Programming Paradigms

—Final Exam—

Department of Computer Science
University of Stuttgart

Summer Semester 2021, September 9, 2021

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.)

   - [ ] A cast is an implicit type conversion.
   - [x] A subtype may be compatible with its supertype.
   - [ ] A coercion is an explicit type conversion.
   - [ ] Types must be the same to be compatible.
   - [ ] Type coercions never cause any information loss.

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

   - [x] In Scheme, names are bound to values with `let`.
   - [ ] The `cdr` keyword of Scheme is used to create a function.
   - [ ] Functional languages, such as Scheme, do not provide loops.
   - [ ] Scheme is a statically typed language.
   - [ ] In Scheme, conditional expressions are evaluated first-come-first-serve.

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

   - [ ] Table-based scanners are typically written by hand.
   - [ ] The "longest possible token rule" specifies the maximum length of tokens in a language.
   - [x] A scanner reads a stream of characters to recognize tokens.
   - [ ] The table used by a table-based scanner represents FIRST, FOLLOW, and PREDICT sets.
   - [ ] A deterministic finite automaton may have epsilon transitions.

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

   - [ ] In a language with dynamic scoping, built-in objects always have priority.
   - [ ] In a language with dynamic scoping, subroutines may never be nested.
   - [ ] In a language with static scoping, the binding of a name depends on the control flow.
   - [x] In a language with static scoping, the binding of a name can be derived from the program text.
   - [ ] In a language with static scoping, two variables in a program cannot have the same name.

Part 2 [12 points]

This task is about computing the parse table for an LL(1) parser. Suppose the following grammar of a simple programming language:

(1) $\mathbf{P} \rightarrow \mathit{begin}\ \mathbf{Stmts}\ \mathit{end}$
(2) $\mathbf{Stmts} \rightarrow \mathbf{Stmt}\ ;\ \mathbf{Stmt}\ ;$
(3) $\mathbf{Stmts} \rightarrow \epsilon$
(4) $\mathbf{Stmt} \rightarrow \mathbf{Assign}$
(5) $\mathbf{Stmt} \rightarrow \mathit{nop}$
(6) $\mathbf{Assign} \rightarrow \mathbf{Id} = \mathbf{Num}$
(7) $\mathbf{Id} \rightarrow a$
(8) $\mathbf{Id} \rightarrow b$
(9) $\mathbf{Num} \rightarrow \mathit{23}$
(10) $\mathbf{Num} \rightarrow \mathit{42}$

The non-terminals of the language, printed in bold font, are:
P, **Stmts**, **Stmt**, **Assign**, **Id**, **Num**

The terminals of the language are:
begin, *end*, ;, *nop*, *a*, *b*, =, *23*, *42*

1. Compute the FIRST and FOLLOW sets of all non-terminals. Provide your solution in the following table:

Non-terminal	FIRST	FOLLOW
P	*begin*	*EOF*
Stmts	*nop*, *a*, *b*, ϵ	*end*
Stmt	*nop*, *a*, *b*	;
Assign	*a*, *b*	;
Id	*a*, *b*	=
Num	23, 42	;

2. Compute the PREDICT set of each grammar rule. Provide your solution in the following table:

Rule	PREDICT
(1)	*begin*
(2)	*nop*, *a*, *b*
(3)	*end*
(4)	*a*, *b*
(5)	*nop*
(6)	*a*, *b*
(7)	*a*
(8)	*b*
(9)	23
(10)	42

3. Compute a parse table suitable for a table-based, predictive parser. Provide your solution in the following table:

Non-terminal	Terminal								
	begin	*end*	;	*nop*	*a*	*b*	=	*23*	*42*
P	1								
Stmts		3		2	2	2			
Stmt				5	4	4			
Assign					6	6			
Id					7	8			
Num								9	10

Part 3 [8 points]

This part is about expressions and their evaluation order. Consider a toy programming language with unary, binary, and ternary operators that have the following precedence and associativity rules. A lower precedence number means that an operator should be applied first.

