Week 1 - Tuesday (2019-04-04)

Donald E. Knuth "The Art of Computer Programming"

TEX Was written in Pascal

Literate Programming

- have comments that are written in LaTeX, allowing the code to process the source code as both a piece of software and also as documentation
- main idea is to maintain documentation of source code inside of the source code so you don't have to update two things at once
- example: Java, source code for the Java library is maintained in the documentation in the same file

Quiz Solution:

```
tr -c -s 'A-Za-z' '[ln*]' | sort | uniq -c | sort -rn
```

- tr -c means complement, so translate everything that's not a letter character
- tr -s means squeeze repeated outputs (generates a single newline between each word)
- uniq -c means count the duplicates
- sort -rn means sort by count (reverse and numeric sort)

Main Concepts of the Course

1. Principles and Limitations of programming models
2. Core Programming Paradigms
3. Learn about the various tradoffs for different types of programming languages

Theory	Practice
Language Design	Ocaml
Syntax	Java
Semantics	Prolog
Functions	Scheme
Names	Python
Types	? (Kotlin)
Control	
Objects	
Exceptions	

BASIC Developed on the GE 225 Mainframe (1961)

- 40 micro-seconds to add integers
- 500 micro-seconds to divide integers
- ~40 KB of RAM
- Timeshared with 20 other users

C (C++) Developed on PDR11 (~1975)

- 4 micro-seconds to add
- 16 KB RAM
- 1.2 micro-seconds memory cycle time. (3x faster than adding)
  - This is the reason that C coding revolves around pointer arithmetic.
  - **Note:** that the assumption that following pointers is faster now is wrong

Stable Languages Die, Successfull Languages Evolve!

1. Syntax Evolution: Idea: Can we change the language and add new features without changing the compiler?
   - Example of how to change C language

```
#define ELTS(a) (sizeof(a)/sizeof(a)[0])
#define CALLMAN(f, args) (f)(ELTS(args), args)
/* These are now equivalent:
 * Foo(f, args); <=> CALLMAN(f, args);
 */
#define CALLN(f, ...) CALLMAN(f, ((obj[]) { __VA_ARGS__ }))
/* This has now changed the language of C through the use of macros,
 * This is called metaprogramming
 */
```

   - Steve Bourne created the shell, and modified the C code using this:

```
#define IF if(
#define THEN ) {
#define ELSE } else {
#define FI }
```

Week 1 - Thursday 2019-04-04

Language Design Choices

1. **Orthogonality**
   - The choice of X does not affect Y
2. **Efficiency**
   - How fast is it?
   - How much RAM does it use?
   - Power/Energy
   - Network throughput/bandwidth/latency
3. **Simplicity**
   - Easy to learn
   - Easy to use with intuitive syntax
   - Good and simple documentation
4. **Convenience**
   - Simple to use
   - Potentially having lots of users is also a convenience
5. **Safety**
   - Static vs Dynamic
     - Static type checking is typically safer to use
   - Type checking
6. **Abstraction**
   - Can your programs scale cleanly?
   - Use of classes is an attempt at abstraction
7. **Exceptions**
   - Proper exception handling
   - Proper indication of errors
8. **Concurrency**
   - Must have proper support for parallel programmng
   - Cannot just be a third party library that adds this functionality
9. **Mutability/Extensibility**
   - Must have support for change in the future
   - Example of C with its macros that allow changes to the language without changing the compiler.

Types of Languages

Imperative	Functional	Logic
C++, C, Java	Lisp, ML, F#	Prolog

1. Imperative Languages are composed of statements
   - Statements are listed in order: `S_1; S_2; S_3;`
2. Functional Languages are composed of functions
   - Functions are linked like:
   $$y = F_1(F_2(x), F_3(x)) \tag{1}$$
   - Functional languages lack I/O
3. Logic Languages are composed of predicates
   - Predicates are linked together through the use of predicates: `(P_1 & P_2) | P_3`

Functional Programming

1. Clarity: use notations from mathematics
2. Parallelizability: performance is good, code structure incentivizes you to write parallel code.

- Function: mapping from a domain to a range
- Functional Form: function where either domain or the range is a function type
  - Example of higher order function in functional form:

$$\int_1^{100} f(x)dx \tag{2}$$

  - Example in C++:
    ```
    y = integral(f, 1, 100);
    ```

Ocaml Introduction

1. Static type checking is supported
   - Similar to Java in that all objects have a type
2. Type Inference
   - type checking is done at compile time, types for variables do not have to be declared
3. Automatic Storage Management
   - Garbage Collection is assumed inside of Ocaml system
4. Good Support for Higher-Order Functions
   - Really easy to implement these features (This led to creation of lambda expressions

Ocaml Basics

- Variables

```
# let my_value = 5;;
val my_value = 5
```

- List

```
# let numbers = [ 1; 2; 3; 4; 5 ]
```

  - All elements must have the same type
  - Under the hood, lists are immutable singly-linked lists
  - List Operations:
    - Adding new elements is right-associative
      `0::[1;2;3] -> [0;1;2;3]`
    - Cannod append to the end
  - Lists are immutable

- Functions
  - Functions in Ocaml can be relatively similar to other languages

```
# let average a b =
    (a + b) / 2;;;
val average : int -> int -> int = <fun>

(* float version *)
# let average a b =
    (a +. b) /2.0;;
val average : float -> float -> float = <fun>
```

  - `let` binds a function with parameters a and b to name average

- Recursive Functions
  - Recursive functions must be specified explicitly (`let rec`), otherwise the compiler will give an error about an undefined function

```
# let rec factorial a =
    if a = 1 then 1 else a * factorial (a-1);;
val factorial : int -> int = <fun>
```

- Defining Local Variables in Functions
  - add keyword `in` after the let statement to make the value available in the following statement

```
# let average a b =
    let sum = a +.b in
    sum /. 2.0;;
val average : float -> float -> float = <fun>
```

