cs-131 notes

yash lala

2020 winter semester

TODO: fill in and reformat this notes files

syntax, semantics, and grammar

syntax: form of a language. independent of the meaning of a language (which is referred to as the semantics). both are independent to a degree; for example:

> "colorless green ideas sleep furiously" — noam chomsky syntactically, but not semantically valid.
>
> "ireland has leprechauns galore" — paul eggert not syntactically valid; galore is an adjective. phrase survives as a vestige from when english absorbed some of gaelic.
>
> "time flies" syntactically *and* semantically ambiguous. depending on which word is the noun and which the verb, we're either following flies with a stopwatch or catching up with friends.

reasons to pick one syntax over another:

- inertia: most important force (eg. we're stuck with PEMDAS in C, etc).
- simple + regular: has very few + intuitive rules.
- it's "readable": `//\.(.\.[_-]*)/\\/\//\/` is unreadable and kinky.
- it's "writable": eg. `c++` is easier to type than `c=c+1`.
- it's redundant: some checks (like matching closing parentheses instead of optional parentheses) help us avoid common mistakes.
- it's unambiguous

we can define a system of syntactical rules and components (known as a grammar). a grammar consists of:

- tokens: a member of a specified finite set (eg. ASCII characters or words).
- strings: finite sequences of tokens.
- language: set of strings.

sample language: $a^n b^m$, n 'a's followed by m 'b's

tokens and tokenizers

tokenizers serve to break the input (eg. a byte or character stream) into distinct tokens (eg. `int varname = 8` -> `int`, `varname`, `=`, and `8`). not every character has to be a token; whitespace and comments, for example, are often ignored.

tokenizers are usually 'greedy'; if a tokenizer runs into a token, it will keep eating characters until it finishes reading. for example, in C, `a+++++b` will fail; it's interpreted as `((a++)++) + b`. as the second `++` doesn't modify an lvalue, the compiler gives us an error. `(a++) + (++b)` is an alternate (and correct!) way to parse the expression, but C doesn't do it that way.

grammars

tokenizers usually break an input byte/character stream into distinct operators, identifiers, numbers, etc. the grammar may give certain words a special meaning; these are called *keywords* (eg. `if` in C). these keywords are usually *reserved words* as well. this means that we can't, for example, create a variable `int if` in C; we can't

create an identifier with the reserved name. not all languages choose to make keywords reserved words (this helps backwards compatibility), although the pattern is common.

usually, languages allow for 'identifiers' (such as variable names). these usually consist of

context-free grammars

1. a finite set of terminal symbols (tokens): eg. "dog" for english.
2. a finite set of nonterminal symbols: eg. 'noun-phrase' for english. these will map to other symbols later — 'noun-phrase' might map to "dog" or cat".
3. a finite set of grammar *rules*, each with

- a left hand side that is a nonterminal symbol
- a right hand side that is a finite sequence of symbols (either terminal or nonterminal).

4. a starting, nonterminal symbol.

for example, let's break down the previously-discussed email message header grammar into a context-free grammar.

LHS

TODO

ISO EBNF

symbol	definition
a, b	a concatenated with b
{a}	zero or more 'a's

ISO EBNF rules in ISO EBNF

(note that ... isn't part of the ISO EBNF syntax.

```
syntax = syntax rule, {syntax rule};
syntax rule = meta id, '=', definitions list, ';';
definitions list = definition, { '|' definition };
definition = term, { ',', term};

identifier = letter list - reserved word
identifier = 'a' | 'b' | 'c' ... '>' | '_' | 'a', 'a' | ...
```

problems with grammars

grammars are like programs; they have specified notations, and are subject to 'bugs' when being defined. for the following sections, assume a simple grammar syntax as follows: nonterminal symbols are capitalized, spaces delimit different symbols, `->` assigns a symbol to a particular RHS. `S` refers to our start symbol.

`S -> a S -> Tb S -> Sc`

what problems can we run into when defining a grammar?

1. we might use a nonterminal symbol on the RHS without defining it. technically, this doesn't invalidate the rest of the grammar; the containing rule is useless, so we can remove it without changing the possible valid parse trees.
2. we might define a nonterminal symbol without ever using the symbol. eg. `S -> a`, `S -> Sb`, and `T -> abc`.
3. we might run into a nonterminal that cannot be reached from the start symbol. for example, take

   ```
   S -> a
   S -> Sb
   T -> Ta
   ```

```
T -> c
```

while the grammar abides by rules 1 and 2, the last two rules aren't actually reachable by the start symbol.

4. rules might be duplicated (eg. `S -> a`, `S -> Sb`, `S -> a`).
5. your rules might include extra grammatical constraints that your grammar doesn't contain (your rules are too generous). let's model english, with noun phrases (NP) and verb phrases (VP).

```
# RULES
S -> NP VP
NP -> N
NP -> Adj NP
VP -> V
VP -> VP Adv
N -> 'dog'
N -> 'cats'
V -> 'prowl'
V -> 'runs'
Adj -> 'blue'
```

as per this, 'blue cats prowl' is a valid sentence (as it is). however, 'blue dog prowl' is not valid english, even though it's allowed by our grammar. our grammar doesn't take plurality into account. we can fix this by complicating our grammar, adding different categories for plural and singular nouns and then sorting our terminal symbols. however, each such category will double the number of rules in our grammar `NP` must be subdivided into `SNP` and `PNP`; this is a limitation of context-free-grammars. use something else to talk about this property (such as a language rule that's not in the grammar).

