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

Sample Solution 11
C0t

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 11.1:

1. Take a look at the following C0p program and state the type environment Γ' after execution of the Block rule, starting with the environment $\Gamma = \{\}$.

```
{
    char c;
    char *pc;
    char **ppc;
    int i;
    int *pi;
    i = 42;
    ...
}
```

2. Derive the type of the following expressions with respect to the type environment Γ'.
   (a) `i - 42`
   (b) `*pc + i - 1337`
   (c) `**ppc - **&pc != 7198`

3. Revisit the expression from 2a. Which type must `i` have so that you cannot apply any derivation rules to the expression?

Solution

1. $\{\texttt{c} \mapsto \texttt{char}, \texttt{pc} \mapsto \texttt{char*}, \texttt{ppc} \mapsto \texttt{char**}, \texttt{i} \mapsto \texttt{int}, \texttt{pi} \mapsto \texttt{int*}\}$

2. (a)

$$[\text{TArith}]\dfrac{[\text{TVar}]\dfrac{\Gamma\ \texttt{i} = \textbf{int}}{\Gamma \vdash \texttt{i} : \textbf{int}} \qquad [\text{TConst}]\dfrac{-2^{31} \le 42 < 2^{31}}{\Gamma \vdash 42 : \textbf{int}}}{\Gamma \vdash \texttt{i - 42} : \textbf{int}}$$

   (b)

$$[\text{TArith}]\dfrac{[\text{TArith}]\dfrac{[\text{TIndir}]\dfrac{[\text{TVar}]\dfrac{\Gamma\ \texttt{pc} = \textbf{char*}}{\Gamma \vdash \texttt{pc} : \textbf{char*}}}{\Gamma \vdash \texttt{*pc} : \textbf{char}} \qquad [\text{TVar}]\dfrac{\Gamma\ \texttt{i} = \textbf{int}}{\Gamma \vdash \texttt{i} : \textbf{int}}}{\Gamma \vdash \texttt{*pc + i} : \textbf{int}} \qquad [\text{TConst}]\dfrac{-2^{31} \le 1337 < 2^{31}}{\Gamma \vdash 1337 : \textbf{int}}}{\Gamma \vdash \texttt{*pc + i - 1337} : \textbf{int}}$$

   (c)

$$[\text{TCmp}]\dfrac{\text{Help} \qquad [\text{TConst}]\dfrac{-2^{31} \le 7198 < 2^{31}}{\Gamma \vdash 7198 : \textbf{int}} \qquad \dfrac{}{\textbf{int} \leftrightarrow \textbf{int}}}{\Gamma \vdash \texttt{**ppc - **\&pc != 7198} : \textbf{int}}$$

Help:

$$\text{[TArith]}\ \dfrac{\text{[TIndir]}\ \dfrac{\text{[TIndir]}\ \dfrac{\text{[TVar]}\ \dfrac{\Gamma\ \texttt{ppc} = \textbf{char**}}{\Gamma \vdash \texttt{ppc} : \textbf{char**}}}{\Gamma \vdash \texttt{*ppc} : \textbf{char*}}}{\Gamma \vdash \texttt{**ppc} : \textbf{char}} \qquad \text{[TIndir]}\ \dfrac{\text{[TIndir]}\ \dfrac{\text{[TAddr]}\ \dfrac{\text{[TVar]}\ \dfrac{\Gamma\ \texttt{pc} = \textbf{char*}}{\Gamma \vdash \texttt{pc} : \textbf{char*}}}{\Gamma \vdash \texttt{\&pc} : \textbf{char**}}}{\Gamma \vdash \texttt{*\&pc} : \textbf{char*}}}{\Gamma \vdash \texttt{**\&pc} : \textbf{char}}}{\Gamma \vdash \texttt{**ppc - **\&pc} : \textbf{int}}$$

3. We can do arithmetic on integer types using the `[TArith]` rule and on pointer types using the `[TPtrArith]` rule. So the only type we cannot do arithmetic with is the `void` type.

Exercise 11.2:

Using the static semantics of C0t, check whether the given program contains type errors under the following type environment:
$\Gamma := \{\texttt{x} \mapsto \texttt{char}, \texttt{y} \mapsto \texttt{char*}, \texttt{z} \mapsto \texttt{int}, \texttt{p} \mapsto \texttt{int**}\}$

```
{
    x = *y + z;
    *p = &z;
    x = **p;
}
```

Solution

The program is well-typed under the given type environment.

$$\text{[TSeq]}\ \dfrac{\text{[TBlock]}\ \dfrac{\text{[TSeq]}\ \dfrac{(a) \qquad \text{[TSeqS]}\ \dfrac{(b) \qquad (c)}{\Gamma \vdash \texttt{*p = \&z; x = **p;}}}{\Gamma \vdash \texttt{x = *y + z; *p = \&z; x = **p;}}}{\Gamma \vdash \texttt{\{x = *y + z; *p = \&z; x = **p;\}}} \qquad \text{[TTerm]}\ \dfrac{}{\Gamma \vdash \varepsilon}}{\Gamma \vdash \texttt{\{x = *y + z; *p = \&z; x = **p;\}}}$$

a)

$$\text{[TAssign]}\ \dfrac{\text{[TArith]}\ \dfrac{\text{[TIndir]}\ \dfrac{\text{[TVar]}\ \dfrac{\Gamma\ \texttt{y} = \textbf{char*}}{\Gamma \vdash \texttt{y} : \textbf{char*}}}{\Gamma \vdash \texttt{*y} : \textbf{char}} \qquad \text{[TVar]}\ \dfrac{\Gamma\ \texttt{z} = \textbf{int}}{\Gamma \vdash \texttt{z} : \textbf{int}}}{\Gamma \vdash \texttt{*y + z} : \textbf{int}} \qquad \text{[TVar]}\ \dfrac{\Gamma\ \texttt{x} = \textbf{char}}{\Gamma \vdash \texttt{x} : \textbf{char}} \qquad \dfrac{}{\texttt{int} \leftrightarrow \texttt{char}}}{\Gamma \vdash \texttt{x = *y + z;}}$$