- Useful List Operations - map
  - Map transforms a list by applying a function on each element

```
# List.map(fun x -> x*x) [1; 2; 3; 4; 5];;
- : int list = [1; 4; 9; 16; 25]
```

- Useful list operations - rev
  - rev reverses the list:

```
# List.rev [1; 2; 3; 4]
- : int list = [4; 3; 2; 1]
```

- Useful list operations - filter
  - filter filters elements of the list that do not fit a certain critera

```
# List.filter(fun x -> x = 2) [1; 2; 3; 4]
- : int list = [2]
```

- Useful list operations - for_all
  - Returns true if all elements match the condition

```
# List.for_all(fun x -> x = 2) [1; 2; 3; 4; 5];;
- : bool = false
# List.for_all(fun x -> x = 2) [2; 2; 2];;
- : bool = true
```

- Useful list operations - exists
  - Exists checks if any elemnt in the list matches a condition

```
# List.exists(fun x -> x = 3) [1; 2; 3; 4; 5];;
- : bool = true
# List.exists(fun x -> x = 6) [1; 2; 3; 4; 5];;
- : bool = false
```

- Pattern Matching
  - More powerful version of the switch statement used in some other languages
  - Pattern matching allows you to list all the different cases in a clean way

```
# let is_zero x =
    0 -> true
  | _ -> false;;
```

  - Patterns can also include conditions using when statement

```
# let rec factorial a = match a with
    x when x < 2 -> 1
  | x -> x * factorial (x-1);;
```

- Data Types - Native Types
  - More native types:
    - `list([1; 2; 3; 4; 5], ["foo"; "bar"])`
      - Elements are tuples
- Data Types - Our Own Data Types
  - The most simple use case is to wrap an existing type

```
type age = int;;
type name = string;;
type person = age * name;;

let print_name(p: person) = match p with
| (p_age, p_name) -> print_string p_name;;

let my_person = (111, "Bilbo": person);;
print_name my_person;;
```

- Data types - Variants
  - Used when there are multiple subtypes of one main type

```
type ccle_user =
    Student of string
    | TA of string
    | Professor of string;;

type my_type =
| A of string
| B of int;;

let my_print x = match x with
| A a -> print_string a
| B b -> print_int b;;

my_print(A "some string");;
"some string"
my_print(B 5);;
5
```

Week 2 - Tuesday 2019-04-09

Ocaml Pattern Matching

1. Patterns types

Type	What they match
0, 19, []	constants only match themselves
P_1::P_2	identifier matches anything but y is bound

- Examples of pattern matching

```
match 3+9 in
| x -> x+1

match [3; -5; 12;] with
| [] -> 0
| [x] -> x+3
| x::y -> x
```

- Note that tuples in Ocaml must have a length known at compile time.

2. Recursion

- Say that you want to sum up all the elements of a list

$$\sum_{i \in \{\}} a_i = 0 \tag{1}$$

- Why does it start of 0? This is because 0 is the identity element of min on positive nums
- We expect this behavior to be consistent

$$\sum_{i \in A \cup B} a_i = \sum_{i \in A} a_i + \sum_{i \in B} a_i \tag{2}$$

- Suppose now that B = {}, then we expect that

$$\sum_{i \in B} a_i = 0 \tag{3}$$

- Example of Recursion in Ocaml (reverse)

```
# let rec reverse l = function
| [] -> []
| h::t -> (rev t) @ [h];;
reverse : 'a list -> a' list = <fun>

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

let reverse = revapp []
```

- Note that in the previous example, the second method is preferred and follows functional programming as a paradigm better

Week 2 - Thursday 2019-04-11

Syntax

- A "solved" problem with classic solutions and theory
- Form is independent of meaning
- English lets you construct a sentence that can have two different meanings, for example "time flies" which could be interpreted as a (N,V) or (V,N) combination.
- Ambiguity needs to be avoided in programming languages

What Makes Good Syntax?

1. **Inertia**: Syntax that people are used to so that they are familiar with it
   - This is why a lot of programming syntax comes from math - something people are familiar with
2. **Simplicity**: Syntax that is simple and allows users to read it and follow along intuitively
3. **Readability**: Just because syntax is simple doesn't mean is readable
   - For example, say you are trying to do `i = j*k;` in lisp, this would be `(set! i (* j k))`
4. **Writability**: The language should not take forever to write code in.
5. **Its Redundant**: This is actually useful in catching human errors
6. **Unambiguous**: Want the syntax to be clear in exactly what a piece of code is trying to accomplish

Overall Program Structure

- In Java/C: You have lots of .c files that are then linked together
- Smalltalk: A large set of classes.
- Bottom up construction: start with tokents then process them

$$\text{tokens} \rightarrow \text{char streams} \rightarrow \text{lexeme streams} \rightarrow \text{token steam} \tag{1}$$

Tokenization

1. Tokenization is expensive - costs several machine cycles per byte
2. Most of the program is non-token white space like `\r \n \v \t \b ...`
3. Keywords (special tokens) and reserved words (keywords that exclude identifiers with the same name)
   - An example in C:

     ```
     #defined FOO && FOO > 1
     // defined here is a keyword but not reserved, can do this:
     #define defined 27
     ```
   - Note that tokenizers are greedy, you cannot for example is assume that just because you have while right now, that it is the reserved keyword, it could just be the start of a variable name like whilex
   - Tokenizers once they finish their task, they return a token streams

Grammars

- Defined as by Backus Naur Form (BNF)
- The grammar structure is defined as

```
grammar = finiteset of nonterminal symbols
        + finite set of terminal symbols
        + finite set of rules which each
        have a nonterminal symbol that points to another symbol
        + a starting nonterminal symbol
```

- Then the language is defined as a set of sentences where sentences are finite sequences of terminal symbols that follow the structure defined by the grammar.