6. grammars can be ambiguous. this is not always obvious. take, for example, a grammar for arithmetic. we might be tempted to define an operation on an expression `E` as `E -> E - E`. however, this leads to problems; depending on the order in which we evaluate, say, `3 - 2 - 1`, we can either get `2` or `0`. to deal with this sort of ambiguity, it's often helpful to make our operator associative. using `E -> E + T` and `E -> T`, our grammar is parsed as `(3 - 2) - 1` (aka it's left-associative; operators group to the left). think about this; the same technique can be used to solve operator precedence problems. think about it (or get someone else's notes TODO). these problems are usually solved by complicating grammars. however, this makes the parse tree structure much more complicated.

programming paradigms

there are three main categories of programming languages.

1. **imperative** or **procedural** languages: eg. C. programs contain commands (aka statements) that tell the computer what to do. programs are chained together by sequencing them; they run in a particular order.
2. **functional** languages: eg. Ocaml, Lisp. programs contain expressions that the computer evaluates. if you program with these, you give up on dealing with program states, global values, and assignments (which are examples of *side effects*; side-effect-free (pure) functions should return the same result given the same input without modifying anything else).
3. **logical** languages: programs consist of predicates that add constraints to a solution that you want; it's the computer's job to find a solution on its own. if you program in this, you give up side effects or functions.

functional programming

20 years ago, google had to do big backend queries on a database in a distributed environment. C++ didn't scale up, so they implemented MapReduce. this more or less implemented functional programming's previous work as far back as the 60s.

functional programing aimed to allow for:

- clarity: without side effects, functions could be easily debugged (problems would stay localized). implementations would also 'skip over' the details, uncluttering programs.
- parallelizability: we can escape from the *Von Neumann bottleneck* (disparity between speed of processor and speed of memory accesses that limits how fast a processor can load its own instructions). as functions are without side effects (aka *pure* functions), they can always be parallelized with no interference problems. because of this, two arguments to a function can be evaluated in parallel with no problems; the ordering of program execution is determined by the calls and arguments. TODO write about **referential transparency** `let x = 5 in let y = x + 5 in x * y` can be subbed in with no mistake.

why call it 'functional programming'? thinking back to math, a function can be expressed as a *relation* (set of ordered pairs from a domain set to a range set). *functions* are relations in which no two ordered pairs have the same first element. *partial* functions don't cover the full domain set.

note that nowhere in the above definition did we specify functions had to be over numbers. in fact, most functional languages use **functional forms** (aka higher order functions) can take a function as an argument. this allows for more interesting programming.

the first mainstream functional language was ML (in this course, we deal with an ML variant, OCaml). why did ML take off (in some places)? ML included some features that were ahead of its time as a result of its functional mindset.

- **type inference**: compile-time type checking. types are usually checked statically, for reliability and performance (python, etc. works but is slower and has a possibility of a hidden type error). this means we don't have to write type names all the time; the compiler will figure the types out depending on the contents of the expression. this isn't bound to a programming paradigm (to a limited extent, it can be used in declarative programming as well), but it's much easier for functional languages.
- good support for memory management: because of its functional nature, memory management can be done more or less automatically; values are all immutable, meaning there's no real reason for `new del` or `malloc()`. garbage collectors can be implemented very efficiently.
- good support for higher order functions: TODO move higher order functions notes here

ocaml: every function is a mapping from a domain to a range. this means that every function takes only one argument; functions with two arguments **CURRYING**

what type should we make an empty list? need a placeholder for any type: 'a

functions are left associative

```
f x is how you call a function
f x y is really (f x) y; (f x) evaluates to a function, which is then passed
'y'

output syntax
value: domain -> range = type
```

as it turns out, even currying is syntactic sugar for what's really going on.

1. functions can be anonymous. `fun x -> x + 1`, often referred to as a lambda function.
2. when you define a function, you often use shorthand.

```
let f x = x + 1 (* is really shorthand for *);;
let f = (fun x -> x + 1);;

let cons x y = x :: y;; (* becomes *)
let cons = (fun x -> (fun y -> (x :: y)));;
```

patterns in Ocaml: used in `match x with` statements.

- `0`, `27`, `[]`, etc: all patterns like these match themselves.
- `_`: matches anything.
- any identifier `a`: " " (TODO: ???) and binds itself to that value.
- `[pattern1, pattern2; pattern3]`: matches all 3-element lists.

