
CS 320 Principles of Programming Languages ample Midterm Notes
---

These notes provide some solutions for the sample midterm. The notes also include:

- Cross references to the main lecture slides, written in the form [W:NN] where W is a week number and NN is a slide number (or a range of slide numbers).

- Advice and comments with suggestions on how you might answer questions like these and avoid common pitfalls or mistakes in an exam situation. These comments are not part of the solutions and are marked by enclosing them in {- ... -} brackets.

These sample questions do not include much material related to the Week 4 topic of Types: you have been focusing on that recently already for your work on the corresponding homework assignment. However, you can expect that the actual midterm will include questions on all of the topics covered in the first five weeks.

---
1)a){- Repeated on slides in Weeks 1, 2, 4, and 5: -}

- Syntax: "the written/spoken/symbolic/physical form; how things are communicated" [1:10]

- Semantics: "what the syntax means or represents" [1:11]

{- The definitions given here are taken directly from the slides. It is okay to quote these verbatim if you wish; you do not need to worry about stating these in your own words, although it is also good if you are able to do that. The question does not ask for examples or detailed explanations, so you do not need to include those elements in your answer. Volunteering additional information that is not requested is unlikely to earn you extra credit and will take time that could have been spent on other questions. A brief, crisp reply that focusses on the essential details is preferable to one that rambles and potentially confuses multiple issues. -}

b) The crosswalk scenario, applying the definitions as given above: [1:13]

Syntax comes into play in the lights/signals that are used to communicate to the pedestrian when it is safe to cross, or when they should stop. The button press and/or the associated signal that it generates are also forms of syntax. {- Actions, behaviors, intentions, mechanisms, algorithms, processes, machines, equipment, paper, buttons, reactions, and so on are not *syntax* when judged by the definition in Part (a). -}

Semantics: an illuminated "walk" sign means that it is (should be) safe for the pedestrian to cross; an illuminated red light means that vehicles should not pass; etc. {- Best to stay on examples like this where you can point to clear "meanings" rather than more abstract examples (related to intentions, for example) or to speculation about how the crosswalk system might be implemented. -}

The child and tiger scenario:

Syntax: the written and spoken word, as well as the picture, are all examples of syntax. {- The act of showing these items to the child would NOT meet the definition of syntax, and neither would the paper where the text or picture is drawn. -}

Semantics: the items here all reference the concept of a particular kind of animal --- a big cat with orange and black stripes, and that is the meaning of the various items of syntax in this case. {- You could also argue that the child's reaction confirms that the symbols mean the same thing to them too. -}

c) It is important to have a well-defined syntax for a programming language so that programmers know what notation that they will need to use when writing code.

It is important to have a well-defined semantics for a programming language so that programmers can accurately predict what their code will do / how it will work / whether it will work correctly.

{- Be careful to avoid confusing syntax and semantics in this question; it is legitimate to say that we need a well-defined syntax if we want to be able to give a well-defined semantics, but that does not explain the inherent importance of specifying syntax. -}

---

2)a) EQUIVALENT: [1:59-70]
We can prove that two expressions are equivalent by using logical equivalences (e.g., commutativity, etc..)

OR
We can demonstrate that they are equivalent by showing that they produce the same output on all input combinations; a simpler way to say this is just that they have the same truth table.

{- You would only need to give *one* of the answers above. Saying something like "show that the two expressions have the same semantics" is providing a definition of what it means for the expressions to be equivalent, but does not address the part of the question that asks you to show *how* you would determine this.

Unfortunately, it is not sufficient to say that you can determine equivalence by reducing the two expressions and testing to see if the results are the same: in general, the expressions may contain variables with unknown values. For example, A and NOT NOT A are equivalent, but neither one can be reduced to the other. It is also not enough to say that you would evaluate the two expressions with a given environment to see if they give the same result: in general, you need to evaluate the two expressions on every possible environment before you can be sure of an equivalence. -}

EQUAL:
Two expressions are "equal, ignoring irrelevant details of syntax" if they have the same abstract syntax tree. In this case, "irrelevant" implies that we won't worry about spacing, comments, etc... as well as parentheses (which play no role in Prop once we have built our AST).

An alternative approach (and an acceptable answer here) would be to compare token streams, concluding that two expressions are equal if they produce the same token stream. This would distinguish between expressions like NOT TRUE and NOT (TRUE) but still ignores spaces, etc. as we do when we compare ASTs.