Ocaml Review

1. Lambda Functions
   - Anonymous functions that are useful forms of syntactic sugar

```
# (fun a -> a + 1) 5;;
-: int=6
```

2. Functions with Multiple Arguments
   - The function that takes in multiple arguments will be converted into a functions that each take one argument

```
# let sum a b = a + b
(* This gets converted internally in Ocaml to *)
# let sum = (fun a -> (fun b -> a + b));;
val sum : int -> int -> int = <fun>

# Can have subfunctions that fullfil some arguments
# let add_one = sum 1
val add_one : int -> int = <fun>
```

3. Function types
   - Ocaml will try to determine the input and output types automatically

```
# let sum a b = a +. b
val sum : float -> float -> float = <fun>
```

   - Sometimes the exact type is not known

```
# let rec my_map func my_list = match my_list with
| [] -> []
| head::tail = (func head)::(my_map tail)
val my_map : ('a -> 'b) -> 'a list -> 'b list = <fun>

# my_map (fun x -> x+1);;
-: int list -> int list = <fun>
```

4. Mutual Recursion
   - Useful when you want to alternate recursing with two functions

```
let rec is_even = function
| x when x = 0 -> true
| x -> is_odd (x-1)

and is_odd = function
| x when x = 0 -> false
| x -> is_even (x-1);;

# is_even 10;;
-: bool = true
# is_odd 8;;
-: bool = false
```

5. Infix Functions
   - Defining your own infix operators makes you code cleaner
   - Example for string concatination

```
let (+) a b = a ^ b;;

# "abc" + "def";;
-: string = "abcdef"
```

   - Warning: function definitions always override existing functions - there is no overloading in Ocaml

- You can also use infix operators as regular functions

```
# (+) 1 2;;
-: int 3
```

6. Pattern Matching
   - There are three ways to do pattern matching:

```
# let first x = match x with
| (left,_) -> left;;

# let first = function
| (left,_) -> left;;

# let first (left, _) -> left;;

(* All these functions will return: *)
# first (1,2);;
-: int = 1
```

7. Ocaml Types
   - Built in types include `int, float, char, string, bool, unit, tuple, list, function, ...`
   - Variant types: our types can be constructed using multiple types

```
type ('nonterminal, 'terminal) symbol =
| N of 'nonterminal
| T of 'terminal;;

type awksub_nonterminals = Expr | Lvalue | Incrop | Binop | Num;;

# [T"("; N Expr; T")"]
-: (awksub_nonterminals, string) symbol list = [T "("; N Expr; T ")"]
```

   - Option types
     - Sometimes functions cant come up with return values (ex: divide by 0 error)
     - Can come up with an alternative with the option type:

```
let divide a b = match a,b with
| x,y when y = 0.0 -> None
| x,y -> Some ( x /. y );;

# divide 1.0 0.0
-: float option = None
# divide 1.0 2.0;;
-: float option = Some 0.5
```

     - Downside is you have to handle this in your following functions

```
let print_division a b = match (divide a b) with
| Some x -> print_float x
| None -> print_string "Undefined";;
```

   - Recursive types - can use types to define recursive data structures like trees

```
# type tree =
| Leaf of int
| Node of tree *tree;;

# let my_tree = Node (Node (Leaf 1, Leaf 2), Node (Leaf 3, Leaf 4))

# let rec first_leaf = function
| Leaf x -> x
| Node (left, right) -> first_leaf left;;
```

```
#first_leaf my_tree;;
-: int = 1
```

8. More On the List Module
   - List.fold_left
     - useful when you want to summarize a list to one value, (this method is called reduce in some other languages)

```
# let sum my_list = List.fold_left (fun acc x _> acc + x) 0 my_list;;
# sum [1; 2; 3];;
-: int = 6
```

   - List.flatten
     - Function that removes all dimensions in the list

```
# List.flatten [["a"; "b"]; ["c"; "d"]; ["e"; "f"]];;
-: string list = ["a"; "b"; "c"; "d"; "e"; "f"]
```

   - List.Mapi
     - mapi is similar to map, except you can use the index of the element also

```
# List.mapi (fun i x -> (i, x)) ["a"; "b"; "c"; "d"; "e"];;
-: (int * string) list = [(0, "a"); (1, "b"); (2, "c"); (3, "d"); (4, "e")]
```

   - List.combine
     - Combines elements from two lists into a list of tuples (known as zip in python)

```
# let grades = [70; 80; 90];;
# let students = ["a", "b", "c"];;
# let new_list = List.combine grades students
val new_list: (int * string) list = [(70, "a"), (80, "b"), (90, "c")]
```

   - List.Split
     - Does the opposite of combine

```
# List.split [(70, "a"), (80, "b"), (90, "c")]
-: int list * string list = ([70; 80; 90], ["a", "b", "c"])
```

Week 3 - Monday 2019-04-16

Grammars Continued

1. ISO EBNF Standard:

Item	Specification
syntax	syntax rule, {syntax rule}
syntax rule	meta id, '=', defns list, ';'
defns list	defn, {'
defn	term, {',', terms}
term	factor ['-', exception]
exception	factor

2. Where the following syntax means::
    - [option]
    - {repetition}
    - (grouping)
    - (* comment *)
3. What can go wrong with grammars?
    - Nonterminal used but not defined
    - Nonterminals defined but not used
    - useless rules
    - extra constraints not easily formulated in BNF/EBNF

Week 3 - Thursday 2019-04-18

Java

Types

1. Type: a set of values and a set of operations
2. Abstract types:
   - Opaque
   - Must use methods (functions)
3. Exposed types:
   - Transparent
   - Users can look at values + reprs
   - Methods are nice too?