Operator	Multiplicity	Precedence	Associativity
♦ ..	Unary	3	Right
.. ♠ ..	Binary	1	Left
.. ♣ ..	Binary	2	Left
.. ❤ ..	Binary	2	Right
.. ✈ ..	Binary	4	Right
.. ☎ .. ✆ ..	Ternary	5	Right

Given the rules above, show the order of evaluating subexpressions in the following expressions. To indicate your solution, add parentheses into the given expressions that unambiguously describe the order in which subexpressions are evaluated. We leave an unusually large amount of space between the operands and operators to allow for adding the parentheses.

- Expression 1: ♦ ((a ♠ b) ♣ c)
- Expression 2: a ☎ (b ✈ (c ✈ d)) ✆ e
- Expression 3: ♦ (a ❤ ((b ♠ b) ♠ c))
- Expression 4: x ✈ (x ✈ (♦ x))

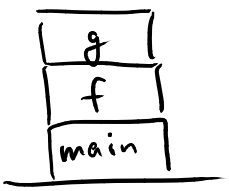
Part 4 [9 points]

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

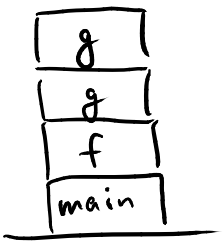
```
1  #include <string.h>
2
3  void h(char *str) {
4      char buffer[16];
5      strcpy(buffer, str); // copies the string pointed to by 'str'
6                           // to the buffer pointed to by 'buffer'
7  }
8
9  void g(int i, int j) {
10     if (i > 0)
11         g(i - 5, j);
12 }
13
14 void f(int a, int b) {
15     g(a, b);
16     g(-2, -3);
17
18     char large_string[256];
19     int i;
20     for(i = 0; i < 255; i++) {
21         large_string[i] = 'A';
22     }
23     h(large_string);
24 }
25
26 void main() {
27     f(5, 3);
28 }
```

1. The program reaches line 10 three times. For the first two times the line is reached, provide a sketch of the function call stack right after the reaching this line. Your sketch does not need to include the details of what is stored inside a stack frame, but simply show the stack frames that exist at a given point in time, along with the function that each frame corresponds to. The following is an example of how to draw the stack, where `foo` and `bar` are function names (obviously, this is not the correct solution):

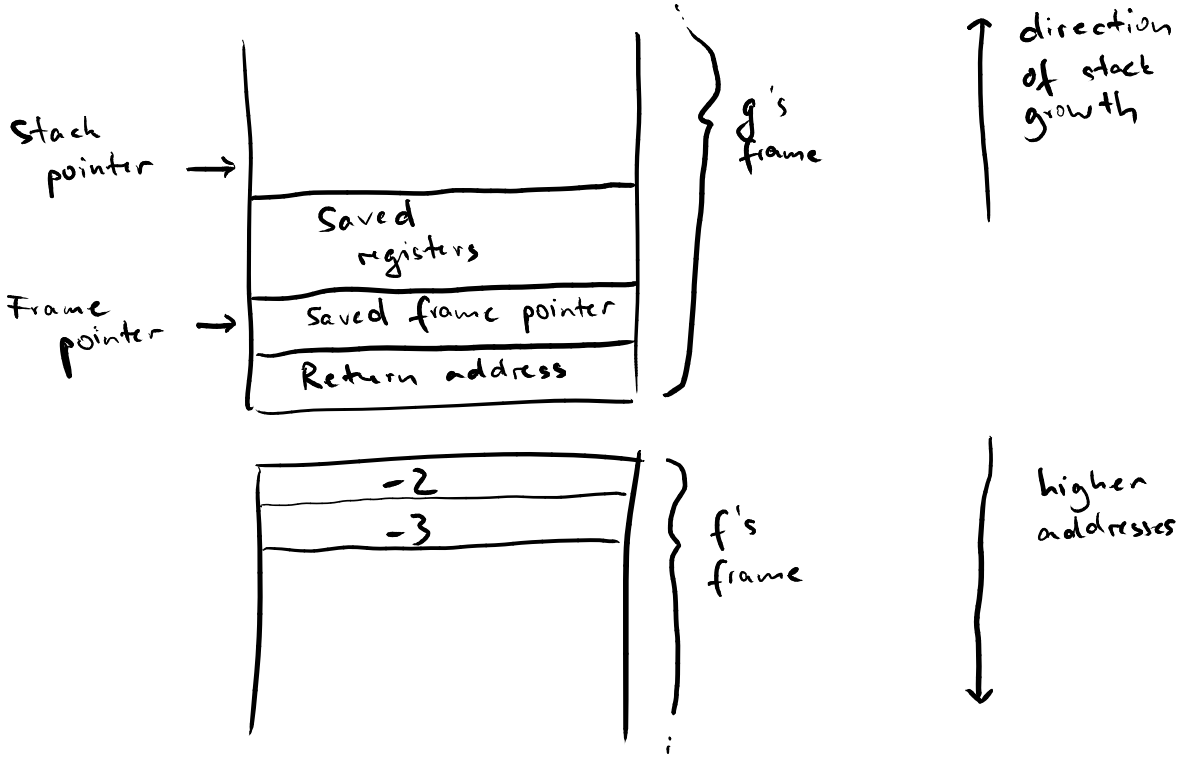
   (a) Call stack at the first time that line 10 is reached:



