Topics:

* Regarding the interpreter project: Be able to answer questions about what we did at each stage so far, and why we did it. Be able to write variations of aspects of the project by writing C code, modifying provided C code, or describing what your technique might be. Specifically, be able to do the above regarding questions about the linked list, talloc, and tokenizer portions of the project. (Likewise for the warmup vector assignment.)
* Be able to reason about C memory, including memory allocation, memory leaks, strings, and valgrind errors. Look at detailed C pointer-related code, and be able to accurately predict output.
* Be able to distinguish between a compiler and an interpreter, describe how each works in "classic" simple scenarios as well as in "hybrid" modern scenarios (such as with DrRacket and CPython), and reason about tradeoffs between them and ramifications of using one vs. another.
* Be able to motivate what purpose BNF serves. Be able to precisely produce BNF to describe a language that I describe in words and/or examples, as well as use derivations in order to show if a particular "sentence" is valid in a particular language. Be able to draw parse trees, and be able to demonstrate if a grammar is ambiguous. Be able to discuss the merits of different parse trees that describe the same sentence from the perspective of which is more desirable.
* Be able to distinguish clearly between tokenizing and parsing.
* Evaluate lambda calculus expressions, and define well-known logical or arithmetic operations on Church booleans or numerals. Students should be able to work through problems of the same style that were done for the lambda calculus homework assignment.
* Describe, trace, or otherwise answer questions about the "Scheme-specific parsing algorithm" we'll be using.
* Be able to explain what LL and LR parsing are, and how they are distinguished from one another.

Parsing

ID\_LIST::= ID, ID\_LIST\_TAIL (i.e. IDT)

ID\_LIST\_TAIL::= , ID ID\_LT

;

EXP::= ID-ID

This is LL1 because we only need to look 1 token ahead to know if it’s a syntax error or what we need to do next.

LR grammars we build from the bottom up.

Languages don’t have to be parsed in any order – that’s why both LL and LR parsers are valid.

, ID EXP IDT

LR does IDT and sees that it turns into a semicolon before doing the middle EXP

For LR:

* ID\_LIST
* ID IDT
* ID, ID Exp IDT
* ID, ID EXP ;
* ID, ID ID-ID

As soon as we see something we expand it, so it’s the left first most. But because our algorithm works in reverse time, the left thing is first for our algorithm but last for the derivation, so it’s right in the derivation.

For LL, it appears we expand the ones that are on the left first. LR we also do the leftmost first, but the order is reserved so in our derivation it looks like the rightmost stuff is first.

To get order of operations in a language w/ addition and multiplication:

EXP:= MEXP \* MEXP

EXP:= EXP = EXP

MEXP:= NUM || MEXP \* MEXP

MEXP:= EEXP \*\* EEXP // \*\* is exponentiation

This forces us to choose which one is next (but this language is not LL1 parseable and would be difficult to parse)

When making the grammar, start with the expression or the leaves and think about the recursive step/how to make an infinite sequence, and how it terminates, and that it meets the restriction

Lambda Calculus:

* Free variable: x is free in a if a = x
* X is definitely free in a lambda y.a if x != y and x is free in a
  + Parameters can never be free
* Expressions: abc if x is free in any of these then it is free in all of them
* e.g. lambda x (lambda y.y) z) // y is not free because it’s bound to the function, but z is free
* lambda x.lambda y.x // x is the same as x, so x is not free
* Bound variable means that there’s still stuff to it
* Free variable is like a rock – you can’t do anything to it
* Bound variable can be changed - there’s work to be done
* How do we now know if substitutions can be made?
  + Lambda y.y y =?= (lambda x. (lambda y. xy)) y

= (lambda x. (lambda y­1.xy1))y2

!= lambda y1. y2 y1

­­Study:

* Predecessor function can be used to subtraction
* May also want to look at comparison (<, >)
* Or/and that take in 3 arguments
* Nand/nor
* Plus 2
* Y combinator – just think about it in a surface level but not likely to be on the test
* Also write down all the ones from the homework – multiplication, church numerals, etc.

BNF

* Bnf is context-free grammar, a way of expressing a language and what’s syntactically legal in it
  + Defines what’s legal
  + Provides framework for extracting a structure (a parse tree) from that language
* Since we need a parse tree, BNF is what’s legal and illegal in that language

C

* What are memory leaks? We allocate memory for different processes/programs. When we allocate memory, Valgrind remembers where you allocated that memory and checks to see if you free it so something else can use it. If the program only runs for a short time it’s fine, since when it’s over the OS overwrites all of it. If we leave the program open, if it allocates a little bit of memory and then we don’t free it, eventually all of the program’s allocated memory will be stuff we don’t know about, and we’ll run out of memory for that process.
* Memory leaks are memory that are not free and not accessible at all.
* Other Valgrind errors:
  + When you rely on uninitialized values
    - Int \*p; if (\*p) {…} //this is bad because \*p is uninitialized, so it will behave differently on different runs or seg fault
  + When you don’t initialize memory or don’t free it
* Can practice writing pseudocode for the programs you wrote for homework
* \* says follow the pointer, and & tells you what address it’s at
* E.g.

Int a = 1;

Int \*b = &a;

Printf(“%i”,a); // prints1

Printf(“%i”,b); //prints memory address of a

Printf(“%i”,\*b = 1);

* When to pass in \*a vs a?
  + It depends on the type
  + If it’s foo(int a) then it won’t modify anything outside of this function
  + If it’s foo(int \*a) then we’re passing in a memory address, so we can have side effects and change memory outside of our function