

Programming Paradigms

—Final Exam—

Department of Computer Science
University of Stuttgart

Summer Semester 2022, September 8, 2022

Note: The solutions provided here may not be the only valid solutions.

Part 1 [4 points]

1. Which of the following statements is true? (Only one statement is true.)

- The C expression `*(ptr)` uses postfix notation.
- The C expression `n++` uses postfix notation.
- The C expression `*ptr--` uses infix notation.
- The C expression `23+42` uses prefix notation.
- The C expression `--in` uses infix notation.

2. Which of the following statements is true? (Only one statement is true.)

- Dereferencing a pointer means to free the memory that the pointer refers to.
- The memory layout of multi-dimensional arrays is the same across all programming languages.
- Languages that use the reference model cannot support nested structs.
- In languages with garbage collection, recursive data structures prevent the problem of having insufficient memory.
- A record is a data structure that stores different data items of possibly different types.

3. Which of the following statements is true? (Only one statement is true.)

- Implicit receipt means that all statements in a code block are executed concurrently.
- Thread pools allow a programmer to explicitly define tasks and their dependencies.
- The launch-at-elaboration syntax specifies that a thread gets started concurrently once a subroutine gets invoked.
- Using fork-join means that a single thread gets reused across multiple tasks.
- Parallel loops are, by design, free of data races.

4. Which of the following statements is true? (Only one statement is true.)

- The rows and columns of a unitary matrix that represents a gate in quantum computing correspond to the two facets of superposition.
- Interference in quantum physics causes every gate in quantum computing to be reversible.
- In quantum computing, a Hadamard gate is used to measure the value of a qubit.
- Every gate in quantum computing can be represented as a unitary matrix.
- Gates in quantum computing are typically represented in bracket notation.

Part 2 [12 points]

This question is about grammars and recursive descent parsing. Consider the following grammar of a tiny and very limited programming language. The non-terminals of the language are *Program*, *Statement*, and *Expression*. The terminals, i.e., the tokens of the language, are *identifier*, *=*, *if*, *then*, *else*, *skip*, *number*, and *+*. The start symbol is *Program*.

- (1) $Program \rightarrow Statement$
- (2) $Statement \rightarrow identifier = Expression$
- (3) $Statement \rightarrow if Expression then Statement else Statement$
- (4) $Statement \rightarrow skip$
- (5) $Expression \rightarrow identifier$
- (6) $Expression \rightarrow number$
- (7) $Expression \rightarrow Expression + Expression$

1. Which of the following is a legal program according to the grammar? Mark legal programs with “yes” and syntactically illegal programs with “no”.

- (a) `identifier = number` → Yes
- (b) `if skip then identifier = identifier else identifier = number` → No
- (c) `identifier = identifier + number + identifier` → Yes
- (d) `if identifier then if number then skip else skip else identifier = number`
→ Yes
- (e) `if identifier then skip else identifier = number` → Yes
- (f) `if number then else identifier` → No

2. The following page shows is an incomplete recursive descent parser for the above grammar, implemented in Python. In line 46, the parser reads a list of tokens, where each token simply is a string. If the parser successfully parses the input, it will terminate without any output. Otherwise, it will print “Error”. (An actual parser would also create a parse tree, which is omitted in this task.)

Provide code to fill into the gaps marked with ??? so that the parser indeed has the functionality described above. Write your solution right next to the line that contains ???. There are four gaps in total.

```

1 def program():
2     statement()
3
4 def statement():
5     if next_token == "identifier":
6         match("identifier")
7         match("=")
8         expression()
9     elif next_token == "if":
10        match("if")
11        expression()
12        match("then")
13        statement()
14        match("else")
15        statement()
16    elif next_token == "skip":
17        match("skip")
18    else:
19        error()
20
21 def expression():
22     if next_token == "identifier":
23         match("identifier")
24         if next_token == "+":
25             match("+")
26             expression()
27     elif next_token == "number":
28         match("number")
29         if next_token == "+":
30             match("+")
31             expression()
32     else:
33         error()
34
35 def error():
36     print("Error")
37
38 def match(target):
39     global next_token
40     if next_token == target:
41         tokens.pop(0) # remove first element of list
42         next_token = tokens[0] if len(tokens) > 0 else None
43     else:
44         error()
45
46 tokens = ... # get input as a list of strings
47 next_token = tokens[0] if len(tokens) > 0 else None
48
49 # start parsing
50 program()
51 if next_token != None: # check that we've reached end of input
52     error()