   (b) Call stack at the second time that line 10 is reached:

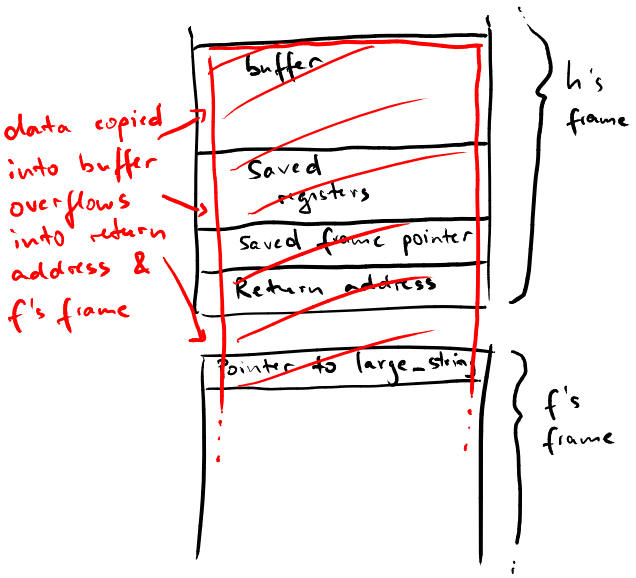


2. When line 10 is reached for the third time, i.e., right after the call at line 16, complete a more detailed sketch of the call stack. You should capture the state right after the prologue of function `g` has executed. Please use the following template to provide your solution and follow the conventions presented in the lecture. Note: All you need to add to the template are the arguments passed from `f` to `g` and the return address.



3. When the program executes line 5, something unusual happens. Explain what happens. To illustrate your explanations, provide a sketch of the call stack, including the `buffer` variable.

The data written into `buffer` is 256 bytes long, i.e., much longer than the memory reserved for `buffer`, which is only 16 bytes. Because `strcpy` does not check the size of the target location, it will write beyond the bounds of `buffer`, overwriting the return address of `h`'s stack frame and even data in `f`'s stack frame. This kind of stack-based buffer overflow can be used by attackers, e.g., to force the program execution to continue at another code address than the return address stored on the stack.



Part 5 [7 points]

This task is about static and dynamic method binding in a class-based language.

