Directions—PLEASE READ

This exam comprises a mix of multiple-choice and short-answer questions, plus one essay-style extra-credit problem. Values are indicated for each. The regular questions total 60 points (including 2 points for putting your name on every page). The extra credit question is worth up to 8 additional points; it isn’t part of the 60, and it won’t factor into your exam score, but it may help to raise your letter grade at the end of the semester.

This is a closed-book exam. You must put away all books and notes. Please confine your answers to the space provided. For multiple choice questions, darken the circle next to the single best answer. Be sure to read all candidate answers before choosing. No partial credit will be given on the multiple-choice questions.

In the interest of fairness, the proctor will generally decline to answer questions during the exam. If you are unsure what a question is asking, make a reasonable assumption and state it as part of your answer.

You must complete the exam in class. Any remaining exams will be collected promptly at 11:40 am. Good luck!

1. (required) Per college policy, please write out the following statement and add your signature: “I affirm that I will not give or receive any unauthorized help on this exam, and that all work will be my own.”

2. (2 points) Put your name on every page (so if I lose a staple I won’t lose your answers).

Multiple Choice (2 points each)

3. Language implementations based on interpretation are usually slower than implementations based on compilation. Why, then, do people still use interpreters?

   ○ a. Interpreters tend to produce more helpful error messages.
   ○ b. Interpreter-based implementations tend to require less memory than compiler-based implementations.
   ○ c. Interpreters accommodate late binding.
   ☒ d. All of the above.

4. The grammars used in many production compilers contain error productions, which allow the parser to accept certain input programs that are, according to the manual, syntactically invalid. What purpose do these error productions serve?

   ○ a. They improve compilation speed by catching common errors more quickly than a general-purpose error recovery scheme would.
   ○ b. They support language features that can be parsed bottom-up, but not top-down.
   ☒ c. They provide more helpful error messages than a general-purpose error recovery scheme would.
5. What is a sentential form?

- a. One line of a derivation.
- b. An arbitrary string of grammar symbols.
- c. The content of the LL(1) parse stack.
- d. The string of symbols between the current point in the parse and the root of the parse tree.

6. Which of the following is a checked dynamic semantic error in C (one the language implementation is required to catch at run time)?

- a. use of an uninitialized variable
- b. access off the end of an array
- c. wrong number of parameters passed to a function
- d. divide by zero

7. The static link of a subroutine is

- a. the address to which the subroutine should return
- b. the frame pointer of the lexically surrounding subroutine
- c. the frame pointer of the calling subroutine
- d. the base address of global variables

8. Why, in the general case, must local variables in OCaml be allocated in the heap instead of on the stack?

- a. Because we don’t know their sizes at compile time.
- b. Because local variables in OCaml can have unlimited extent.
- c. Because OCaml subroutines don’t follow strict LIFO calling conventions.
- d. Because the OCaml garbage collector is unable to manipulate addresses in the stack.

**Short Answer** (3 points each)

9. Why does the scanner keep the text of identifiers, but not of keywords?

**Answer:** Because all instances of a given keyword are spelled the same, but instances of an identifier differ. The semantic analyzer needs to to know which is which, and all phases of the compiler need the spelling for error messages.

10. Why do most compilers perform semantic analysis on a syntax tree instead of on a parse tree?

**Answer:** Because the syntax tree is simpler and better captures the structure of the program.

11. Why do compiler families (collections of compilers for several languages and several target architectures) typically employ a common intermediate form?

**Answer:** So we can combine \( n \) front ends and \( m \) back ends instead of writing \( n \times m \) whole compilers.
Multiple Choice (3 points each)

Recall, for the following three questions, that OCaml, like most modern languages, uses static scope and deep binding.