- `p1 :: p2`: matches a list of length >= 1.
- `p1, p2, p3`: matches all 3-tuples.

```
let ucons (h, t) = h :: t;;

let ucons p = match p with | (h, t) -> h :: t;;
```

`fun` is for currying

```
fun x y z -> x + y * z ;;
fun x -> fun y -> fun z -> x + y * z
int -> int -> int -> int
```

`function` is for pattern matching

```
function | 0 -> -1 | n -> n + 1;;

fun x -> x + 1;; (* use for currying *)
function | x -> x + 1;; (* use otherwise *)
```

function that takes the first element of the list in lisp: car

```
(* ocaml will complain about the below two *)
let car h :: _ = h;;
let car l = match l with | h :: _ -> h;;

(* some cases aren't covered! if the empty list exists, there could be a bug *)
```

optimizing recursive functions with OCaml

```
let rec reverse = function
  | [] -> []
  | h :: t -> (reverse t) @ [h]
```

this function runs in order $O(n^2)$. the list concatenation operator `@` has a runtime of $O(n)$, and we do it for every step in the list. this is abysmal; can we improve it?

we'll solve the more general case of reverse-and-appending two lists, and see how we can apply it to the rest of our problems. for an example, `revapp [3;9;2] [1;7;6]` becomes `[2;9;3;1;7;6]`

```
let rec revapp l a =
  match l with
  | [] -> a
  | h :: t -> revapp t (h :: a)
```

now, we use the `cons` (`::`) operator, which is effectively inserting into a linked list — aka $O(1)$. we can now rewrite our reversal function as

```
let reverse l = revapp l []
```

this makes use of a more general technique — an **accumulator**. an accumulator is an "extra" argument passed to a recursive function that 'accumulates' the part of the answer computed so far. in this example, 'a' stores the unreversed portion of the list (ie. the portion of the list we have to append at the end). we use this by gradually adding items from the front to the 'unreversed', 'stored' section.

we can make our code a little bit nicer by making a few modifications. rather than immediately match our function argument, we can use the 'function' keyword. keep in mind the 'function' argument is appended to the end of the argument list, so we'll have to reorder our arguments: `let rec revapp a l`. overall, we refactor into:

```
let rec revapp a = function
  | [] -> a
  | h :: t -> revapp (h :: a) t
```

```
(* this also lets us take advantage of currying! why bother passing
 * in an argument when we can just equate the two functions? *)
(* 'let reverse l = revapp [] l' turns into: *)
let reverse = revapp []
```

writing good functions

with functional languages, we want to keep our functions as general as possible.

```
let rec minvalhelper h = function
  | [] -> h
  | h :: t ->
        let r = minval2 h t in
        if lt h r then h else r

let minval = function
  | [] -> None
  | h :: t ->

  Some minval2 h t
        match minval t with
        | Some s -> Some (if s < h then s else h)
        | _ -> None (* not needed; will never happen *)
```

HOMEWORK 2 NOTES

parsing problems:

- recursion in the grammar. (alex hint: keep expanding until you run out of tokens)
- concatenation: `s -> a b c`
- alternation (logical or): grammars where there are multiple alternate rules for the same nonterminal symbol.

if we solve these, we're pretty much done.

- recursion: apparently comes for free with ocaml. functions can be made recursively. when dealing with complex recursion, don't worry about it. *trust* your subroutines work; work one level at a time.
- concatenation: apparently not too hard either. you can evaluate them in order, feeding the output of one into each other. `return (parseC(parseB(parseA(x))))`.
- alternation: the hardest one by far.

we can do conditionally:

y = parseA(x) if y = nil then return parseB(x) else y

but the thing is, if we allow parseA

in the homework, `make_or_matcher` solves the alternation problem

interpreting languages: java

early 1990s, Sun Microsystems believed that massively networked devices were the future (what we now dub IoT).

TODO rest

problems with C++:

1. software tended to be unreliable because programmers would use undefined hardware-specific features.
2. multiple CPU architecture
3. executables consisted of machine code, and were *big* (making them slow to download over the internet).

walked over to Xerox Palo Alto Research Center (Xerox PARC), and decided to steal an idea. at the time, Xerox was working on Smalltalk, an interpreted object-oriented language.

compiler: translates souce code to machine code, then executes the machine code. **interpreter**: very close to the source code level. 'runs' that code.

rather than purely deal in source code or machine code, smalltalk dealt with **byte codes**, compact representations of instructions to the 'abstract machine'. this solved the second and third C++ problem: the abstract machine was reimplemented as necessary, so code was portable, & the bytecode was much more compact than source code or executable code (because the instructions to the abstract machine were simply represented eg. 'add a, b').

Sun's new language family improved on this, fixing the first problem with C++ as well. because it was interpreted, the abstract machine could add runtime checks for undefined behavior (pointers, checked subscripts, etc). this slowed the interpreter, but it was slow anyways — the tradeoff was deemed worth it.