b)

$$\text{[TAssign]}\ \dfrac{\text{[TAddr]}\ \dfrac{\text{[TVar]}\ \dfrac{\Gamma\ \texttt{z} = \textbf{int}}{\Gamma \vdash \texttt{z} : \textbf{int}}}{\Gamma \vdash \texttt{\&z} : \textbf{int*}} \qquad \text{[TIndir]}\ \dfrac{\text{[TVar]}\ \dfrac{\Gamma\ \texttt{p} = \textbf{int**}}{\Gamma \vdash \texttt{p} : \textbf{int**}}}{\Gamma \vdash \texttt{*p} : \textbf{int*}} \qquad \dfrac{}{\texttt{int} * \leftrightarrow \texttt{int} *}}{\Gamma \vdash \texttt{*p = \&z;}}$$

c)

$$\text{[TAssign]}\ \dfrac{\text{[TIndir]}\ \dfrac{\text{[TIndir]}\ \dfrac{\text{[TVar]}\ \dfrac{\Gamma\ \mathtt{p} = \textbf{int**}}{\Gamma \vdash \mathtt{p} : \textbf{int**}}}{\Gamma \vdash \mathtt{*p} : \textbf{int*}}}{\Gamma \vdash \mathtt{**p} : \textbf{int}} \qquad \text{[TVar]}\ \dfrac{\Gamma\ \mathtt{x} = \textbf{char}}{\Gamma \vdash \mathtt{x} : \textbf{char}} \qquad \dfrac{}{\mathtt{int} \leftrightarrow \mathtt{char}}}{\Gamma \vdash \mathtt{x\ =\ **p;}}$$

Exercise 11.3:

State the resulting type environment Γ after executing the Block rule, beginning with an empty environment. Then, check whether the following block statements contain any type errors under Γ using the C0t type semantics.

```
{
  int a;
  char c;
  void* v;
  char* cp;
  int** i;

  ...
}
```

1.

```
{
  cp = v;
  while (cp)
    a = v;
}
```

2.

```
{
  i = &cp;
  **i = c + a;
  v = &i;
}
```

3.

```
{
  cp = &c;
  while (cp) {
    int* a;
    a = c;
  }
}
```

4.

```
{
  if (a)
    *cp = a;
  else
    *cp = &a;
}
```

5.

```c
{
  c = a;
  if (c)
    abort();
  else
    *i = &cp;
}
```

Solution

Type environment: $\Gamma = \{\texttt{a} \mapsto \textbf{int}, \texttt{c} \mapsto \textbf{char}, \texttt{v} \mapsto \textbf{void*}, \texttt{cp} \mapsto \textbf{char*}, \texttt{i} \mapsto \textbf{int**}\}$

1. The statement is not well-typed.

(i)

$$\text{[TAssign]}\dfrac{\text{[TVar]}\dfrac{\Gamma\ \texttt{cp} = \textbf{char*}}{\Gamma \vdash \texttt{cp} : \textbf{char*}} \quad \text{[TVar]}\dfrac{\Gamma\ \texttt{v} = \textbf{void*}}{\Gamma \vdash \texttt{v} : \textbf{void*}} \quad \dfrac{}{\texttt{char} * \leftrightarrow \texttt{void} *}}{\Gamma \vdash \texttt{cp = v;}}$$

(ii)

$$\text{[TAssign]}\dfrac{\text{[TVar]}\dfrac{\Gamma\ \texttt{a} = \textbf{int}}{\Gamma \vdash \texttt{a} : \textbf{int}} \quad \text{[TVar]}\dfrac{\Gamma\ \texttt{v} = \textbf{void*}}{\Gamma \vdash \texttt{v} : \textbf{void*}} \quad ???\dfrac{???}{\texttt{int} \leftrightarrow \texttt{void} *}}{\Gamma \vdash \texttt{a = v;}}$$

(iii)

$$\text{[TWhile]}\dfrac{\text{[TVar]}\dfrac{\Gamma\ \texttt{cp} = \textbf{char*}}{\Gamma \vdash \texttt{cp} : \textbf{char*}} \quad (ii.)}{\Gamma \vdash \texttt{while (cp) a = v;}}$$

$$\text{[TBlock]}\dfrac{\text{[TSeqS]}\dfrac{(i.) \quad (iii.)}{\Gamma \vdash \texttt{cp = v; while (cp) a = v;}}}{\Gamma \vdash \texttt{\{cp = v; while (cp) a = v;\}}}$$

2. The statement is not well-typed.

(i)

$$\text{[TAssign]}\dfrac{\text{[TVar]}\dfrac{\Gamma\ \texttt{i} = \textbf{int**}}{\Gamma \vdash \texttt{i} : \textbf{int**}} \quad \text{[TAddr]}\dfrac{\text{[TVar]}\dfrac{\Gamma\ \texttt{cp} = \textbf{char*}}{\Gamma \vdash \texttt{cp} : \textbf{char*}}}{\Gamma \vdash \texttt{\&cp} : \textbf{char**}} \quad ???\dfrac{???}{\texttt{int} ** \leftrightarrow \texttt{char} **}}{\Gamma \vdash \texttt{i = \&cp;}}$$

