CS131: Programming Languages
Lecture Notes

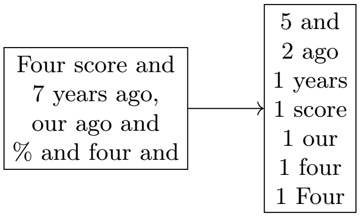
Henry Genus

Spring 2020

Contents

1 Choice of Notation

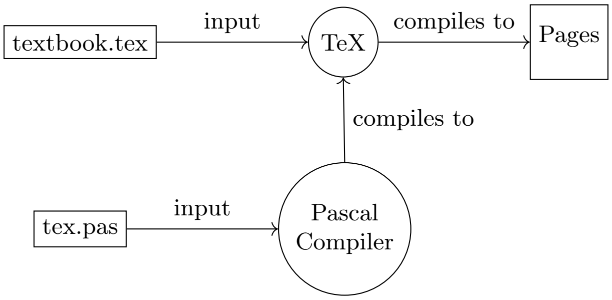
We will begin this class with a quiz. We wish to develop a program that takes a sequence of ASCII characters and outputs a frequency-sorted concordance of all words in the input.
We define a word with the following regular expression: [A-Za-z]+
Words must be bounded by white space. An example usage of our program is:



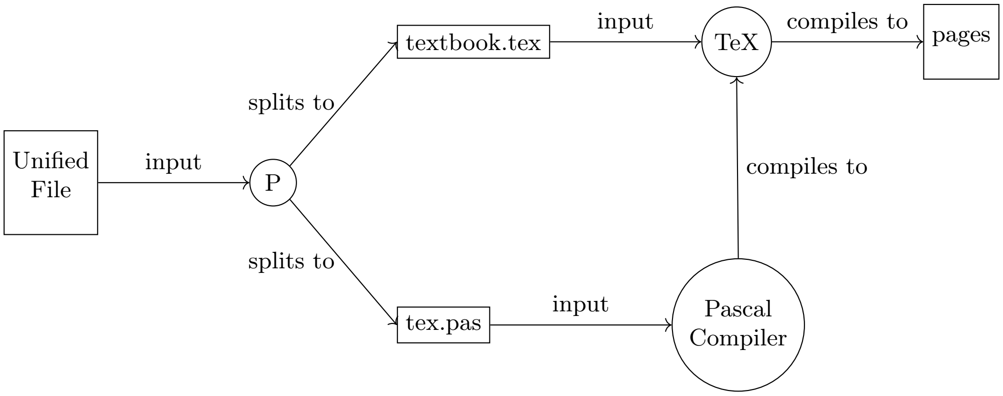
```
Four score and
7 years ago,
our ago and
% and four and
```

```
5 and
2 ago
1 years
1 score
1 our
1 four
1 Four
```

This was used by Turing Award winner DE Knuth as press for his book, The Art of Computer Programming
The motivation came from difficulties in creating the initial press for his book;
The lead type press could not handle code or equations, which were fundamental!
So Knuth invented a language to produce correct images, called Tex.
He developed a textbook to bring attention to his language, which explained the framework. The framework took the form:



This presents a large (and common) issue – code and documentation must be kept in agreement.
Knuth developed the idea of a unified, interleaved file to combat this.
Thus his program took the following final form:



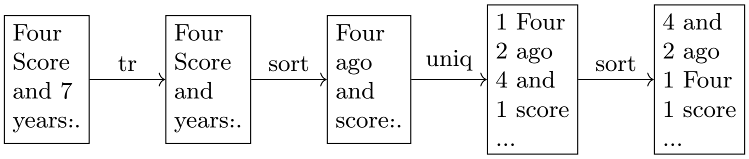
This left him with a nearly 500 page textbook; how would he drum up interest?
Like a good computer scientist, he chose to write a paper.
He coined his approach **literate programming**.
This has become standard practice in many disciplines; we can see JavaLibrary for proof.

Knuth approached Doug McIlroy, the manager for the development of UNIX, for help.
McIlroy suggested our problem as an example
Knuth wrote a solution using Hash Tries via literate programming!
Running it through TeX gives pages and through pascal gives the paper!

As impressive as this was, the afterword was what stuck.
McIlroy proposed the following trivial BASH solution (he was the brain behind UNIX pipes after all)

```
tr -c 'A-Za-z' '\n' | sort | uniq -c | sort -rn
```

The pascal solution was, say, 1000 lines.
It was faster
It was more "checked" by compilers
BUT the bash solution is exceedingly simple and documents itself as follows:



This demonstrates an issue fundamental to computer science called **choice of notation**.
There are pros and cons to any choice of notation, and it is up to the engineer to weigh them!
As this is a discussion of languages, consider the following fundamental linguistic hypothesis:

Sapir-Whorf Hypothesis.

a) There is no limit on the structural diversity of language.

b) The structure of a language determines a native speaker's perception of experience.

This was proposed by linguists for natural languages, but (b) was softened.
Even though it is false, it stuck: see “Eskimos have 19 words for snow”.
Even if false for natural languages, it is true for programming languages!
We explore this in the next lecture

2 Syntax

The core of programming languages comes down to 3 things:

1. principles and limitations of programming models
2. notations and user supports for programming models
3. methods of evaluation of programming models

These are the things we consider when attempting to decide on a choice of notation.
By the end of the course, our choices will be:
Ocaml – our canonical functional language
Java – our canonical imperative language
Prolog – our canonical logic language
These are our "big three" languages, which cover the main paradigms.
We will also cover Scheme, Python, etc.

The general goal is to learn, evaluate, and use languages.
How do we evaluate a language? We seek to minimize costs.
These come in the form of:

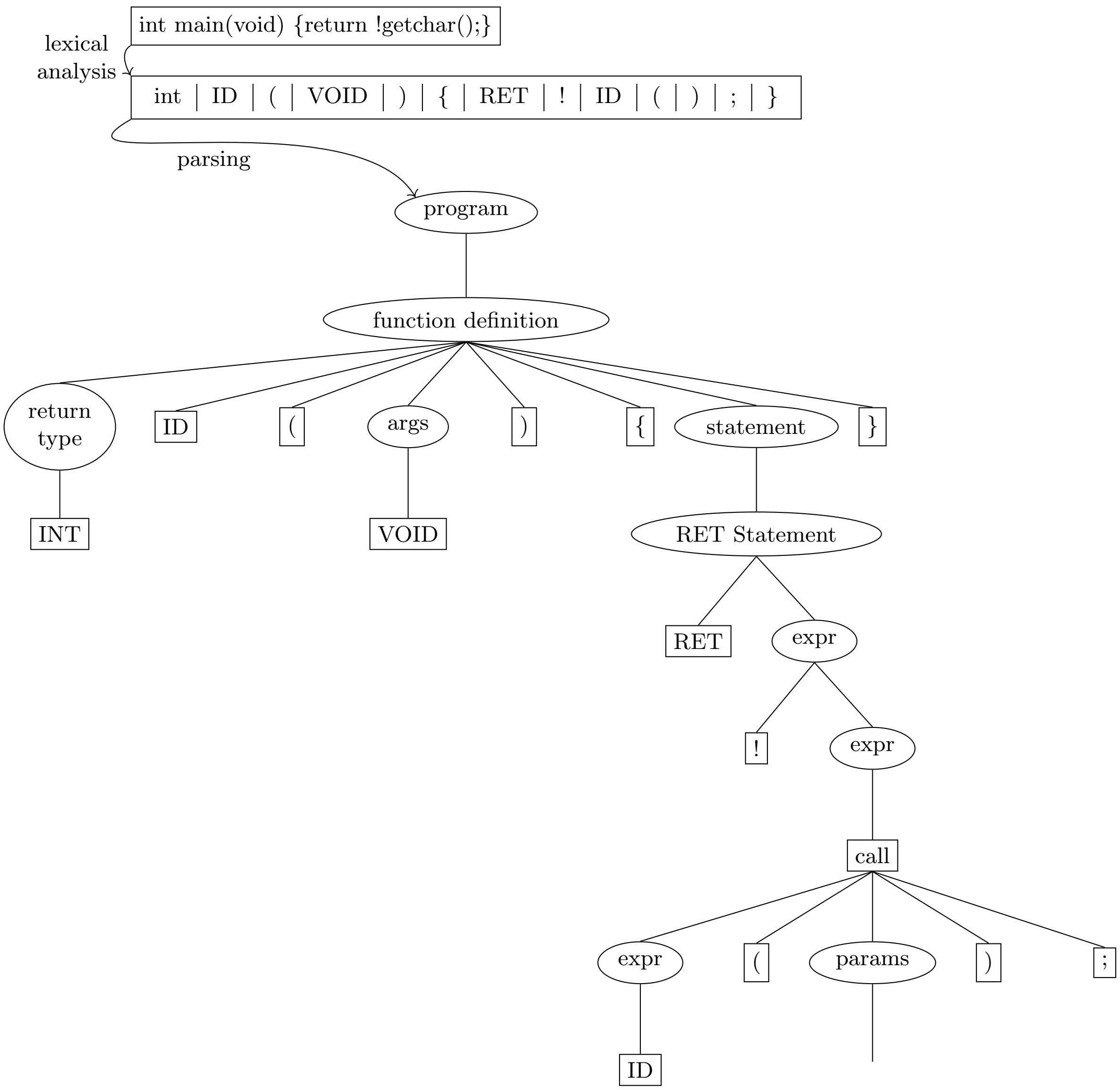
1. Primary Costs
   (a) maintainability
   (b) reliability
   (c) training
   (d) program development
2. Secondary Costs
   (a) execution overhead (exception: critical in machine learning)
   (b) licensing fees
   (c) build/compilation overhead
   (d) porting overhead

These choices can often be political (Apple vs Google, etc).

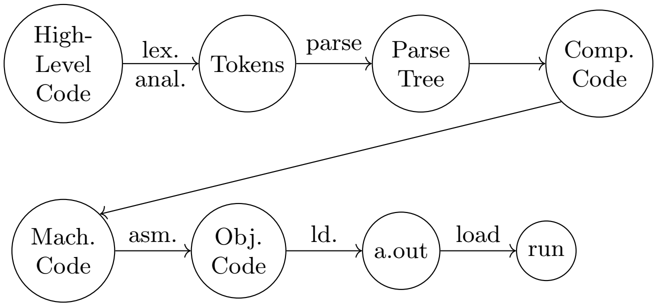
The main issues of language design, on the other hand, are:

1. orthogonality
   For example, all C types except arrays are returnable, which presents difficulties with typedef!
2. (runtime) efficiency
   This includes CPU, RAM, energy, network access, etc.
3. simplicity
4. convenience
   C allows i++, ++i, i = i + 1, i += 1, choosing convenience over simplicity.
5. safety
   Errors can be checked in two ways:
   (a) static checking – safe, compile-time checking
   (b) dynamic checking – simple, runtime checking
6. abstraction support
7. exceptions
8. concurrency
9. evolution/mutability
   "As long as a language is alive, it is mutating"

We will now move on to considering more computer language specific issues.
The translation of a language's code goes as follows:



We use this tree to build our compiler-level code.
It is portable and convenient for the compiler creator.
This puts the code → run process as:



Our first two assignments focus on the token → tree step.
This is where we consider **syntax**

Syntax

≡ the form of a language independent of meaning/**semantics**

- a sentence can have good syntax and bad semantics
  "Colorless green ideas sleep furiously." – Noam Chomsky
- a sentence can have good semantics and bad syntax
  "Ireland has Leprechauns galore." – Paul Eggert

- a sentence can be syntactically or semantically ambiguous
  "Time flies." – unknown; probably a politician, since doublespeak helps them

How do we evaluate a syntax?
Consider the example of 3+4*5 (C) vs 345*+ (Forth) vs (+3(* 4 5)) (LISP)

- inertia – no one likes to have to change
  C > Forth ≈ Lisp, since C is familiar to standard form
- simplicity/regularity – few, regularly applied rules
  Forth (never parentheses) > LISP (always parentheses) > C (some parentheses)
- human readability – code shows algorithm/operations C ≡ LISP ≡ Forth due to Leibniz's Criteria
  ≡ a proposition's form should reflect reality
  ex. "if (0 < x && x < n)" > "if (x > 0 && x < n)" since 0 < x < n is natural
- human writability
  C > LISP ≈ Forth due to comfortability with standard form
- redundancy – repetition helps catch errors (inapplicable in this example)
- unambiguity – code has only one interpretation
  Forth ≡ LISP > C, since C relies on an implicit order of operations

Syntax is build upon **tokens**

Tokens

Tokens are a subset of **lexemes**, which are built from characters.
Tokens can be broken up into three categories

- identifiers
- operators
  Operators are often user-defined, but some are **keywords**
  keyword ≡ a word with a special meaning in a language
  Some operators can also be categorized as **reserved words**
  reserved word ≡ words which are not allowed to be used as a name
  C, for example, has reserved all words of the form _[[: capital :]].∗.
  This is annoying, since it makes us use _Noreturn instead of noreturn.
  Consider the example 'if':
  'if' is a keyword since it tests an expression, BUT:
  - in C it is reserved, ⟹ 'int if = 12' is invalid
  - in PL1 it is not reserved, ⟹ 'if (if=3) if = 4' is valid.
- numbers
  This includes 27, 0x19, and 5e-12, but not -127, because itemizers are greedy.
  To understand this, consider the example: a - - - - - b; this should be valid as (a - -) - (- - b),
  BUT the tokenizer reads it as ((a - -) - -) - b, and we cannot decrement a constant.

It may be a natural assumption that tokens may only consist of characters, but this is overkill.
We must consider both white space, and ambiguities like 'ifx', which would cause slow parsing.
Luckily, tokens are a level higher than that, and can thus be parsed with C.

Grammars

If we were attempting to define a language in mathematical form, we may do it like so:

$$L = [a^m b^n | m < n], \text{ tokens: a, b}$$

This would give us the language "", "b", "ab", "abb", ...
This doesn't scale well to programming languages. Instead we abstract to **grammars**

A few definitions are necessary to begin this discussion:
grammar ≡ a description of a language's syntax
language ≡ a set of sentences
sentence ≡ a string within a language
string ≡ a finite sequence of tokens
The specific grammars in this course are called **Context-Free Grammars**
A CFG is a set containing

- a finite set of tokens/terminal symbols (leaves)

- a finite set of non-terminal symbols (intermediate nodes)
- a finite set of rules (parent/child relationships)
- a start symbol (root)

Thus they can be defined by a grammar trees like the above.
They are called "context free" because an expression's definition is free of its context.
For clarity, consider a few counterexamples:
in C, 'typedef int foo; ... foo i;' is valid, but 'foo i;' is not.
in Python, 'if (true) pass' is invalid due to indentation rules.

3 Grammar Notation

The basic form of notation for a context-free grammar is the **Backus-Nour Form**.
These map non-terminal identifiers to their options for representation.
Consider the example BNF grammar:
S → (S)
S → a
A similar grammar can be expressed more efficiently in the form of a regular expression:
S → ab*
But sadly, regular expressions cannot be used on just any grammar.
For example, in attempting to emulate the BNF grammar above, the closest we can come is (*a)*.
This does not account for matching parenthesis count!
This is because regular expressions cannot handle nested recursion.

Consider the example of email-RFC5322. This requires all emails contain a line similar to the following:

Message-ID:<eggert."93-5xx27"@cs.ucla.edu>

This uniquely identifies any email message to allow for easy filtering of duplicates.
The grammar takes the form of an extended BNF.
It uses / for OR, * for repeatable nonterminals, () for optional, and ” for terminals:
msg-id = ’<’ dot-atom-text ’@’ id-right ’>’
id-right = dot-atom-text / no-fold-literal
dot-atom-text = 1*atext (. 1*atext)
no-fold-literal = ’[’ *dtext ’]’
atext = ALPHA | DIGIT | ’!’ | ’#’ | ...
dtext = %33-90 / %94-126 (the set of printable characters excluding ’[’, ’/’, and ’])

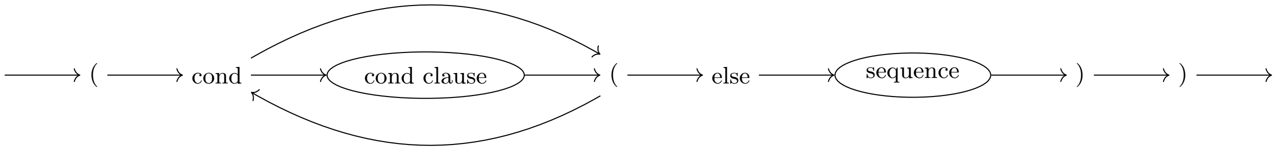
Since this is an EBNF grammar, it must be expressible in BNF form.
Consider the fourth rule in the list. It can be expressed in BNF form as the set of
no-fold-literal = ’[’ dtexts ’]’
dtexts =
dtexts = dtext dtexts
We could perform similar expansion for the rest of the rules, but this would get unwieldy quick.
We thus need a more efficient form for expressing grammars.
Luckily, the International Standards Organization established a standard called **ISO BNF**

ISO-EBNF (a top-down view):
"terminal symbol"
[optional]
{repeat zero or more times}
*repeated
(* comment *)
A - B (* A except B, set division *)
A,B (* concatenate A and B *)
A|B (* A or B *)
A = BC; (* grammar rule *)
The gave an English summary of their grammar for grammars, then demonstrated it with their own notation:
syntax = syntax-rule , {syntax-rule};
syntax-rule = meta-id , ’=’ , defns-list , ’;’ ;
defns-list = defn , ’|’ , defn;
defn = term , ’,’ , term
term = factor , [’-’ , exception];
exception = factor;
factor = [integer , ’*’]
Note that exception is unnecessary, but was included for clarity.