12. What is the value of the following?

```
let x = 2 in
let f y = x + y in
let x = 3 in
f 4;;
```

- a. 5
- x. 6
- c. 7
- d. none of the above: the program is incorrect

13. How about this one?

```
let x = 2 in
let f y = x + y in
let x = 3 in
f x;;
```

- a. 4
- x. 5
- c. 6
- d. none of the above: the program is incorrect

14. And this one?

```
let rec f p n =
  let g () = n in
  if n = 0 then p ()
  else if n > 2 then f g (n - 1)
  else f p (n - 1) in
f (fun () -> 0) 3;;
```

- a. 3
- x. 2
- c. 1
- d. 0
Short Answer (3 points each)

15. In a table-driven top-down parser, why can’t semantic attributes be kept in the symbols on the parse stack (even with L-attributed attribute flow)?

   Answer: Because the parse stack contains only the symbols expected to be seen in the future, and we often need attributes for some of the symbols seen in the past.

16. Why doesn’t a C program need to create a subroutine closure when it passes a function as an argument?

   Answer: Because functions don’t nest in C: every variable is either local or global, so there’s no need for a static chain.

17. Describe, in 15 words or fewer, the language generated by the extended regular expression 
   \(( \text{digit}^+ \cdot \text{digit}^* ) \mid ( \text{digit}^* \cdot \text{digit}^+ )\) where \text{digit} is \(0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9\) and the dots are characters, not meta-characters.

   Answer: Strings containing a decimal point and at least one decimal digit.

Questions 18 through 21 (3 points each) refer to the following LL(1) CFG for arithmetic expressions in postfix notation.

\[
P \rightarrow E $$$
E \rightarrow \text{num} \ ET
ET \rightarrow \text{un.op} \ ET
   \rightarrow \ E \ \text{bin.op} \ ET
   \rightarrow
un.op \rightarrow ~ \mid \ \backslash
bin.op \rightarrow + \mid - \mid * \mid /
\]

Here the convention is that an operator \textit{follows} its arguments, rather than preceding or falling between them. The conventional infix equivalent of \(3 \ 2 \ 5 \ + \ * \ - \) is \(((3 \ * \ (2 \ + \ 5)) \ - \ 2)\). You may think of \(\sim\) as denoting additive inverse (what is usually marked \(+/–\) on a calculator); you may think of \(\backslash\) as denoting multiplicative inverse (what is usually marked \(1/x\) on a calculator).

18. What is \textit{first}(ET)?

   a. \(\{ \text{up.op, E} \}\)
   b. \(\{\sim, \backslash, \text{num}\}\)
   c. \(\{+, \sim, *, /\}\)
   d. \(\{+, \backslash, \text{num, } \epsilon\}\)

   \(\bigotimes\) 18. What is \textit{first}(ET)?

19. What is \textit{follow}(E)?

   a. \(\{ET\}\)
   b. \(\{$$, bin.op\}\)
   c. \(\{\sim, \backslash, \text{num}\}\)
   d. \(\{$$, +, \sim, *, /\}\)

   \(\bigotimes\) 19. What is \textit{follow}(E)?
20. What is $\text{PREDICT}(\text{ET} \rightarrow \epsilon)$?

- a. $\{\epsilon\}$
- b. $\{\epsilon\}$
- c. $\{\text{un\_op}, \text{bin\_op}\}$
- d. $\{\text{$$}, +, -, *, /\}$

21. The left-most derivation of the string $3 \sim 4 \ 2 + *$ is 14 lines long. What is the 7th line?

- a. $\text{E \ bin\_op \ ET}$
- b. $\text{num \ un\_op \ E \ * \ $$}$
- c. $\text{num \ ~ \ num \ ET \ bin\_op \ ET \ $$}$
- d. $\text{num \ un\_op \ num \ E \ bin\_op \ ET \ bin\_op \ ET \ $$}$

Questions 22 and 23 (2 points each) refer to the following attribute grammar, based on the previously given CFG for postfix expressions.

