CMPSCI 377 Operating Systems **Spring 2022**

Lecture 4

Lecturer: Emery Berger *Scribe: Huy Hoang, Sam Stern*

4.1 Clarifications on Locality

We were discussing locality using a linked list as an example. This is a bit of a red herring.

4.1.1 Example

Given a sufficiently large number of objects that you have to allocate, a better example to describe the difference between "good locality" and "bad locality" is how you choose to allocate these objects. If you opt to scatter these objects around the entire memory space of the program, the locality is generally considered to be bad. However, if you were to place them next to each other in a contiguous block of memory (such as an array) the locality would generally be considered to be good.

4.1.2 Types of Locality

4.1.2.1 Spacial locality

This is the property that the example was exploiting. A program has good spacial locality if, given some object access, it is very likely that adjacent objects are accessed. This is a function of how things are laid out in memory.

4.1.2.2 Temporal Locality

A program has high temporal locality if, given an object access, it is likely that the same object will be accessed soon. This is a factor of recency of use and is a property of the program

Q: *I got an interview question about the tradeoffs between arrays and linked lists*

A: We will return to locality later.

Today, we are discussing ALGOL-60 and LISP

4.2 ALGOL-60

ALGOL-60 an attempt to put programming languages and semantics of PL on a firmer foundation. This happened in the light of then-recent innovations in the understanding of formal languages. Dijkstra wanted recursion add to ALGOL60 thus he did it himself without permission.

4.2.1 The Chomsky Hierarchy

Noam Chomsky at MIT showed that languages form a hierarchy. These are:

- Regular
- Context-Free
- Context-Sensitive
- Recursively Enumerable

4.2.1.1 Regular Languages

These are powerful, but many things cannot be represented

Q: *Is it recursive or recursively enumerable?*

A: Cannot answer this question on the fly

Answer from Emery on Slack

> In response to a question regarding the Chomsky hierarchy (https://en.wikipedia.org/wiki/Chomsky_hierarchy) about where recursive fits in, which I wisely refrained from answering on the fly - "Note that the set of grammars corresponding to recursive languages is not a member of this hierarchy; these would be properly between Type-0 and Type-1."

Regular expressions are "fine" but unmaintainable, though they can be useful for certain tasks.

An example of a regular expression is one that matches any of the characters in the English alphabet, which would be marked as `[a-zA-Z]`.

Regular expressions also have several control characters. Most notable are the "kleene star" (`*`), which matches 0 or more of the previous token, and the plus (`+`), which matches 1 or more of the previous token.

However, regular expressions have some ambiguity. One of the major questions in parsing a regular expression is how far ahead one matches and whether the match is greedy or "ungreedy".

As a response to these inconsistencies, the "Perl Compatible Regular Expression" library was implemented, which aimed to match the features of Perl. This more-standard form was such a significant engineering feat that many other "implementations" of regular expressions have just been PCRE ported into other languages.

However, if you tried to use regular expressions to parse a language, you would quickly run into problems. The most popular example is that regular expressions cannot be used to match only strings that contain balanced parentheses since **regular expressions cannot count**, furthermore; parsing was computational expensive at given time.

Thus, regular expressions/languages are only suitable for lexing, or lexicographic analysis/tokenization.

An example of lexicographic analysis is lexing a theoretical program of the form "function foo(a) {", which could theoretically be lexed as `F FUNCTION_NAME("FOO") ARGLIST[ARG("A")]`

Lexicographic analysis was first formally done in ALGOL-60! FORTRAN's lexing was ad-hoc and LISP did not really need a powerful lexer.

4.2.2 Parsing

Program information has to be structured in a computer-readable form. For maximum expressiveness, one could feasibly use a language high up in the hierarchy, such as a recursively enumerable language. However, in the worst case, these can take huge amounts of time and space to parse.

The innovation of ALGOL-60 is that, rather than trying to make the language as expressive as possible, the designers wanted to make it as easy to parse as possible.

It is very easy to efficiently parse context-free grammars in only a single pass with very little additional memory. Context-sensitive grammars are inappropriate because, in the worst case, one could feasibly have to read the entire program into memory in order to determine the nature of the first token.

With the memory and CPU restrictions of computers at the time, there was a deep need to read tokens as they come in and not store too much extra information. In particular, because of the very low speed of storage devices, a parser should not have to scan the program an arbitrary number of times. This is called backtracking, and it is to be avoided at all costs.

In the modern day, compilation space and time is still an issue. Emery recently worked with a C++ build that consumed 128 GB of disk space and took 8 hours.

As it turns out, the parser wasn't the problem in Emery's case, but rather the linker was.

4.2.2.1 Interlude: C/C++ Compilation

When you have a c program, you might want to call a function defined in a separate library, such as `printf`. What you don't do is combine all object code, so what you do instead is have pointers that are eventually resolved to libraries.

In UNIX, libraries are contained in `.a` or `.so` files. `.a` files are static libraries, meaning they are likely meant to be compiled a binary. `.so` files "shared objects", meaning that they are loaded at runtime.