Some grammars, like SQL data typing extensions, are best expressed through **syntax charts**.
Here is an impractically small example (in SQL notation):

<cond> → (<cond clause>*) | (cond <cond clause>* (else <sequence>))

This is equivalent to the following diagram:



We can use this to program a parser:

```
void cond(void) {
        eat("(");
        eat("cond");
        while (cond-clause()) continue;
        eat(")");
        ...
}
```

It would be nice to be able to express this as a finite state machine, but there is recursion.
Thus we would need to use a push-down automaton to express this.
Grammars progress into programs along grammar → design → parser code
But code can be buggy, and bugs in the parser would be bugs in the grammar, adding to complication!

Grammar Issues

Useless Rules

S → Sa
S → B
C → Cd
While this is a valid grammar, it defines a pointless subroutine, wasting space.
These errors can further be broken down into

1. Nonterminal used but not defined
   S → Sa
   S → Bc
   S → d
   The second grammar rule above is analogous to calling an undefined subroutine.
2. Nonterminal defined but not used
   S → Sa
   S → d
   B → aS
   The third grammar rule above is analogous to defining an uncalled function.

While these are both valid grammars, they are space-inefficient.

Uncaptured Constraints

Consider the following subset of English:
S → NP VP
NP → N
NP → Adj NP
VP → V
VP → VP Adv
for tokens N, V, Adj, Adv
This grammar has errors: for instance, it allows the sentence "Dog Bark".
To account for subject-verb agreement, we must complicate the grammar.
We would double the size of our grammar just to solve a boolean issue!

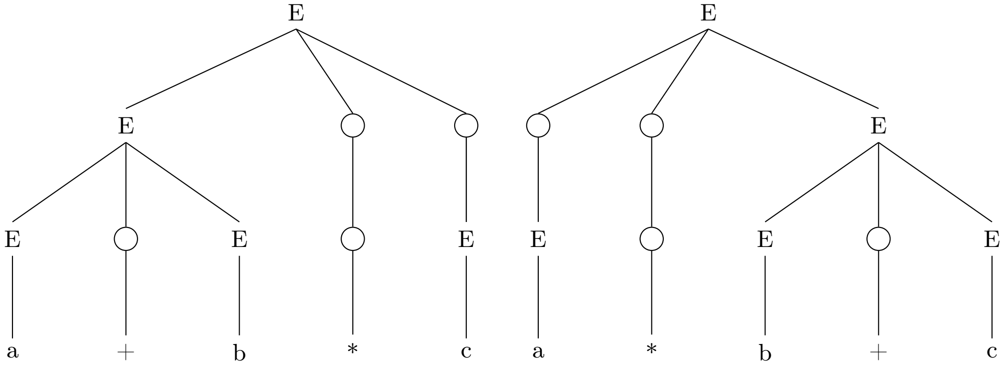
There are other types of errors, which would each cause exponential growth.
We can demonstrate some of these issues in OCaml:

```
let x=5 in x*y;; (* undefined y *)
5 / 3.0;; (* error: bad type float *)
```

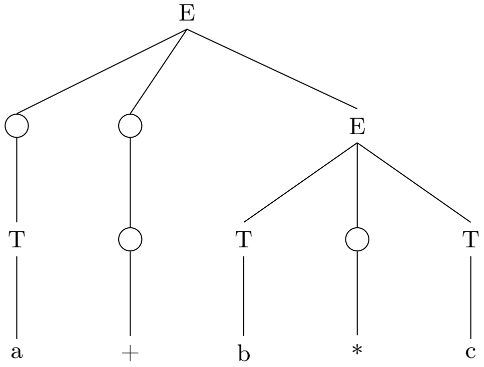
How do we fix these? WE DON'T. DON'T DO IT.

4 Ambiguity

Consider a simple grammar:
E → E + E
E → E * E
E → (E)
E → id
This has an ambiguity, as can be seen by the two parse trees below:



Parsers will build the left tree, whereas humans would build the right.
We must complicate the grammar to filter out the bad trees.
E → E * T
E → E + T
E → T
E → (E)
E → id
Now we have a single correct tree:



Our edits specify two new properties:

1. (*) has higher precedence than (+)

2. both (*) and (+) are left-associative

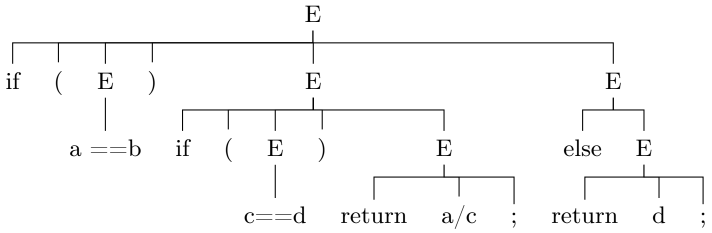
Let's now look at a subset of C's grammar:
stmt:
';'
'break' ';'
'return' ';'
'return' expr ';'
expr ';'
'goto' 'ID' ';'
'ID' ';' stmt
'while' '(' expr ')' stmt
'do' stmt 'while' '(' expr ')' ';'
'if' '(' expr ')' stmt
'if' '(' expr ')' stmt 'else' stmt

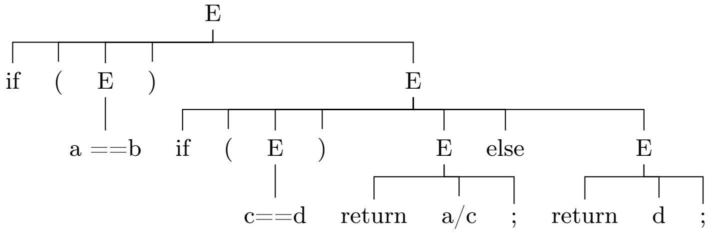
We may naturally wonder why some of these choices were made; for instance – why parentheses?
Consider the following altered right-hand-sides

1. ’while’ expr stmt:
   This is ambiguous – consider while i * p = = 3;

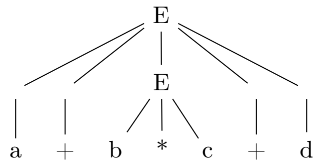
2. ’do’ stmt ’while’ expr ’;’
   This is not ambiguous, since we know where our expr ends. Parentheses are used for consistency.

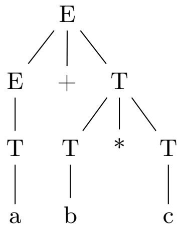
Not all ambiguities are as obvious as our original example, however; this C sub-grammar even has one! It is one we have undoubtedly encountered.... The dangling else!
Consider the following two trees:





We know the latter is correct, but what rule is causing this?
As it turns out.. ’if ’(’ expr ’)’ stmt ’else’ stmt *is too generous!*
We could fix this by adding a <full-stmt> nonterminal that requires an else, but C doesn’t.
To see why, we must consider the following two trees for the statement a + b * c + d:





The left tree is an **abstract parse tree**, which uses an ambiguous grammar and smart compiler.
The right tree is a **concrete parse tree**, since its grammar is unambiguous.
Since abstract parse trees are smaller and simpler, the C compiler uses the abstract tree.
The C standard thus leaves fixing the ambiguity to the programmer, in "the usual way".

For an example of ambiguity, consider Prolog:
Prolog lets us define operators in the form :- op(precedence, associativity, [operators]),
where lower precedence means operating first, and y is the associative parameter. Thus:

```
:-op(500, yfx, [+, -]).
:-op(400, yfx, [*, /]).
% so for a*b*c -> (a+b)*c, we violate (*)'s precedence and associativity!
:-op(200, xfy, [**]).
:-op(200, fy, [+, -]).
/*      Therefore:
        a**b**c -> a**(b**c)
        a**(-b) -> -b**a -> -(b**a) */
:-op(700, xfx, [=, >=,...]).
% these are non associative ^^^ so a==b==c is an error!
if (a <= b <= c) ...; % valid code! but translates to
if ((a<=b) <= c)...; % which is equivalent to
if ([0,1] <= c)...; % which is not the expected behavior!
```

Ambiguity is not the only runtime error we may face though; consider the following C code:

```
int a, b;
int f(void) {return (a=1) + (b=2);}
// this is valid & harmless, albeit pointless, BUT
int g(void) {return (a=1) + (a=2);}
// this is undefined behavior in C!
// it could change a to 15, or return 42, or even dump core!
```

Compare how C and Java deal with this problem:
C – undefined behavior
Java – L → R semantics; this is valid, but prevents compiler optimization.
Thus side effects in expressions tend to be bad news, and programs with them are hard to maintain.
Undefined Behavior is dependent on semantics; it is the problem of competing side effects

These problems were evident to J. Backus after he made FORTAN ambiguous for performance's sake.
He regretted it, and thus proposed **functional programming**
He had two primary motivations in doing this:

1. clarity (use of well defined math notation)
   even the name of an imperative language (C++ -> C = C + 1) is nonsense!
2. performance (via parallelization)
   escape the von Neumann LOAD-EXECUTE-STORE model

One can now see Backus was a visionary:

1. IOT devices occasionally recompute a value rather than load from RAM to save power.
2. RAM access can have massive time cost when distance separated from CPU
3. modern computers run many CPU's in parallel

This makes his paradigm worth discussing, but some terms need to be defined first:
function ≡ a mapping from a domain to a range
domain ≡ a set of values
range ≡ a set of values
partial function ≡ a function which maps a subset of the domain
no side effects ≡ the same arguments to the same function will always the same result
functional form/higher-order function ≡ a function with a function as a parameter

So if functional languages can't modify state, how do they do I/O?
The short answer? They don't; it can't be done functionally.
The long answer? Consider getc(). It moves the stream pointer, so it has side effects.
We can thus emulate a pure functional I/O function by

```
define (f1, c) = getc(f)
```

Functional programming has a few very beneficial properties:

```
(* 1. evaluation order is not controlled by sequencing *)
# h(f(a), g(b))
(* the order must be g,f -> h; since g \& f have no order, this is partial order *)
(* 2. referential transparency *)
# let x = 1 in
   let f = fun f -> x + f in
   let x = 2 in
   f 2;;
- : int 3
(*      f uses the definition of x within its code block -- this x cannot change value.
        The variable/function f exist in different namespaces => they are unambiguous.
        Thus the CPU can cache and optimize even faster.
        This is so strong that there will be no errors, even on reuse: *)
# let square square = square * square;;
(* perfectly valid code -- function square & variable square are in different namespaces *)
```

5 Software Construction for Programming Languages

There are a few main approaches to running code:

The Compiler Model

think: C, C++, Fortran, etc.
gcc's process takes c source code, and runs like so:

1. Preprocessing
   (a) lexing → tokens
   (b) parsing → parse tree
   (c) checking (identifiers, type) → checked parse tree
   (d) Machine Independent Code generation → MIC
   (e) optimization → optimized intermediate MIC
2. Compilation
   (a) machine code generation → machine dependent code
   (b) optimization → assembly language
3. Assembly
   (a) assembly → machine code (.o)
4. Linking
   (a) linking (ld) → a.out
5. Running
   (a) loading by kernel → run

Each of these steps can be broken into a separate program using gcc options and tools.
Compilers can make programs quite difficult to debug, since code is translated away from the source code.
For instance, consider the following two translations:
i *= 2 → shl %rax
1 = a/b; r = a % b; → ldivq %rdl
By the above, it can be seen that some translations are more direct than others.
The transformations compilers can perform include:

- aggregation of multiple lines of source code into one machine instructions
- splitting of a line of source code into multiple machine instructions
- swapping the order of machine commands

Since debugging machine code can be incredibly difficult, we developed:

The Interpreter Model

The program is left in a form near to the source code and interpreted/run directly.
The form of the code is designed for an abstract machine and human-readable.
Code is checked prior to translation.
The interpreter is written in a high-level language and thus easy to implement.

As we can see, interpreters have many advantages over compilers, so why do we use compilers?
When running code, an interpreter must walk through a tree or load bytes from MIC.
This can result in up to 10 machine instructions for each line of source code.
In practice, much of this is optimized away, but the principle stands; we wish to improve this.

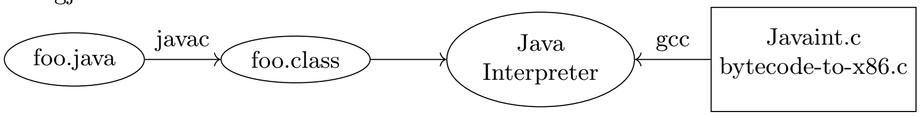
Integrated Development Environment

think: Eclipse, Emacs, etc.
The original was Smalltalk by Xerox, which:

- handles dependencies and partial recompilation
- keeps "one big happy program", including:
  - memory management
  - exception handling
  - dumping core
  - parallelism
- interacts with and controls the screen, keyboard, and mouse
- reads and works either directly from source code, or from a form very close to it, which allows
  - direct debugging of source code
  - continuous execution of program while editing

Java Virtual Machine (JVM)

The gjc model is as follows:



The steps

- the interpreter uses bytecode-to-x86.c to profile the code
  ≡ the compilation of recorded hotspots to machine code
- jump to machine code rather than fetching interpreted instructions
- progressively compile learned hot spots that pop up during execution

In a sense, the profiler acts like the backend of a compiler. This has a few advantages:

- space – shared code need only be stored once rather than once per program
- time – recompilation is not necessary on change of source code
- space – unused parts of the library are not unnecessarily loaded

This is called a **just-in-time/hotspot compiler**.

Conclusion

Real life is unsurprisingly more complicated; this behavior exists on a spectrum:

- pure interpreter – the intermediate language IS the source code
- tokenization – whitespace and comments are removed (the most common level)
- partial compilation — intermediate code could either be compiled or run directly on an interpreter
- full compilation – code is fully compiled; the intermediate code is machine code

Emacs and eclipse both fall somewhere in the middle of this spectrum.
The JVM approach is used for Javascript on browsers, which requires varying code by machine.
This is no big deal – we already needed different machine code anyway.

IDE's and Virtual Machines require one more functionality: **Dynamic Linking**
We take a piece of a program/file and link it into a running program.
Examples include:

- emacs takes shared machine code object files and links it using emacs Lisp
- JVM and Javascript do the same thing with their own native methods

In effect, this creates self modifying code, which is an idea fundamental to CS! (See: AI, the stack, etc.)
There are two ways to dynamically link:

1. complete linkage on load time
   This is safe, but much slower.
2. load functions on usage
   This can be risky, since

(a) functions may change between calls

(b) malicious or bad code may cause errors

Ideally, we would like the power of self modifying code without the risk.
This class's focus so far has been on imperative languages, which are command execution based.
Our attempt to improve dynamic linking is by using functional languages.

There are based in evaluation of expressions.

These avoid the side effects and cache invalidation of imperative languages.

This gives us the advantage of building a program on the fly, without the dangers of dynamic linking!

6 Identifiers

In addition to hardware/software interface evolution, languages can evolve too.
Languages like C/C++ have evolved so much that programs from the 70s couldn't run today.
There are even "software archaeologists" who specialize in translating this code.
The point is, we have a compatibility issue that we need to plan to address.

Let's consider some specific examples:

- BASIC:
  - was developed for the GE model 225 CPU. Specs:
    - about 40 microseconds per addition, and 500 microseconds per division.
    - about 40 KiB RAM, which was huge for the time.
      This is big enough that the device could fit all of BASIC in its RAM!
  - BASIC is a small base language which supports many extensions.
  - While this was critical at the time due to limited memory, it is now inconsequential.
- C
  - was developed for the PDP II Minicomputer CPU. Specs:
    - about 4 microseconds per addition and 1.2 microsecond cycle time
    - about 16 KiB RAM
  - C allows for fast memory access, but comparatively slow computation;
    This means that pointers were a very efficient way to manipulate data.
  - Modern computers can do computation easily, but access RAM comparatively slowly.

These languages can still run on modern devices, but their performance is worse.
We can, however, anticipate these changes
    For instance, modern languages are not conducive to GPU (SIMD)/cloud computing (MIMD).
All of these issues are issues of **scale**; languages may not scale well to:

- larger machines with increased array size and multiple CPUs
- complex problems that require multiple programmers

We address these issues in a few object-oriented ways.