{- It is important to distinguish between expressions being equal and the values that they represent being equal. TRUE and NOT FALSE, for example, will both have value True, but they are not equal as expressions (i.e., they are written in different ways).
-}

{- With two points allocated for this question, and two parts to the question, you can probably guess that you are likely to receive up to one point for each part. -}

b) It is useful to enumerate the different forms of expression as the constructors of a data type:

- To specify all of the ways that expressions can be constructed

- To specify all of the cases that must be considered when we define a function on expressions (via pattern matching).

{- My hope here is that you would be able to come to an answer along these lines based on the description of the Prop type [1:30-31], the examples of the vars [1:39] and eval [1:53-58] functions, and the examples in the labs, such as the eval and beval functions in Lab 4. -}

{- Note that there are two points allocated to this question, and also about as much space for an answer as there was for the previous part. You might take this as a hint that you will either need to give more than one reason, or else that you will need to give a more substantial answer. But if you can only think of one

thing to say, then don't waste time saying it twice in different words; you won't receive extra credit for that! And don't try to pad out a brief answer with details that are not relevant (or worse, that are not correct) because again you won't receive credit for that (and you might lose credit if your answer suggests a deeper lack of understanding). -}

c) {- This question is a reference to [1:71]; the four answers suggested there (only two required here) are listed below. -}

- abstraction: the ability to name patterns and structures to promote code reuse and manage complexity

- variables: for holding intermediate and shared results

- control structures: adding constructs for loops, conditionals, recursion, etc. to allow more flexible ways of describing the construction of circuits

- types: to classify values and ensure correct usage.

{- A mistake that some students might make here would be to talk about properties of *implementations* rather than *languages* [3:27]. For example, facilities to support input/output to files, debugging, static analysis, optimization, or error checking, might well be useful in a compiler/interpreter/analysis tool for Prop, but are not part of the language.

Adding support for precedence, grouping, fixity would not be a good answer here because the initial version of Prop uses a prefix notation, so there is no need for precedence (indeed, there is no ambiguity in the grammar).

Suggestions that involve adding new operations or types would also not be good answers here. Although these might be useful as part of a larger package, they are relatively small additions from the perspective of language design (i.e., in comparison to the items listed above). For example, adding support for an XOR operator, or for integer values, would not change the overall character of the language in fundamental ways. XOR, for example, can already be coded in Prop, and integer values would not be useful without some of the additional features listed above such as control structures, types, and variables. -}

---

3)a) [1-9][0-9]*

{- Regexp syntax on [2:34,47], example [2:49], [5:59] -}

{- The question isn't very clear about whether or not zero should be included, so I would actually allow both. (A suitable regexp that includes zero is 0|[1-9][0-9]*). The question very clearly

specifies "integer" values so do not interpret "decimal notation"
as meaning that your regular expression should allow for a decimal
point and a fractional component; those are not part of an integer
value. "Decimal notation" is really just another way of saying
"base 10". -}

b) Break this down in to pieces: Numbers 16-19, 20-79, and 80-87,
which gives us the following regexp:

```
1[6-9] | [2-7][0-9] | 8[0-7]
```

{- Look out for edge cases. It wouldn't surprise me too much if
some people wrote incorrect regular expressions that match the
numbers 15 and 88. But the question says "greater than 15", for
example, not "greater than or equal". In other words, be sure you
read the question carefully! -}

c) {- Derivations were discussed extensively in lectures
[2:55,60-73] and in the Week 2 Lab and homework. The hint here is
a reference to the kind of tree that we produced using the
derivationTree function, which was shown as one of the early
examples of using the Grammar Toolkit in Lab 2. -}

{- For ease of presentation, I'll draw the following tree
structure from left to right rather than top to bottom. You don't
actually need to draw a tree, but doing that means you don't have
to keep writing down the same initial portion in a lot of the
derivations, and that will likely save you some time. -}

```
   0 steps    1 step     2 steps
 1) P
 2)       ==> TRUE
 3)       ==> FALSE
 4)       ==> NOT P
 5)                ==> NOT TRUE
 6)                ==> NOT FALSE
 7)                ==> NOT NOT P
 8)                ==> NOT P AND P
 9)       ==> P AND P
10)                 ==> TRUE AND P    -- expand leftmost P
11)                 ==> FALSE AND P
12)                 ==> NOT P AND P
13)                 ==> P AND P AND P
14)                 ==> P AND TRUE    -- expand rightmost P
15)                 ==> P AND FALSE
16)                 ==> P AND NOT P
17)                 ==> P AND P AND P
```

There are 17 derivations here.