Float (C, Java)

- Floats in C and Java are represented as 32 bits
  - These 32 bits are split up as follows:
    * 1 signed bit (s)
    * 8 exponent bits (e)
    * 23 mantissa/fraction bits (f)

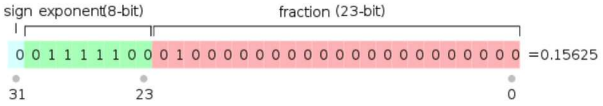


Figure 1: Representation of floats

- Tiny Numbers
  - These numbers have a value $e = 0$
  - They have a value range of $2^{e-126}f$
  - Why have these numbers? With tiny numbers, underflow like this never happens

```
if (a != b && a -b == 0) {
    ouch();
}
```

  - There are two values for zero: 0 and -0

```
if (a == b && memcmp(&a, &b, sizeof a )) {
    ouch();
}
```

- Other values
  - When $e = 255$ and $f = 0$ you can have $\pm$inf
  - When $e = 255$, and $f \neq 0$ you can get `NaN`

Primitive vs Constructed Types

- Classic primitive types includes stuff like `bool, float, int, char, ...`
- Constructed Types: `String, Array, List, Map, int * float, etc...`
  - These are constructed by humans, and have a constructor

Uses of Types

1. Annotations (put these into your program)
   - Should be easier to understand
   - Compiler can generate more efficient code
2. Inference (look at at a var and figure out type)
   - In C/C++ for example there is the keyword `auto` which will let the compiler determine what the type of that variable should be

Type Checking

- Static type checking:
  - Happens at compile time
  - Has high levels of reliability and performance
  - Used by static-typed languages like C++, C, Java, etc
- Dynamic type checking
  - Checking occurs at runtime
  - Used by scripting languages like Javascript, python, and sh...
  - More flexible and simpler
- Strongly Typed
  - Cannot escape type checking
  - All operations' types are checked
  - No way to escape this
  - C++ is not strongly typed (can read int as char for example)

Type Equivalence

- Let === Mean same type
- Structural Equivalence (same representation)
  - Pass around during run time
- Name Equivalence (same name)

```
typedef int dollars;
typedef int pennies;
// Created two aliases for int
// Here, dollars === pennies (structural equivalence)

struct s { int val; struct s* next; };
struct t { int val; struct t* next; };
// Here, struct s !=== struct t (Here C uses name equivalence)
```

Subtypes

- Pascal:

```
type lc = 'a' ... 'c'
var x:lc;
var y:char;
y := x; // Safe
x := y; // Unsafe (runtime check will see if it works)
```

- Common Lisp:

```
(declare (type (and integer (satisfies evenp)) x))
```

- C++:
  - You can assign a child as the value to a variable of type parent, but not the other way around

```
class animal {
    ...
}
class mammal : animal {
    ...
}
mammal dog;
animal a = dog; // Works
```

  - Can also do more with non-const pointers vs const pointers

```
char const *p;
// You can't store through p
// But you can change p

char *o;
o = p; // Invalid
p = o; // Valid
```

  - Here, `char*` is a **subtype** of `char const*` since you can do more stuff with `char*`
  - `char const **` is a **subtype** of `char* const*`

Morphisms

- A morphism is a function that takes many forms
  - What data types does it accept and produce?
  - Ex: cosine in Fortran:
    - Can have `cos(float)`, `cos(double)`, are both ok
    - Two implementations: compiler will look at the type of argument and then choose the matching implementation of the function
    - This is implemented using **name mangling**
- Name Mangling: changes the name of the function at a lower level (machine code?)
- Overloading: essentially the same idea, lets you overload a function to take in many different types. The compiler will look at the code and determine which matching signature you want
- Coercion: implicit type conversion

```
float f;
double d;
d = f; // runtime action required, no information loss here
f = d; // will lose information since double takes more memory
```

  - Problems with coercion: can cause problems with signed vs unsigned arithmetic

```
unsigned u;
if (u <= -1) {
    ouch(); // This runs because int is coerced to unsigned
}
```

Parametric Polymorphism

- Pioneered in ml (the predecessor to ocaml)

- The type of function can contain parameters

  ```
  length: 'a list -> int
  ```

- Form 1 (C++ and Ada): Templates
  - Write a template: a function with type variable
    - After instantiation at compile time, compiler replaces variables with real types.
  - An example in C++:

    ```
    template <class T>
    T add(T a, T b) { return a + b; }
    ```

- Form 2: (ML, Ocaml, Java, . . .)
  - Generics:
    - Looks similar to template, but is different
    - Compiler will see if this code will work no matter what the type is
    - Less code bloat (only one copy of the code)
    - All type checking is performed at once

Java Introduction

- Compiled into byte code that is then run ob the Java Virtual Machine
- The use of the Java Virtual Machine allows compiled bytecode to be run on any machine
- Java bytecode is a commpromise vs compiled and interpreted code that allows platform independence and some performance (since interpreted code is difficult to optimize)
- Everything in Java is inside of a class
- Java Memory Model defines how threads interact through memory.
  - Each thread has its own stack, but share the heap
- When there is only one thread, the Java compiler is allowed to change the order of the code as long as the result of execution is the same
- With multiple threads, it becomes more challenging to do these types of complications.
  - Here there is a need for human input to set constraints for what can be optimized

The `volatile` Keyword

- Defining the variable `volatile` guarantees that other threads will see the changes immediatly

  ```
  public class SharedObject {
      public volatile int counter = 0;
  }
  ```

Week 4 - Tuesday 2019-04-23

Paramentric Polymorphism

- Generics (without):

```
for (iterator i = c.iterator(); i.hasNext();) {
    if (((String) i.next()).length()) == 1) {
        i.remove();
    }
}
```

- Generics (with):

```
for (Iterator<String> i = c.iterator(); i.hasNext();) {
    if (i.next().length() == 1) {
        i.remove();
    }
}
```