Names/Identifiers

These are arbitrary within a program, so why is this the first thing we discuss?
    Choosing naming conventions is a major issue within software engineering.
There are two aspects to identifiers within a program:

- Binding Time
- Scope

We face a couple common issues:

1. Permitted Characters
   - In FORTAN, spaces were permitted within names. This led to serious errors.
     In the code controlling the Mariner 1 rocket, the following code was intended

     ```
     DO 10 I = 1,10
       FOO(I)
     10 continue
     ```

     but a period was typed instead of a comma, causing the code to be interpreted as

     ```
     DO 10 I = 1.10
     FOO(I)
     ```

     This resulted in the crashing of the rocket!
   - In C++, an identifier is of the form [a-zA-z_][a-zA-Z_0-9]*
     Characters from other languages were initially prohibited, but this changed in C++11
     This added complexity, since the latin and Cyrillic o's look similar but are encoded uniquely.
     Additionally, only some parts of names are case sensitive:

     ```
     double e = 1e16;
     double E = 1E16;
     assert(e == E > memcmp(e, E, sizeof(e)));
     ```

2. Reserved Words

   ```
   if (class < 12) return; // legal in C, not in C++
   ```

   Clearly, adding keywords to a language can add complexity and invalidate legacy code.
   We thus reserve names early in a language's development.
   For example, C/C++ reserve names starting with '_', so _Noreturn could safely be added in C11.

Binding

≡ an association between a name and a value.
We often access bound data via its mapping in the symbol table. These bindings can be deceptively simple:

```
short a = 10; // a is bound to 10
short *p = &a; // pointer is bound to the address of a
int *p = &10; // EXCEPTION -- literals do not have memory addresses
assert(sizeof(10) == sizeof(a)); // NO -- 10 is 4 bytes but a is only one
```

Binding is done in one of two ways:

1. **Explicit Binding** ≡ binding written directly into a program
2. **Implicit Binding** ≡ binding done as a consequence of a program

Implicit binding is often considered bad practice, so explicit binding is encouraged.
In many languages, types must be declared explicitly so as to avoid errors.
These languages bind variables to values between compilation and linkage, in a time called **binding time**.
To understand the different periods of binding, consider the following code:

```
void main()
{
  for (int i = 0; i < 10; i++) foo(i);
}
```

Portions of this function were bound at different times (listed chronologically):

1. 'void' and 'for' were bound at language **definition/authorship time**
2. 'int' was bound to 4/8/etc bytes at **language implementation time**
3. 'foo' was bounded to its definition and reference at **link time**
4. 'foo' and 'main' were bounded to their memory locations at **load time**
5. 'i' is bounded to multiple values at **runtime**

Even with the support for explicit binding, even C cannot avoid it completely.
Macros are expanded prior to runtime, but allow for implicit binding like so:

```
#define Foo 27
#IFDEF FOO ...
// if we typed FOO instead,
// we would get no error,
// but the if branch would be skipped
```

OCaml is considered a safer language than C, but OCaml uses nearly all implicit binding.
How is this possible? OCaml is safe because of redundancy and type checking.
We can do this in gcc with an option which bans implicit binding
  BUT even that wouldn't fix the above issue.

Types

Finding an exact definition of types is difficult, so we will use a few. This is the simplest:
  ≡ a set of values/objects
Defining a type explicitly via listing is called **enumeration**, and is used in functional languages like Lisp.
In computer languages, these enumerations are often defined in terms of **native types**.
Any type that can be constructed of native types is called a **constructed type**.
Object-Oriented languages often use another definition:
  ≡ a set of values and a set of operations on those values
Adding the idea of operations allows for observation of values and performance of actions; consider:

```
class complex
{
  double re;
  double im;
  double dist() {return sqrt(re*re + im*im);}
}
```

The above definitions can be used to analyze the C primitive ’float’:

- is float a set of values? YES – 0, 0.2, 0.22, ...
- is float a representation? YES – defined as follows:

For many years, this implementation was hidden;
since not specified by the spec, implementations tended to be machine-dependent.
In the current day, one layout has won out, called IEEE-754:
Numbers are 32 bits, and represented by the following sequential sections:

1. the sign bit
2. the eight bit exponent value
3. the 23 bit fractional value

We interpret the float value (sef) according to the following rule:

$$\text{value}(sef) = \left\{ \begin{array}{lr} (-1)^s * 2^{e-127} * 1.f & \text{for } o < e < 255\text{(normalized)} \\ (-1)^s * 2^{e-127} * 0.f & \text{for } e = 0 \text{ (denormalized)} \\ (-1)^s * \infty & \text{for } e = 255 \text{ \&\& } f = 0 \\ \text{Not a Number (NaN)} & \text{for } e = 255 \text{ \&\& } f! = 0 \end{array} \right\}$$

This implementation results in a couple special cases:

- there are two distinct zeroes
  → two variables can have different data with the same value.
- NaN evaluates false no matter what it is compared with
  → two variables can have the same data and different values.

Floats can trigger a few exceptions:

1. overflow – exponent too large (1e308 x 1e308)
2. underflow – exponent too small (1e-307 x 1e-307)
3. bad arguments – (0.0/0.0, sqrt(-1))
4. inexact results – (1.0/10 != 0.1)

Traditionally, hardware would trap on exceptions 1-3 and ignore 4, but now they all return special values.

Types have a few ways to avoid exceptions:

- **Annotations**
  Types document programs, and the compiler uses this documentation to report errors. (ex. int val;)

- **Inference**
  The compiler infers the type from the code (ex. 1.0, 1.0f, 1)
  There are two standard approaches:
  - bottom-up – C: 3 + 4.5 + ’a’ → float
  - top-down – OCaml: [3, 4, 5] = (f y z) → int -> int -> int

- **Checking**
  In a strongly typed language, all operations are checked for type matches prior to being run.
  The benefits: it is safer, and there is no mistaking the type.
  The downsides: it is finicky, and can disallow legal operations.
  C and C++ are strongly typed (except void pointers can subvert this typing).
  OCaml and Java are strongly typed with no exceptions.
  Python is weakly typed with no exceptions.

7 Polymorphism

Types can be broken into two major categories:

1. **Concrete Types** have an implementation directly visible to the programmer.
   For example:

   ```
   typedef struct complex
   {
     double re, im;
   }
   ```

2. **Abstract Types** hide their implementation from the user, and are defined by operations.
   We can think of 'float' as abstract, since f, e, and s vary in location by endian-ness.

There is a sort of battle between abstraction and efficiency (low level access).

Given that we can define multiple types, there must be a well defined definition of equivalence.
There are two standard approaches to this:

1. **Structural Equivalence**
   $\equiv$ two types are the same if and only if their structure is the same.
   This is equivalent to saying that two types are assumed to be the same unless they behave differently.
   This approach tends to be most applicable to concrete types.

   ```
   typedef int T;
   typedef int U;
   T v;
   U *p = &v;
   ```

2. **Name Equivalence**
   $\equiv$ two types are the same if and only if they have the same name.
   This approach tends to be most applicable to abstract types.

   ```
   struct R {double re, im;}
   struct S {double re, im;}
   struct R v;
   struct S *p = &v; // ERROR
   ```

Suppose we wanted to implement structural equivalence in C.
Then R and S would be equivalent, which leads to complications, for instance we can do:

```
struct T {int val; struct U* pair;};
struct U {int val; struct T* pair;};
// and these would be equivalent as well
```

These complications were avoided for C, which was optimized for efficiency.

Type equivalence is more complicated when it comes to subtypes.
The behavior we want is: $\mathrm{T} == \mathrm{U} \iff \mathrm{T} \subseteq \mathrm{U}$ && $\mathrm{U} \subseteq \mathrm{T}$.
Thus consider the following code in PASCAL

```
type alpha = 'a'..'z';
var a: char; var b: alpha;
a := b; (* legal *)
b := a; (* trouble -- might be out of range *)
```

What do we do in this situation? We have two options:

1. report an error at compile time
2. report a runtime error if the bounds are violated.

Pascal chooses the latter, so we say it uses *soft* type checking.
This is called **subtype polymorphism**.
It appears in object-oriented languages within the idea of a subclass.
A subclass is more restrictive than its parent, which lets it have more operations

```
Class C : public p;
P *p = C; // ok
C *p = P; // bad
```

For an example, consider C's char* and char const*; which is the type and which is the subtype? Consider:

```c
char buf [2000];
char *p = buf;
char const *q = buf; // allowed, we just can't modify buf via q
*q = 'x'; // BAD
// whereas
char const str [] = "xyz";
char const *q = str; // OK
char *p = str; // allowed, but can cause runtime errors
*p = 'x'; // allowed, but we may get s SIGSEGV
```

Since char* allows for operations that const char* doesn't, char* is a subclass of char const*

A harder example: char const* const* vs char **q

```c
char **q = char const * const *p; // BAD
char const * const *p = char **q; // BAD in C due to special type rules;
// was decided to be a special case in C++
```

Since we know const restricts our operations, char** is a subclass of char const * const *.

Rule Complexoty is a double-edge sword, since strong type checking increases complexity.
This introduces the idea of **polymorphism**
≡ the property of a symbol operation to accept multiple 'forms'
For example, in C a + b works whether a,b are 'int'/'float'/'long'/'ulong'.
This is an example of compile-time polymorphism, which is used in statically checked languages.
In Python, this happens at runtime.
How does this happen (in C)?

```c
double f(double a, double b) {return a + b;}
float g(float a, float b) {return a + b;}
// these compile to different machine code
```

Polymorphism can fall into two major categories:

1. **Ad-Hoc Polymorphism**
   This has the following properties:

   - accept a finite number of types.
   - grew up organically.
   - have coherent individual rules, but little overall organization.

   This can be broken down into two subcategories:

   (a) **Overloading** ≡ the act of identifying operations and functions by types/content of parameters.
   Functions with the same name will have different ABI/API, and the compiler chooses by type.
   The above two functions are an example, which may beg the question,
   "Wouldn't it be better to just operate on the 64 bit float always?"
   NO

   - it would be slower
   - it would round incorrectly for 32 bit arguments

   Thus we choose to implement two functions of the same name.
   During assembly, the compiler does **name mangling** for identification
   ≡ adding illegal characters into names specified by the user.
   This makes compiling C with other languages tough, since names won't match.

   (b) **Coercion**
   ≡ implicit type conversion required for an expression to be valid.
   Consider the '+' operator in C. It:

   - accepts {int, uint, long, ulong, long long, ulong long float, double, long double}.
   - does not accept {char, uchar, short, ushort}.

   We would thus expect to have 196 variations of the operator, but with coercion we only need 4.
   This is because coercion follows the following rules:

   - if sizeof(T) < sizeof(int), then cast T to int.
   - if sizeof(T_1) > sizeof(T_2), cast T_1 to type(T_2).
   - else, cast both numbers to unsigned.

   This can result in some unexpected behavior:

```
int i, j;
long l;
i = j + l; // trap
// sometimes the values can even change
int i = -1;
unsigned j = i; // changed value; j == UINT_MAX
unsigned short k = i; // loses info
assert(i == j);
// this can even happen implicitly on comparison
int i = -1;
unsigned z = 0;
assert(i < z); // EXCEPTION -- i is converted to unsigned
// even literals are not immune to this behavior
assert(-1 < 2^34); // EXCEPTION -- 2^34 is converted to int, where it overflows
assert(-2^32 < 0); // EXCEPTION -- 2^32 is converted to unsigned, then negated,
                   // but since it is unsigned, it is still nonnegative
```

These two types of ad-hoc polymorphism can interact to cause undefined behavior:

```
inf f(double x, int y);
int f(int x, double y);
f(3, 5); // which one does it call?
```

2. **Parametric Polymorphism** ≡ the ability of a function or operation to take an unlimited number of types.
   Consider Java; prior to 2005, there was no polymorphism, so a class may be of the form:

```
public interface List {
  void add (Object);
  Iterator iterator();
} // these are required for any object
public interface Iterator {
  void remove(); // remove the last item accessed
  Object next(); // gets the next item in the list
  bool hasNext();
}
List l;  ... l.add("abc"); ...
for (Iterator i = l.iterator(); i.hasNext())
  if ((string)(i.next()).length() == 1) i.remove();
  // removes all length 1 strings
```

Java users complained about this runtime check and clutter for years, so Java added **generic types**.

```
public interface List <E> {
  void add (E);
  Iterator<E> iterator();
}
public interface Iterator <E> {
  E next();
  bool hasNext();
  void remove();
}
for (Iterator<E> i = l; l.hasNext())
  if (l.next().length() == 1) i.remove();
```

We have seen this idea before with C++ **templates**.
These two ideas form the basic techniques of parametric polymorphism:

(a) Templates (C++, Ada)
≡ a stand in for not yet compiled code

- faster and lower level
- full checking and optimization permitted

For instance: List<E> stands for List<int>, List<double>, etc,
BUT for each required type, a separate form of the function must be compiled.

(b) Generics (OCaml, Java, newer languages in general)
≡ only the translated machine code for types used is copied

- all checking is done on compilation

- more streamlined, less optimizeable
- represents all objects by pointers

8 Scope and Error Handling

Scope ≡ the set of program locations where a given identifier is visible.
Thus scope allows us to "hide" identifier information.

In small applications we manage scope via nesting, but this doesn't extend well.
The object oriented method of handling scope is to label each name as to how public it is going to be.
This is the approach taken by Java classes.

- public — visible to all
- protected — visible to subclasses and other classes in the same package
- default — visible to other classes in the same package
- private — visible to only self

This approach tends to be too restrictive, thus we would like a more general option.
We instead decide to have a separate section of a program with package API's.
This is the approach taken by Java interfaces.
The usage of this method is growing more popular over tine.
OCaml uses this as well; it has structures and signatures:

- structures are analogous to classes and define implementation.
- signatures are analogous to interfaces and define API, for example:

```
(* signature: *)
module type Q = sig type 'a queue;; (* gives the API for a queue *)
(* structure: *)
type 'a queue = struct
  Empty | node of int * 'a * 'a queue
  (* code for queue *)
end;;
```

But OCaml goes beyond Java with something called functors.
These compile the functions from functions to functions.
These let you "edit" the source code to conform to signatures.
Thus you can choose your visible interface!
This is more flexible, but also more complicated.

The elephant in the room in terms of scope is separate compilation.
We want to be able to compile parts of our program separately.
Scope rules are often determined by that goal.

We also want to be able to prevent propagation of errors.
Programming has a hierarchy of errors (from least to most severe):

1. implementation restriction
   The program isn't at fault; it is following the spec.
   The spec, however, limits the implementation.
2. unspecified behavior
   The spec defines a range of acceptable behaviors.
   We must be portable & robust (able to handle all cases).

   ```
   (eq? '(x) '(x))
   ; option a: two singleton lists may be created -> #f
   ; option b: the same object is used -> #t
   ```

3. signaled error
   an exception is required by the spec

   ```
   (open-input-file "foo") ; with inexistent "foo"
   ```

4. undefined behavior
   Should NEVER happen.
   Behavior is undefined (implementations are not required to even detect the error)

   ```
   (car 0)
   (car '())
   ```

Programmers often think of exceptions as the standard way of dealing with errors,
BUT we have many methods of dealing with errors:

1. fire programmers
   This does not tend to work, since errors ALWAYS appear.

2. Have the compiler check for mistakes
   This occurs in C:

   ```
   char *p;
   ...
   p = nullptr;
   ...
   *p...; // undefined behavior -- often causes a crash
   ```

   as well as in OCaml:

   ```
   let p = None or ... in
   ...
   match p with
   | None -> xxxxs
   | Some x -> ...x...;;
   ```

   But even this static checking doesn't always work; consider an index exception in C!
   We could define a class including a subset of integers that prevents this if we wanted.

3. List the assumptions made about the arguments to any bit of code
   The callee assumes the preconditions are met, and the caller is responsible for that.

   ```
   char index(char *p, int i) {return p[i];}
   // p!= nullptr, 0<= i <= sizeof(array p)
   // We can imagine a  language which uses
   // preconditions as part of its syntax as follows
   char index(char *p, int i)
   precondition(p!= NULL \&\& 0 <= i \&\& i <= isizeof(p))
   {return p[i];}
   ```

   This can be implemented via a runtime check such as an assertion.
   Static checking partially does this!

4. Define behavior for all bad inputs, returning a special value.
   The caller is responsible for checking for the value, but it can be passed along indefinitely.

   ```
   char *p = NULL; *p; // returns -128 as a special marker "bad char"
   ```

   This is not often implemented in this way specifically because it restricts the domain.
   Most machines use it as in the method of 'floats':

   ```
   double a = 127;
   double b = 0;
   double c = a/b; // returns infinite (in Java)
   double d = c + 7; // sets d to infinity
   ```

   This is great because it is safe and thus very popular!

5. Do not specifically define behavior on errors.

   ```
   int[3] a;
   a[3] = 0; // modifies some random unassuming variable
   ```

   This is often used in languages where performance is paramount, like C, C++, FORTRAN.