{- The hint in this question refers to derivation trees, but there
is nothing here about parse trees. Drawing parse trees for each

of the different derivation steps (instead of the sentences shown above) would take an unreasonably large amount of time and space. Drawing a single parse tree would suggest a deeper misunderstanding of the question. -}

d) The discrepancy is due to ambiguity [2:74]: Lines 8 and 12 and also Lines 13 and 17 each result in the same final string, but generate different parse trees.

{- A common mistake here might be to suggest that strings like TRUE and NOT FALSE have the same meaning. But when we are talking about grammars and derivations like this, syntax is all that matters! -}

---

4)a) Two parts required here: [3:23]

- For every valid input, the compiler should produce valid output ("validity")

- The semantics of the output should be the same as the semantics of the input ("preserves semantics")

{- Saying that the compiler must always "give the same output for any given input" is just saying that the compiler should behave like a function. That's a useful property, but not fundamental to correctness in the same way as the two properties above.

It would be easy to make a mistake with phrasing here: It is not accurate to say that a compiler translates source languages to target languages. Instead, a compiler translates the *programs* in a given source language in to (corresponding) *programs* in the target language. Details like this matter! -}

b) {- As the hint on this question suggests, you will not get any credit for a question like this if you just answer Yes or No; a more thoughtful analysis, demonstrating a clear understand of the notion of "compiler correctness", is required here. -}

A correct compiler will remove one source of errors in the programs that run on the autonomous vehicle (mistakes in translation), so it should increase our confidence a little. However, even if there are no bugs in our compiler, there is no guarantee that the software we are compiling will be bug free. If there is a mistake in the logic that is used to control the car's brakes, for example, then a correct compiler will ensure that the same problem is present in the machine code that runs on the car.

{- Although it won't receive additional credit, one could also point out that there are other issues to worry about beyond what was described above. For example: Is the hardware (sensors, actuators, etc.) free from bugs? Is the proof of compiler

correctness free from bugs? Is the compiler that we used to compile the correct compiler free from bugs? Are the other drivers on the road going to drive in ways that the software can predict/handle? A correct compiler is definitely a good things to have (one less thing to worry about), but that alone won't ensure the overall safety of the system. -}

{- Although the question mentions "you", it does still require a technically informed answer rather than a statement of your personal beliefs! It would not be acceptable, for example, for you to state that you will avoid the risks by choosing not to ride in an autonomous vehicle. (and if that is truly your position, do you know how many computers you might be relying on already when you drive or ride in a modern (non-autonomous) car? :-)] -}

c) Key idea: interpreters run programs, compilers translate programs [3:15-16]

Other factors to consider might include performance, interactivity, portability, support for large programs, developer tools/environment. [3:17-18,24-25].

{- The question is about making a choice between an interpreter or a compiler, so you should try to avoid factors that might apply equally to either one. For example, it is not necessarily true that compilers are better for error checking or that interpreters are better for testing; the first depends on properties of individual implementations, while the second might depend on what form of testing procedures you were using.

The question states that both the interpreter and compiler support the same source language, so you can tell that this is a question about implementations rather than languages. An answer along the lines of "The team should use Python rather than C because it is easier to use ..." would not receive credit because it misses the point of the question and the constraint that is stated in the parenthesized part of that question. -}

d) General goals of modularity in programming: [3:62]

- modular implementations can be easier to write, test, debug, understand, and maintain than monolithic implementations

- components can be developed independently, reused in other contexts, usefule as independent/standalone units.

{- An answer containing the content of either one of the bullets above would be sufficient for the one point allocated for this question. -}

An example demonstrating the benefits of modularity in the construction of a practical compiler system:

- The compiler pipeline, which splits the compilation task into
  multiple distinct phases [3:43-60]

- The Unix C compiler is implemented as a pipeline of compilers
  (ccp, cc1, and as) [3:63-66]

- The ability to mix and match different front- and back-ends that
  share a common intermediate language [3:69-70]

{- Any one of the three examples above would be a satisfactory answer by itself here: the question only asks for *a* specific example, so there is no need to provide more than one. I didn't actually go over the slides for the third example above in class, but it was also discussed on an earlier slide [3:21] and should be easy to understand as you read through the slides. -}

---

5)a) See [3:50] for diagram, [3:45-49] for summaries of each component.