During the link step, `.a` and `.o` connect the files. In the case of Emery's build, the link step creates a colossal number of intermediate files.

Q: *Is there a better linker out there?*
A: The one Emery is using, `lld`, actually *is* the better one.

4.2.2.2 Continuing with Algol

In terms of submission-to-executable time, compiling ALGOL took a very long time. An ALGOL programmer working on mainframes, programmers had to

1. Write the program
2. Turn the program into punch cards
3. Give punch cards to operator
4. Wait for operator to run code and for the mainframe to produce the output
5. Wait for the operator to take the readout and give it to programmer

This is what Emery had to do when programming ALGOL-60.

The modern equivalent to this is "Clusters", which have the following workflow.

1. Give executable and context to cluster, put in queue
2. Eventually, the cluster schedules the job (in the meanwhile, you're waiting)
3. Eventually the job is done, and the submitter will get mail

This means that, when you make a mistake (because you will), you only figure out later.

4.2.2.3 Returning to parsing

As stated before, tokenization can be done with regexes! However, these regexes do not "know" the syntax itself, it requires a higher-level formal language to be able to parse.

FORTRAN was a context-sensitive grammar. Its parser just went through the program with if-statements and branches. For all intents and purposes, it was an ad-hoc state machine. By modern standards, this is considered to be a bad thing.

ALGOL introduced use of Backus-Naur form, which lets you write the grammar as

```
expr ::= '(' expr ')' | number | unop expr | expr binop expr
```

With grammars specified like this, you can simply look at the next token to figure out the construct that you're dealing with rather than the entire program. The number in grammar names like `LR(1)` or `LALR(1)` refers to how many tokens of lookahead are needed to make a decision on a construct. These are very easy to parse.

The question of parsing was fully established as an area of study starting with ALGOL, and from that time

This was until recently considered a solved problem. Recently, HTML, Javascript, and JSON have made reopened the field. In theory, there is a grammar for HTML, but no HTML parsers are actually correct as per this grammar. Even though a validation tool is available, most websites are badly formed.

Since the fundamental purpose of modern web browsers is to be resilient, most HTML parsers are designed to parse as much as possible before giving up.

For example:

```
1 <p>
2 <font color="RED">
OH NO
-1 </p>
```

We know that this is ill-formed, since there is not `</font>`, but every modern HTML parser will implicitly add a `</font>` (or, more generally, close level 2 when the parser sees a -1 tag before a -2 tag)

4.2.2.4 Interlude: injection attacks

XKCD made a comic about this!

One way an injection attack can occur is if SQL statements are dealt with incorrectly. An example is if you had a statement like the following:

```
sql.search(‘SELECT * FROM students WHERE LASTNAME=‘ + lastname)’
```

If a program were to naïvely take user input and jam together with query string, then you can construct malicious query

An example is if an adversary were to send a `lastname = "; DROP TABLES;"`.

Another type of injection could occur if raw data containing angle brackets were written to a webpage (which would then render it as HTML)

4.2.2.5 Returning again to parsing

Should this line compile?

```
int c = c;
```

Most of the class says "no"

A context-sensitive grammar would be able to specify that the names on each side of the `=` must be different. However, this type of malformed program is not handled in most parsers at all. People designing parsers largely favored the "efficiency" side of the tradeoff space.

However, the grammars that are easiest to parse are not the best in all situations.

Some early parsers used context-sensitive grammars, and recently (and without historical knowledge) people invented a context-sensitive parser called `packrat`

`packrat` is actually a fairly efficient way of compiling context-sensitive grammars. In the common case, parsing takes linear time. In the absolute worst case, it requires cubic ($O(n^3)$) time.

One might not care quite so much about efficiency when there is one large compiling machine compiling for multiple targets, so it might be a use-case for more expressive but less efficient grammars.

One reason is that, since compilers are not directly user-facing, they can afford to be slightly slower.

Q: *With the tradeoff between expressiveness and ease of parsing, why not compile something slightly harder to parse but more expressive on a separate computer?*

A: This is moderately common practice, especially with building for architectures other than the one the programmer is on.

Q: *Is there a time when build farms came into common use?"*

A: Building remotely has been in use since programs have existed. For the first part of computing, personal computers simply didn't exist. All computing, including compiling, was done on a central machine or a mainframe. Whenever anyone could work directly on a computer, it was a terminal to the mainframe.

When PCs came into existence in the 1980s, people wanted to do more compiling on their personal computers because, even though the actual compilation may have been slower, the end-to-end time of getting a compiled program from source code was faster because they didn't have to deal with the time-sharing of mainframes. At that point, mainframes were increasingly used for more business-critical functions.

Since then, the standard has swung back and forth from compiling programs on the sorts of computers they were meant to be run on and compiling p;rograms on other computers.

For instance, people do not compile mobile applications on phones, but rather use a desktop or a laptop and test on a simulation.

For IOT devices, programmers do the same thing.