- Problem with generics:

```
List<String> ls = _; // Ok
List<Object> lo = ls; // reference assignment
lo.add(new Thread()); // Ok
String s = ls.get(); // Ok
```

- The problem here is that line 2 is an error in Java, since `List<String>` not is a subtype of `List<Object>`, even though `String` is a subtype of `Object`
  - Subtype is a **Superset of operations**
- Note that the collection of objects must not be a collection of strings

```
void printAll(Collection<Object> c) {
    for (Object o : c) {
        System.out.println(o);
    }
}
```

- Java supports wildcards that let functions to take in any type for their operations, in printAll for example, can use the `?` wildcard which represents any type

```
void printAll(Collection<?> c) {
    for (Object o : c) {
        System.out.println(o);
    }
```

- If you want to copy an array of string to a collection of Objects, Java wont allow it unless T is the same type

```
static <T> void convert (T[] a, collection<T> c) {
    for (T o : a) {
        c.add(o);
    }
}
```

- Can also add constraints for the generic types, such as U here is a supertype of T

```
static <T, U super T> void convert (T[] a, collection<U> c) {
    for (T o : a) {
        c.add(o);
```

```
    }
}
```

Erasure

- During a cast, there is no real type checking when you do a cast

```
List<String>   -> List<Object>
// Compiletime -> Runtime
```

- Should avoid writing code that uses dynamic casting of types during runtime, since the compiler will not do a check for these casts, and the error messages will be very difficult to decipher.
- The Java compiler does away with most types during compile time

Duck Typing

- Ask for methods of objects instead of types
- The compiler or interpreter does not care about type, only about functionality
  - This leads oftentimes to simpler languages
- Loss of readability
- Python is an example of a language that uses duck typing

Week 4 - Thursday 2019-04-25

Lanugage Environments

1. In Java, order of evaluation is always left to right
   - less undefined behavior
   - code is more portable
   - lower performance
2. There are several tiers to compiling good code, initially, the compiler will write first do type checking and optimize for any machine, after that, there is another tier of optimizations that occurs that are architecture specific

Week 5 - Tuesday 2019-04-30

Prolog

1. Prolog Lets you define your own operators

```
:-op(700, xfx, [==,\=,=<])
:-op(500, yfx, [+,-])
:-op(400, yfx, [*,/])
:-op(200, yfx, [**])
:-op(200, fy, [+,-])
```

2. Suppose we want to create a Prolog parser, parsing in general has two main problems that need to be solved:
   1. Disjunction - try different rules with same NonTerminal
   2. Concatenation - Handle rules with multiple pieces
3. Here is an example parser for regular expressions using functional programming:

```
let rec mm = function
| Or (a, b) -> let ma = mm a and
                let mb = mm b
                in (fun acc frag ->
                 match ma acc frag with
                 | Some x -> Some x
                 | None -> mb acc frag)
| Concat (a,b) -> let ma = mm a and
                    let mb = mm b
                    in (fun acc -> ma (mb acc))
```

   - Note that here: Or and Concat represent data structures that represent the text fragment

Java Abstract Classes

1. Java allows the definition of abstract classes

```
abstract class X {
    int i;
    public void incr() { i++; }
    public void decr();
}
```

2. Note in the previous operation, cannot run `new X` since X does not fully implement the interface

3. Must create a child class that inherits from X and finishes the interface

```
public class SX extends X {
    public void decr() { i--; }
}
```

4. Abstract classes can be replaced with an interface and regular classes
5. Final class: cannot overwrite methods or extend, this prevents child class from making mistakes
    - Final allows the compiler to inline calls to final methods since you cannot override the method and thus can make optimizations
    - Final methods don't have to deal with generics and worrying about being overwritten by subclasses - essentially trades flexibility for speed

Java Library Definition of the Object Class

1. The Object abstract class requires the following from all the inherited objects:

```
public class Object {
    public Object();

    // Note that '==' will compare addresses while equals can be defined to do
    // something else
    public boolean equals(Object obj); //default ==
    public int hashCode(); // default is int(this)
    public String toString(); // Note that yes, String is a subclass of Object, this
                              // is allowed in Java
    public final Class getClass(); // Note this actual declaration is more
                                   // complicated than this. (Note that Class is
                                   // capitalized, this is a description of the class)
    protected void finalize() throws Throwable; // For garbage collection
    protected Object clone() throws CloneNotSupportedException; // For copying
}
```

2. Note that for two objects, o_1 and o_2, we have the following implication:

```
o1.equals(o2); // This implies that:
o1.hashCode() == o2.hashCode();
```

3. This means that if you plan on overwriting `equals()` you are probably going to have to overwrite 'hashCode()'

Java Multithreading

1. Thread's life cycle

```
Thread t = new Thread(...); // First it is created
t.start();  // Tells the OS to allocate resources (pthread_create() invocation)
t.run() // you can either pass a runnable object or implement the Runnable interface
        // or subclass Thread

public interface Runnable {
    void run();
}
sleep()    // ->
wait()     // ->
// during I/O ->   BLOCKED (waiting for I/O)
```

```
            //      WAITING
            //      TIMEDWAITING
            //      Threads in these states are frozen and do not take up CPU power
exit() // exit the run method resulting in status TERMINATED, here it still exists
       // but can't do anything
// Garbage collector cleans it up
```

2. Synchronization

   - Synchronization methods:
     - Before calling, lock an object level spin lock
     - After calling, unlock the object level spin lock
   - wait methof of an

Week 6 - Tuesday 2019-05-07

Java Synchronization

- Race conditions
  - Lead to nondeterminism
  - Lead to Crashes
- `synchronized` keyword around critical sections/monitors
  - Prohibit races where two threads access the same object
  - Can be seen as too simple and leads to synchronization problems
  - Synchronized is used to be implemented by spinlocks, now it there are more sophisticated implementations that are faster
- High-level Synchronization Library
  1. Semaphores - allow for up to a set number of threads to access simultaneously
  2. Exchanger