{- Potential mistakes might include:

- Forgetting the names of phases (e.g., lexical analysis, parsing,
  static analysis). This is mostly just a matter of memorizing
  the component names.

- Getting the phases in the wrong order (e.g., putting lexical
  analysis after parsing). It is harder to make a mistake here if
  you understand what each phase does!

- Mischaracterizing the role of phases (most commonly, suggesting
  that static analysis catches syntax errors). Don't go so fast
  that you make careless mistakes, and be sure you understand how
  each phase contributes to the overall compilation process.

- Treating syntax analysis as a single phase (it is the
  combination of lexical analysis and parsing, and typically
  source input too). If you do that, you might run out of other
  phases to include, and you will definitely miss out on being
  able to describe the separate roles of the individual syntax
  analysis components.

-}

b) As described in the HW3 solution, any error in the input program should be detected in one of the front end phases. So the last step in the pipeline should not generate any errors in response to bugs in the input program.

On the other hand, it is still possible for an error to occur in the last stage: (1) if there is a bug in the compiler, causing an

internal error or inconsistency; or (2) if there is a problem in
the environment that the compiler is running in, such as not
enough disk space to save the compiled program.

{- The latter is one reason why I don't have to worry about which
of the code generation or optimization stages you might decide to
list last in your answer to Part (a): whichever one it is, it will
typically have to write the final program to disk. -}

---
6)a) procedural paradigm: programs described in terms of a
collection of procedures that can operate on a shared state and
call each other in a LIFO fashion. [5:27] {- Be sure to include
producedures in your answer; execution of commands or instructions
without that is just "imperative programming". -}

functional paradigm: Computing by calculating/evaluating
expressions. [5:9-21] {- Pure, first-class functions are an
important part of many functional languages, but if you only talk
about that then you'll be missing the important point that they
can also work with many other types of value. In other words,
it's not just about functions ... -}

object-oriented paradigm: Computations are performed by
collections of objects that invoke methods or send messages to one
another to request a service or deliver a result. Encapsulate
state, inheritance, dynamic dispatch are often key. [5:35-26]

{- When defining and distinguishing between different paradigms,
it is usually best to focus on the different views of computation
that each one provides rather than any perceived advantages or
disadvantages of each one. For example, an an answer that
characterized object-oriented programming as being "good for
modularity" or functional programming as being good for
"mathematical calculations" does not adequately describe the
paradigms: there are languages in all of these paradigms that have
good support for both modularity and mathematical work. -}

b) Python implementation:

```
  def getNumber(cs, n):
      i = 0
      while i<len(cs) and isDigit(cs[i]):
          n = 10*n + digitToInt(cs[i])
          i += 1
      return n
```

{- The key challenge here is to recognize that the getNumber
function can be implemented using a loop with an accumulating
parameter. An interesting detail here is the relationship between
the use of local state and looping in the Python code, and the
explicit passing of the full input and recursion in the Haskell

version. This exercises obviously requires some understanding of Haskell, at least to read and understand the supplied code (Labs 1, 3, 4, Notes on Haskell). Generating the Python version requires some familiarity with the basics of procedural programming in Python [5:68-75], which was actually covered in the labs rather than the lecture]. The requirement in the question that there are "no function calls" rules out the use of recursive definitions. Note also that the initial value of n must also be passed in as a parameter. -}

c) {- This is an open-ended question that I would grade flexibly accepting any reasonable answers that satisfy the prompt in the question. Honesty and openness would be appreciated/rewarded. -}

Possible examples of benefits might include:

- reinforce understanding of fundamental concepts

- new ways to think about programming and problem solving

- preparation for learning new languages

Possible examples of obstacles might include:

- unfamiliar syntax

- difficulty "unlearning" old habits

- overcoming biases

- taking new ideas on board

- adapting to new ways of thinking

- struggling to find or navigate documentation

- ...

{- There are a total of two points available, and I would expect to assign one for a response about "primary benefits" and the other for a response about "obstacles". If you wrote an extensive answer but only addressed one of these areas, then you wouldn't get the point for the other one ...

Do remember: (a) you're not expected to be an expert Haskell/Python/... programmer by the end of this class; we're only scratching the surface; and (b) if you get stuck, with Haskell/Python/... or anything else, please reach out for help! -}
