad \textit{Helper} \quad S = \texttt{sw}}{\textit{codeS offs} \ [\![x = *y + 2;]\!] = c_{assign}}[\text{CAssign}]}{\textit{codeS offs} \ [\![\texttt{while(1)} \, x = *y + 2;]\!] = c}[\text{CWhile}] \quad \dfrac{}{\textit{codeP offs} \ [\![\epsilon]\!] = c_{term}}[\text{CTerm}]}{\textit{codeP offs} \ [\![\texttt{while(1)} \, x = *y + 2;]\!] = c}[\text{CSeq}]$$

Helper:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathit{offs}\ \mathtt{y} = 4}\,[\mathrm{CVar}]}{\mathit{codeL}\ \mathit{offs}\ \mathtt{\$t1 :: \$t2} :: rs\ [\![\mathtt{y}]\!]\ \mathtt{int*} = c_{var,2}}\ L = \mathtt{lw}}{\mathit{codeR}\ \mathit{offs}\ \mathtt{\$t1 :: \$t2} :: rs\ [\![\mathtt{y}]\!]\ \mathtt{int*} = c_{ltor,2}}\,[\mathrm{CLtoR}]}{\mathit{codeL}\ \mathit{offs}\ \mathtt{\$t1 :: \$t2} :: rs\ [\![*y]\!]\ \mathtt{int} = c_{ltor,2}}\,[\mathrm{CIndir}]}{\mathit{codeR}\ \mathit{offs}\ \mathtt{\$t1 :: \$t2} :: rs\ [\![*y]\!]\ \mathtt{int} = c_{ltor,1}}\,[\mathrm{CLtoR}]\quad L = \mathtt{lw} \qquad \dfrac{}{\mathit{codeR}\ \mathit{offs}\ \mathtt{\$t2} :: rs\ [\![2]\!]\ \mathtt{int} = c_{const,2}}\,[\mathrm{CConst}]}{\mathit{codeR}\ \mathit{offs}\ \mathtt{\$t1 :: \$t2} :: rs\ [\![*y+2]\!]\ \mathtt{int} = c_{binary}}\,[\mathrm{CBinary}]$$

$c_{const,1}$:

```
li $t0 1
```

$c_{const,2}$:

```
li $t2 2
```

$c_{var,1}$:

```
addiu $t0 $sp 0
```

$c_{var,2}$:

```
addiu $t1 $sp 4
```

$c_{ltor,2}$:

```
addiu $t1 $sp 4
lw $t1 $t1
```

$c_{ltor,1}$:

```
addiu $t1 $sp 4
lw $t1 $t1
lw $t1 $t1
```

c_{binary}:

```
addiu $t1 $sp 4
lw $t1 $t1
lw $t1 $t1
li $t2 2
addu $t1 $t1 $t2
```

c_{assign}:

```
addiu $t0 $sp 0
addiu $t1 $sp 4
lw $t1 $t1
lw $t1 $t1
li $t2 2
addu $t1 $t1 $t2
sw $t1 ($t0)
```

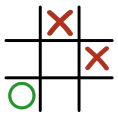
c_{term}:

```
# c_term
```

c:

```
 b  T
L:
    addiu $t0 $sp 0
    addiu $t1 $sp 4
    lw $t1 $t1
    lw $t1 $t1
    li $t2 2
    addu $t1 $t1 $t2
    sw $t1 ($t0)
T:
 li $t0 1
 bnez $t0 L
# c_term
```

Exercise 12.3:



Define an appropriate inference rule which can be used to generate code for a for-loop:

```
1 for (s_1; e_2; s_3)
2     s_4
```

In contrast to C, assume that s_1 and s_3 are statements.

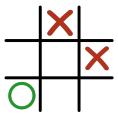
Solution

In the following, `T` and `L` must be fresh labels.

$$\text{[CFor]}\ \frac{\begin{array}{c} codeS\ offs\ [\![s_1]\!] = c_1 \quad codeR\ offs\ (r :: rs)\ [\![e_2]\!]\ k = c_2 \quad codeS\ offs\ [\![s_3]\!] = c_3 \\ codeS\ offs\ [\![s_4]\!] = c_4 \end{array}}{codeS\ offs\ [\![\texttt{for}\ (s_1;\ e_2;\ s_3)\ s_4]\!] = c_{for}}$$

```
1   #c_for
2
3   c_1
4   b     T
5 L:
6   c_4
7   c_3
8 T:
9   c_2
10  bnez r L
```

Exercise 12.4:



Define an appropriate inference rule that allows us to generate code for function calls `f(`e_1`, ..., `e_n`)`. You may assume that $n \leq 4$.