(ii)

$$\text{[TAssign]}\dfrac{\text{[TIndir]}\dfrac{\text{[TIndir]}\dfrac{\text{[TVar]}\dfrac{\Gamma\ \texttt{i} = \textbf{int**}}{\Gamma \vdash \texttt{i} : \textbf{int**}}}{\Gamma \vdash \texttt{*i} : \textbf{int*}}}{\Gamma \vdash \texttt{**i} : \textbf{int}} \quad \text{[TArith]}\dfrac{\text{[TVar]}\dfrac{\Gamma\ \texttt{c} = \textbf{char}}{\Gamma \vdash \texttt{c} : \textbf{char}} \quad \text{[TVar]}\dfrac{\Gamma\ \texttt{a} = \textbf{int}}{\Gamma \vdash \texttt{a} : \textbf{int}}}{\Gamma \vdash \texttt{c + a;} : \textbf{int}} \quad \dfrac{}{\texttt{int} \leftrightarrow \texttt{int}}}{\Gamma \vdash \texttt{**i = c + a;}}$$

(iii)

$$\text{[TAssign]}\dfrac{\text{[TVar]}\dfrac{\Gamma\ \texttt{v} = \textbf{void*}}{\Gamma \vdash \texttt{v} : \textbf{void*}} \quad \text{[TAddr]}\dfrac{\text{[TVar]}\dfrac{\Gamma\ \texttt{i} = \textbf{int**}}{\Gamma \vdash \texttt{i} : \textbf{int**}}}{\Gamma \vdash \texttt{\&i} : \textbf{int***}} \quad \dfrac{}{\texttt{void} * \leftrightarrow \texttt{int} ***}}{\Gamma \vdash \texttt{v = \&i;}}$$

$$\text{[TBlock]}\dfrac{\text{[TSeq]}\dfrac{(i.)\qquad \text{[TSeqS]}\dfrac{(ii.)\qquad (iii.)}{\Gamma \vdash \texttt{**i = c + a; v = \&i;}}}{\Gamma \vdash \texttt{i = \&cp; **i = c + a; v = \&i;}}}{\Gamma \vdash \texttt{\{ i = \&cp; **i = c + a; v = \&i; \}}}$$

3. The statement is not well-typed.

   (i)

$$\text{[TAssign]}\dfrac{\text{[TVar]}\dfrac{\Gamma\ \texttt{cp} = \textbf{char*}}{\Gamma \vdash \texttt{cp} : \textbf{char*}} \qquad \text{[TAddr]}\dfrac{\text{[TVar]}\dfrac{\Gamma\ \texttt{c} = \textbf{char}}{\Gamma \vdash \texttt{c} : \textbf{char}}}{\Gamma \vdash \texttt{\&c} : \textbf{char*}} \qquad \dfrac{}{\texttt{char} * \leftrightarrow \texttt{char} *}}{\Gamma \vdash \texttt{cp = \&c;}}$$

   (ii)

$$\text{[TAssign]}\dfrac{\text{[TVar]}\dfrac{\Gamma[\texttt{a} \mapsto \texttt{int}]\ \texttt{a} = \textbf{int*}}{\Gamma[\texttt{a} \mapsto \texttt{int}] \vdash \texttt{a} : \textbf{int*}} \qquad \text{[TVar]}\dfrac{\Gamma[\texttt{a} \mapsto \texttt{int}]\ \texttt{c} = \textbf{char}}{\Gamma[\texttt{a} \mapsto \texttt{int}] \vdash \texttt{c} : \textbf{char}} \qquad \text{???}\dfrac{\text{???}}{\texttt{int} * \leftrightarrow \texttt{char}}}{\Gamma[\texttt{a} \mapsto \texttt{int}] \vdash \texttt{a = c;}}$$

   (iii)

$$\text{[TWhile]}\dfrac{\text{[TVar]}\dfrac{\Gamma\ \texttt{cp} = \textbf{char*}}{\Gamma \vdash \texttt{cp} : \textbf{char*}} \qquad \text{[TBlock]}\dfrac{\text{[TSeq]}\dfrac{(ii.) \qquad \text{[TTerm]}\dfrac{}{\varepsilon}}{\Gamma[\texttt{a} \mapsto \texttt{int}] \vdash \texttt{a = c;}}}{\Gamma \vdash \texttt{\{ int* a; a = c; \}}}}{\Gamma \vdash \texttt{while (cp) \{ int* a; a = c; \}}}$$

$$\text{[TBlock]}\dfrac{\text{[TSeqS]}\dfrac{(i.) \qquad (iii.)}{\Gamma \vdash \texttt{cp = \&c; while (cp) \{ int* a; a = c; \}}}}{\Gamma \vdash \texttt{\{ cp = \&c; while (cp) \{ int* a; a = c; \} \}}}$$

4. The statement is not well-typed.

   (i)

$$\text{[TAssign]}\dfrac{\text{[TIndir]}\dfrac{\text{[TVar]}\dfrac{\Gamma\ \texttt{cp} = \textbf{char*}}{\Gamma \vdash \texttt{cp} : \textbf{char*}}}{\Gamma \vdash \texttt{*cp} : \textbf{char}} \qquad \text{[TVar]}\dfrac{\Gamma\ \texttt{a} = \textbf{int}}{\Gamma \vdash \texttt{a} : \textbf{int}} \qquad \dfrac{}{\texttt{char} \leftrightarrow \texttt{int}}}{\Gamma \vdash \texttt{*cp = a;}}$$