Java (one of Sun's resulting creations) improved on this by implementing easy support for multithreading. this caused it to take off, even though Sun executives were about to terminate the project.

bytecode performance could be improved using the better runtime profiling allowed for by interpreters. when the interpreter notices a method that's being used a lot, it will compile the specified bytecode, loading the resulting machine code when the method is called. this optimization method is commonly called **just-in-time** (aka **JIT**) **compilation**. this is obviously not portable; however, it's also not strictly *necessary*, as the bytecode can be executed on architectures where the interpreter isn't equipped with a JIT mechanism; it'll just be slower.

the interpreter will still have to be compiled for every different architecture. however, those who write the interpreter can be careful to only write portable C code, so the interpreter should still work. the bytecode interpreter will have to be manually ported over.

java

synchronization

1. semaphores
2. exchanger

optimization and the JVM

java memory model

all Java objects inherit from the toplevel `Object` type.

- `public Object()`
- `public int hashCode()`
- `public boolean equals(Object obj)`
- `public String toString()`
- `public final Class getClass()`: a bit special; a class object is a data structure that defines the Java class.
- `public void finalize() throws Throwable`: called by the garbage collector when an object has no reference to it. this is analogous to a C++ destructor; while they aren't needed for freeing memory (Java's memory is automatically managed anyways), they can still be used to close OS resources, etc. the default implementation doesn't do anything. it's deprecated; ideally, programmers should do cleanup code themselves rather than expecting the garbage collector to do it.
- `protect Object clone() throws CloneNotSupportedException`: can duplicate an object. is discouraged in Java; the use of `new` is recommended instead.

singleton pattern: a class that will have exactly one member object (will only be instantiated once).

types

a set of values, along with a set of common operations on these values. a Java class has instance variables (corresponding to values) and instance methods (common operations).

why are types useful? by classifying data into certain types, we can optimize programs (good for performance) *and* prevent 'misclassifying' data (good for reliability).

most languages distinctions between primitive types (built-in), and constructed types (defined by the library or app, usually composed of primitive types).

`NaN`, etc. returns false when compared to anything (cheap in hardware, doesn't "infect" the program with a growing number of `NaN`s, which is what happened if `3 + NaN = NaN`. why use this special value as opposed to just throwing an exception?

main uses of types:

- annotations: useful info while reading the program.
- inferences: allows you to infer the types of expressions by knowing their subcomponents.
- checking: can be either **static checking** (compiles and checks types before execution) vs **dynamic checking** (checks during execution). statically checked languages can defer type checking to runtime by using casts (assumes it works during compilation, but adds a runtime check).

type systems come in several broad classes.

- strongly typed: can't cheat the type system; types cannot be compared if not of the same type (is allowed to be overridden via casts, etc).
- weakly typed: types can be automatically converted as necessary (eg. C lets you compare doubles and integers, etc).

abstract types vs exposed types: - abstract: keeps the implementation secret. eg. classes with well defined interfaces, certain primitive types. - exposed: lets users fiddle with the implementation (eg. C structs). tight coupling between use and implementations.

type equivalence:

- **structural equivalence**: types are the same if the underlying data structures are the same. for example, in C, given `typedef long s` and `typedef long t`, `s + t` is a valid expression; we can interchange `s` and `t` in any context.
- **name equivalence**: two types are the same if they have the same name. for example, in C, given `struct s { double a; int b;}` and `struct t { double a;  int b}`, we *cannot* pass a `struct s` to a method that demands a `struct t`; the two structs are *not* interchangeable in any context.
- subtyping

in other type systems, you can compare types if one type is a subtype of the other. (eg. subclasses and superclasses). if t is a subset of u, then:

1. `T`'s values are a subset of `U`'s values.
2. `U`'s operations are a subset of `T`'s operations. (aka `T`'s operations are a superset of `U`'s operations; imagine inheritance and think about it).

polymorphism: principle that claims a function can accept and return many different types. we must be careful when doing so; conversions can often lose information (eg. double to int), so we must try and keep generality.

when doing **type coercion** (also known as ad hoc polymorphism) we'll also end up with some arbitrary cases. for example, the C '+' operator is polymorphic — when given an unsigned int and an int, the unsigned int will dominate, converting the other operand.

if we compare `int i = INT_MAX` (aka 2^31-1) and `float f = i`, we won't get the mathematically correct answer if we do `i < f`. converting the integer to a float won't be precise for large integers (because the float uses exponential representation internally, it'll "round" the integer to a nearby number, in this case 2^31. now, when we compare the two, we'll get an erroneous result.

overloading (smarter part of polymorphism): a single identifier (eg. `cos` or `+`) represents multiple objects (`cos` with doubles, `cos` with ints, etc). the context (eg. argument type) indicates to the compiler which object to use. other languages, such as Ada, overloads based on the *result* type as well.

ad hoc polymorphism doesn't really scale, especially if we keep adding user defined types as in object-oriented programming. modern languages usually deal with this via **parametric polymorphism**, in which a function's type contains 'type variables' (eg. `List<t>` where T can be any reference type — no primitive types). Java generics give you parametric polymorphism.