```

3. Show the steps taken by your completed parser for the last of the inputs given in Question 1 above (i.e., `if number then else identifier`). Provide your solution in the following table, where each row corresponds to calling one function in the parser. In the first column, provide the name and (if any) the arguments of the called function. In the second column, provide the remaining tokens just before calling the function. In the third column, provide the grammar rule due to which the function gets called.

Call	Remaining tokens just before the call	Grammar rule
<code>program()</code>	<code>if number then else identifier</code>	(Start symbol)
<code>statement()</code>	<code>if number then else identifier</code>	Rule 1
<code>match("if")</code>	<code>if number then else identifier</code>	Rule 3
<code>expression()</code>	<code>number then else identifier</code>	Rule 3
<code>match("number")</code>	<code>number then else identifier</code>	Rule 6
<code>match("then")</code>	<code>then else identifier</code>	Rule 3
<code>statement()</code>	<code>else identifier</code>	Rule 3
<code>error()</code>	<code>else identifier</code>	(None)

Part 3 [12 points]

This question is about names, binding, and scoping. Consider the program given in the first column of the following table. The program is written in a simple toy language with variables, functions, and assignments.

	Values after executing the line under rule set 1			Values after executing the line under rule set 2			Values after executing the line under rule set 3		
	a	b	c	a	b	c	a	b	c
<code>fun f() {</code>									
<code>// entry</code>	7	U	5	7	U	U	3	U	U
<code>b = a</code>	7	7	5	7	7	U	3	3	U
<code>c = 6</code>	7	7	6	7	7	6	3	3	6
<code>}</code>									
<code>fun g() {</code>									
<code>// entry</code>	3	U	4	U	X	U	U	X	U
<code>a = 7</code>	7	U	4	7	X	U	7	X	U
<code>c = 5</code>	7	U	5	7	X	5	7	X	5
<code>f()</code>									
<code>}</code>									
<code>a = 3</code>	3	U	U	3	X	U	3	X	U
<code>c = 4</code>	3	U	4	3	X	4	3	X	4
<code>g()</code>									
<code>// end</code>	7	7	6	3	X	4	3	X	4

Suppose three sets of rules for how scoping works in this language:

1. Rule set 1: There is a *single global scope*.
2. Rule set 2: There are function-level scopes and a global scope. The language uses dynamic scoping. A variable that gets written in a function before it is read in the same function is a local variable of this function; all other variables are globals.
3. Rule set 3: Same as rule set 2 but with static scoping.

For each rule set, follow the execution of the program and enter a value into every box () . For each box, enter the value that the corresponding variable name (a, b, or c) refers to right after the execution of a specific line. If the variable does not exist in the current scope, enter "X". If the variable exists in the current scope but is not yet defined, enter "U" .

Part 4 [9 points]

This question is about control flow constructs and how they get compiled to assembly code. You are given two snippets of Ada-inspired source code and the corresponding assembly code. The source code should be understandable without detailed knowledge of Ada (hints: `/=` is the not-equals operator, `=` is the equals operator, `4..5` means a range of values, `6 | 9` means either of the two values). Case-switch statements do not use fall-through semantics, but instead the cases are mutually exclusive. The assembly code uses the toy assembly language known from the lecture.

Some parts of the code are missing. Your task is to fill in the missing parts in such a way that the source code and the assembly code are semantically equivalent. Each missing part of the code is marked with `???`. Provide the correct code to insert by writing it right next to the given code on the corresponding line.