   (ii)

$$\text{[TAssign]}\dfrac{\text{[TIndir]}\dfrac{\text{[TVar]}\dfrac{\Gamma\ \texttt{cp} = \textbf{char*}}{\Gamma \vdash \texttt{cp} : \textbf{char*}}}{\Gamma \vdash \texttt{*cp} : \textbf{char}} \qquad \text{[TAddr]}\dfrac{\text{[TVar]}\dfrac{\Gamma\ \texttt{a} = \textbf{int}}{\Gamma \vdash \texttt{a} : \textbf{int}}}{\Gamma \vdash \texttt{\&a} : \textbf{int*}} \qquad \text{???}\dfrac{\text{???}}{\texttt{char} \leftrightarrow \texttt{int} *}}{\Gamma \vdash \texttt{*cp = \&a;}}$$

$$\text{[TBlock]}\dfrac{\text{[TIf]}\dfrac{\text{[TVar]}\dfrac{\Gamma\ \texttt{a} = \textbf{int}}{\Gamma \vdash \texttt{a} : \textbf{int}} \qquad (i.) \qquad (ii.)}{\Gamma \vdash \texttt{if (a) *cp = a; else *cp = \&a;}}}{\Gamma \vdash \texttt{\{ if (a) *cp = a; else *cp = \&a; \}}}$$

5. The statement is not well-typed.

(i)

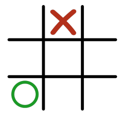
$$[\text{TAssign}]\dfrac{[\text{TVar}]\dfrac{\Gamma\ \texttt{c} = \textbf{char}}{\Gamma \vdash \texttt{c} : \textbf{char}} \qquad [\text{TVar}]\dfrac{\Gamma\ \texttt{a} = \textbf{int}}{\Gamma \vdash \texttt{a} : \textbf{int}} \qquad \dfrac{}{\texttt{char} \leftrightarrow \texttt{int}}}{\Gamma \vdash \texttt{c = a;}}$$

(ii)

$$[\text{TAssign}]\dfrac{[\text{TIndir}]\dfrac{[\text{TVar}]\dfrac{\Gamma\ \texttt{i} = \textbf{int**}}{\Gamma \vdash \texttt{i} : \textbf{int**}}}{\Gamma \vdash \texttt{*i} : \textbf{int*}} \qquad [\text{TAddr}]\dfrac{[\text{TVar}]\dfrac{\Gamma\ \texttt{cp} = \textbf{char*}}{\Gamma \vdash \texttt{cp} : \textbf{char*}}}{\Gamma \vdash \texttt{\&cp} : \textbf{char**}} \qquad ???\dfrac{???}{\texttt{int} * \leftrightarrow \texttt{char} **}}{\Gamma \vdash \texttt{*i = \&cp;}}$$

$$[\text{TBlock}]\dfrac{[\text{TSeqS}]\dfrac{(i.) \qquad [\text{TIf}]\dfrac{[\text{TVar}]\dfrac{\Gamma\ \texttt{c} = \textbf{char}}{\Gamma \vdash \texttt{c} : \textbf{char}} \qquad [\text{TAbort}]\dfrac{}{\Gamma \vdash \texttt{abort();}} \qquad (ii.)}{\Gamma \vdash \texttt{if (c) abort(); else *i = \&cp;}}}{\Gamma \vdash \texttt{c = a; if (c) abort(); else *i = \&cp;}}}{\Gamma \vdash \texttt{\{ c = a; if (c) abort(); else *i = \&cp; \}}}$$

Exercise 11.4:



In this exercise, you are given small C0t code snippets. You should first check whether a code snippet is well-typed and if it is, execute it. For this, you should create a derivation tree and if it is well-typed, also an execution protocol for each code snippet.

a) Starting state: $\Gamma = \{\}$

```
{
    int x;
    int* px;

    x = 1337;

    px = &x;

    *px = 420;
}
```

b) Starting state: $\Gamma = \{\texttt{c} \mapsto \texttt{char}, \texttt{i} \mapsto \texttt{int}, \texttt{p} \mapsto \texttt{char*}\}$

```
i = 42;
p = &c;

if (i > 21)
    p = &i;
else
    *p = 21;
```

c) Starting state: $\Gamma = \{\texttt{v} \mapsto \texttt{int}, \texttt{pv} \mapsto \texttt{int*}, \texttt{pv2} \mapsto \texttt{int*}, \texttt{c} \mapsto \texttt{char}\}$

```
v = 69;
pv = &v;
pv2 = pv + 2;
c = pv2 + 2;
```

Solution

a) The code snippet is well-typed

Derivation Tree:

$$\dfrac{\dfrac{\Gamma' = \Gamma[\texttt{x} \mapsto \texttt{int}, \texttt{px} \mapsto \texttt{int*}] \qquad \dfrac{\dfrac{\text{Subtree 1}}{\Gamma' \vdash \texttt{x=1337;}}\,[\texttt{TAssign}] \qquad \dfrac{\dfrac{\text{Subtree 2}}{\Gamma' \vdash \texttt{px=\&x}}\,[\texttt{TAssign}] \qquad \dfrac{\text{Subtree 3}}{\Gamma' \vdash \texttt{*px=420;}}\,[\texttt{TAssign}]}{\Gamma' \vdash \texttt{px=\&x; *px=420;}}\,[\texttt{TSeqS}]}{\Gamma' \vdash \texttt{x=1337; px=\&x; *px=420;}}\,[\texttt{TSeq}]}{\Gamma \vdash \texttt{\{int x; int* px; x=1337; px=\&x; *px=420;\}}}\,[\texttt{TBlock}] \qquad \dfrac{}{\Gamma \vdash \epsilon}\,[\texttt{TTerm}]}{\Gamma \vdash \texttt{\{int x; int* px; x=1337; px=\&x; *px=420;\}}}\,[\texttt{TSeq}]$$