parametric polymorphism gives us a few problems. consider the following code, in which each step makes sense but the result is clearly wrong.

```
// generates a list of strings.
List <String> ls = ...;

// we assign it to a
List <Object> lo = ls; // copies the reference

// adding an object to an object list. OK
lo.add(new Thread());

// we're returning a "string" from a string list. OK.
String s = ls.get()

// but the result isn't OK! we just returned a thread into a string variable!
```

this operation should clearly fail. where should the compiler complain? it complains for line 2. we claim `List <String>` is *not* a subtype of `List <Object>`, even though `String` is a subtype of `Object`. this fixes the problem, but causes even more problems. what if we want to print any object?

```
interface Collection<T> {
    // ...
}
void printAll(Collection<Object> c) {
    for (Object i: c)
        System.out.println(i)
}
```

now, if we try `printAll(cs)` (with `Collection<String> cs`), this won't actually work — we just defined the parametric types as not related to each other, so this won't work. (it doesn't matter that `Object` is a superclass of `String`). we need arbitrary inputs or wildcards.

```
void printAll(Collection<?> c) {
    for (Object i: c) // works because ? will be a subtype of Object
        System.out.println(i)
}
```

or

```
void printAll(Collection<T> c) {
    for (T i: c) // T is an arbitrary type. we print it.
        System.out.println(i)
}
```

but what if we need two arbitrary types? aaah! `printall(Collection<a, b>)`. but this causes more problems. oh no.

you can see how this is a problem. there is a strong minority of programmers that believes that the problems introduced by these type systems aren't really worth the technical edge cases and generic rules they introduce (think of complicated generics. scary, huh?). they choose not to worry about typing and subtyping rules, worrying instead about whether the object in front of them can do what they want. this is called **duck typing**, and is popular in a lot of dynamically tped languages (eg. Python). in duck typed languages, types are less 'first class objects' and more 'collections of behavior'. if an object supports `o.waddle()` and `o.quack()`, then it's a duck; just check for associated behavior and throw an exception if it's wrong.

sacrificing the rigid type system means there's less compile-time checking; a python program can unexpectedly error out at runtime. however, it also makes the language a lot more flexible.

implementation of polymorphic types differs:

- **generics**: (Java, OCaml, etc). type check generic methods once, then compile them once. the same machine code can deal with them, because in these languages objects are handled via fixed size pointers anyways. type safety is known when you compile the method.
- **templates**: (C++, Ada, etc). type check generic code when it's *instantiated* (if we do `List<int>`, compile the method and check if the method works with ints). this is necessary because data types are different sizes (`sizeof` is needed, etc). we effectively have one piece of machine code that deals with every single thing we use it with. this sacrifices a bit of safety (for example, we might try to compile `List<double>` and *then* find out our List constructor was written wrong. on the other hand, the machine can optimize each piece of code for each specific data type. more copy pasting at compile time, so harder — but more efficient at run time, so faster.

prolog

don't think of prolog statements as functions; think of them as relations between sets of arguments (eg. "for a relation, these particular things have to be true"). order matters, though — running predicates backwards might result in an infinite loop (eg. run append with the first `argument` as a variable).

in practice, most predicates don't run efficiently. if we want to write an efficient program (using accumulator, etc), often our program is optimized for running in *one direction*. for example,

`reverse(L, R)` might only work if 'L' has no variables in it; aka if L is totally fixed. make sure to put this in a comment.

debugging prolog: most prolog implementations have a 4-port debugger. what does this mean? if we enable debugging with the 'trace' predicate, it focuses on the current clause, treating it as a blackbox. (eg. `G1, G2, G3.`; it can focus on `G2`.

```
call ->             -> succeed (move on to G3)
fail <-             <- backtrack
```

and there are 4 corresponding paths.

what `trace` does is put breakpoints on all of these 'gates'.

efficient prolog

prolog uses a stack to keep track of the axioms it hasn't proven yet (akin to a to-do list).

it also has to remember its state because it might need to backtrack.

generally, this stack grows on success and shrinks on failure.

given `G1, G2, G3.`, `G1` will be on the top of the stack, followed by `G2` and `G3`. After `G1` is proven, it's popped from the stack. this is useful for logical conjunctions ('and' statements, `,`). to deal with 'or', aka `;` statements, prolog uses **choice points**; something that you can push on to the stack to help with backtracking. if we have `G1, G2; G3` and `G2` fails, we keep popping values from the stack until we reach a choice point just before `G1`, and then proceed to `G3` (pretty much it's just an indicator on the stack that notes where to backtrack to in the case of a failure).

the stack gets longer on success because it keeps track of what's been evaluated so far (needed to backtrack).

many prolog programs don't even need the heap; because there's no garbage collection, it can become fast.

terms are also added to the stack; `f(a, g(Y), Z)`; if a variable is created, it's added to the stack. when a variable is resolved, it copies the value into the spot where the variable was (so any variable lookups will return the resolved value). this isn't very helpful for backtracking, though. because of this, variables are also 'trailed' on the stack; variables hold records of their previous values.