1. Code snippet 1 (4 missing parts):

(a) Source code:

```
1  if ((A /= B) and ((C < D) or (C = A)))
2  then
3    // branch 1
4  else
5    // branch 2
6  end if;
```

(b) Assembly code:

```
1    r1 := A
2    r2 := B
3    if r1 = r2 goto L2
4    r2 := C
5    r3 := D
6    if r2 < r3 goto L1
7    if r2 = r1 goto L1
8    goto L2
9  L1: code in branch 1
10   goto L3
11  L2: code in branch 2
12  L3: ...
```

2. Code snippet 2 (5 missing parts):

(a) Source code:

```
1 case E
2 if
3   when 6 | 9 => branch 3
4   when 3     => branch 1
5   when 4..5 => branch 2
6   others    => branch 4
7 end case;
```

(b) Assembly code:

```
1 T:  &L1
2     &L2
3     &L2
4     &L3
5     &L4
6     &L4
7     &L3
8
9     r1 := result of evaluating expression E
10    if r1 < 3 goto L4
11    if r1 > 9 goto L4
12    r1 := r1 - 3
13    r1 := T[r1]
14    goto *r1
15
16 L1: code in branch 1
17     goto L5
18 L2: code in branch 2
19     goto L5
20 L3: code in branch 3
21     goto L5
22 L4: code in branch 4
23 L5: ...
```

Part 5 [8 points]

The following is about formally defined type systems. Recall the toy language for typed expressions introduced in the lecture. For your reference, the syntax and type rules are reproduced here.

Syntax:

$\langle t \rangle ::=$ true
 | false
 | if $\langle t \rangle$ then $\langle t \rangle$ else $\langle t \rangle$
 | 0
 | succ $\langle t \rangle$
 | pred $\langle t \rangle$
 | iszero $\langle t \rangle$

Type rules for *Bool*:

$\frac{}{\text{true} : Bool}$ T-True
 $\frac{}{\text{false} : Bool}$ T-False
 $\frac{t_1 : Bool \quad t_2 : T \quad t_3 : T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T}$ T-If

Type rules for *Nat*:

$\frac{}{0 : Nat}$ T-Zero
 $\frac{t_1 : Nat}{\text{succ } t_1 : Nat}$ T-Succ
 $\frac{t_1 : Nat}{\text{pred } t_1 : Nat}$ T-Pred
 $\frac{t_1 : Nat}{\text{iszero } t_1 : Bool}$ T-IsZero

1. Consider the following typed expression:

if iszero (pred 0) then false else succ (succ 0)

(a) Is this expression type-correct?

No.

(b) If yes, provide the type derivation tree, including the names of the rules you apply.

If no, then explain why not.

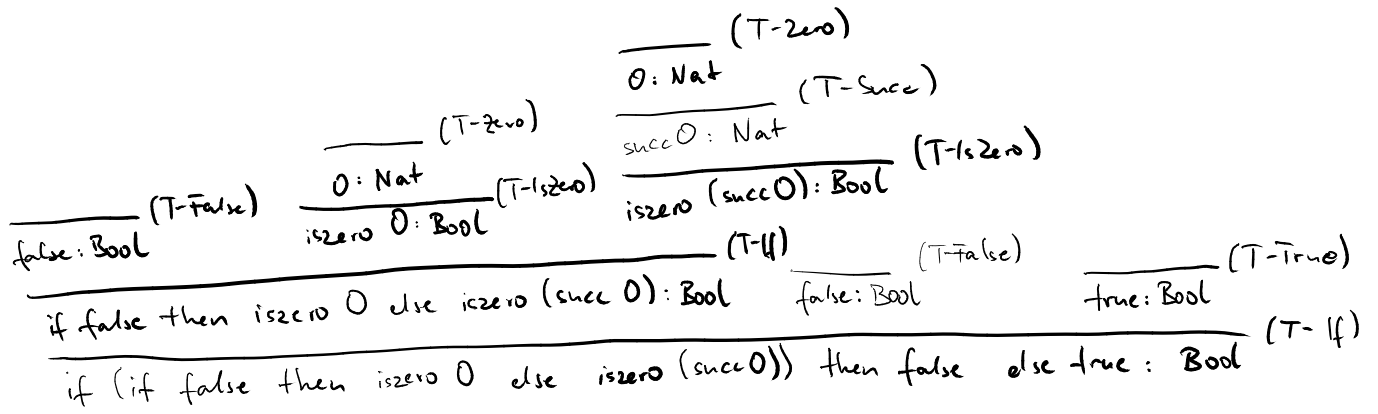
The then-branch and the else-branch yield values of different types (*Bool* and *Nat*, respectively), which is not allowed according to the T-If rule.