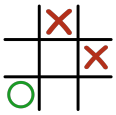
Solution

$$\text{[CCall]}\ \frac{\begin{array}{c} codeR\ offs\ (r_1 :: r_2 :: \ldots :: r_n :: rs)\ [\![e_1]\!] = c_1 \\ codeR\ offs\ (r_2 :: r_3 :: \ldots :: r_n :: rs)\ [\![e_2]\!] = c_2 \\ \vdots \\ codeR\ offs\ (r_n :: rs)\ [\![e_n]\!] = c_n \end{array}}{codeR\ offs\ (r_1 :: r_2 :: \ldots :: r_n :: rs)\ [\![\texttt{f}(e_1, e_2, \ldots, e_n)]\!] = c_{\mathsf{Call}}}$$

```
1 #c_Call
2
3 # code for argument expressions
4 c1
5 c2
6 ...
7 cn
8
9 # save caller-save registers
10 # here we save all, in an implementation we would keep track of which ones are
11 # used and only save those
12 addiu $sp $sp -52
```

```
13 sw     $t0 0($sp)
14 sw     $t1 4($sp)
15 ...
16 sw     $t9 36($sp)
17 sw     $v0 40($sp)
18 sw     $v1 44($sp)
19 sw     $ra 48($sp)
20
21 # pass function args
22 move   $a0 r1
23 move   $a1 r2
24 ...
25 move   $an-1 rn
26
27 # call function
28 jal    f
29
30 # move result into register r1 before restoring $v0
31 move   r1 $v0
32
33 # recover caller-save registers
34 # again, we would only restore those that were used and, in particular,
35 # not r1, since it contains the result
36 lw     $t0 0($sp)
37 lw     $t1 4($sp)
38 ...
39 lw     $t9 36($sp)
40 lw     $v0 40($sp)
41 lw     $v1 44($sp)
42 lw     $ra 48($sp)
43 addiu  $sp $sp 52
```

Exercise 12.5:



In this exercise, you should complete No Hau's unfinished project *Expressions*. The project implements a representation of simple arithmetic expressions. The corresponding `git` repo is:

`https://git.prog2.de/shared/exercise-repos/expression`

The English skeleton for the task is on the `main` branch of the given repository.
The documentation of the project provides the specification. Please proceed as follows:

1. Create classes that implement the expressions with binary operators $+, -, *, \div$ as well as the unary minus. These classes should inherit from `BinaryExpression` or `UnaryExpression`, respectively. Extend the classes with fitting constructors.

   Remark: To ensure that the tests work as expected, you should name the classes in the following way: `AddExpression`, `SubExpression`, `MulExpression`, `DivExpression`, `NegExpression`

2. Implement the method `public String toString()`, that returns the **abstract syntax** as a string of the expression, in the three abstract classes `UnaryExpression`, `ConstExpression` and `BinaryExpression`.

3. Implement `public boolean equals(Object o)` in appropriate classes. Make sure that the method considers expressions that evaluate to the same value as equal.

4. Complete the project by implementing `public int eval()` and add a `Main` class with a main method such that the project can be used to evaluate arbitrary expressions.

5. *(Bonus):* Create classes to support the additional operators $\wedge$(`&&`), $\vee$(`||`), $\neg$(`!`), $=$ (`==`) and $\oplus$(`^^`) (`AndExpression`, `OrExpression`, `NotExpression`, `EqualsExpression`, `XorExpression`). You will also have to extend the `Enum Operators`. For subtasks 1, 2 and 4 proceed as before. For subtask 3 you should think about whether a definition analogous to the arithmetic expressions is the right way to go. Before you start implementing this, think about what the best way to structure these classes would be.

   Remark: To make sure that a correctly implemented project without the bonus task still builds without errors, the tests pertaining to the bonus task are commented out. If you do implement the bonus task, you can uncomment the tests to let them run on your implementation.

Solution

The solution of the task is on branch `solution` of the given repository. The solution of the task with bonus is on branch `bonus`.
Remark: The full solution also implements a visitor pattern. Don't be confused by this, it is not necessary to solve the bonus task. It's mostly helpful to output the concrete syntax again in a visually appealing way.

Tip: You can generate a visual representation of the AST that the `main` function generates, by first running said `main` function and then executing the `build_ast.sh` script in the project directory to convert the generated `ast.dot` file into a PDF containing the AST. But note that this only works with the full solution contained in the `bonus` branch.