6. Stop the program before it can do more damage.
   I/O — crash in C/ C++ on x86-64/SEASNet (related to the abort() call)

7. Stop the program, but allow the programmer to deal with it.
   This is called **exception handling**.

```
try {
  f(x, y);
  g(w, y);
} except(Exception e) {
  // executed iff f/g throws an Exception
} finally {
  // ALWAYS executed, NO MATTER WHAT
  cleanup();
}
```

This method requires us to change the source code and handle in a structured way

In Java (and many other languages), exceptions form a hierarchy:

- Throwable
  - Error — (not the fault of the programmer, can happen anywhere; ie virtual memory exhaustion)
  - Exception
    - IOException
    - FileNotFoundException
    - . . .

Java uses static checking to ensure the user is prepared for any contingency.
Each method must declare any exception it can throw in its signature.

Exception handling is a complication:

```
int x = f(a, b);
// acquire a lock
int y = g(x, b);
// release a lock
// These operations may be performed out of order,
// or we may not return, instead performing a nonlocal jump
//
// For delicate code, exception handling may not be
// our best option; instead we may do:
int x = f(a, b);
// acquire lock
if (x != FAILURE) {
  int y = g(x, b);
  f (y != FAILURE) // release lock
}
```

We would ideally want code that lists all normal cases and follows with the unusual ones.

How are exceptions implemented?
We want to look dynamically through the stack for a stack (down to main if need be).
Thus the catcher pushes a marker, and the thrower walks down the stack looking for the first.
This cannot be done statically.

9 Memory Management

Memory management within a program is broken into two main categories:

1. Stack Management
2. Heap Management

Stack Management

Each function has an activation record called a frame that records the instance of the call.
Each frame shares the code, but the values are different.
A frame includes:

- return address
- arguments
- local variables
- temporary instances

Frames developed in a few stages:

1. FORTRAN (1958)
   Frames were static rather than dynamic; thus FORTAN had no stack or recursion
2. C (1972)
   Frames statically defined frame layout, but the instantiation is left to runtime.
   This allows recursion!
   This does not allow for a heap, or global variables.

   ```
   int f(int n) {
     ...
     char[n] a;
     ...
   } // NOT ALLOWED
   ```

3. Algol (1960)
   Frames are stored in a stack which can grow as the program runs.
   Data is accessed by two pointers (%rbp and %rsp) to the base and top of the stack, respectively.

   ```
   int f(int n) {
     ...
     char[n] a;
     ...
                   } // OKAY
   ```

   We can thus introduce the idea of **<u>nested functions</u>**

   ```
   int f(void) {
     int x;
     int g(void) {
       int y;
       ...
       return x + y;
     }
     g();
   }
   f();
   ```

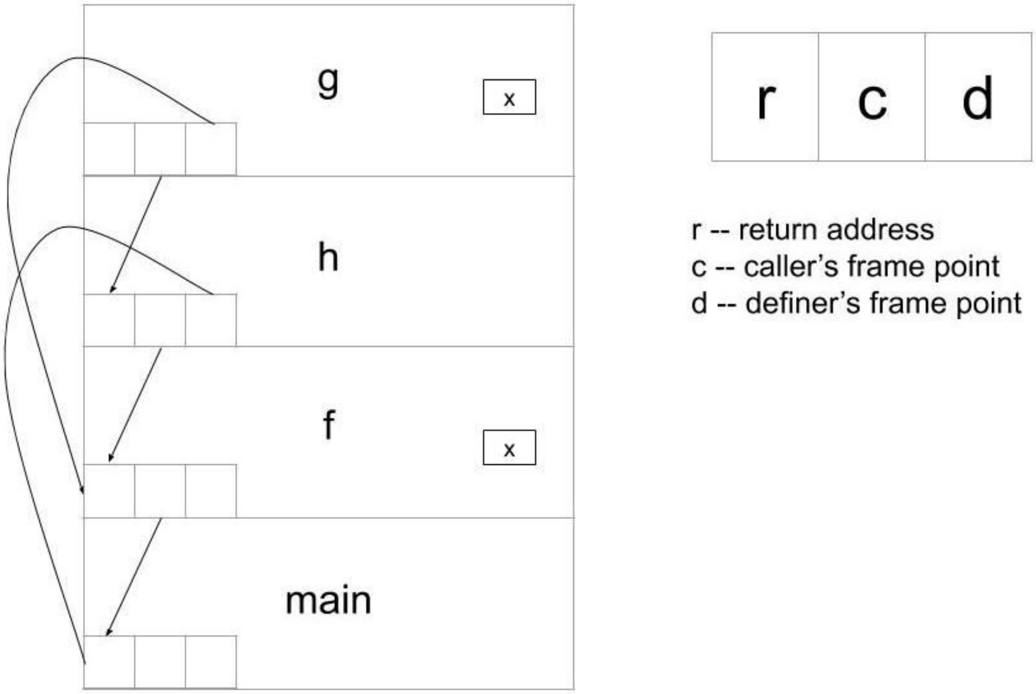


Figure 1: A model nested function call

We store two linked lists of addresses.

(a) **The Static Chain**
This forms the connection of definer frames.
The depth is the nest level.
This tends to be pretty short.

(b) **The Dynamic Chain**
This forms the connection of caller frames.
The depth is the call level.
This can be relatively long.

To make this work, functions are represented by a pair of pointers

(a) a pointer to the code (ip)

(b) a pointer to the defining frame (op)

A function is thus an (ip, op) pair.
These pointers are called FAT, since they hold two words as opposed to thin C pointers.
This is how continuations work!

4. ML A nesting function can return whilst the nested is still valid (currying).

```
(fun x -> fun y -> x + y) 3;;
-: 'a -> 'a = <fun>
(* this returns a function that remembers the 3! *)
```

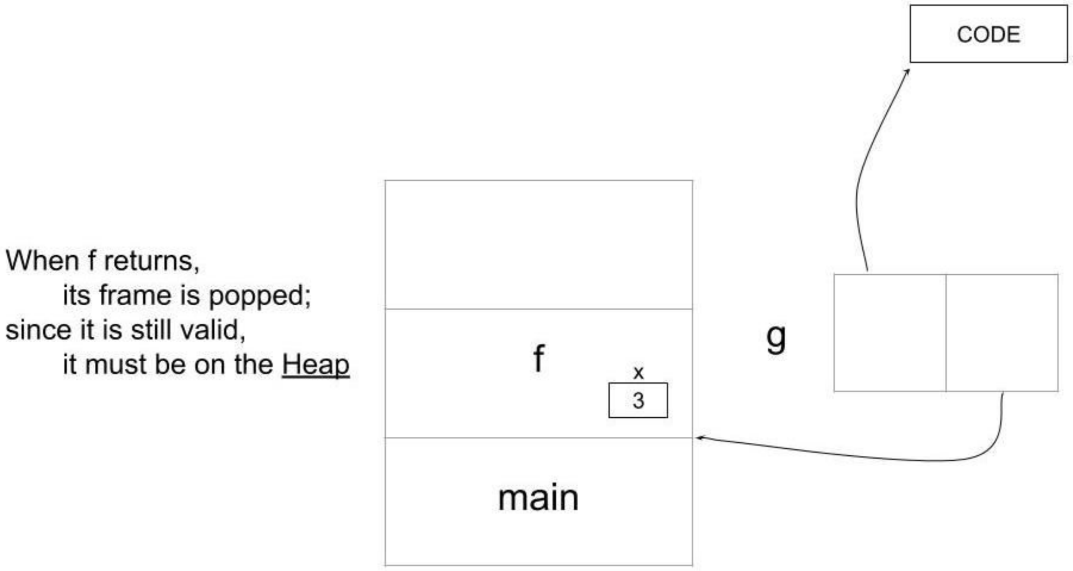


Figure 2: A currying frame

The stack can be expanded or contracted with a single pointer addition instruction.
Thus heap management is far more expensive, so function calls are thus not as efficient.

Heap Management

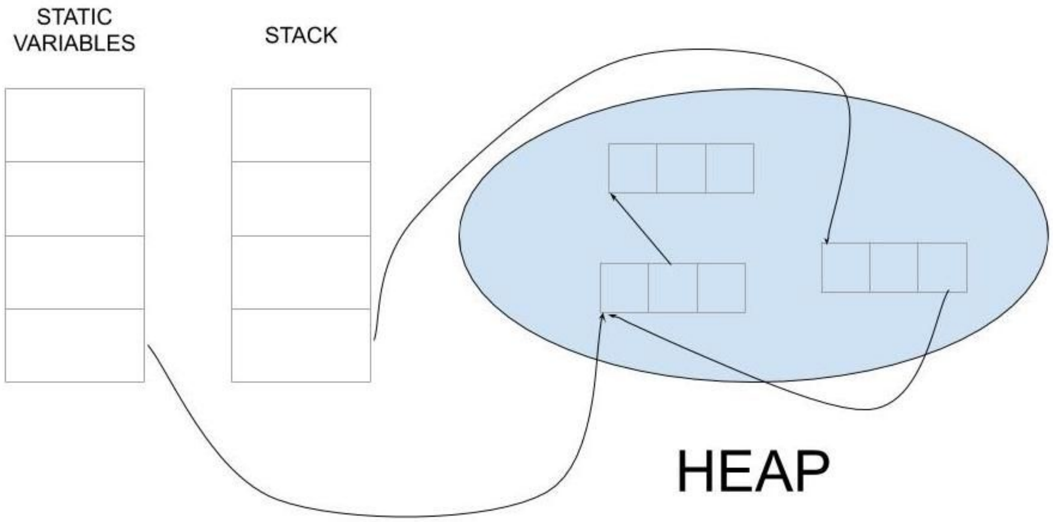


Figure 3:

How do we arrange to free unused storage?
The simplest approach: **explicit free operation**
Problems:

- It is a hassle for the programmers.
- A programmer may forget to free. (memory leak!)
- A programmer may preemptively free. (dangling pointer → undefined access!)

We can prevent this with **garbage collection**
≡ there is no 'free' — the underlying memory management system handles it.
Problems:

- How do we kept track of pointers into the heap (roots)?
  A static table is kept with a frame template and indirect pointers to the heap.
- How do we free indirect pointers to the heap?
  mark & sweep; recursively mark all reachable elements, then free all unmarked.
- How does the heap managed keep track of used/unused space?
  We keep a free list in unused block 1 for fast allocation

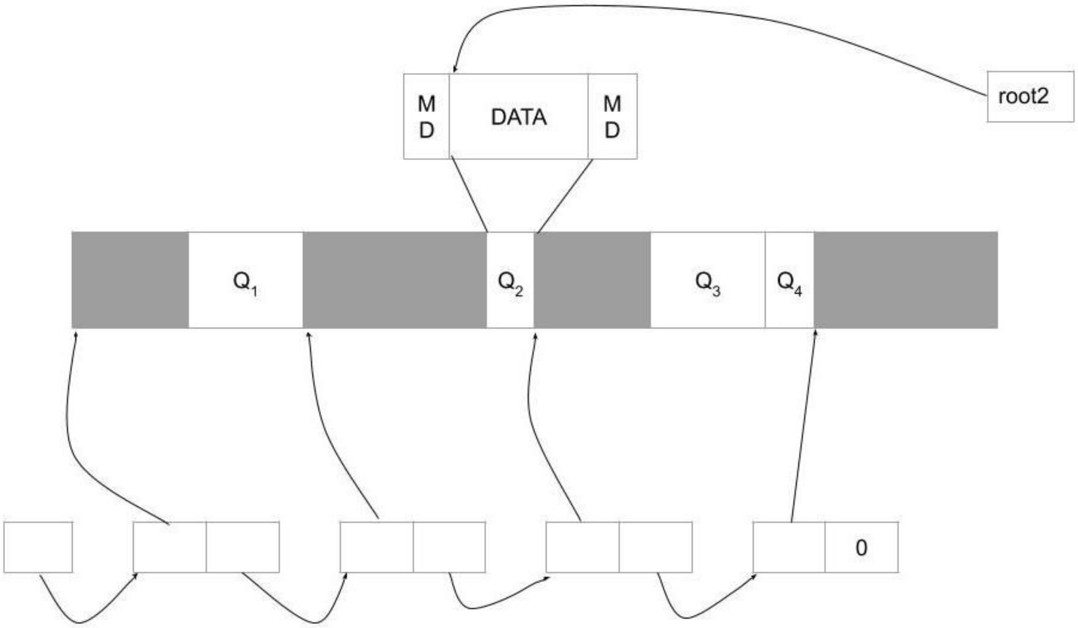


Figure 4: A model of a free list

The heap can become very large as the program's data gets very large.
This can lead to delays, as memory allocation can take up to linear time.
Additionally, blocks near the beginning of the heap will tend to be more fragmented.
If they fragment too much, they chew up CPU time as runts.
Thus we use a **roving pointer/next-fit list**
≡ leave the free pointer at the location of the last successful allocation

We must be sure to make a distinction between using a free/del operator & a garbage collector.
If we are using free/del, keeping track of roots is delegated to the program.
The program is then responsible for not misusing the primitive (arbitrary free, double free, etc.).
This can be more efficient if the program is well written, BUT it is less reliable, since:

- Dangling Pointers (usage of freed storage)
- Memory Leaks (allocated and never freed memory)
- Imposter Pointers (free (void *) 2323; — luckily 2323 is not an address on SEASNet)

The garbage collector is by far the most common choice in modern languages.
Thus garbage collectors avoid the above problems; how?

- Storage is not reclaimed if it is reachable from roots.
- We want the GC to be as accurate as possible in determining which storage is reachable
  BUT reclaiming immediately can be expensive, so we sometimes wait.
- Languages do not allow forging of pointers (in at least Java and OCaml)

Mark and Sweep is generally a pretty good algorithm, but it makes malloc O(object count).
How can we speed it up?

1. Mark and Sweep in a separate thread? NO — the locks will almost always be a bottleneck.
2. Combine GC with C/C++?
   Our reflex is to say no, since
   (a) C lets us theoretically access any address.

(b) The compiler doesn't share roots or object layout with the program

BUT it is possible with **conservative garbage collection**.
The garbage collector doesn't know which data is a pointer, but it does have memory bounds.
The objects that the heap manager knows about are:

- in registers (16 in x86-64)
- on the stack (%rsp in x86-64)
- in static variables (*.o are static files)

We search in these locations and assume any pointers within range are valid
This can lead to issues:

```
char msg[] = "\300\147...";
long long int gates_balance = 347140743208473;
// might happen to be a pointer to the heap.
malloc(1); // returned (char *)347140743208473 a while ago.
// The conservative GC will think that 1-byte object is still in use.
```

BUT if we are careful, wrong-assumptions should be rare:
We place our heap is in obscure locations, with addresses not likely to look like actual numbers.
If the wrong assumptions are rare, then we won't have a lot of leaks.

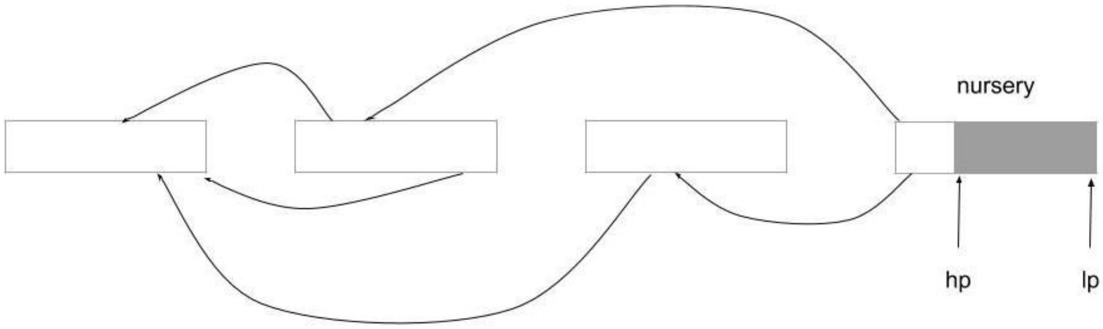
Who actually does the garbage collection?
Typically garbage collection is done in user-mode code without much formal help.
This is done for performance reasons.

The heap manager knows where objects are but does not show the OS.
For instance, one could find the source code for malloc() and free() written in C.
(Mostly, except malloc could request a huge chunk of RAM)
This is a reasonably popular option and is used in the C++ gcc code and C emacs code.
The idea of this is that we never need to call free!

```
#define free(p) ((void*) p)
```

There are two major garbage collection methods:

1. **Generation-Based Copying Collector (Java)**



Older objects appear farther to the left; new allocations are taken from the nursery.
Objects tend to point to older objects that rarely mutate, so garbage is in newer generations.
In actuality, there is some overlap, since there are exceptions for backward pointers.
We thus only need to garbage collect the nursery, and the roots to a degree.
The generation-based keyword refers to the the separately collected generations.

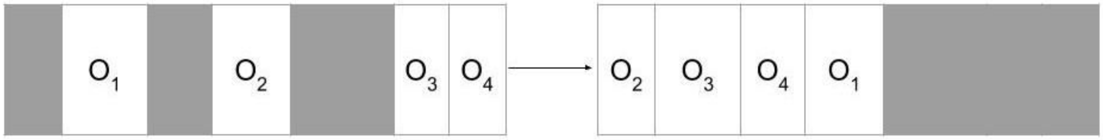


Figure 5: Removing fragmentation from the nursery

The copying keyword comes from the merging to remove fragmentation.