     ```
     Exchanger x = ... ;
     // Thread 1
     v = x.exchange(w);
     // Thread 2
     m = x.exchange(s);
     ```
  3. CountDownLatch
  4. Cyclic Barrier
- Volatile Keyword
  - Tells the compiler to not optimize accesses to the variable
  - When you access a volatile variable, you must do a load, and if you access two volatile variables, you must load them in the same order they are encountered
  - **Note**: volatile does not mean atomic

Language Implementation Optimization

- "As If" Rule: you can implement it any way you like, so long as its done as if the model were executed
  - Commonly invoked under the assumption of single-threaded code
- HW: Read Chapter 1-15 + Prolog

Scope

- Essentially allows you to break up your program into multiple self-contained modules
- Scope of a name: set of program locations where the name is visible
- **Block Scope**

  ```
  {
      // The scope of n is this block, however that inside n is a different variable,
      // so inside the nested scope, the outer n is invisible
      int n = ...;
      { int n = ...; }
  }
  ```
  - Note that being invisible is different from not existing
  - The scope is usually static. The compiler can see when av variable is and isn't viable
  - Lifetime is dynamic, you need to run the code to see the lifetime of a variable

Week 6 - Prolog - Thursday 2019-05-09

Prolog

- Prolog has no traditional functions - it has predicates, which can be thought ofas just functions that return a boolean. This is the hallmark of a logic programming language
- In Prolog, you declare what a correct solution looks like and Prolog searches for it. This makes Prolog adeclarative programming language
- Prolog attempts to separate an Algorithm into logic and control, where logic dictates correctness and control dictates efficiency. The programmer writes logic, and the Prolog compiler figures out how to achieve that. Contrast this with Java, where the two are combined.

Example: Sorting

- When atempting to create `sort`, first define what it means for something to be sorted:

```
sort(L, S) :- sorted(S), perm(L, S).
```

- So Here, `S` is a sorted version of `L` if `S` is sorted and `L` and `S` are permutations
- In a way, Prolog "if" is reversed compared to other programming languages; the predicate for "if" comes after the implication

```
sorted([]). /* empty list is sorted */
sorted([_]). /* singleton list is sorted */
sorted([X, Y | L]) :- X =< Y, sorted([Y, L]). /* x,y | L means x :: y :: L in OCaml */
```

- Now define `perm`

```
perm([], []).
perm([X], [X]).
perm([X | L], R) :-
append(P1, [X | P2], R),
append(P1, P2, P),
perm(L, P)
```

- Now define `append`

```
append([], L, L).
append([X | L], M, [X | LM]) :- append(L, M, LM).
```

- **Executing code**: To run the code, Prolog has a prompt:

```
| ?- sort([3, -5, 7], R).
R = [-5, 3, 7].
```

- **Reordering predicates**: However, Prolog has a determistic behavior, going from left to right. Consider `sort(L, S)`; Prolog first considers `sorted(S)`, trying all possible `S` (starting with the empty list) so that it fits `perm(L, S)`
- A way to fix that is to just reverse the order of the predicates, note that only changed the control - not logic, as the "and" operation is commutative

```
sort(L, S) :- perm(L, S), sorted(S).
```

- Note, this algorithm first looks at all possible permutations of `L`, making this algorithm $\mathcal{O}(n!)$ time
- Prolog is used to see if an algorithm is logically correct; if proven to be so then people would rewrite the code in someother language

Second example: list member

- Consider the following clauses

```
member(X, [X | _])
member(X, [_ | L]) :- member(X, L).

| ?- member(Q, [5, 3, 1]).
Q = 5 ?
```

- Here, we have the option of letting Prolog give us a new valid answer (;), or stop (ret), if we input ;, thenm we get

```
| ?- member(Q, [5, 3, 1]).
Q = 5 ? ;
Q = 3 ? ;
Q = 1
yes
```

- This feature could be described as automatic backtracking, what happens if we try a list containing 9?

```
| ?- member(9, L).
L = [9|_] ? ;/* derived directly from the first rule */
L = [_,9|_] ? ;/* derived first from the second rule *
L = [_,_,9|_] ? ;
L = [_,_,_,9|_] ?
...
```

- This program enters an infinite loop
- What if we try to find a three-element list that contains 9?3

```
| ?- member(9, [A, B, C]).
A = 9 ? ;
B = 9 ? ;
C = 9 ? ;
```

- If we reject all of the possible predicates, the interpreter returns no. Now consider how `append` works

```
| ?- append([3, 1], [9], R).
    X1 = 3, L1 = [1], M1 = [9], R1 = [3, LM1]
        append([1], [9], LM1)
            X2 = 1, L2 = [], M2 = [9], R2 = [1| LM2] = LM1
                append([], [9], LM2)
                    L3 = [9] = LM2
R = [3, 1, 9]
```

- We could also run append backwards:

```
| ?- append(L, M, [3, 1, 9])
L = [], M = [3, 1, 9] /* due to first rule */? ;
    [3 | L1] = L, M1 = M, [X1 | LM1] = [3, 1, 9] => X1 = 3, LM1 = [1, 9]
        append(L1, M1, [1, 9])
            L1 = [], M1 = [1, 9] /* due to first rule */
L = [3], M = [1, 9] ? ...
```

Prolog Arithmetic

- Consider the following expression:

```
| ?- X = 2+3.
X = 2+3
```

- The `+` constructs a tree data structure, with a root node as `+`, to actually do arithmetic in Prolog, use the `is` primitive

```
| ?- X is 2+3.
X = 5

/* This is also equivalent */
| ?- is(X, 2+3).
```