Consider the following source code, which is written in a toy language with Java-inspired syntax. Note that the semantics of the language is *not* as in Java, but as given below.

```
1  class A {
2          void m() {}
3          void n() {}
4  }
5
6  class B extends A {
7          void m() {}
8          override void n() {}
9          void o() {}
10 }
11
12 A x = new A();
13 B y = new B();
14 A z = y;
15
16 x.m();  // Call 1
17 x.n();  // Call 2
18 y.m();  // Call 3
19 y.n();  // Call 4
20 y.o();  // Call 5
21 z.m();  // Call 6
22 z.n();  // Call 7
23 z.o();  // Call 8
```

Suppose three different ways of method binding:

1. *Dynamic*. The language uses dynamic method binding.
2. *Static*. The language uses static method binding.
3. *Static except for "override"*. The language uses static method binding, except when an overriding method is declared with `override`, in which case dynamic method binding is used for calls to the method.

For each of the eight calls at the end of the program, indicate the method that gets called under a specific way of method binding. For example, if method `m` of class `A` gets called, then write `A.m`. If the language does not allow a specific call, please indicate "error". Use the following table to provide your solution:

Call	Dynamic	Static	Static except for "override"
Call 1	`A.m`	`A.m`	`A.m`
Call 2	`A.n`	`A.n`	`A.n`
Call 3	`B.m`	`B.m`	`B.m`
Call 4	`B.n`	`B.n`	`B.n`
Call 5	`B.o`	`B.o`	`B.o`
Call 6	`B.m`	`A.m`	`A.m`
Call 7	`B.n`	`A.n`	`B.n`
Call 8	`B.o`	error	error

Part 6 [11 points]

This tasks is about "true" iterators, as provided in Python via the `yield` keyword.

Consider the following Python program:

```
1  def f():
2      n = 5
3      while n > 0:
4          yield n
5          for j in g(n):
6              n = n - j
7
8  def g(n):
9      yield 1
10     print(n)
11     yield 1
12
13 if __name__ == "__main__":
14     for i in f():
15         print(i)
```

1. What does this code print?

   5
   5
   3
   3
   1
   1

2. How often are lines 2, 4, and 6 executed? Provide your solution using the following table:

Line	Number of times executed
2	1
4	3
6	6

3. Give a single-character change of the above program (i.e., replacing a single character with another character) that modifies the printed output to:

   5
   5

   In line 9, change `1` to `5`.

Part 7 [9 points]

The following is about a concurrent program written in a Cilk-like, C-based language. The language allows programmers to `spawn` function calls to be run as logically concurrent tasks, and to `sync` all tasks spawned by the calling task. Each function marked as `cilk` is not only a function but also a task.

Consider the following program:

```
1  int x = 1;
2
3  cilk void a() {
4      int s, t;
5      s = spawn b(2);
6      t = spawn b(3);
7      sync;
8      x += s + t;
9      s = spawn b(4);
10     t = spawn b(5);
11     sync;
12     x += s + t;
13 }
14
15 cilk int b(int n) {
16     return x + n;
17 }
18
19 int main() {
20     a();
21 }
```

1. What are the possible values of `s` and `t` after the execution of line 7? Explain your answer.

   The tasks spawned at lines 5 and 6 will run concurrently and return 3 and 4, respectively. As a result, `s=3` and `t=4` after line 7. Because both tasks are only reading `x`, but not writing to any shared memory, there is only one possible behavior.

2. What are the possible values of `x` after the execution of line 8? Explain your answer.

   The `sync` at line 7 forces the program to wait until both previously spawned tasks have finished and written values into `s` and `t`, as described above. When reaching line 8, the program adds 3 and 4 to the previous value of `x`, which is 1, so that `x=8` after line 8. Because in this part of the execution only a single task is active, there is only one possible behavior.

3. What are the possible values of `s` and `t` after the execution of line 11? Explain your answer.

   The tasks spawned at lines 9 and 10 will run concurrently and both add their respective argument, i.e., either 4 or 5, to the current value of `x`, which is 8. The `sync` forces the program to wait until both tasks have finished, and as a result, `s=12` and `t=13` after line 11. Again, there is only one possible behavior.

4. What are the possible values of `x` after the execution of line 12? Explain your answer.

   Similar to question 2, the program adds 12 and 13 to the current value of `x`, which is 8, giving `x=33`.