Upsides:

(a) Runtime is O(bytes used).

(b) We need not access free areas.
This might seem small, but avoiding a free list means we don't have to fill the L1 & L2 cache.
This is huge if the objects are small.

Downsides:

(a) We have to do a LOT of copying.

(b) We may need to free data not directly controlled by the GC.
We need to change any pointer referring to these objects.
We call Object.finalize() to collect garbage from a given object.
This is a method that by default does nothing, and is used for cleanup.
Mark and sweep can solve the finalize() issue.
Upon sweep, unmarked objects have finalize() called.

Since this collector doesn't sweep, Java has to blend the two approaches:
Java does generation-based garbage collection for standard objects.
It does mark and sweep for objects with a user-defined finalize() method.

ISSUE — Multithreading.
The naive multithreaded garbage collector requires synchronization, and thus is slow.
Java avoids a global lock by giving each nursery its own lock.
Typically, threads share only old objects, so in practice this works well.
In the worst case, this reduces to the global lock case.

ISSUE — Real-Time.
In this case, we don't care particularly about the speed of 'new', but predictability.
Thus we do an incremental garbage collection with some marking or sweeping each call.
This can be tricky, but is often used in Java applications.

2. **Reference-Counting Garbage Collector (C-Python)**
Each object keeps track of the number of references to it.

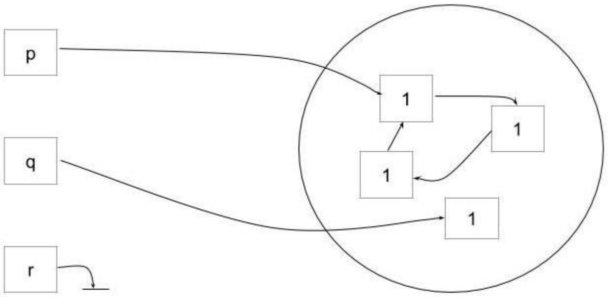


Figure 6: A model nested function call

Positives:

(a) quick garbage discovery/reclamation

Negatives:

(a) cyclic object patterns are not reclaimed
(b) Lost references

Not having cycles isn't inherently terrible, but people laughed off Python's method.
This is because they either needed avoid cycles or remember to break them.
Nowadays, Python integrates an extra phase of mark & sweep.
The timing of the mark and sweep is dependent on multiple heuristically tuned parameters.
Therefore, we need to optimize these when building a large application.

A trick that sometimes works and sometimes fails miserably is called **object polling**.
Say we know we are going to allocate a bunch of 'struct cons' objects. We then do

```
struct cons { struct cons * car; struct cons *cdr; };
// We thus allocate en masse and keep our own free list
struct cons *free_conses;
struct cons *next_cons(void) {
  if (free_conses) {
    struct cons * r = free_conses;
    free_conses = r->car;
    return r;
  }
  return malloc(sizeof(struct cons)); // expensive
}
void free_cons(struct cons *p) {
  p->car = free_conses;
  free_conses = p;
}
```

This works with mark and sweep, is but terrible for generation, since it copies the free list.

10 Object-Oriented Programming

We have worked with object-oriented languages before,
BUT there is a difference between OO-Languages and OO-Programming.
We can write object-oriented programs in C (or even assembly)
We would use 'struct' for classes.
We would use a table of pointers to functions for virtual functions.
...
BUT we can also write non object-oriented programs using OCaml.

But even object-oriented programming itself is not a monolith:
Consider the implementation of Python and that of Java:

- Python uses dynamic typing whereas Java uses static.
  Thus Java has a more complicated syntax (interfaces. abstract classes, generics)
- Many properties, however, are shared:
  - Classes bundle together fields and functions.
  - Instantiation is the way to create objects.
  - Inheritance is class-based.
  - Classes are types.
  - Classes control namespaces

One area of large variance is in the version of inheritance that the language uses:
Java uses single inheritance — classes form a tree rooted at 'Object'.

```
o.m();
// look in o's class,
// then its parent,
// grandparent, ad in infinitum
```

Python uses multiple inheritance — classes form a DAG, and rules are more complicated.

```
o.m()
# look in o's class,
# up each of its parents' ancestry trees,
# sequentially
```

So is Java object oriented? some people say no, since it has primitives.
Is Scheme object oriented? in some ways, since all data is treated as an object.
We can thus see that we need to be careful when others say 'object oriented'.

Javascript, however, takes a different approach to object-oriented programming:
This approach is called prototype-based (as opposed to class-based).
In Java, there are two main ways to create an object of type T.

```
T x = new T();
// OR
T o = ...;
T x = o.clone();
```

Javascript only allows instantiation via the 'clone' method.

```
Thread t;
// clone a class ("prototype")
greenthread = t.clone();
// modify it to be how you like
greenthread.color = "green";
// we then treat greenthread as a prototype in itself
gt = greenthread.clone();
```

Thus inheritance is left to us, which gives us more control and simplifies at the low level.
There has been a lot of ink spilled debating prototype vs class-based oo-programming.
Prototyping tends to work well with dynamic type checking.
The class-based approach works well with static type checking.

Another area with multiple approaches is the method of parameter passing a language uses.
This, as well as syntax comes down to the issue of correspondence;
How do we match the caller's arguments with the callee's parameters?

1. Positional Correspondence ≡ match based on position in the parameter list.
   This is an approach taken by C, LISP, and most languages since FORTRAN
2. Varargs Correspondence ≡ match a fix number of parameters, then pool the rest.
   This can be seen in Scheme:

   ```
   (define foo (lambda (x . rest) ...))
   (foo 1 2 3 4) ; -> {x/1, rest/(2, 3, 4)}
   ```

   Python takes a similar approach:

   ```
   def foo(x, *rest):
     ...
   foo(1, 2, 3, 4) # -> {x/1, rest/[2, 3, 4]}
   ```

3. Keyword/Key Correspondence ≡ the caller specifies the argument/parameter binding.
   Python allows this:

   ```
   def arctan(y, x):
     ...
   arctan(x=5, y=10)
   ```

   Python even allows mixing of the three

   ```
   def f(x, *y, **z):
     ...
   f(1, 2, 3, x=10, y=20)
   # -> \{x/1, y/(2, 3), z/\{’x’=10, ’y’=20\}\}
   ```

Semantics come down to the issue of calling conventions.
These are more important to programmers since they affect correctness.
There are many methods of calling functions:

1. **call by value**
   ≡ caller evaluates arguments, gets values, and passes to callee;
   callee then has a copy and can do whatever it wants with it

   ```
   // caller:
   y = f(x);
   assert(y == x);
   // callee:
   x++;
   return x;
   ```

   This can cause performance problems:

   (a) Suppose a large value space (ie a long array)
   (b) Suppose an expensive function call
   (c) Suppose an unused result or error — extra computation!

2. **call by reference**
   ≡ caller passes a copy of a pointer that the callee can inspect/verify
   This is commonly used in C:

   ```
   int f(int &x) {return x++;}
   z = 9;
   return f(z);
   // in C, this is compiled as if it were
   int f(int *px) { return (*p)++; }
   int z = 9;
   return f(&z);
   ```

   This avoids copying overhead, but introduces problems with aliasing:

   ```
   void trouble(int &x, int &y) {
     x = 5;
     y = x + 7;
   }
   int z = x;
   trouble(&x, &z); // TROUBLE
   void trouble (int &x, int &y) {
   ```

```
  x = 5;
  y = x + 7;
  z = x; // x has gotta be 5 here, normally
} //  (BUT NOT ALWAYS); consider:
z = x = 5;
y = 12;
// Faster and equivalent, right?
// Nope, not if you have call by reference.
//
// aliasing introduces complications:
int n = 10;
int z;
trouble (n, n);
// compiler cannot optimize because aliasing is possible
```

There are, however, ways to get around these optimization limits

```
char *strcpy(char restrict *dist, char const restrict *src);
strcpy(x, x); // GCC compiler warning -- no aliasing
```

3. **call by result**
≡ caller tells callee where the result should go, and the result is copied there
This lets a function return a varying result.
This lets functions communicate in the opposite direction as call by value.
The variable is instantiated on call and initialized on return.
This is commonly used in ‘Ada’, but is used in C as well:

```
<unistd.h>
  ssize_t read(int filedes, void *buf, size_t nbyte);
  // buf is really a call by result!
```

4. **call by unification**
≡ parameters are unified by compiler.
This is the approach of Prolog.

5. **call by value-result**
≡ parameter is passed by value and returned by result.
The callee thus has a copy it can modify which isn’t changed until return.

```
int x = 5;
int f(int n) {n++; return n + x;}
f(x); // copy n in and copy result to n
```

6. **call by macro expansion**
≡ arguments are a built in piece of the program, and so is the result.
Utilized by Scheme, C/C++, etc

7. **call by name**
≡ caller does not evaluate arguments, but instead tells the callee how to evaluate.
The callee can then choose to utilize the description whenever it likes.
This treats each parameter like an anonymous function.
Call by name is to atoms as call by referral is to pointers.
This can be done in C++++:

```
int f(int (*xf) (void)) {return (*xf)() + 3;}
// this is compiled like:
int z = 9;
int zhelp(void) {return z;} // this is a 'thunk'
return f(zhelp);
// This gives us a way to wait to evaluate!
void print_avg(int arg, int n) {
  if (n == 0) print(****);
  else print("\%d iters, avg is \%d\n", n, avg);
}
print_avg(100/0, 0);
// this succeeds with call by name because of lazy evaluation
// this succeeds with call by value because of eager evaluation
```

Our programs thus just set up a computation to be done later (often via thunks).
Nothing happens until output is needed.

This can be extended to **call by need** .
≡ call by name + cache results of thunks (Haskell, etc.).
The 'anonymous functions' returned for parameters need only be run once.
It's easy to say let x = (list of all prime numbers).
It's easy to compute with infinite objects.

The syntax we have been focusing on all quarter tends to be the easier part of language. Semantics tend to be much harder, and is broken into two categories:

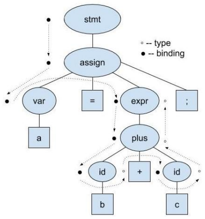
1. **static semantics** ≡ meaning you can tease out before the program runs
2. **dynamic semantics** ≡ meaning that won't be known until the program is run

Among semantics, static is considered easy, and dynamic hard
We can deal with each of these elements with the following tools:

- syntax — BNF, EBNF, syntax charts, etc
- static semantics — attribute grammars Each node in the BNF parse tree has extra information, called "attributes". These say something about the running of the program beyond the BNF.

Consider the expression 'a = b + c';
This can form the tree (right).
Downward inference is called **inheritance**.
Upward inference is called **synthesis**.
These let us derive information from grammar rules

```
plus → id + id
    type(plus) =
        if type(id1) == type(id2) == int
          then int
          else float
expr → plus
    type(expr) = type(plus)
```



Dynamic semantics must further be broken into three broad categories:

1. **operational**
   ≡ describe writing an interpreter — derived from imperative languages.
   For example, Python is defined by an interpreter, C-Python, written in C.
2. **axiomatic**
   ≡ logic you can use to reason the program — derived from logic languages.
   For example, the first semantics for lisp were defined in Lisp.
   This is philosophically "bad" but interesting in practice, like an English dictionary
3. **denotational**
   ≡ function: program → meaning — derived from functional languagea.

This is simplified of course.
Allow us to attempt to use method (c) to define a subset of OCaml in Prolog:

```
m(E1+E2, C, Vsum) :-
  m(E1, C, V1),
  m(E2, C, V2),
Vsum is V1 + V2).
% we have thus defined a high level '+' by a lower level '+'
m(K, C, K) :- integer(K).
% defines constants
m(Var, C, Val) :- member(Var=Val, C).
% defines variable binding
m(let(X, E1, E2), C, Val) :-
  m(E1, C, V1),
  m(E2, [X=V1|C], Val).
% defines 'let' binding
m(fun(X, E), C, lambda(X, E)).
% functions need not be evaluated
m(call(E1, E2), C, Result) :-
  m(E1, C, lambda(Xf, Ef)),
  m(E2, C, V2),
  m(Ef, [Xf=V2|C], Result).
```

These semantics use dynamic scoping; static would be more complicated.

The history of programming languages in a few lines, according to Paul Eggert:
"Lots of attempts to 'take over the world'"
"None have succeeded"
"We still have a lot of variety, and that is not going away"
Thus we must keep always the basic principals in mind

A OCaml

The basic approach:
Rather than modify existing code, like an imperative language, we glue functions together.
Advantages:

- compile-time checks for types (like C++, Java, & other "Professional" languages) to avoid exceptions.
- types need not be specified (like scheme, python, & other scripting languages)

  ```
  Thread x = new Thread (); //Java
  ```

  Note that pure interpreters would fail this, since they can't infer well.

  ```
  let x = Thread ();;
  (* OCaml uses type inference *)
  ```

- memory need not be manually managed (like Java, not C).
- good support for concurrency
- good support for higher-order functions

Our book chooses to use ML rather than OCaml, so why do we use OCaml?

- Intel & other companies use OCaml.
- OCaml is cleaner than ML
- It is good to see multiple forms of the same language

  ```
  (* ML -> OCaml *)
  x<y andalso z<w -> x<y \&\& z<w
  [1, 3, 5] -> [1; 3; 5]
  val x = 5 -> let x = 5
  3.0 + 4.0 -> 3.0 +. 4.0
  ```

Note that OCaml's type inference is not as fancy as ML's.

Properties of Code:

There is no redefinition of variables; only addition of new definitions.

```
# let x = 1 in
let f = fun y -> x + y in
let x = 2 in
f 2;;
- : int 3
(* f uses the definition of x within its code block *)
```

OCaml checks all types even if the code is unreachable.

```
# if 1<2 then "a" else "b";;
- : string "a"
#if 1<2 then "a" else 0;;
(* ERROR: OCaml is conservative,
   and assigns types to entire statements;
   We can get errors on non-errors, but if it compiles.
   we are guaranteed that there will be no type errors *)
```

Tuples have set length, lists have set type.

```
# (1, "a");;
- : int * string = (1, "a")
# [1; "a"];;
(* ^^^ ERROR: this expr has type string
but expr was expected of type int *)
# [1; 10; -3; 4*7] = [1; 7];;
- : bool = false
# (1, 10, -3, 4*7) = (1, 10);;
(* ^^^^^^^ ERROR: this expr has type int * int
but expr was expected of type int * ... * int *)
```

```
# [1; 5; -3] = [];;
- : bool = false
# ["a"; "b"; "c"] = [];;
- : bool = false
(* huh? How is [] both int list and char list? *)
# [];;
- : 'a list = []
(* this is called a generic type and 'a is a type variable *)
```

Functions take 1 parameter and return 1 result

```
# let f = fun x -> x+1;;
val f = int -> int = <fun>
(* <fun> is a stand in for the tree
or machine/byte code OCaml uses internally *)
# let tup = (3, 12);;
val tup = int * int = (3, 12)
# let add = fun (x, y) -> x + y;;
vall add : int * int -> int
# add tup;;
-: int = 15
(* a tuple is a way of emulating multiple parameter functions *)
```

Currying is the canonical way to emulate multi-variable functions

```
(* say we are attempting to emulate Lisp's 'cons' keywork *)
(* we do this with currying: using functions that pass/return functions *)
# let ccons = (fun a -> (fun b -> (a::b)));;
val ccons : 'a -> 'a list -> 'a list = <fun>
(* we can use this in the standard two-variable way: *)
# ccons 2 [5; 9; 27];;
-: int list = [3; 5; 9; 27]
(* but we can also take advantage of the function return value *)
# let prepend_3 = (ccons 3);;
val prepend_3 : int list -> int list
(* thus we can create various functions without recompilation! *)
(* this cannot be done with either element of a tuple! *)
(* in practice, OCaml lets us simplify the declaration *)
# let ccons = fun a b -> a::b;;
val ccons : 'a -> 'a list -> 'a list = <fun>
(* because of this, function calls are left-associative:
   f g h = ((f g) h)
   but function types are right-associative:
   int -> float -> int = (int -> (float -> int))
   and the keyword 'list' has the highest priority \& '->' the lowest
   int -> float * float list = int -> (float * (float list)) *)
```

Pattern matching is the canonical way to test expressions

```
# match 1 with
| int n -> true
| _ -> false ;;
(* this returns true for ints and false otherwise *)
(* we can use this in function parameter lists as shorthand *)
# let car (h::_) -> h;;
# let car = fun l -> match l with | (h::_) -> h;;
(* ^^^^^^^^^^ WARNING:
   This pattern matching is not exhaustive
   Here is an example of a case that is not matched:
   [] *)
val car : 'a list -> 'a list = <fun>
(* this compiles, but throws an exception if we violate it *)
# cdr [];;
Exception: Match_failure("//toplevel//", 1, 8).
(* we would thus like to make car safer : *)
# let safer_car = fun l -> match l with
| (h::_) -> h
| _ -> 0;;
val safer_car : int list -> list = <fun>
(* this only works for integers, we would like it to be more general *)
# let scar = fun d -> fun l = match l with
| h::_ -> h
| [] -> d;;
val scar : 'a -> 'a list -> 'a = <fun>
(* we can apply this specifically to turn it into our earlier safer_car *)
# let safer_car = scar 0;;
val safer_car = int -> int list -> int = <fun>
```

One can define custom types (in particular, unions).