\[
\begin{align*}
P & \rightarrow \ E \ $$ \quad \triangleright \ P.r = E.r \\
E & \rightarrow \ \text{num \ ET} \quad \triangleright \ E.T.s = \text{num}.r \\
& \quad \quad \triangleright \ E.r = E.T.r \\
\text{ET}_1 & \rightarrow \ \text{un\_op \ ET}_2 \quad \triangleright \ ET_2.s = \left(\sim + \text{un\_op}.r + \text{ET}_1.s + \sim\right) \\
& \quad \quad \triangleright \ ET_1.r = ET_2.r \\
\text{ET}_1 & \rightarrow \ \text{E \ bin\_op \ ET}_2 \quad \triangleright \ ET_2.s = \left(\sim + \text{ET}_1.s + \text{bin\_op}.r + E.r + \sim\right) \\
& \quad \quad \triangleright \ ET_1.r = ET_2.r \\
ET & \rightarrow \ \epsilon \quad \triangleright \ ET.r = ET.s \\
\text{un\_op} & \rightarrow \ \sim \quad \triangleright \ \text{un\_op}.r = \sim \\
\text{un\_op} & \rightarrow \ \backslash \quad \triangleright \ \text{un\_op}.r = 1/ \\
\text{bin\_op} & \rightarrow \ + \quad \triangleright \ \text{bin\_op}.r = + \\
\text{bin\_op} & \rightarrow \ - \quad \triangleright \ \text{bin\_op}.r = - \\
\text{bin\_op} & \rightarrow \ * \quad \triangleright \ \text{bin\_op}.r = * \\
\text{bin\_op} & \rightarrow \ / \quad \triangleright \ \text{bin\_op}.r = / \\
\end{align*}
\]

Here the intent is that in a fully decorated tree, the $r$ attribute of the root ($P$) node will contain the (fully parenthesized) infix equivalent of the postfix expression that constitutes the leaves of the tree.

22. Attributes $s$ and $r$ are

- a. inherited and synthesized, respectively
- b. synthesized and inherited, respectively
- c. both inherited
- d. both synthesized
23. The attribute grammar as a whole is
   ○ a. S-attributed
   X b. L-attributed
   ○ c. both
   ○ d. neither

24. (3 points) Here is a parse tree for $3 \sim 4 \ 2 + \ *$. If we decorate the tree using the attribute grammar, the $r$ attribute of the root $P$ node will be the character string $((- 3) * (4 + 2))$, which is the infix equivalent of the parsed expression. What will be the $s$ attribute of the circled $ET$ node?
   ○ a. the empty string
   ○ b. $3 \sim 4 \ 2 +$
   X c. $(4 + 2)$
   ○ d. $(- 3) * (4 + 2)$

25. (Extra Credit; 8 points max)
   Some languages (e.g., Python) allow the programmer to omit variable declarations entirely. Other languages (e.g., Scheme) require the programmer to declare all variables, but do not require them to declare the variables’ types. Still other languages (e.g., Java) require the programmer to declare both variables and their types.
   Give a short (1–2 sentence) argument in favor of each approach. Which argument do you find most convincing? Why?

Answer: The Python approach leads to shorter programs that are arguably easier to write and read. It provides polymorphism “for free,” and relieves the programmer of the need to prove to the compiler that the program is type safe.

The Scheme approach is also polymorphic, but avoids errors due to misspellings and to accidental use of the same variable name for multiple purposes. It also makes the scope of names more obvious.

The Java approach is self-documenting, and arguably forces the programmer to think through things more carefully, reducing the incidence of bugs. Bugs that do occur are likely to be found at compile time. Given detailed knowledge of types, the compiler can produce more efficient code than it can for Scheme or Python.

Notably, OCaml and its relatives achieve most of the “free polymorphism” of Python and Scheme while still performing all type checking at compile time.

Which approach is better is largely a matter of personal preference. Experience suggests, however, that very large systems are easier to maintain (and run much faster) when written in statically typed languages (with or without type inference).

[NB: A full-credit answer would need to touch on issues of programmer convenience, polymorphism, error reporting time, execution speed, and, ideally, scoping. It would also express and defend a preference.]