Subtree 1:

$$\dfrac{\dfrac{-2^{31} \leq 1337 < 2^{31}}{\Gamma' \vdash \texttt{1337 : int}}\,[\texttt{TConst}] \qquad \dfrac{\Gamma'\texttt{x} = \texttt{int}}{\Gamma' \vdash \texttt{x : int}}\,[\texttt{TVar}] \qquad \dfrac{}{\texttt{int} \leftrightarrow \texttt{int}}}{\Gamma' \vdash \texttt{x=1337;}}\,[\texttt{TAssign}]$$

Subtree 2:

$$\dfrac{\dfrac{\dfrac{\Gamma'\texttt{x} = \texttt{int}}{\Gamma' \vdash \texttt{x : int}}\,[\texttt{TVar}]}{\Gamma' \vdash \texttt{\&x : int*}}\,[\texttt{TAddr}] \qquad \dfrac{\Gamma'\texttt{px} = \texttt{int*}}{\Gamma' \vdash \texttt{px : int*}}\,[\texttt{TVar}] \qquad \dfrac{}{\texttt{int*} \leftrightarrow \texttt{int*}}}{\Gamma' \vdash \texttt{px=\&x;}}\,[\texttt{TAssign}]$$

Subtree 3:

$$\dfrac{\dfrac{-2^{31} \leq 420 < 2^{31}}{\Gamma' \vdash \texttt{420 : int}}\,[\texttt{TConst}] \qquad \dfrac{\dfrac{\Gamma'\texttt{px} = \texttt{int*}}{\Gamma' \vdash \texttt{px : int*}}\,[\texttt{TVar}]}{\Gamma' \vdash \texttt{*px : int}}\,[\texttt{TIndir}] \qquad \dfrac{}{\texttt{int} \leftrightarrow \texttt{int}}}{\Gamma' \vdash \texttt{*px=420;}}\,[\texttt{TAssign}]$$

Execution Trace:

$$\begin{array}{llll}
 & \langle \texttt{\{int x; int* px; x=1337; px=\&x; *px=420;\}} & |\{\};\{\}\rangle & \\
\rightarrow & \langle \texttt{x=1337; px=\&x; *px=420;}\ \blacksquare & |\{\},\{\texttt{x} \mapsto \triangle, \texttt{px} \mapsto \square\};\{\triangle \mapsto ?, \square \mapsto ?\}\rangle & [\texttt{Scope}] \\
\rightarrow & \langle \texttt{px=\&x; *px=420;}\ \blacksquare & |\{\},\{\texttt{x} \mapsto \triangle, \texttt{px} \mapsto \square\};\{\triangle \mapsto 1337, \square \mapsto ?\}\rangle & [\texttt{Assign}] \\
\rightarrow & \langle \texttt{*px=420;}\ \blacksquare & |\{\},\{\texttt{x} \mapsto \triangle, \texttt{px} \mapsto \square\};\{\triangle \mapsto 1337, \square \mapsto \triangle\}\rangle & [\texttt{Assign}] \\
\rightarrow & \langle \blacksquare & |\{\},\{\texttt{x} \mapsto \triangle, \texttt{px} \mapsto \square\};\{\triangle \mapsto 420, \square \mapsto \triangle\}\rangle & [\texttt{Assign}] \\
\rightarrow & \langle \varepsilon & |\{\};\{\}\rangle & [\texttt{Leave}]
\end{array}$$

b) The code snippet is *not* well-typed

$$\dfrac{\dfrac{\dfrac{-2^{31} \leq 42 < 2^{31}}{\Gamma \vdash \texttt{42 : int}}\,[\texttt{TConst}] \qquad \dfrac{\Gamma\texttt{i} = \texttt{int}}{\Gamma \vdash \texttt{i : int}}\,[\texttt{TVar}] \qquad \dfrac{}{\texttt{int} \leftrightarrow \texttt{int}}}{\Gamma \vdash \texttt{i=42;}}\,[\texttt{TAssign}] \qquad \dfrac{\dfrac{\text{Subtree 1}}{\Gamma \vdash \texttt{p=\&c;}} \qquad \dfrac{\dfrac{\text{Subtree 2}}{\Gamma \vdash \texttt{i > 21 : int}} \quad \dfrac{\text{Subtree 3}}{\Gamma \vdash \texttt{p=\&i;}} \quad \dfrac{\text{Subtree 4}}{\Gamma \vdash \texttt{*p=21;}}}{\Gamma \vdash \texttt{if (i > 21) p=\&i; else *p=21;}}\,[\texttt{TIf}]}{\Gamma \vdash \texttt{p=\&c; if (i > 21) p=\&i; else *p=21;}}\,[\texttt{TSeqS}]}{\Gamma \vdash \texttt{i=42; p=\&c; if (i > 21) p=\&i; else *p=21;}}\,[\texttt{TSeq}]$$

Subtree 1:

$$
\texttt{[TAssign]}\ \dfrac{\texttt{[TAddr]}\ \dfrac{\texttt{[TVar]}\ \dfrac{\Gamma \texttt{c} = \texttt{char}}{\Gamma \vdash \texttt{c : char}}}{\Gamma \vdash \texttt{\&c : char*}} \quad \texttt{[TVar]}\ \dfrac{\Gamma \texttt{p} = \texttt{char*}}{\Gamma \vdash \texttt{p : char*}} \quad \dfrac{}{\texttt{char*} \leftrightarrow \texttt{char*}}}{\Gamma \vdash \texttt{p=\&c;}}
$$

Subtree 2:

$$
\texttt{[TCmp]}\ \dfrac{\texttt{[TVar]}\ \dfrac{\Gamma \texttt{i} = \texttt{int}}{\Gamma \vdash \texttt{i : int}} \quad \texttt{[TConst]}\ \dfrac{-2^{31} \leq 21 < 2^{31}}{\Gamma \vdash \texttt{21 : int}} \quad \dfrac{}{\texttt{int} \leftrightarrow \texttt{int}}}{\Gamma \vdash \texttt{i > 21 : int}}
$$