unification: like the ocaml `match` pattern match statement, but it's two-way. think about this:

```
match l with
| h :: t ->
    true
```

this ocaml code binds `h` and `t` to point to the subcomponents of `l`. however, there's no real way for this match to work in both directions (eg. bind `l` to the `cons` of `h` and `t`). prolog does this automatically; `p(a, X)` can give us an answer, as can `p(Y, b)`. this is *unification*; it's powerful, but intrinsically can run into a few problems.

```
e(X, X).
e(z, f(z))
{x = z, x = f(z), thus z=f(f(f(f(... }
```

TODO: copy in peano arithmetic thing.

```
p(X, Y) :- generator(X, Y, Z), member(X, Z), expensive(Z, X, Y).
```

the generator generates prolific answers, the member function acts as a 'filter' on correct solutions, and then the expensive function checks if the solution is valid. this can have a huge performance problem! what if the generator creates a binding such that `X=a`, and `Z=[a, b, c, a]`? the `member` filter will succeed, and the expensive function will run and fail. now, when we *backtrack*, the `member` function will find another way to succeed; there's another `a`, so it will bind X to the new `a` and recompute `expensive(Z, X, Y)` (which we know will fail, because it has the same arguments and failed before). because of this, prolog includes the *cut* predicate, spelled as `!`. what this will do is

scheme

TODO:

- named let
- tail calls
- continuations

continuations:

call with current continuation: `call-with-current-continuation`, aka `call/cc`.

- to create a continuation: `(call/cc p)`. this statement ccan return multiple times.
- `(p k)`. to use a continuation, we just call it with one argument.
- `(k v)` will set `%rip` to `k`'s instruction pointer, and set the environment pointer to `k`'s environment pointer; aka it arranges for `call/cc` to return `v`.
- create k TODO

```
(let [cont (call/cc (lambda (k) (k)))]
  (if (integer? cont)
    cont
    (cont 27)))
```

this is a complicated way of returning 27.

write a program to compute the product of a list of integers. some of these integers could be very large (billions of digits long). however, zeroes are frequent; if we encounter one, we should just return zero and save ourselves the hassle. we can do these with continuations.

```
(define (pr ls)
  (call/cc (lambda (break)
    ; this is a named let. look it up.
    (let pr (ls ls) (a 1))
      (cond
        ((null? la) 1)
```

```
        ((zero? (car ls)) 1)
        (else (* (pr(cdr ls)) (car ls)))))))
```

in a way, this is similar to a nonlocal `goto`. we can also implement continuations in C.

```
#include <setjmp.h>

jmp_buf                 // type: holds rip, rep, etc.
int setjmp(jmp_buf j)   // create continuation, store it into j

// restores interpreter registers from j, causes longjmp to return r
_Noreturn void longjmp(jmp_buf, int r);

// using continuations:

static jmp_buf j;

int main(void) {
    while(true) {
        if (setjmp(j) == 0) {
            // ordinary computation
        } else {
            // error recovery
        }
    }
}


//...later in a helper function
int helper() {
    // we run into an error! using longjmp, we 'time travel' / 'jump' to
    // where we originally made 'setjmp'. setjmp(j) now returns -1!
    longjmp(j, -1);
}
```

green threads / coroutines

throughout the following code snippet, we're mutating variables and using side effects — very nonfunctional programming.

```
; 'green_thread' = green thread

(define green_thread_list '()')

; thunk is a parameterless procedure called for its side effect.
; when we pass in a 'thunk', it adds it to the end of the green-thread-list
(define (green_thread thunk)
  (set! green_thread_list (append green_thread_list (list thunk))))

; start takes no arguments; it's used for its side effect, which is to run all
; threads.
(define (start)
  (let (first_thread (car gtl))
    (set! green_thread_list (cdr green_thread_list))
    (first_thread)))
```

what this will do is run through every thread in the list, executing each one. this supposes that each thread will give up the CPU every now and then; can we make this happen?

```
(define (yield)
  (call/cc (k)
    (green_thread (lambda () (k #f)))
    (start)))
```

now, we can run these!

```
; create a thread that says 'h', then yields, then loops back to 'h'.
(green_thread (lambda () (let f () (display "h") (yield) (f))))
; create a thread that says 'i', then yields, then loops back to 'i'.
(green_thread (lambda () (let f () (display "i") (yield) (f))))
; create a thread that says `\n`, then yields, then loops back to `\n`.
(green_thread (lambda () (let f () (newline) (yield) (f))))
; start running threads!
(start)
```

this will output: `hi\n` in an infinite loop.

however this style uses continuations via the seemingly-magic `call/cc` that rips `%rip` and the environment pointer out of the scheme machine. can we do this in a 'pure' way?

in C, we can implement object oriented programming if we program in an OO style — every function accepts an extra argument, which is the 'object' to which it refers (in higher languages, we claim the `this` object is being invoked. think back; all python methods need `this` (or really *any* variable name) as a reference to the object itself. similarly, in scheme we can use a continuation-passing style. TODO copy in pictures.