If we were to define a custom type in C

```
union u (long l; char *p;);
union u v;
// both of the following share an address location
v.l = 12;
v.p = "abc";
// so consider a function
int f(union u a) { // is a.l valid or is a.p? We can't know!

(* in OCaml, data type is known
    note that type constructors in OCaml are capitalized *)
# type mytype =
| Foo
| Bar of int
| Baz of int * int;;
(* this leaves us with three constructors: # Foo; # Bar 12; #Baz (3, 5);
   we can use these constructors for pattern matching \& thus determine type *)
# match myvalue with
| Foo -> 0
| Bar x -> x
| Baz (y, z) -> y + z;;
(* these unions can be generic: *)
# type 'a option = | None (* None = type 'a option *)
| Some of 'a;; (* Some + 'a = datatype *)
(* this typing can be used to define an option for any type, string included
   We have thus defined a safer version of the nullptr;
   The nullptr can give runtime errors,
   but the OCaml compiler would give thisto us as a compile-time error *)
(* we can even define types recursively *)
# type 'a list =
| []
| 'a::'a list;;
```

We can use union pattern matching to define "polymorphic" functions:

```
# fun x -> match x with
| None -> 0
| Some y -> y;;
(* this is common enough that we have a shorthand *)
# function | None -> 0 | Some y -> y
(* this is the second major shorthand we have seen*)
# fun x y -> x * x + y = fun x -> fun y -> x * x + y
(* we could combine these notations; the following 3 are equivelant *)
# let car (h:::_) -> h
# let car = fun h_::_ -> h
# let car = fun x -> match x with (h::_) -> h
(* BUT we don't; we use fun for currying \& function for recursion*)
```

OCaml is very conducive to recursion

```
(* let's try our canonical recursive function; reversing a list *)
# let reverse = function
| [] -> []
| h::t -> (reverse t)@h;;
```

```
Characters 69-76:
| h::t (reverse t)@h
Error: unbound value reverse
(* OCaml has an iron-clad rule -- no usage pre-definiton
   thus even x = x + 0 throws an error
   We need this functionality for recursive functions,
   so we use keyword rec *)
# let rec reverse = function
| [] -> []
| h::t -> (reverse t)@h;;
val reverse : ’a list list -> ’a list = <fun>
(* that’s not right; We want ’a list -> ’a list! See: *)
# reverse [[3;4]; [-1;-3]; [5]; [6;9;12]];;
int list = [6; 9; 12; 5; -1; -3; 3; 4]
(* the static type checking caught our mistake! *)
# let rec reverse = function
| [] -> []
| h::t -> (reverse t)@h;;
(* this works, but @ is O(n), so reverse is O(n^2); we need better!
   we need an accumulator; solve a specific problem by solving a general
   this is the opposite of C, in which we get more specific to get speed *)
# let revapp a = function
| [] -> a
| h::t -> (* ? *);;
# let reverse l = revapp l [];;
(* we just need to determine what goes in place of the question mark
   |h|t| |a| -> |t’|t|a| -- |t|a| is cheap to compute! Thus our function is:
   PLUS the program doesn’t have to save stack since it won’t return! *)
# let rec apprev a = function
| [] -> a
| h::t -> apprev (h::a) t;;
```

Function type definition is only as general as the most specific operator/variable.

```
(* we would like to find the minimum value in a list *)
# let rec minlist = function
| h::t -> let m = minlist t in
if h<m then h else m
| [] -> (* ? *);;
(* if we were doing sumlist, we would use the addition identity 0;
   if we had a minlist identity we would use it, we instead pass this to user *)
# let rec minlist id = function
| h::t -> let m = minlist id t in
if h<m then h else m
| [] -> id;;
val minlist : int list -> int = <fun>
(* (<) binds the function to integers; we must pass the comparison to user! *)
# let rec minlist lt id = function
| h::t -> let m = minlist lt id t in
if lt h m then h else m
| [] -> id;;
val minlist (’a -> ’a -> bool) -> ’a -> ’a list -> ’a = <fun>
# minlist (<) 100000000 [3; -5; 7; -20];;
- : int = -20
```

B Java

In Java, all types are a subset of the general Object.
Thus, we may write the following code:

```java
List<String> ls;
List<Object> lo = ls;
lo.add(new Thread());
String s = ls.get();
// BOOM ~ type error; we assigned a thread to a string
```

The Java compiler can catch this, but when should it send the message?

The error is at line (2); implicit type conversion, which subverts the type mechanism.
Therefore, Java doesn’t allow this, since List<String> !<= List<Object>.

This is counterintuitive; it feels like List<String> should be a subclass of List<Object>.
The problem is that lists are mutable; => our intuitions on subtypes don’t extend to subclasses.

This introduces another issue: what if we want to print an Object List?

```java
void printList(List<Object> lo)
{
  for (Object o: lo)
    System.out.println(o);
}
printList(ls); // ERROR ~ List<String> is missing this method
```

We could write a polymorphic function, but this would require us to write duplicate code.
Thus we use a **wildcard**.

```java
void printList(List<?> l)
{
  for (Object o: lo)
    System.out.println(o);
}
printList(lo); // OK
printList(ls); // OK
```

Now suppose the method to be called is restricted in type:

```java
void displayShapes(List<?> l)
{
  for (Shape s: l)
    displayShape(s);
}
displayShapes(List<String>);
// allowed by ? but String has no displayShape()
```

Thus we must use a **restricted wildcard**.

```java
void displayShapes(List<? extends Shape> l)
{
  for (Shape s: l)
    displayShape(s);
}
/* ... */;
l.add(new Shape());
// may be disallowed since ? doesn't necessarily = Shape
```

Now suppose we want a function with two parameters

```java
void convert(Object[] ao, List<Object> lo)
{
  for (Object o: ao)
    lo.add(o);
} // adds all of ao to lo
convert(String[], List<String>);
// ERROR ~ List<Object> != List<String>
```

Again, we could just duplicate code, but that is inefficient.
We also cannot just use a wildcard

```
void convert(Object[] ao, List<?> lo)
{
  for (Object o: ao)
    lo.add(o);
} // ERROR ~ casting ? to Object
```

We instead use a type parameter

```
<T> void convert(t[] ao, List<T> l)
{
  for (T o: ao)
    l.add(o);
}
String[] as = /* ... */;
List<Object> lo - /* ... */;
convert(as, lo);
// ERROR ~ List<Object> is not a supertype of List<string>
```

We can enforce this type rule with another bounded wildcard

```
<T> void convert(T[] a, List<? super T> l)
{
  for (T o: ao)
    l.add(o);
}
```

This complicated type algebra came naturally to meet demand;
  How is it implemented?
The Java language does not specify an implementation, but a common approach is:

- Every Object is a piece of storage with an address.
- The storage begins with a type ‘tag’.
- This ‘tag’ points to a runtime representation of the type.

This, however, does not capture the notion of generic types;
  We don’t have a separate representation for each implementation of the generic.
  Instead we point ‘tag’ to List<?> and trust the compiler to type check; this is called **erasure**.
This simplifies the system and lets us implement Java on simpler JVM’s without generics.
This also lets us have only one copy of function code and static variables.

This is not perfect, however; consider the following usage

```
List<Shape> ls = /* ... */;
List<Rectangle> lr = (List<Rectangle>) ls;
// the standard compiler cannot do this check with the above information
```

As a result, we do not cast to generic types;
  instead we utilize Duck Typing

Duck Typing

“If it waddles like a duck and quacks like a duck, it is a duck." Objects don’t have types, only methods.
Instead of worrying about what an object ‘is’, we worry about what it ‘does’.

For example, instead of this standard approach:

```
if (isaDuck(o))
{
  print "is a duck; it will waddle";
  o.waddle();
  o.quack();
}
```

We simply attempt the behavior with a contingency (Pseudocode):

```
try {
  o.waddle();
  o.quick();
} except {
  //deal with non-ducks
}
```

This approach is common in languages like Python, Ruby, Smalltalk, etc..

Is it possible to implement duck typing at compile time?
Yes! it was even one of the major motivations for Java:

History of Java

In 1994, Sun Microsystems was working on Solaris

- kernel in machine code/apps in C/C++
- ran on SPARC workstations and multiprocessor servers

Sun knew C worked well on their complicated machinery, but what about IOT? They built a lab to address the following issues:

1. C/C++ crash easily, which is unacceptable.
2. SPARC is expensive, so code needs to be made for all machines.
3. Networking is VERY slow, so C/C++ updates would be long.
4. C++ is too complicated, you can't know all of the rules.

They considered two approaches:

1. Use C++, but remove the confusing stuff = C+-
2. Steal from XEROX PARC
    - invented the mouse, internet, IDE, Smalltalk (OO dynamic typing)
    - build workstations for text processing (but charged too much)

Smalltalk solved some problems of C, but introduced some new ones:
The good:

- interpreted, so no crashing
- garbage collection rather than new/del, so more reliable
- operated via byte code, so machine independent
- Object-Oriented

The bad:

- weird syntax
- single-threaded
- terrible marketing

Solaris ultimately blended the two and called the result "Oak" The marketing department renamed it "Java" They decided to market it by writing a browser The main browser at the time was Mosaic bu UIUC

- crashed a lot (C++)
- required rewriting & recompiling to extend
- would ultimately diverge:
- netscape → mozilla → firefox
- internet explorer → microsoft edge

So Sun decided to create an extensible browser;

- it would accept Java bytecode and run it in a sandbox
- it was popular for demos but isn't used much anymore
- it convinced Eggert that Java was great for server-side applications

Java Functionality

- compile-time checking — to avoid crashing like Smalltalk
- variables are always initialized — to avoid inconsistency like SmallTalk
- left-to-right order of evaluation — avoid crashing (*p++ / *p++ is undefined in C)
- well defined primitive types — consistency (a*b+c → fmul+fadd, not fmuladd)
- wrapped pointers — to avoid inconsistencies in behavior (because ptr length)

Let's consider how this last one works with the edge case of Arrays:

- are a reference type
- mandate subscript checking
- are all held on the heap (and thus return/passable)
- have fixed size when allocated
- are typed

Much of the rest of Java is analogous to C++, the exception being inheritance

Inheritance

Java has single inheritance, so the class hierarchy forms a tree.
Duck typers may be worried about this rigidity
eg. suppose you have a length() and a width() method;
A hierarchy cannot represent all versions of both!
We address this problem using Interfaces; they declare functions, but leave the class to implement them.

```
interface Lengthable
{
  int length();
}
interface Geometry2D extends Lengthable
{
  int width();
}
class Canvas implements Geometry2D
{
  int length() {return -3;}
  int width() {return 27;}
}
```

There is a hierarchy here, but

- a class can implement multiple interfaces.
- parents pass code, but interfaces pass "obligations"/API.
- interfaces form some type of a static version of duck typing.

Interfaces are often not powerful enough though;
if we use the same one many times, we may end up duplicating code.
We can solve this with abstract classes, which merge interfaces and classes

```
abstract class C
{
  int length() {return 1;}
  abstract int width();
}
new C(); // ERROR ~ width() is undefined!
abstract class D extends C
{
  int height() {return 3;}
}
new D(); // ERROR ~ width() is undefined!
class E extends D
{
  int width() {return 2;}
}
C x = new E();
foo(x);
```

Final classes are in some sense the opposite of abstract.
Abstract Classes force you to subclass and define methods.
Final Classes don't allow you to subclass and define sub-methods.

A Final Class thus forms a leaf in the class hierarchy.
Both classes and methods can be declared as final & thus can't be overwritten.
What are they good for?

1. Provides a barrier preventing subclasses from misbehaving
2. Low level reasons we won't discuss

A bioengineer wrote a simulator of the contents of the cell in Java.
It was a mess, unsurprisingly, since it was written by a bioengineer.
Everything was declared 'final'... Why? "because it made things faster!"
Finals make things faster! Why? It lets the compiler 'inline' the body of the method.

Concurrency

Concurrency is one of the biggest advantages of Java.
To discuss it, consider the question: what is the world's fastest computer?
This will let us think ahead and plan such that our language lasts The world's fastest computer: SUMMIT:
CPU in C/assembly to squeeze out performance.
RedHat operating system (like SEASNet).
Most of the power goes toward cooling but still relatively power efficient.
Power cost of roughly a million laptops!
Speed is determined by LINPAC.
≡ Matrix multiplication floating point operations (FLOP) per second
This is the speed of 149e6 laptops!
Java was designed at the supercomputers of 1990!

The idea was to lash many processors/cores together.
There were two main philosophies for this type of parallelism:
SIMD — multiple instructions on one data set MIMD — multiple instruction pointers for multiple data sets (ex MIPS) The computer they looked at (POWER) does both:
it has Nvidia (GPU-like devices) & array instructions for small arrays (SIMD).
it has simple instructions for complicated structures (MIMD).

Java decided to focus on the latter:
The Java abstraction for MIMD is called the "Thread" Each Thread corresponds to one computation with its own IP The life cycle of the Thread:

1. Thread t = new Thread(); — NEW
2. t.start(); — RUNNABLE
   (a) compute — RUNNABLE
   (b) yield — RUNNABLE
   (c) IO — BLOCKED
   (d) wait — WAITING
   (e) sleep — TIMEDWAITING
   (f) return from run — TERMINATED

This leaves the Thread Object as the Threads "tombstone".

They are used in one of two ways

1. the classic object-oriented method — subclass Thread & override the run() method

```
class Foo extends Thread
{
  void run() {
    /* this code will be run when you start the thread */}
}
Foo f = new Foo;
f.start(); // allocates resources and calls t.run()
```

2. the preferred, duck typing method — define a new class implementing Runnable

```
// the interface is along the lines of:
interface Runnable
{
  void run();
  ....
}
class Bar implements Runnable
{
  void run() {/* ... */}
}
Bar b = /* ... */;
new Thread(b);
```

Threads have a major problem though, and that is the problem of Thread communication.
Java's solution is shared memory.
A Thread T can communicate with a Thread U by modifying a shared Object o

```
// T:
O.set(x);
// U:
...
u = get()
```

The main problem with this is the problem of race conditions.
get() may be called earlier or even concurrently with setting, causing a garbage get().
Race Conditions are a big enough problem that Java has multiple techniques.

1. Embarrassing Parallelism
   this is the simplest and easiest, but is SLOW

   ```
           Thread t = /* ... */;
           t.join(); // waits for t to finish
   ```

2. Volatile Instance Variables
   The 'volatile' keyword tells the compiler it is intended for communication.
   T he compiler then gets the variable on each usage, so access is not optimized away (some compilers place this constraint on themselves; this is very specialized & slow) Not only does this influence speed, but also correctness:

   ```
   y = f(x) + g(x); // x may change between function calls
   y = x - x; // optimization would make y zero
   while (x == x) ; // optimization would make this an infinite loop
   // volatile variables still leave room for errors, though:
   class C
   {
     volatile int x, y; // invariant: x < y
     int set(int newx) { x = newx; y = newx + 1;}
     int getx() {return x;}
     int gety() {return y;}
   }
   int a = getx();
   int b = gety();
   if (a >= b) disaster_strikes();
   // we may grab x before a change and y after
   ```

3. Synchronized Methods/Monitors
   Only one synchronized method may be used on an object at the same time.
   This slows you down, since Threads much wait to execute.
   Effectively places a lock on the object → methods must be VERY fast.
   This fixes many errors that volatile doesn't.

   ```
   class C
   {
     volatile int x, y;
     synchronized void set(int newx)
     {
       if (newx < newx + 1)
       {
   ```

```
      x = newx; y = newx + 1;
    }
  }
  int getx() {return x;}
  int gety() {return y;}
  synchronized long getBoth () {return ((long) x << 32) + y;}
}
long ab = getBoth();
long a = ab >> 32;
long b = ab & 0xFFFFFFFF;
if (a >= b) disaster_strikes();
```

4. Waiting for events
   We use this when our method is too slow to synchronize

   ```
   o.wait(); // remove all locks and wait until all are free again
   o.notify(); // wake up one of the waiting Threads (FIFO is common)
   o.notifyAll(); // wakes ALL the waiting threads
   // can be used to allow programmer's choice of next Thread
   ```

   While Wait() does work in all addressed scenarios, it is too low level for many people.