Subtree 3:

$$
\texttt{[TAssign]}\ \dfrac{\texttt{[TAddr]}\ \dfrac{\texttt{[TVar]}\ \dfrac{\Gamma \texttt{i} = \texttt{int}}{\Gamma \vdash \texttt{i : int}}}{\Gamma \vdash \texttt{\&i : int*}} \quad \texttt{[TVar]}\ \dfrac{\Gamma \texttt{p} = \texttt{char*}}{\Gamma \vdash \texttt{p : char*}} \quad \dfrac{\textbf{Error}}{\texttt{int*} \leftrightarrow \texttt{char*}}}{\Gamma \vdash \texttt{p=\&i;}}
$$

The error occurs here, because the value of `int*` cannot be implicitly converted into a value of type `char*`.

Subtree 4:

$$
\dfrac{\texttt{[TConst]}\ \dfrac{-2^{31} \leq 21 < 2^{31}}{\Gamma \vdash \texttt{21 : int}} \quad \texttt{[TIndir]}\ \dfrac{\texttt{[TVar]}\ \dfrac{\Gamma \texttt{p} = \texttt{char*}}{\Gamma \vdash \texttt{p : char*}}}{\Gamma \vdash \texttt{*p : char}} \quad \dfrac{}{\texttt{int} \leftrightarrow \texttt{char}}}{\Gamma \vdash \texttt{*p=21;}}\ \texttt{[TAssign]}
$$

c) The code snippet is *not* well-typed

Derivation Tree:

$$
\texttt{[TSeq]}\ \dfrac{\texttt{[TAssign]}\ \dfrac{\texttt{[TVar]}\ \dfrac{\Gamma \texttt{v} = \texttt{int}}{\Gamma \vdash \texttt{v : int}} \quad \texttt{[TConst]}\ \dfrac{-2^{31} \leq 69 < 2^{31}}{\Gamma \vdash \texttt{69 : int}} \quad \dfrac{}{\texttt{int} \leftrightarrow \texttt{int}}}{\Gamma \vdash \texttt{v=69;}} \quad \dfrac{\dfrac{\text{Subtree 1}}{\Gamma \vdash \texttt{pv=\&v;}} \quad \dfrac{\dfrac{\text{Subtree 2}}{\Gamma \vdash \texttt{pv2=pv+2;}} \quad \dfrac{\text{Subtree 3}}{\Gamma \vdash \texttt{c=pv2-pv;}}}{\Gamma \vdash \texttt{pv2=pv+2; c=pv2-pv;}}\ \texttt{[TSeqS]}}{\Gamma \vdash \texttt{pv=\&v; pv2=pv+2; c=pv2-pv;}}\ \texttt{[TSeq]}}{\Gamma \vdash \texttt{v=69; pv=\&v; pv2=pv+2; c=pv2-pv;}}
$$

Subtree 1:

$$
\texttt{[TAssign]}\ \dfrac{\texttt{[TAddr]}\ \dfrac{\texttt{[TVar]}\ \dfrac{\Gamma \texttt{v} = \texttt{int}}{\Gamma \vdash \texttt{v : int}}}{\Gamma \vdash \texttt{\&v : int*}} \quad \texttt{[TVar]}\ \dfrac{\Gamma \texttt{pv} = \texttt{int*}}{\Gamma \vdash \texttt{pv : int*}} \quad \dfrac{}{\texttt{int*} \leftrightarrow \texttt{int*}}}{\Gamma \vdash \texttt{pv=\&v;}}
$$

Subtree 2:

$$
\texttt{[TAssign]}\ \dfrac{\texttt{[TVar]}\ \dfrac{\Gamma \texttt{pv2} = \texttt{int*}}{\Gamma \vdash \texttt{pv2 : int*}} \quad \texttt{[TPtrArith]}\ \dfrac{\texttt{[TVar]}\ \dfrac{\Gamma \texttt{pv} = \texttt{int*}}{\Gamma \vdash \texttt{pv : int*}} \quad \texttt{[TConst]}\ \dfrac{-2^{31} \leq 2 < 2^{31}}{\Gamma \vdash \texttt{2 : int}}}{\Gamma \vdash \texttt{pv+2 : int*}} \quad \dfrac{}{\texttt{int*} \leftrightarrow \texttt{int*}}}{\Gamma \vdash \texttt{pv2=pv+2;}}
$$

Subtree 3:

$$\text{[TAssign]}\ \dfrac{\text{[TVar]}\ \dfrac{\Gamma\,\texttt{c} = \texttt{char}}{\Gamma \vdash \texttt{c} : \texttt{char}} \qquad \text{[TPtrArith]}\ \dfrac{\text{[TVar]}\ \dfrac{\Gamma\,\texttt{pv2} = \texttt{int*}}{\Gamma \vdash \texttt{pv2} : \texttt{int*}} \qquad \text{[TConst]}\ \dfrac{-2^{31} \leq 2 < 2^{31}}{\Gamma \vdash 2 : \texttt{int}}}{\Gamma \vdash \texttt{pv2+2} : \texttt{int*}} \qquad \dfrac{\textbf{Error}}{\texttt{char} \leftrightarrow \texttt{int*}}}{\Gamma \vdash \texttt{c=pv2+2;}}$$

The error occurs here, because the value of `int*` cannot be implicitly converted into a value of type `char`.

Exercise 11.5:

For each of the following snippets, first perform static semantic analysis using the derivation rules from Definitions 6.6.6 and 6.6.7. Then use small-step operational semantics of *C0b* to evaluate the program. Do expression evaluation seperately.