2. Consider the following typed expression:
 if (if false then iszero 0 else iszero (succ 0)) then false else true

(a) Is this expression type-correct?

Yes.

(b) If yes, provide the type derivation tree, including the names of the rules you apply.
 If no, then explain why not.



Part 6 [9 points]

The following is about calling sequences. Consider the following C program:

```
1 void bar(char c) {
2     int a, b;
3     // HERE
4 }
5
6 void foo(int x, int y) {
7     char z;
8     bar('q');
9 }
10
11 int main() {
12     foo(23, 42);
13 }
```

We assume the conventions for calling sequences introduced in the lecture. In addition, assume the following:

- The size of a `char` is 1 byte, and the size of an `int` is 4 bytes.
- Addresses are 4 bytes long.
- The compiler does not optimize the stack memory, but it allocates exactly the space required for all local variables declared in a function, even if they are not used.
- The callee-saved registers always require 8 bytes. There are no caller-saved registers.
- All arguments are passed via the stack (i.e., not via registers).

1. Using the template given on the next page, draw the function stack when the program reaches the location marked with `HERE`. For each stack frame, indicate which function it is for. For each memory region in a stack frame, indicate what is stored in it. If you know the concrete value that gets stored, then provide it; otherwise, provide the name or a description of what is stored. In the template, double lines separate stack frames, whereas single lines separate memory regions used for different purposes. The stack grows toward the top of the page, i.e., toward lower addresses.

A memory region of this height corresponds to 1 byte (and higher regions proportionally):

a
b
Caller-saved registers
Saved frame pointer
Return address
'q'
z
Caller-saved registers
Saved frame pointer
Return address
23
42
...

Frame of
function
bar

Frame of
function
foo

Frame of
function
main

2. Assume the code at location `HERE` wants to read the parameter `c` of `bar`. How can it access this parameter? Hint: Your explanation should use some, but not necessarily all, of the following terms: stack pointer, frame pointer, caller, callee, address, bytes.

The parameter `c` is stored in the frame of the caller, i.e., function `foo`. To access it from location `HERE`, the code starts from the frame pointer, which points to the saved frame pointer in the frame of function `bar`, and then adds the offset necessary to reach the value `c='q'`. That is, if the frame pointer is `f`, then the parameter will be accessed at `f + 8` bytes.

3. Stack frames typically store a return address. Explain what this address represents, why it is necessary, when it is used, and how it gets used.

The return address points into the memory region that stores the code of the currently executing program. More specifically, the return address stored in the frame of some function `callee` points to the instruction right after the call of `callee` in some function `caller`. Once the execution returns from `callee`, it will jump to the return address to continue execution in `caller`.

Part 7 [6 points]

The following is about the meaning of Scheme programs and about the differences between applicative-order and normal-order evaluation. Consider the following Scheme code:

```
1 (define g
2   (lambda (a b c) (if a b c))
3 )
4
5 (define f
6   (lambda (a b) (> a b))
7 )
8
9 (g (= 2 3) (f 7 "a") 42)
```

1. Provide a step-by-step evaluation of the expression at line 9 under *applicative-order* evaluation. Use the following template to provide your solution.

(g (= 2 3) (f 7 "a") 42)

⇒ (g #f (f 7 "a") 42)

⇒ (g #f (> 7 "a") 42)

⇒ Runtime type error: cannot compare number 7 and string "a"

2. Provide a step-by-step evaluation of the expression at line 12 under *normal-order* evaluation. Use the following template to provide your solution.

(g (= 2 3) (f 7 "a") 42)

⇒ (if (= 2 3) (f 7 "a") 42)

⇒ (if #f (f 7 "a") 42)

⇒ 42