1 Ground Rules for Discussion

The following ground rules will ensure that our discussions this semester remain productive and enjoyable.

• Monitor how much you're talking: try not to talk too much nor too little
• Monitor the quality of your comments – are you contributing something new to the conversation?
• Respect others by not interrupting
• Actively listen – e.g., look at the person speaking, ask clarifying questions, build off others’ ideas
• Do not disparage another person or their ideas – e.g., avoid disparaging language like “stupid” or “wrong”
• Respect others by not abusing technology during class time

2 How To Prepare for Discussion

Being prepared for discussion means that you have done the following:

• Completed the reading assignment
• Answered the written response question(s)
• Posted a discussion question
• Printed a hardcopy of the paper(s) to bring to class
• Spent time reflecting and thinking about the reading so that you have ideas and questions to share with the class.

You are expected to contribute to each discussion even if you are not the discussion leaders. The following questions can help you reflect more deeply on the reading:

• Why were these papers chosen? What is their purpose? What is their value?
• How is the technology described in the paper similar to today’s technology? How is it different?
• Is the problem the paper is addressing still a problem today? To what extent?
• What assumptions are made in the paper? Do you agree with these assumptions or not?
• Historically, what was the original purpose of the technology? How was the technology received at the time? How is it perceived now?
3 Discussion Leaders

Discussion leaders should first read the papers on their own (see above). Leaders will then meet together to
discuss the papers and to create a plan of action for leading discussion. At this meeting, someone should
type meeting notes. You will submit these meeting notes to me as part of your grade for this activity.

The meeting notes should be in the form of “minutes”. That is, they must include:

• The time the meeting started and the time the meeting finished
• The names of those in attendance
• A bullet-pointed listing of the ideas you discussed from the papers with each idea attributed to its originator.
• A bullet-pointed listing of the ideas/game plan for leading the in-class discussion

The meeting should take between 1 to 1.5 hours and should contain two parts: your discussion of the
readings and then your discussion of how you want to lead the in-class discussion – i.e., what questions do
you want to ask, what are the main ideas you're hoping people take away from the discussion, are there any
activities you want to do with the class, any videos you want to show, etc.

It should be clear from your meeting notes who contributed which ideas – i.e., you should take care to
attribute ideas to people: “Person X noted how numbers and operators are tied together in ancient Babylo-
nian algorithms whereas Lovelace makes a big deal out of keeping them separate in the Analytical engine”
or “Person Y had a great idea for an activity we could do in-class to show why base 60 is so convenient to
work in”. I've included an example meeting note from a previous year (attached as an appendix to this
guide). The discussion leaders had an 80-minute meeting about the readings, and formulated a “plan of a
attack” with solid discussion questions.

Submit these minutes to me by Friday before the Tuesday discussion. I may or may not have feedback
for you before your discussion day.
**Min. 1-15:** We began by answering some of the general discussion questions provided on the discussion guide. Nicole noted that this paper bridged the gap between the theoretical developments of programming languages and models to how we implement these in classes/work today. She also noted that it’s easier to understand what it takes for languages to be improved upon if we know the historical significance of past, seminal developments. Lucca found it interesting to find the similarities and evolution of older and modern languages/models. How would things have developed differently if there was more communication among programmers and institutions? Jess noted that the publicity or popularity of a work was largely dependent on the institution it was associated with (i.e. military institutions were more likely to keep works classified).

We then shifted to talking about persistent problems the paper highlighted. Jess brought up abstraction. That is, when designing a language or model, what is the programmer expected to know (e.g. mathematical notation, machine language). Nicole and Lucca brought up what was written on p55-56 where there was a shift toward significant abstraction — novices should be able to program without knowing much about the computer (i.e. machine) at all. From there Jess also noted a growing divergence between computer science and mathematics (e.g. Math/CS department versus a math department and a CS department). Nicole brought up that even within computer science there are wide-ranging topics with little overlap (e.g. computer engineering versus computer programming).

Lucca brought up how much thought was put into operating symbols and the variety of them. Jess compared the differences between Hopper and Glennie’s comments on the use of mathematical notation. Nicole noted that there may be down-sides to abstracting too far into human language.

**Min. 16-30:** We moved on to discussing things that are of note. Lucca found it interesting that the proposed languages were so diverse (esp. The flow chart). Going off of that, Jess found it interesting that so many of the included languages/models were hypothetical and not implemented. We talked a bit about how flowcharts today are used for brainstorming (high level of abstraction), but they were originally a lot more complicated and actually included computations. Jess wondered why it was that data structures were not an emphasis for a very long time (or at least not nearly as important as they are today). Nicole noted that Zuse was really ahead of his time in this respect. She wondered why, on a machine level, it’s important to have strongly or weakly typed languages (e.g. Java versus Python)? Does it help the programmer or the machine more? Lucca and Nicole discussed that it may have to do with how data is represented at the bit level. Jess and Nicole pondered over how Strings would be compiled into machine code (e.g. using ASCII).