1. ```
   {
       int x;
       x = 42 / 0;
   }
   ```

2. ```
   {
       int x;
       x = x + 42;
   }
   ```

Solution

1. Type checking: With Type environment: $\Gamma = \{\}$

$$\text{[TSeq]}\ \dfrac{\text{[TBlock]}\ \dfrac{\Gamma' = \Gamma[\texttt{x} \mapsto \textbf{int}] \qquad \text{[TSeq]}\ \dfrac{\text{[TAssign]}\ \dfrac{\text{[TArith]}\ \dfrac{\text{[TConst]}\ \dfrac{-2^{31} \leq 42 < 2^{31}}{\Gamma' \vdash 42 : \textbf{int}} \qquad \text{[TConst]}\ \dfrac{-2^{31} \leq 0 < 2^{31}}{\Gamma' \vdash 0 : \textbf{int}}}{\Gamma' \vdash \texttt{42 / 0} : \textbf{int}} \qquad \text{[TVar]}\ \dfrac{\Gamma'\,\texttt{x} = \textbf{int}}{\Gamma' \vdash \texttt{x} : \textbf{int}} \qquad \dfrac{}{\texttt{int} \leftrightarrow \texttt{int}}}{\Gamma' \vdash \texttt{x = 42 / 0;}} \qquad \text{[TTerm]}\ \dfrac{}{\Gamma' \vdash \varepsilon}}{\Gamma' \vdash \texttt{x = 42 / 0;}}}{\Gamma \vdash \texttt{\{ int x; x = 42 / 0; \}}} \qquad \text{[TTerm]}\ \dfrac{}{\Gamma \vdash \varepsilon}}{\Gamma \vdash \texttt{\{ int x; x = 42 / 0; \}}}$$

   Then performing the small-step operational semantics:

$$\begin{array}{lll} & \langle \texttt{\{ int x; x = 42/0 \} \}} \mid \{\}; \{\} \rangle & \\ \rightarrow & \langle \texttt{ x = 42/0;} \ \blacksquare \mid \{\}, \{\texttt{x} \mapsto \triangle\}; \{\triangle \mapsto ?\} \rangle & \text{[Scope]} \end{array}$$

   We need to evaluate the expression `42/0` seperately. We do this by using the small-step operational semantics of *C0b* for expressions: Let $\sigma := (\rho_0, \rho_1; \mu)$ be $(\{\}, \{\texttt{x} \mapsto \triangle\}; \{\triangle \mapsto ?\})$. Since $R[\![\texttt{42/0}]\!]\sigma$ is undefined, and the execution gets stuck at this point. Consequently, the entire program becomes stuck because the Assignment statement necessitates the evaluation of this expression.

2. Type checking: With Type environment: $\Gamma = \{\}$

$$\text{[TSeq]}\ \dfrac{\text{[TBlock]}\ \dfrac{\Gamma' = \Gamma[\texttt{x} \mapsto \textbf{int}] \qquad \text{[TSeq]}\ \dfrac{\text{[TAssign]}\ \dfrac{\text{[TArith]}\ \dfrac{\text{[TVar]}\ \dfrac{\Gamma'\,\texttt{x} = \textbf{int}}{\Gamma' \vdash \texttt{x} : \textbf{int}} \qquad \text{[TConst]}\ \dfrac{-2^{31} \leq 42 < 2^{31}}{\Gamma' \vdash 42 : \textbf{int}}}{\Gamma' \vdash \texttt{x + 42} : \textbf{int}} \qquad \text{[TVar]}\ \dfrac{\Gamma'\,\texttt{x} = \textbf{int}}{\Gamma' \vdash \texttt{x} : \textbf{int}} \qquad \dfrac{}{\texttt{int} \leftrightarrow \texttt{int}}}{\Gamma' \vdash \texttt{x = x + 42;}} \qquad \text{[TTerm]}\ \dfrac{}{\Gamma' \vdash \varepsilon}}{\Gamma' \vdash \texttt{x = x + 42;}}}{\Gamma \vdash \texttt{\{ int x; x = x + 42; \}}} \qquad \text{[TTerm]}\ \dfrac{}{\Gamma \vdash \varepsilon}}{\Gamma \vdash \texttt{\{ int x; x = x + 42; \}}}$$

   Then performing the small-step operational semantics:

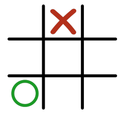
$$\begin{array}{lll} & \langle \texttt{\{ int x; x = x + 42 \} \}} \mid \{\}; \{\} \rangle & \\ \rightarrow & \langle \texttt{ x = x + 42;} \ \blacksquare \mid \{\}, \{\texttt{x} \mapsto \triangle\}; \{\triangle \mapsto ?\} \rangle & \text{[Scope]} \end{array}$$

We need to evaluate the expression `x + 42` seperately. We do this by using the small-step operational semantics of *C0b* for expressions: Let $\sigma := (\rho_0, \rho_1; \mu)$ be $(\{\}, \{\mathtt{x} \mapsto \bigtriangleup\}; \{\bigtriangleup \mapsto ?\})$.

$$\begin{aligned}
R[\![\mathtt{x\ +\ 42}]\!]\sigma &= R[\![\mathtt{x}]\!]\sigma + R[\![42]\!]\sigma \\
&= \mu(L[\![\mathtt{x}]\!]\sigma) + 42 \\
&= \mu(\rho_1(x)) + 42 \\
&= \mu(\bigtriangleup) + 42 \\
&= ? + 42
\end{aligned}$$

As addition is only defined for values (and especially not for ?), we get stuck during expression evaluation. Thus, our operational semantics are stuck as well.