errors in scheme programs

- undefined behavior, eg. `(car 0)`
- errors signalled, eg. `(open-input-file "nonexistent.txt)`. implementation checks for errors, then reports them.
- unspecified behavior: program does things that are allowed by standard, but isn't allowed to error. eg. `(eq? '(a b) '(a b))`. these `'(a b)` constants might be different objects in memory, so `eq?`'s pointer comparison will return `#f`. however, a compiler might optimize and store both versions of `'(a b)` as the same constant, so the implementation might return `#t`.
- implementation errors: out of memory, etc. not the programmer's fault.

storage management

we've seen in CS 33 how stack management works; it breaks down execution into discrete frames. how does this work for dynamically allocated memory?

heap management

assume we have `malloc` and `free`. how do we implement these? we need a way to keep track of the array blocks we haven't used yet.

free block management

there are a few strategies:

linked list strategy: the free-list can be stored as a linked list within the empty blocks themselves; it's stored as a singly linked list. each free section contains a record of its own size and a pointer to the next block of free memory. this way, the heap effectively records itself.

this *fixed-pointer* approach has a few problems; traversing the list may be difficult if the heap manager memory is swapped to disk (I/O accesses will be slow af). in addition, the first memory block will tend to 'shrink' (eg. we have a 900 byte block. after 3 consecutive 299 byte mallocs, we have 3 bytes (which must include pointer to the next block, etc etc). now, any subsequent mallocs (which are almost always more than 3 blocks) will skip this entry, then exhaust the second free block, with more and more traversals every time. because of this, we

often use a *roving-pointer* method, in which the free memory list is circular, and the pointer moves around to 'distribute' the mallocs.

in addition, we have problems when `free`ing using this approach. we have to keep track of how much memory was `malloc`ed. we might secretly `malloc` a few extra bytes at index `-1`, and store the size of the `malloc`ed block there. we also run into the problem of memory fragmentation; if we add `free` blocks to the list without merging them with adjacent free blocks, our blocks will end up fragmenting into many small free-blocks (with no blocks large enough for us to use). to solve this, we'll need to periodically traverse the linked list, intelligently merging adjacent free blocks in the freelist.

this isn't particularly performant. often, heap managers will take advantage of patterns in their data via **quick-lists**. let's say we're writing a Scheme interpreter in C. we know that we'll be using `cons` a lot. a `cons` object might be structured as: `struct cons { void *car; void *cdr; }`. as we know the size of a `struct cons`, we might as well avoid overhead and just `malloc` a large array of `cons` structures. whenever our Scheme program wants to cons a list, we can just give it an object on our **quick-list**. neat.

free object management

the previous methods work fine provided we explicitly allocate and free memory. nowadays, another method is becoming popular; **garbage collected** languages.

pros:

- simpler
- no dangling references
- avoid storage exhaustion

cons:

- complexity in implementation: GC bugs are found often, especially with multithreaded programs (what if objects point in a loop?).
- performance: GC, by definition, must scan all of program memory to read all pointers. this causes a 'hiccup'. overhead, CPU and memory (we need an extra bit of memory to mark objects).

simplest garbage-collection algorithm: **mark-and-sweep**. each object gets a 'mark' bit. the GC process goes as follows:

1. clear all mark bits.
2. find all objects immediately reachable from the root object (a pointer or something, either in stack memory or on a register).
3. find all unmarked objects immediately reachable from a marked object. mark them.
4. repeat step 3 as needed.
5. add all unmarked objects to the freelist. (*sweep* all unmarked objects).

we can easily make steps 1-4 (the *mark* steps) depth-first instead.

this method isn't useful in real-time systems, due to the time 'skip' when the GC is called. cost of garbage collecting is proportional to the number of objects.

how do we know where the 'root's are? they could be in a register or in the stack.

1. compiler tells the GC the location of all roots (common in Java or Ocaml, languages with object layouts). this isn't possible in languages such as C, because the compiler convention is to not differentiate between pointers and values (a value could be interpreted as an integer or a pointer).
2. **conservative GC**: the memory manager knows where objects, stack storage, and registers are. it looks through every object for something that might even resemble a pointer. it might make mistakes (eg. a number in Bill Gates' bank account might be a very large integer *or* a pointer), so it plays it safe and assumes they could be pointers (we can't dereference them to check bc we might get a segfault). because of this, we have OK garbage collection, but might have a memory leak because we played it safe. conservative collectors also can't move objects around in memory (because it might be changing int values).

how can we make garbage collectors even better?

real-time GC: `new` is given an upper bound in terms of CPU time. this isn't guaranteed for mark-and-sweep, so it requires more sophisticated algorithms. the cost of a GC cycle is amortized over various calls to `new`; each `new` call marks a few more objects, leading to a gradual GC with less overhead. these are very difficult to make, and are very expensive (think about how simple pointer assignment would work: to ensure we don't undo all the work done in the last GC semi-sweep, every pointer assignment would have to clear the 'mark bits' of the objects. it's hard, and slow — luckily for real-time systems, slow is better than unpredictable.).