Week 6 - Thursday 2019-05-16

Scheme Introduction

- Scheme is a functional programming language that is part of the LISP language family
- Pioneered many new concepts such as garbage collection, recursion, dynamic typing, and program code as a data structure
- Scheme is a dialect of lisp that was created by the MIT AI lab, and historically very popular in academia
- Hello world in scheme can be done with the following: create a helloworld.ss file:

```
#lang racket

(display "Hello, world!\n")
```

- Then to run this file on the command line:

```
$ racket helloworld.ss
"Hello, world!"
```

Basic Syntax

- **Comments**
  - Semi-colon `;` starts a line comment
  - Block comments are done with `#|` as the start, and `|#` as the end
- **Numbers**: similar to most languages:

```
1, 1/2, 3.14, 6.02e+23
```

- **Strings**: done with double quotes

```
"Hello, World!"
```

- **Booleans**: represented with `#t` and `#f`
- **Function Calls**
  - In scheme, the function name always comes first in function calls

```
> (display "Hello")
Hello
> (+ 1 2)
3
> (+ 1 2 (- 4 3))
4
> (/ (+ 1/3 1/6) 2)
1/4
```

- **Definitions**: defining variables and functions have similar syntax:

```
(define pi 3.14)
> pi
3.14

(define (print-name name)
        (display (string-append "Hello, " name)))
> (print-name "Steve")
Hello, Steve
```

- **Lambda Functions**: anonymous functions that can be defined with `(lambda (args*) expr)`

```
> (lambda (a b c) (+ a b c))
#<procedure>

> ((lambda (a b c) (+ a b c)) 1 2 3)
6
```

- **Local Bindings**: Let keyword defines a new variable inside an expression

```
(define (say-hello)
    (let ([name "John"]
          [greeting "Hello, "])
        (display (string-append greeting name))))
> (say-hello)
Hello, John
```

- **Identifiers**
  - Scheme is very liberal with identifiers, and allows all the following:

```
+
Hfuhruhurr
integer?
pass/fail
john-jacob-jingleheimer-schmidt
a-b-c+1-2-3
; The following are forbidden:  ) [ ] { } " , ' ` ; # | \
```

- **Checking Types**
  - Scheme provides functions to check types

```
> (number? 5)
#t
> (string? "My string")
#t
> (list? (list 1 2 3 4))
#t
> (pair? (cons 1 2))
#t
```

- **Lists**
  - Scheme internally uses linked lists similar to Ocaml and Prolog
  - To access head you can use `(car my-list)` or `(first my-list)`
  - To access tail, use `(cdr my-list)` or `(rest my-list)`
  - Empty list represented by `'()`, can check with

    `(empty? '()) -> #t`

  - Can define lists as follows:

```
> (define my-list (list 1 2 3))
> (car my-list)
1
> (first my-list)
1
```

```
> (rest my-list)
'(2 3)
```

– Other list operations:

```
> (map (lambda (x) (+ x 1)) '(1 2 3))
'(2 3 4)
> (filter (lambda (x) (> x 2)) '(1 2 3 4))
'(3 4)
> (foldl (lambda (a b) (+ a b)) 0 '(1 2 3 4))
10
> (sort '(5 4 3 2 1) <)
'(1 2 3 4 5)
> (length `(1 2 3 4 5))
5
> (reverse `(1 2 3 4 5))
'(5 4 3 2 1 )
```

CS 131 Readings

Chapter 1 - Programming Languages

- Programming Languages are grouped into four families
  1. Imperative
  2. Functional
  3. Logic
  4. Object-Oriented
- Hallmarks of imperative languages
  - Assignments
  - Iteration
  - Order of execution is critical
- Hallmarks of functional languages
  - Single-valued variables
  - Heavy use of recursion
- Hallmarks of logic languages
  - Program expressed as rules in formal logic
- Hallmarks of object-oriented languages
  - Usually imperative
  - Constructs to help programmers use "objects" to do things
- Language Standards
  - Documents that define the language standards
  - Can be a slow and rancorous process

Chapter 2 - Defining Program Syntax

- Syntax and Semantics
  - Programming language syntax: how programs look, their form and structure, defined by a formal grammar
  - Programming language semantics: what programs do, their behavior and meaning
- How the Grammar Works
  - The grammar is a set of rules that say how to build a parse tree
  - You put <S> at the root of the tree
  - The grammar's rules say how children can be added at any point in the tree
- A Parse Tree of the Gramar:

```
<S> ::= <NP> <V> <NP>
<NP> ::= <A> <N>
<V> ::= loves | hates|eats
<A> ::= a | the
<N> ::= dog | cat| rat
```

- Programming Language Grammar
  - Here is an example expesssion:

```
<exp> ::= <exp> + <exp> | <exp> * <exp> | ( <exp> )
                        | a | b | c
```

  - An expression can be one of the variables a, b, or c or it can be a two expressions along with an operator
  - The parse tree for this looks like:

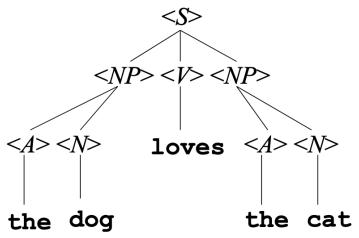


Figure 1: Example English Parse Tree

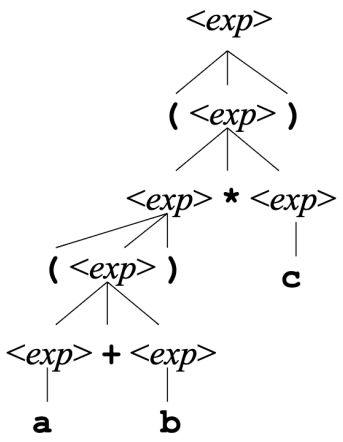


Figure 2: Parse tree for ((a+b)*c)