For the original Mac computer, programs could only be compiled on the original LISA, which (even in the 1980s) cost 10,000 dollars.

Clearly, in many case it can be hard to program on the device itself.

Another modern example is how Google has large render farms and devotes many resources to making development easier. This varies significantly by the industry, since non-software companies might not dedicate these resources.

ALGOL-60 went down the path of formalizing things. They divorced the specification from the implementation so that, in theory, anyone could implement a compiler from the specification.

Until 1977, this was not the case with FORTRAN. The question of "what is FORTRAN" basically amounted to "What is the behavior of IBM's FORTRAN compiler?"

Worth noting is that the behavior of different versions of the same programming language can be radically different. godbolt.org can compile programs in different C versions and compare the output.

FORTRAN, like C is now, was defined by its compiler. ALGOL's behavior is more or less defined by its specification. The idea of ALGOL was to be able to write programs that one could largely understand what it does without compiling it.

ALGOL-60 is specified in both syntax and semantics and, in many ways, was designed as a successor to FORTRAN.

Hoare said: "ALGOL was an improvement on all of its predecessors and most of its successors"

4.3 LISP

LISP came from an entirely different heritage– it emerged from logic itself. LISP is arguably just an executable lambda calculus. LISP's goal was to be a language for AI, which was (at that point) all about logic and logical relationships.

4.3.1 Interlude: AI

There was a startup group called Cyc in Austin whose self-stated goal was to encode all of human knowledge. They employed thousands of domain experts to encode their knowledge. For instance, an expert on keys might write information on church keys, skeleton keys, and keycards.

(For context, the creater of Cyc was named Lenat)

In the Hacker's dictionary, by Eric S. Raymond, the "microlenat" was defined as the "unit measure of bogosity"

Cyc had a colossal amount of money in 1980s and 1990s.

4.3.2 Returning to LISP

LISP was fundamentally a language about symbolic computation, whereas ALGOL and FORTRAN were about numerical computation.

LISP had recursion, which was needed since lambda calculus was basically just logic. LISP also had symbolic computation. The combination of these let LISP essentially be defined in itself, called "Bootstrapping"

The idea in LISP was to implement a few basic operations. With very little effort, one could produce a LISP interpreter by just implementing a few functions, most notably "eval"

In LISP, programs are themselves data. For instance:

```
(cons 1 (cons 2 NIL))
```

represents the list `[1, 2]`

In this way, LISP is somewhat opaque– one can set anything as a string and call `eval` on that strong.

This actually eliminated parsing as a problem.

Emery worked as a LISP programmer professionally for a time! He worked on LISP machines, which were computers that were built to (allegedly) run LISP faster than other computers could. These were invented a few years before Moore's law really started being significant.

4.3.3 Outerlude: Moore's Law and Performance Measurement

Q: *Did any single-language machines really keep up with Moore's law?* A: Most machines are essentially single-language machines designed to run C.

By and large, this is caused by a vicious cycle. Manufacturers were in competition and wanted to claim that their hardware was faster than their competitors'. The question that naturally arises is "faster on *what*?"

In the late 1980s, a consortium came together to develop benchmarks in a suite called `spec`

The problem is, if "being faster on X" is the target, then the thing being measured will become faster on X, but perhaps nothing else.

Recursive fibonacci used to be a standard benchmark, which is a bad benchmark because there are very few 3-line recursive functions. While recursive fibonacci runs in exponential time, there is actually a closed form expression for it.

In 1985, some researchers attempted to make a set of benchmarks "better than fibonacci" and called it no-fib. Among them was the n-queens problem. Haskell still uses these benchmarks.

Fundamentally, the idea of "faster" and benchmarking in the first place is incredibly difficult.

Q: *What are your thoughts on Jazelle, an extension to ARM that ran JVM bytecode natively?*

Thoughts: Every microarchitecture is a composite of many special purpose processors. For instance, the Intel 8088 (inside the IBM PC), had `ADD` and `MUL` as built-in instructions, whereas the `SQRT` instruction trapped to software!

Separately sold was the 8087, a special purpose math coprocessor. It had high-speed implementations of mathematical operations, such as trigonometric operations and SQRT.

IBM shipped its PC with an open socket for its math coprocessor, which could be bought if people needed it.

In the present day, these are called "accelerators". However, lots of things have just gone into modern general purpose processors, such as vector processors and virtualization.

For instance, the original 8088 did not have any sort of memory virtualization. A separate thing was later created called the "memory protection unit", which was later incorporated in the i386 architecture.

Further, Macs used Motorola chips, which did not have memory protection until the late 1990s.

Eventually, though, almost everything was just crammed into main processors. GPUs are modern accelerators– an integrated GPU is an example of an on-chip accelerator.

Fundamentally, the question of what is "special purpose" changes over time. Originally, graphics simply didn't exist, and graphics processors started as a niche interest. When games started making money and when GUIs started becoming widely used, graphics processors moved from being special purpose to being of general interest.