Exceptions

These are error conditions that interrupt the regular flow of a program The most general syntax is much like the syntax of C:

```
System.out.print("1");
try {
  System.out.print("2");
  if (true) throw new Exception();
  // if(true) so no code is unreachable
  System.out.print("3");
} catch (Exception e) {
  System.out.print("4");
} finally {
  System.out.print("5");
}
System.out.print("6");
// 12456 -- finally is run even with an exception!
```

Java implements them in an interesting way:

There is a class called Throwable which contains two subclasses:

- error — exceptions that cannot be caught
- exception — exceptions that can be caught

Exceptions derived from error or exception.runtimeException are unchecked.
Checked exceptions cannot be ignored at ANY LEVEL.
We can use this to implement our own exceptions:

```
public class OutOfGas extends Exception
{
  private int miles;
  public OutOfGas (String details, int m)
  {
    super (details); miles = m;
  }
  public int getMiles()
  {
    return miles;
  }
}
try {
  throw new OutOfGas ("You have run out of gas");
} catch (OutOfGas e)
```

```
{
  System.out.print(e.getMessage());
  System.out.printIn("Odometer: " + e.getMiles());
}
```

This can be done in two major ways, which are built into Java

1. **Cyclic Barrier**
   Threads run independently, but wait at a barrier point.
   Runnable object run() may be called every time the last Thread reaches the barrier.

   ```
   // Example: Matrix Decomposition
   class Solver
   {
     final int N;
     final float[][] data;
     final CyclicBarrier barrier;

     class Worker implements Runnable
     {
       int myRow;
       Worker(int row) { myRow = row; }
       public void run()
       {
         while (!done())
         {
           processRow(myRow);
           try {
             barrier.await();
           } catch (InterruptedException ex) {
             return;
           } catch (BrokenBarrierException ex) {
             return;
           }
         }
       }
     }

     public Solver(float[][] matrix)
     {
       data = matrix;
       N = matrix.length;
       barrier = new CyclicBarrier(N, new Runnable() {
         public void run() {
           mergeRows(...);
         }
       });
       for (int i = 0; i < N; ++i)
         new Thread(new Worker(i)).start();

       waitUntilDone();
     }
   }
   ```

2. **Exchanger**
   Works similarly to a "take a variable, leave a variable" drop box

   ```
   // swap buffers between threads such that
   // filling thread gets an empty one when full
   class FillAndEmpty {
     Exchanger<DataBuffer> exchanger = new Exchanger<DataBuffer>();
     DataBuffer initialEmptyBuffer = ... a made-up type
     DataBuffer initialFullBuffer = ...

     class FillingLoop implements Runnable {
       public void run() {
         DataBuffer currentBuffer = initialEmptyBuffer;
   ```

```
      try {
        while (currentBuffer != null) {
          addToBuffer(currentBuffer);
          if (currentBuffer.isFull())
          currentBuffer = exchanger.exchange(currentBuffer);
        }
      } catch (InterruptedException ex) { ... handle ... }
    }
  }

  class EmptyingLoop implements Runnable {
    public void run() {
      DataBuffer currentBuffer = initialFullBuffer;
      try {
        while (currentBuffer != null) {
          takeFromBuffer(currentBuffer);
          if (currentBuffer.isEmpty())
          currentBuffer = exchanger.exchange(currentBuffer);
        }
      } catch (InterruptedException ex) { ... handle ...}
    }
  }

  void start() {
    new Thread(new FillingLoop()).start();
    new Thread(new EmptyingLoop()).start();
  }
}
```

C Prolog

Let's look back at our approaches:
Imperative — glue together commands via sequencing (side effects are key)
Functional — glue together functions (give up side effects)
Declarative — glue together predicates with (&, |, →)

It may seem like by using logic programming we are giving up a lot,
BUT in return we "don't have to worry about how our program works"
Declarative programs are written by specifying goals rather than methodology
In a sense, we declare constraints on a solution, and our interpreter finds a solution.
Our algorithms have two major parts:

1. Logic — specifications of correct solutions to the problem
2. Control — advises the interpreter for efficiency (does not affect speed)

In a sense, all of our logic programs utilize divide and conquer!
The alternative would be Java, which mixes the two parts:
for (i = 0; i < N; i++) f(i); // if we change to i<N-1, it effects correctness.

The basics:
pred(Arg1, Arg2, ...) forms a goal
',' is AND & '.' is end (at the top level)
':-' is a turnstile that implies "if"
pred(Arg1, Arg2, ...) :- cond1, cond2. forms a rule
This means that pred is true if cond1 and cond2 are true

Our first logic program will be SORTING

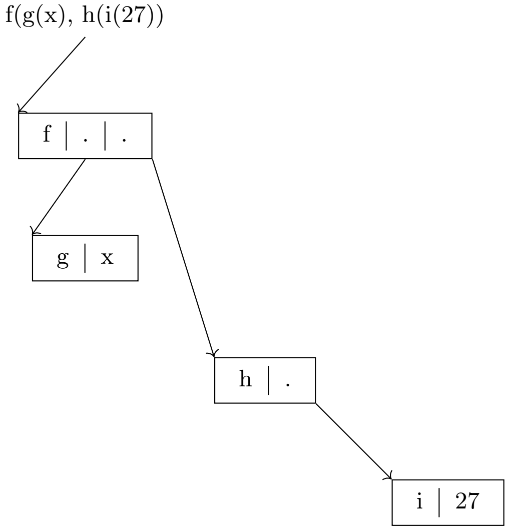
```
% First we write specifications for what it means to sort a list
% L -- unsorted list
% P -- sorted list
% L&P must be permutations of each other
% P must be sorted
% First we state the top level requirements:
sort(L, P) :- permute(L, P), sort(P).
% sort is true for L & P if permute and sort are true
sorted([]).
% the empty list is sorted
sorted([_]).
% a single element list is sorted
sorted([X, [Y|L]) :- X =< Y, sorted([Y|L]).
% X::Y::L is sorted if X <= Y and Y::L is sorted
permute([], []).
% the empty list is a permutation
permute([X|L], R) :-
permute(L, PL),
append(PL1, PL2, PL),
append(PL2, [X|PL1], R).
append([], L, L).
append([X|L1], [X|L2], L) :- append(L1, L2, L).
% a fun consequence of this is that we can run code in either direction!
?- sort([1, 9, -3], R).
R = [-3, 1, 9].
?- append([a, b], [c, d, e], R).
R = [a, b, c, d, e].
?- append(X, [c|Y], [a, b, c, d, e]).
X = [a, b].
Y = [d, e].
% a predicate can even return multiple; we type ';' to see the next
?- append(x, y, [a, b, c, d, e]).
x = [], y = [a, b, c, d, e] ;
x = [a], y = [b, c, d, e] ;
x = [a, b], y = [c, d, e] ;
x = [a, b, c], y = [d, e] ;
x = [a, b, c, d], y = [e] ;
```

```
x = [a, b, c, d, e], y = [] ;
no.
```

This should have been obvious to us; we utilize append in this way in our code!
While our code is simple and documents itself well, append is O(N), so permute and sort are O(N!)
This may lead one to think Prolog is inefficient, but consider the N-Queens problem; Prolog can solve it in 3 lines of code at roughly C speed!
It is up to us to utilize Prolog in an efficient way!
We will now lay out the basic syntax of Prolog (Edinburgh Syntax): A Prolog term is one of:

1. atom — #, $[a - z][A - Za - z\_0 - 9]*$
   These represent constants within our programs.
   Alternatively, we can use single quotes to turn anything into an atom.
   Atoms are unique and equivalent to only themselve.

2. (logical) variable — $[\_A - Z][A - Za - z\_0 - 9]*$
   Variables are bound/instantiated on success. They are only unbound on failure. Bound variable assignments cannot be changed We avoid names beginning with '_'; these are generated by the interpreter
3. structure — atom(term1, term*)
   The starting atom is called the functor.
   We refer to a structure by functor/#args
   The name is apropos; these are represented as trees



That's it! Anything else we have seen is only "syntactic sugar"

- [] == '[]'
- $[a, b, c]$ == '.'(a, '.'(b, '.'(c, '[]')))
- $[X, Y|Z]$ == '.'(X, '.'(Y, Z))
- a :- b, c, d. == ':-'(a, ','(b, ',' (c, d)))

But if predicates are represented as structures, how do we actually evaluate them?
We use the keyword 'is'

```
?- N is 2+2.
N = 4.
?- is(N, 2+2).
N = 4.
```

This tends not to be used much, since declarative statements are rare in Prolog.
Prolog instead tends to represent first-order logic

Structures can take one of three forms:

1. fact — a statement that is true

   ```
   true.
   prereq(CS31, CS131).
   prereq(CS31, CS111).
   equals(X, X).
   ```

2. rules — a predicate that is conditionally true

```
ipr(A, B) :- prereq(A, Q), ipr(X, B).
ipr(A, B) :- prereq(A, B).
% this compiles fine, but the below causes an infinite loop:
ipr(A, B) :- prereq(A, B).
ipr(A, B) :- prereq(A, Q), ipr(X, B).
% all of these rules are considered the same predicate,
% since they share functor and argument count
```

3. goals - queries to be evaluated

```
?- X = 3.
X = 3.
yes.
?- ipr(CS31, Z).
Z = CS131? ;
Z = CS111? ;
no.
```

Failure

In the real world, we have TRUE, FALSE, and UNKNOWN, BUT
in C/Java we have T or F and we deal with unknowns.
in Prolog we have success or failure to prove.
Thus failure becomes a major factor in flow control.

A simple example: Say the registrar provides us with the following predicate:

```
prereq(CS31, CS32).
prereq(CS31, CS111).
prereq(CS31, CS131).
prereq(CS131, CS132).
```

Can we conclude that dance 101 is not a prerequisite for CS131?
Logically NO, we just can't prove it,
BUT in prolog we assume a closed world and thus that the list is complete
Therefore, this would fail!

Ordering

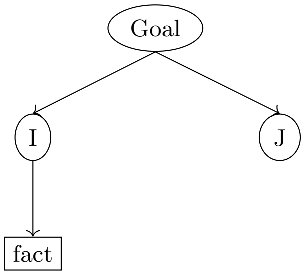
A clause can refer to a predicate not yet defined so long as it is defined at usage time. While ordering does not affect correctness (in general), it affects efficiency.

Prolog is goal oriented, so the search is bottom up.
We backward-chain to build a search tree
leaves = facts
branches = successful rules
Failed rules are removed from the tree
A parse tree is a form of search tree — it proves a sentence validity in a grammar.

At any point in time, Prolog has a partially build tree.
It matches the provided goal to the heads of the clauses, in order, until it finds a match, when:

1. it is a fact => we have a leaf
2. it is a rule => we have a subgoal to evaluate!

The interpreter then works on subtrees from left to right, depth-first, and backtracking.



ipr(A,B) =
match G to
H := I, J
H = ipr(X, Y)
J = prereq(X, M)
I = ipr(M, Y)

Infinite loops are relatively easy to fall into:

```
memb(X, [X|_]).
memb(X, [_, L]) :- memb(X, L).
/* the standard forward usage would be: */
|?- memb(X, [a, b, c, d, e]).
X = a? ;
X = b? ;
X = c? ;
X = d? ;
no.
/* we can get strange repeat cases:*/
|?- memb(a, [a, b, x, y, d, e, z]).
true? ;
X=a? ;
Y = a? ;
Z = a? ;
no.
/* we can even test the goal without any literals */
|?- memb(A, B).
B = [A|_]? ;
B = [_, A|_]? ;
B = [_, _, A|_]? % this gives us an infinite loop!
app([], L, L).
app([X|L], M, [A|LM]) :- app(L, M, LM).
|?- app(X, Y, [a, b, c]).
X = []
Y = [a, b, c]? ;
X = [a]
Y = [b, c]? ;
X = [a, b]
Y = [c]? ;
X = [a, b, c]
Y = []? ;
no.
/* even this predicate is not safe, though: */
```

We will now write our canonical reverse:

```
naive_reverse([], []).
naive_reverse([X|Y], Rx) :- naive_reverse(L, R), append(R, [x], Rx).
% this is O(N^2); we must use a general solution with an accumulator!
revapp([], A, A).
revapp([X|L], A, R) :- revapp(L, [X|A], R).
% now for the application:
fast_reverse(L, R) :- revapp(L, [], R).
% We can compare the two implementations:
|?- naive_reverse([a, b, c], R).
R = [c, b, a].
|?- fast_reverse([a, b, c, d, e, f, g], R).
R = [g, f, e, d, c, b, a].
|?- naive_reverse(L, [a, b, c, d]).
L = [d, c, b, a]? ;
Fatal Error: global stack overflow!
% what happened? can fast_reverse handle it?
|?- fast_reverse(L, [a, b, c, d]).
L = [d, c, b, a]? ;
Fatal Error: global stack overflow!
```

How can we debug this?
We use the Four Port Debug Model

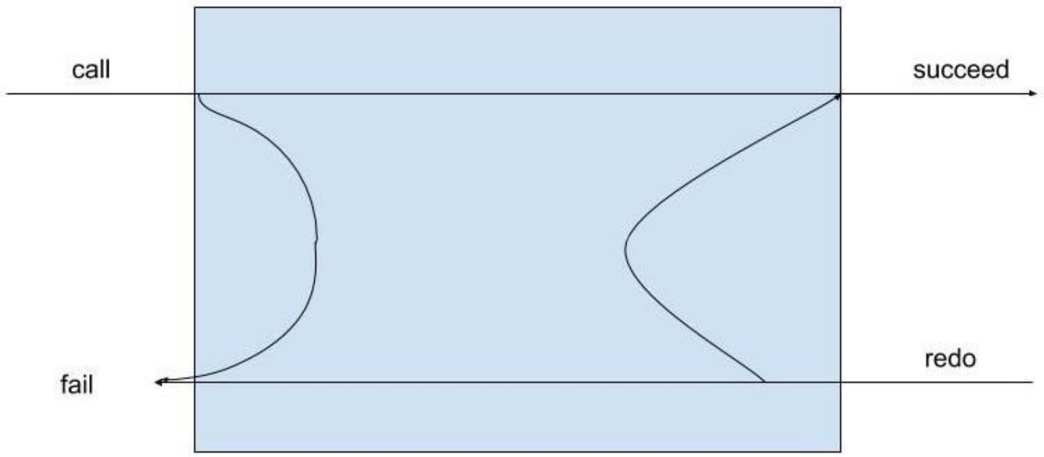


Figure 7: We place a "spy" at each port

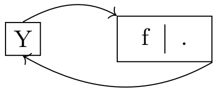
```
| ?- trace, naive_reverse(L, [a,b]).
The debugger will first creep -- showing everything (trace)
1 1 Call: naive_reverse(_23,[a,b]) ?
2 2 Call: naive_reverse(_63,_102) ?
2 2 Exit: naive_reverse([],[]) ?
3 2 Call: append([],[_62],[a,b]) ?
3 2 Fail: append([],[_62],[a,b]) ?
2 2 Redo: naive_reverse([],[]) ?
3 3 Call: naive_reverse(_89,_128) ?
3 3 Exit: naive_reverse([],[]) ?
4 3 Call: append([],[_88],_156) ?
4 3 Exit: append([],[_88],[_88]) ?
2 2 Exit: naive_reverse([_88],[_88]) ?
5 2 Call: append([_88],[_62],[a,b]) ?
5 2 Exit: append([a],[b],[a,b]) ?
1 1 Exit: naive_reverse([b,a],[a,b]) ?

L = [b,a] ? ;
1 1 Redo: naive_reverse([b,a],[a,b]) ?
2 2 Redo: naive_reverse([_88],[_88]) ?
3 3 Redo: naive_reverse([],[]) ?
4 4 Call: naive_reverse(_115,_154) ?
4 4 Exit: naive_reverse([],[]) ?
5 4 Call: append([],[_114],_182) ?
% the system finds a match of the appropriate length,
% and if we continue it looks for a longer one, which it won't find!
% NOTE: we could also perform a manual debug if we wanted ("app" is self-chosen)
|?- append(X, Y, [a, b, c, d]), write(app(X, Y)), nl, fail.
```

Unification

≡ the process of making two terms equal
Consider: f(g(Y), h(Z)) - f(A, h(i(B))) → f(g(Y), h(Z))
A = g(Y), z = i(B) is called the unifier, and it makes the statements equivalent.
When we have earlier said "matching" heads, we were referring to unifying.
We use the following terminology:
Fact: p(a, b)
Goal: p(X, b)
Unifier: X=a
This is used to transmit information from the callee to the caller.

Matching can even bind variables:
Fact: p(X, b) :- q(X)
Goal: p(a, b)
Thus we can see that unifying is more powerful than matching or switches;
these let us take apart values, but Prolog can let you build as well!
A single unification can even communicate in both directions!

Fact: e(X, X)
Goal: e(Y, f(y))
|?- e(Y, f(Y)).
→ Y=f(f(...



Fact: f(g(Y), h(Z))
Goal: f(A, h(i(B)))
BUT this can lead us into trouble...

1. we could get an infinite loop ->
2. we could break the logic of a problem!