Exercise 11.6: MiniLexer



Dieter, a student of Prog2, heard about syntactic analysis in the lecture and got really excited about it. He cannot wait to implement the compiler project and wants to start right away. Thus, he decides to implement a lexer first. Unfortunately, he is not very experienced in programming and needs your help.
His lexer should be able to lex a small subset of Java, which he calls MiniJava. It contains the following lexical categories:

- `Identifier`: A sequence of characters, starting with a letter (a-z, A-Z) and containing letters, digits (0-9) and underscores (_).
- `Operator`: +, -, *, /, ==, <, >
- `Const`: A sequence of digits (0-9)
- `Keyword`: if, else, while, for, return, int, bool
- `ASSIGN`: =, `LPAREN`: (, `RPAREN`:), `LBRACE`: {, `RBRACE`: }

As input the lexer should get a string. If the string is lexically correct, the lexer should return a list of tokens. Otherwise, it should throw an exception. The tokens should consist of the lexical category and the original text (if necessary). Furthermore, they should contain the line of the token in the original string. Lines are separated by the newline character `\n`.

1. If you are not yet familiar with what a lexer is, read chapter 9.1: Syntactic Analysis of the lecture notes.
2. Create a class `Token` which represents a token. It should contain the lexical category, the original text and the line of the token. Furthermore, it should contain a method `toString()` which returns a string representation of the token.
3. Create a class `MiniLexer` which contains a method `lex(String input)`. The method should return a list of tokens if the input is lexically correct. Otherwise, it should throw an exception containing the illegal character as well as the corresponding line.
4. Test your lexer by lexing example programs.

<u>Hint</u>:

- It might be useful to use regular expressions for identifiers and constants.
- A valid input for the lexer could be: `int a = 5 + 3;`.
  The lexer should return a list of tokens, which can simply be printed using `System.out.println`, as you have implemented the `toString()` method of the token class. Your output could look something like this:
  `[(Keyword,int,1), (Identifier,a,1), (ASSIGN,1), (Const,5,1), (Operator,+,1), (Const,3,1)]`

Solution

Note that for all of the operators and the keywords we use a single token kind (`Categories`) each. They are only distinguished by their text. Usually one would use separate token kinds (e. g. one token kind for `if`, one for `else`, on for +, one for -...), but for simplicity and readability we reduced the number of `Categories`.

```java
public class Token {
    public enum Categories{
        Identifier, Operator, Const, Keyword, ASSIGN, LPAREN,
        RPAREN, LBRACE, RBRACE
    }
    private final Categories category;
    private final String value;
    private final int line;

    public Token(Categories category, String value, int line){
        this.category = category;
        this.value = value;
        this.line = line;
    }

    @Override
    public String toString(){
        assert(category!=null);
        if(value != null){
            return "("+category+","+value+","+line+")";
        }
        return "("+category+","+line+")";
    }

}
```

```java
import java.util.ArrayList;
import java.util.List;

public class Lexer {

/**
 * This method should take a string (program code) and return a list of
 * tokens.
 * It supports e.g. the following tokens: if, while, for, else, return,
 * int, bool, =, (, ), {, }, +, -, *, ==, <, >, /, identifiers,
 * (integer) constants.
 *
 * @param s program code
 * @throws Exception if the program code is not valid lexically
 */
public List<Token> lex(String s) throws Exception {
    List<Token> tokens = new ArrayList<>();
    int line = 1;
    for (int i = 0; i < s.length(); i++) {
        Character currChar = s.charAt(i);
        switch (currChar) {
        case '=':
            if (s.charAt(i + 1) == '=') {
                tokens.add(
                    new Token(Token.Categories.Operator, "==", line));
                i++;
                break;
            }
            tokens.add(new Token(Token.Categories.ASSIGN, null, line));
```

```
            break;
        case '(':
            tokens.add(new Token(Token.Categories.LPAREN, null, line));
            break;
        case ')':
            tokens.add(new Token(Token.Categories.RPAREN, null, line));
            break;
        case '{':
            tokens.add(new Token(Token.Categories.LBRACE, null, line));
            break;
        case '}':
            tokens.add(new Token(Token.Categories.RBRACE, null, line));
            break;
        case '+':
        case '-':
        case '*':
        case '/':
        case '<':
        case '>':
            tokens.add(
                new Token(
                    Token.Categories.Operator, currChar.toString(), line));
            break;
        default:
            String var = "";
            String currString = currChar.toString();
            // handle newlines
            if (currChar == '\n') {
                line++;
                break;
            }
            // skip whitespace
            if (currString.isBlank() || currChar == ';') {
                break;
            }
            // handle identifiers
            if (currString.matches("[a-zA-Z]")) {
                while (i < s.length()
                        && s.substring(i, i + 1).matches("[a-zA-Z0-9_]")) {
                    var += s.substring(i, i + 1);
                    i++;
                }
                i--;
                lexKeyword(var, tokens, line);
            // handle constants
            } else if (currString.matches("[0-9]")) {
                while (i < s.length()
                        && s.substring(i, i + 1).matches("[0-9]")) {
                    var += s.substring(i, i + 1);
                    i++;
                }
                i--;
                tokens.add(
                    new Token(Token.Categories.Const, var, line));
            } else {
                throw new Exception(
                    "Unexpected character: " + currString +
                    " at line " + line);
            }
        }
    }
```

```
    return tokens;
}

private void lexKeyword(String var, List<Token> tokens, int line)
    throws Exception {
    switch (var) {
        case "if":
        case "while":
        case "for":
        case "else":
        case "return":
        case "int":
        case "bool":
            tokens.add(
                new Token(Token.Categories.Keyword, var, line));
            break;
        default:
            tokens.add(
                new Token(Token.Categories.Identifier, var, line));
    }
}
}
```