alternatively, we can take advantage of patterns in real world data to improve our algorithms:

generation-based-allocator: most objects are short-lived; longer-lived objects tend to be killed at a lower rate. in addition, most objects point towards older-lived objects. because of this, we can segregate objects into 'generations', with the youngest generation (the 'nursery') stored in a continuous block of memory. this makes `malloc` really simple given the nursery isn't full:

```
malloc(n) {
    t = heap_pointer;
    heap_pointer += n;
    if (heap_pointer is overfilling the nursery)
        garbage_collect();
    return t;
```

all we have to do is increment a pointer. when the nursery gets full, we need to deal with it; a **copying garbage collector** will copy the valid objects in the nursery into a new spot in memory instead of cleaning them in-place (this is to take advantage of consecutive memory accesses, leading to better hardware caching). this runs into problems; eg. if you have a linked list in the nursery, the copying GC will copy all of it, leading to overhead. eventually, the nursery can become an older generation.

another sect of programmers believes that garbage collection is fundamentally too complicated and full of heuristics. Python memory management: takes a hint from file-system development, and uses **refcounting**. every object has an associated number of references that 'point at it'; when the number reaches zero, the object is assumed to be garbage.

this runs into problems with circular references and such; while Python's strategy for a long time was to 'never use circular data structures', they eventually implemented a hybrid approach; they use mark-and-sweep when Python is running low on memory.

object-oriented languages

class-based: either objects can be created (`o = new O()`), or cloned from existing objects. these have complex rules about subclasses and inheritance.

prototype-based: languages think there should be one way to create an object; the should be cloned from existing objects only. more flexible to an extent; no subclass rules, etc: duck typing is enough.

parameter passing

parameter correspondence: how do we assign the arguments of a method call to the parameters of the method? fairly straightforward; we can match based on position (first argument, etc), deal with varying numbers of arguments by passing in a tuple of additional arguments, and use named correspondence (explicitly pass in the name of the parameter in the method call). these correspond to regular arguments, `*args`, and `**kwargs` (keyword arguments) in Python, respectively. look through Python documentation for more details, it's a fairly trivial concept.

parameter passing conventions: how do we pass data in to a subroutine? some traditional ways:

1. **call-by-value**: caller evaluates all expressions, and passes copies of the value to the callee on the stack or in registers. this copying takes time for large objects.

2. **call-by-reference**: the caller evaluates the *addresses* of the expression, passing small *references* to the callee (implemented as pointers that are passed-by-value and automatically dereferenced. saves time for large objects, but pointer dereferencing slows down passing small values. in addition, it can lead to *aliasing* problems; if `f(x, y)` is called as `f(a, a)` (ie both are aliases for the same value), problems may occur. confusing and unpredictable code, slower code (compiler has to take the 'both pointers are equal' case into account, can't optimize code).
3. **call-by-result**: used by Ada. the caller passes in where to put the result, and the callee's responsibility to initialize this parameter (eg `f(x, y, z, v, w)`, where `v` and `w` are pointers to the result; equivalent to `v, w = f(x, y, z)`.
4. **call-by-value-result**: when you call, it's call by value. when you return, it's call by result. no idea wtf this is, look it up TODO.
5. **call-by-name**: in all the above cases, it's been the caller's responsibility to pass in an evaluated expression. this causes overhead; we might not even need the result, so why evaluate it and pass in an argument? as such, the caller passes in a recipe (a parameterless procedure, colloquially known as a *thunk*) that evaluates the result. for example, when we pass in `f(x)` without evaluating x, we're really passing in `f(lambda () (a[100 + 1]))` the program plugs the result of the lambda function when it finally needs `x`. because expressions aren't evaluated till the last minute, this is often called **lazy evaluation** (as opposed to languages where expressions are evaluated as soon as possible, which use **eager evaluation**). this is more robust, because invalid expressions may be ignored (if we evaluate something else instead).
6. **call-by-need**: like call-by-name, but the thunk is evaluated at most once, and the result is cached. if the function has no side effects, then we know evaluating it multiple times gives us the same result.
7. **call-by-unification**: done by Prolog. you have 2 terms, either of which can be evaluated; we match these two arguments. this is very conceptually different than the other few, requires logical language such as Prolog.
8. **macro calls**: arguments are text-substituted (Scheme S-expressions) at compile-time. in a way, similar to call-by-value.

cost models

99% of the time, an

$$O(n)$$

algorithm in Python is preferable to to an

$$O(n^2)$$

algorithm in optimized machine code. that said, there is a time to worry about the constants that each algorithm may imply (in time, space, network access, energy, etc).

develop mental models as to how things work (eg. Lisp lists are implemented as linked lists, so prepending is cheap and appending harder. this is not the case for python lists, where prepending is expensive and appending is easier. occasionally Python lists will have to expand their memory buffer, so performance may be 'jerky' (1 insert will take order N). `strcmp` is order N in C, but certain x86_64 instructions allow for mass byte by byte comparison using SIMD hardware. this will speed up the call massively, but also brings the processor out of its low power mode. the cost of function calls may depend on the parameter passing method described above.