Consider the example of Peano Arithmetic:
We wish to come up with a logical basis for mathematics.
Peano decided to implement nonnegative integer arithmetic. But Godel's Incompleteness Theorem says a logic cannot prove itself!
Let's try anyway:

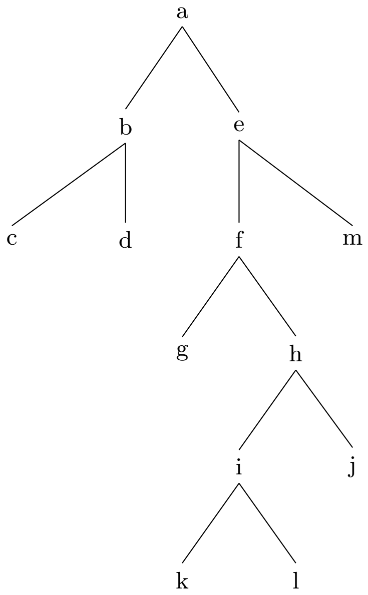
```
% zero = 'zero'
% N+1 = 'succ(N)'
sum(zero, N, N).
sum(succ(N), M, succ(NplusM)) :- sum(N,M,NplusM).
```

Control

We can actually model our tree a different way:
leaves are answers
nodes are "choice points"
In this model, our behavior at choice points determines our efficiency; this is called **control**.
Consider the following examples:

```
fail. % this is hard wired, and produces an empty search tree
true. % this forms a tree with a single node
% the following is akin to C's while(true), and is built in
repeat.
repeat :- repeat.
|?- repeat, write(X), fail. % infinite loop of X's
% this forms a right-recursive search tree, and finds infinite solutions
% the following is not built in, but is logically equivalent to repeat
loop :- loop.
loop.
% this forms a left recursive search tree, and thus will find no solution!
% it therefore cannot be proven
```

Suppose we have the following search tree:



Suppose we are at point g.
Suppose we know that if we fail, the sibling subtree of h will.
Then we can use the cut (!) to prune alternatives.
The cut succeeds on first encounter, but fails on backtrack.
Doing this increases the efficiency of our programs!

The usage is as follows:

```
 complicated(X, Y, Z) :- generate(X, Y, Z), test(X, Y, Z).
 % suppose we know that test/3 is slow and will fail unless X is in Y
 % we can thus perform this check before performing our slow function
 complicated(X, Y, Z) :- generate(X, Y, Z), member(X, Y), test(X, Y, Z).
 % this still has an efficiency problem!
% consider X = a, Y = [a, b, c, a, c, a, d, a]
% our program will perform test 4x more than necessary!
% we can improve this with the cut!
memberchk(X, [X|_]) :- !.
memberchk(X, [_|L]) :- memberchk(X, L).
% thus we can greatly improve our efficiency:
complicated(X, Y, Z) :- generate(X, Y, Z), memberchk(X, Y), test(X, Y, Z).
```

This is great, but the cut is very much the ‘goto’ of prolog: it is risky!

```
 \+(P) :- P, !, fail.
 \+(_).
 % this only succeeds if P is false!
 % it is not equivalent to 'not', consider:
 |?- X=1, \+(X=2), write(ok).
 % writes 'ok'
 |?- \+(X=2), X=1, write(ok).
 % fails, since X is already bound to 1
 % thus \+ is not commutative!
```

This may seem like semantic nitpicking, but the idea is actually central to prolog.
For example, theoreticians would say
  $\vdash$P means “P is provable”
  $\models$P means “P is true”
These may seem equivalent, but they very much are not!
  $\vdash$P but not $\models$P $\Longrightarrow$ your logic is inconsistent (BAD)
  $\models$P but not $\vdash$P $\Longrightarrow$ your logic is incomplete (DISAPPOINTING)
The goal of developing a logic is to meet both requirements
This has been met for some subsets of logic (Euclidian Geometry, etc.).
This is not possible for integer arithmetic due to Godel’s Theorem/Turing’s Halting Problem

Thus \+ really means “not provable” (It even looks like $\not\models$)
The closed world assumption lets us use \+ in some cases:

```
 prereq(CS31, CS131).
 % ...
 |?- \+ prereq(dance101, CS131).
 % succeeds, which is correct with a closed world
 |?- \+(X=1).
 % fails, since X is not currently bound
 % we thus call X a ground term,
% and we insist \+ takes only ground terms
```

D Scheme

Scheme is a variant of LISP, the second programming language created, still in use in AI!
Scheme has the following properties:

1. Objects are dynamically allocated and never freed.
   Like: OCaml, Java, Prolog
   Unlike: C/C++
   We assume a garbage collector is used to clean up memory.

2. Types are properties of objects, not of programs or expressions.
   Like: Python, Prolog
   Unlike: Java, C/C++
   In Scheme terminology, we say types are latent, not manifest.

3. The scope of a variable is known statically
   Like: OCaml, Java, Python, Prolog, C/C++
   Unlike: CLISP, zsh
   We call this static scoping := to find an identifier's reference we search containing blocks
   This is as opposed to dynamic scoping := we search calling functions.
   We can see this clearly in OCaml function definitions:

```
fun b -> fun c -> fun a -> a + b + c;;
(* as opposed to *)
let f = fun a -> a + b
in let b = 10 in f 10;;
ERROR: b is not defined (* This would be allowed in LISP! *)
```

4. Parameters are passed by value.
   Like: OCaml, Java, C/C++ (non-pointer)
   Unlike: Prolog (call by unification)
   call by value := the caller must evaluate parameters, then pass by copy.
   This allows the caller to do (mostly) whatever it wants with the data.
   We say mostly because there are a few occasions with pointer equivalents.

5. A wide variety of build in objects are provided.
   A particularly useful object is a continuation which is effectively a 'goto' on steroids.

6. The syntax is very simple.
   This allows us to treat programs as objects and modify/inspect easily.

7. Arithmetic is high-level and reasonably machine-independent.
   Integers are 'unlimited' in size (6 64-bit memory locations, little endian).
   Rational numbers are exact (represented by numerator/denominator).
   BUT this only holds with addition and multiplication.
   For more complicated operations, this falls back to float.

8. Tail recursion is always supported.
   Tail recursion is not required by the C standard, but consider the following program:

```
int g(int x) {return h(x + 1);}
int f(int x) {int y = x + 5; return g(y+7);
```

   g() and h() are tail calls, which allow an important optimization:

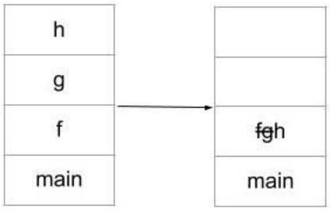


Figure 8: Tail call support

We can reuse stack frames!
While this is arbitrary in the above case, it is key with recursion!
A note: this is dangerous in C due to side effects, but we use Scheme purely functionally.

Syntax

```
; comment
identifiers; string of ascii characters not beginning with number, ’+’, ’-’, ’...’, ’->’
;These are used for 3 things, all of which can be seen in the line:
        (let (x ’abc))
        ; variables -- (x)
        ; keywords --  (let)
        ; symbol -- (’abc)
(pa . ir)
(l i s t); a pair chain ending with ’()
#t #f ; booleans; ONLY #f is false
"str\ning"
#\c ; character
’x ; data item x, non-evaluated
‘(f x (g ,y)) ; evaluates only y
```

Semantics

Nearly everything in Scheme is represented by a function call.

```
; Format:
(function arg1 arg2)
; all elements are expressions evaluated left to right prior to call

; Scheme provides a few standard functions:
(cons x y); yields a pair (x, y)
(car x); yields the first element of pair x
(cdr x); yields the second element of pair x
(eq? x y); yields #t iff x & y are the same object
(eqv? x y); yields #t iff x & y have the same value
        (eqv? 5 5) ; -> #t
        (eq? 5 5); -> undefined (depends on compiler representation)
(equal? x y); yields #t if x & y have the same recursively dereferenced value
( = x y); yields #t iff x & y are numbers with equivalent values
(list? x); yields #t iff x is a list
(null? x); yields #t iff x is the empty list ’()
(eq? x ’()) ; There is only one empty list, so this is equivalent

; there are a few specific special cases:
; (1)
(quote x); yields x without evaluation

;(2)
(lambda FORMALS EXPRS); yields a function: FORMALS -> EXPRS(FORMALS)
; we can allow any number of parameters:
(lambda (FORMAL . ARGV) EXPRS); yields a lamba function w/ FORMAT+ arg count
; binds parameter1 to FORMAL, rest as dotted list to ARGS; similarly
(lambda . FORMAL EXPRS); 0+ parameter special case, since .() is an error
; data is represented as (IMPROPERLIST . ’())
        (list a b c); -> (a . (b . (c . ())))
        (cons a b); -> (a . b)
        ’(a b); -> (quote . ((a . (b . ())) . ()))
; thus we are "hacking" the representation

; (3)
(define NAME VALUE); binds evaluated NAME to VALUE (can be lambda expr)
; though this can be used for constants, it is often for functions
(define (NAME FORMALS) EXPRS); == (define NAME (lambda(FORMALS) EXPRS))
; we can use this with the dot for very concise definitions:
        (define (list . x) x)
        (define (id x) x)
        (define (cons x y) x.y)

; (4)
(if TEST THEN ELSE); evaluate TEST; if #t THEN else ELSE
; why is this necessary; couldn't we do it as a function? Like
        (define (funif test then else) #| flow control|#)
        (funif #t 97 26); -> 97
        (funif #t 0 0/0); -> ERROR
; NO -- if must evaluate lazily rather than eagerly
; we can use this to implement 'not'
        (define (funnot x) (if x #f #t)

; (5)
(and (E1 ... En)); yield #f on first failure, else last expression's value
(or (E1 ... En)); yield first true, else #f
; but wait, can't we just implement this with 'if'?
        (define (funand . args)
          (if (null? args)
            #t
            (if (not ((car args))
              #f
             (funand (cdr args))))))
; NO -- we again cannot prevent eager evaluation

; there is another form that is tempting to include on this list:
(let BINDINGS EXPRS); evaluate vals in BINDINGS, assign to vars, evaluate EXPRS
; but this functions just like lambda; consider:
        (let ((x (+ 3 5))
          (y (* 2 7)))
          (+ (* x x) (* y y)))
; is equivalent to
        ((lambda (x y)
          (+ (* x x) (* y y)))
          (+ 3 5) (* 2 7))
; thus 'let' can be thought of as "syntactic sugar"
; it is included to allow our code to be written in evaluation order
```

Now we will write our canonical reverse function:

```
(define (reverse ls)
  (if (null? ls)
    ls
    (append (reverse (cdr ls)) (list (car ls)))))
; but unsurprisingly, this would be (N^2)
; we then use a named let
(define (reverse ls)
  (let revapp ((ls ls) (a '())
    (if (null? ls)
       a
      (revapp (cdr ls) (cons (car ls) a)))))
; this preserves execution order and lets us use easy tail recursion!
```

We use functional programming with Scheme, so set! is considered taboo,
but there is an element even more controversial:

Continuations

Proponents say they are the "essence of Scheme."
Opponents say they are too low level.
Professor Eggert likes them, but thinks simple ones are often too low level.

What is a continuation?

Note that an interpreter always keeps track of:
1. What to do next? (IP) %rip
2. What env to do it in? (EP) %rbp/%rsp
A continuation packages these two things as a pair.
Therefore the program can easily control flow!

We can make a continuation at any time with call-with-current-continuation.

When we call (call/cc PROCS), the system:

1. Creates a continuation pointing at the return address of call/cc.
2. Calls PROC with the continuation as a parameter.
3. Returns whatever PROC returns

How do we use a continuation? An example:

```
(define (product ls)
  (call/cc
    (lambda (break)
      (let (prod ((ls ls))
        (if (null? ls)
          1
          (if (zero? (car ls))
            (break 0)
            (* (car ls) (prod (cdr ls))))))
; we use a continuation to "escape" from a deeply nested recursion
```

This example allows us to "play with the program's brain" with a "nonlocal goto".
This is a common idea in many languages: consider exception handling

```
        try {/* exception */} except(E e) {/* catching code */;}
```

We can do this with continuations in the following way:

1. Create a continuation k.
2. Give it to the main code.
3. If k is used, we are back at the top level.

We can use continuations in a more powerful way to jump back into a returned function!
A common usage for this is Green Threads — multithreading with one CPU.

This is common in small, single core environments in the IoT.
The basic idea:

1. Use continuations to switch from one thread to another.
2. Cooperative scheduling with yield including continuation construction.

```
; list of threads to run
(define thread-list '())
; add a thunk (no parameter call) to the thread list
(define (new-thread thunk)
(set! thread-list (append thread-list (list thunk))))
; start a given thunk
(define (start)
  (let ((next-thread (car thread-list)))
    (set! thread-list (cdr thread-list))
    (next-thread)))
; give up the CPU
(define (yield)
  (call/cc
    (lambda (k)
      (new-thread (lambda() (k 42)))
    (start))))
; usage:
(new-thread (lambda() ((let (f (display "h") (yield)) (f))))
(new-thread (lambda() ((let (f (display "e") (yield)) (f))))
(new-thread (lambda() ((let (f (display "y") (yield)) (f))))
(new-thread (lambda() ((let (f (newline) (yield)) (f))))
(start) ; -> hey\nhey\nhey\n.........
```

This would traditionally be very hard to do in C but we have two ways of approaching it.
Note: C CODE

```
#include <setjmp.h>
// this library defines:
  #typedef /*...*/ jmp_buf
  // a machine independent array == continuation
  int setjmp(jmp_buf env);
  // uses %rip, %rsp to save to jmp_buf & return 0
  _noReturn void longjmp(jmp_buf env, int val);
  // puts VAL into %eax (return value) and jumps to jmp_buf w/ a continuation
  // this effectively utilizes a try with a long jump ~= throw
  // this is not as powerful as scheme -- we can only jump out 1 frame max
  // -> we can do ’product’ but not ’green-threads’

// we have another more powerful version, but that makes it more controversial
#include <wcontext.h>
  // this library defines:
  #typedef /*...*/ wcontext_t;
  // another version of a continuation
  int getcontext(wcontext_t *p);
  // creates and stores a current continuation
  int setcontext(wcontext_t const *p);
  // resume from continuation *p
```

Scheme is a simple core language with an easy way to extend on syntax.
These are called syntax extensions, and we can use it to redefine ‘let’

```
(define-syntax let
  (syntax-notes ()
    ((let ((name val) ...) body1 body2 ...)
      ((lambda (name) ...) body1 body2 ...) val ...))
        ((let tag ((name val) ...) body1 body2 ...)
          ((letrec ((tag (lambda (name ...) body1 body2 ...)))
        tag)
      val ...))))
; we could also redefine ’and’
(define-syntax and
  (syntax-rules ()
    ((and) #t)
    ((and e) e)
    ((and e1 e2 ...)
    (if e1 (and e2 ...) #f))))
; usage:
(and ((i 1) (< -10 i)) -> (if (< i 1) (< -10 i) #f)
; clearly, we could also do this with ’OR’, right?
(define-syntax syn-or
  (syntax-rules ()
    ((syn-or) #f)
    ((syn-or e) e)
    ((syn-or e1 e2 ...) (if e1 e1 (or e2 ...))))
; BUT this doesn’t work -- consider:
> (or (getenv "PATH") (getenv "PATH") "bin:/usr/bin")
    (if (getenv "PATH") (getenv "PATH") ("/bin:/usr/bin"))
; getenv is evaluated twice -- this is expensive and may have side effects!
(define-syntax syn-or
  (syntax-rules ()
    ((syn-or) #f)
    ((syn-or e) e)
    ((syn-or e1 e2 ...)
    (let ((x e1))
    (if x x (or e2 ...))))))
; but this has an even more subtle issue -- CAPTURE; consider:
(let ((x "usr:/usr/bin"))
  (or (getenv "PATH") x))
; this would be transformed to:
```

```
(let ((x "usr:/usr/bin"))
  (let ((x getenv "PATH"))
    (if x x (or x))))
; Our macro expansion has captured x in a context it doesn't belong!
; Scheme resolves this by resolving scope BEFORE macro expansion
; These are called "hygienic macros", since they are "cleaner" than C
; This doesn't require inter-compiler agreement since it is internal
```

Scope of identifiers in Scheme:

```
; most of the time, Scheme scope functions statically and like C
(let ((x 3))
  (let ((y 4))
    (let ((x (+ y 1)))
      (* x y))))
; -> 20
; BUT there are cases where this is not true
(define (foo)
  (let ((x 3))
    (let ((x (+ x 5))
           (y (+ x 2)))
        (* x y))))
> (foo) ; 40
; Scheme evaluates variables first to last, so we would expect 80 but get 40
; Thus we can see the scope of a variable defined by 'let' does not extend
; into the definitions of other parallel defined variables
; Why is this? Because of the equivalence of 'let' to 'lambda'!
(define (foo)
  (lambda (x)
    ((lambda (x y) (* x y))
    (+ x 5) (+ x 2))
  3))
; Scheme uses the 'let*' keyword for sequential definition:
(define (foo)
  (let* ((x 3))
          (y (+ x 2)))
  (* x y)))
> (foo) ; 15
```