**Min. 31-50:** We began to formulate possible discussion questions as well as some questions we had ourselves (i.e. things we need clarification on).

- (N.) What are some similarities in syntax or structure you noticed? Between what languages?
- (J.) Why did the author include so many hypothetical languages and models? Why not just use what was implemented?
- (N.) How did the state of the art (i.e. machine) limit what languages were most promising?
- (J.) Coming from a a liberal arts school, it’s important to be able to express how this type of education alters our approach to computer science. How do you think the professional backgrounds of the people noted affected their foresight? For example, Zuse had a firmer grasp on data structures than other engineers; why do you think this is so? (interdisciplinary effects on CS)
- (N.) What are some of the factors that influenced the necessity of implementing data types and structures?
○ (N.) Do you think the TPK algorithm was a sound pedagogical choice?
  - (J.) What criterion are good for comparing languages?
  - (L.) Are there any algorithms that can be expressed in all or many languages? Or are we
    working under the assumption that languages should be universal (in that they can
    express any algorithm)? Are we justified in making this assumption?
  - (J.) What is a subroutine call? A loop? A library (e.g. import util)?
  - (J.L./N.) Difference between compilation and object-code time (p35)?
  - (J.L./N.) Modern differences between interpreter and compiler. Do these definitions align with
    the older ones?

- **Min. 51-80:** We spent the last 15 minutes dividing our discussion and related questions into 5 major
  themes + an introduction. We came up with a short introduction to each theme and then cherry-picked
  discussion questions.
  - Introduction: Begin with talking about the characterization and value of this paper. We think the
    characterization is historical and theoretical.
    - What is the significance of going over the historical and theoretical aspects of CS (for
      the purpose of capstone)?
    - Why did the author include so many hypothetical languages and models? Why not just
      use what was implemented?
    - Do you think the TPK algorithm was a sound pedagogical choice?
  - Abstraction (role of programmer): Over time, the base level of knowledge a programmer needed
    to have before making contributions has lessened (p56). For example, one of a programmer’s
    main concerns used to be numerical analysis; today, we generally take mathematical accuracy
    for granted.
    - How would good or clean code differ from before and today?
    - Are there advantages to making it easier for people to code (i.e. without having to know
      about machine language or math)?
    - What’s the importance of human readable code (e.g. comments)?
  - Abstraction (role of machine) [p23]: “...they could not imagine wasting any extra computer time
    for something a programmer could do by himself.” attitudes seem to have shifted — today we
    don’t want to do anything that the computer could do by itself. There’s also been a shift
    regarding machine use (e.g. academic, military, commercial, enterprise, home, etc.)
    - How did the state of the art (i.e. machine) limit what languages were most promising?
  - Divergence (math & CS): There’s been a growing divergence (at least in the paper) between
    math and CS. This is especially noticeable in discussions on whether or what mathematical
    notation should be used. We see this at our own university (i.e. combined math/CS department).
    - Is this divergence exaggerated? Is it justified?
    - How do you think the professional backgrounds of the people noted affected their
      foresight? For example, Zuse had a firmer grasp on data structures than other engineers;
      why do you think this is so? (interdisciplinary effects on CS)
  - Data type & structures: The only language to implement the use of data structures is Zuse’s
    Plankalkul. With the development of languages, our perception of the “important concept of
    hierarchically structured data” [p12] that none of the other languages we examine in the paper
    address has changed, and today data structures are a necessary part of our programming
    education.
What are some of the factors that influenced the necessity of implementing data types and structures?
- What are some similarities in syntax or structure you noticed? Between what languages?
- What are some of the factors that influenced the necessity of implementing data types and structures?

Democratization: Today, open-source programming and transparency regarding data and code is highly regarded. However, in the paper there were several instances where papers went unpublished for a variety of reasons (e.g. classified files, low publicity, the ideas weren’t popular or beyond the possibility of implementation, etc.). Within the CS field, do you think communication is more open or opaque? In what sense?

Related Material:
Differences between interpreter and compiler

<table>
<thead>
<tr>
<th>Interpreter</th>
<th>Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translates program one statement at a time.</td>
<td>Scans the entire program and translates it as a whole into machine code.</td>
</tr>
<tr>
<td>It takes less amount of time to analyze the source code but the overall execution time is slower.</td>
<td>It takes large amount of time to analyze the source code but the overall execution time is comparatively faster.</td>
</tr>
<tr>
<td>No intermediate object code is generated, hence are memory efficient.</td>
<td>Generates intermediate object code which further requires linking, hence requires more memory.</td>
</tr>
<tr>
<td>Continues translating the program until the first error is met, in which case it stops. Hence debugging is easy.</td>
<td>It generates the error message only after scanning the whole program. Hence debugging is comparatively hard.</td>
</tr>
</tbody>
</table>