- BNF Grammar Definition (4 parts)
  - Set of tokens
    - Tokens are smallest unit of syntax and are atomic - not treated as a composition of smaller pieces
  - Set of non-terminal symbols
    - They are strings enclosed in angle brackets, as in
    - They are not strings that occur literally in program text
    - Grammar specifies how these can be expanded out
  - Start symbol
    - the non-terminal that forms the root of any parse tree for the grammar
  - Set of productions
    - the tree-building rules
    - Each one has a non-terminal, the separator ::=, and a sequence of things which can be a tokens or a non-terminals
    - A production gives one possibility for building a parse tree, with each non-terminal having things to put as its children
    - Also possible in BNF grammar to seperate the possible right hand sides with |
- Empty
  - The special nonterminal is for places where you want the grammar to generate nothing
  - For example, this grammar defines a typical if-then construct with an optional else part

```
<if-stmt> ::= if <expr> then <stmt> <else-part>
<else-part> ::= else <stmt> | <empty>
```

- Parse Trees
  - To build a parse tree, put the start symbol at the root
  - Add children to every non-terminal, following any one of the productions for that non-terminal in the grammar, finish when leaves are tokens
- Language Definition
  - The language defined by a grammar is the set of all strings that can be derived by some parse tree for the grammar
- Lexical Structure And Phrase Structure
  - Grammars so far have defined phrase structure: how a program is built from a sequence of tokens
  - We also need to define lexical structure: how a text file is divided into tokens
  - Usually this is dones with two seperate grammars - one on how to construct a sequence of tokens and one says how to construct the parse tree
- Seperate Compiler Passes
  - The scanner the input file and divides it into tokens according to the first grammar, discarding whitespace/comments
  - The parser then constructs the parse tree from the second grammar
- EBNF Variations
  - EBNF Adds Additional syntax to simplify some grammar chores
    - {x} to mean zero or more repetitions of x
    - `[x]` to mean x is optional (i.e. x | `<empty>`)
    - () for grouping
    - anywhere to mean a choice among alternatives
    - Quotes around tokens, if necessary, to distinguish from all these meta-symbols
- EBNF Examples

```
<if-stmt> ::= if <expr> then <stmt> [else <stmt>]
<stmt-list> ::= {<stmt> ;}
<thing-list> ::= { (<stmt> | <declaration>) ;}
<mystery1> ::= a[1]
<mystery2> ::= 'a[1]'
```

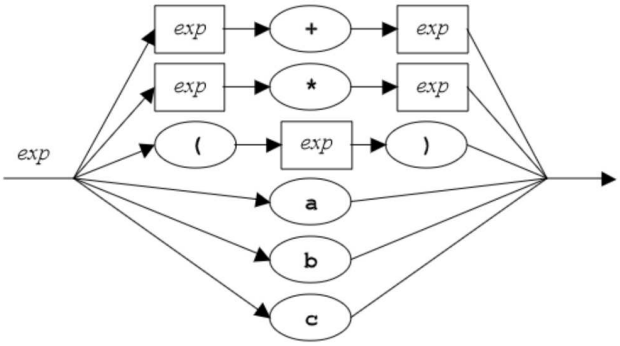
- Syntax Diagrams
  - Start with an EBNF grammar, a simple production is just a chain of boxes
  - For the grammar: `<if-stmt> ::= if <expr> then <stmt> else <stmt>`

  - Can also have Bypasses which are square-bracket pieces from the EBNF get paths that bypass them
  - So if you let the else be optional: `<if-stmt> ::= if <expr> then <stmt> [else <stmt>]` then

  - Can also use branching for multiple productions, so for a grammar like

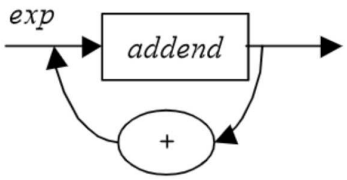
```
<exp> ::= <exp> + <exp> | <exp> * <exp> | ( <exp> )
            | a | b | c
```

– You can generate a parse tree:



– Can also use Loops for EBNF curly brackets, like in the grammar

```
<exp> ::= <addend> {+ <addend>}
```



– Syntax Diagram Pros and Cons

* Easier to read casually, but harder precisely
* Harder to make machine readable

Chapter 3 - Where Syntax Meets Semantics

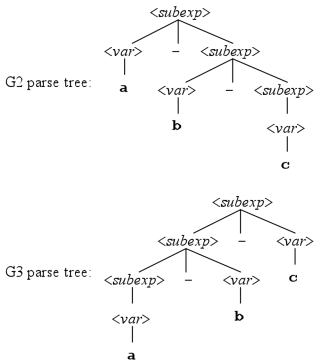
- Here are Three equivalent grammars

```
<subexp> ::= a | b | c | <subexp> - <subexp>

<subexp> ::= <var> - <subexp> | <var>
<var> ::= a | b | c

<subexp> ::= <subexp> - <var> | <var>
<var> ::= a | b | c
```

- These grammars all define the same language but have different syntax trees



- We want the structure of the parse tree to correspond to the semantics of the string it generates

- Precedence of Operators
  - Each operator has a precedence level, and those with higher precedence are performed first
- Precedence In Grammar
  - To fix the precedence problem, modify the grammar so it is forced to put * below + in the parse tree, so transform grammar as follows:

```
<exp> ::= <exp> + <exp>
            | <exp> * <exp>
            | (<exp>)
            | a | b | c

To:
<exp> ::= <exp> + <exp> | <mulexp>
<mulexp> ::= <mulexp> * <mulexp>
                | (<exp>)
                | a | b | c
```

  - Now our grammar will not generate parse trees with bad precedence

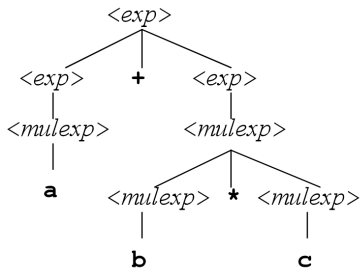
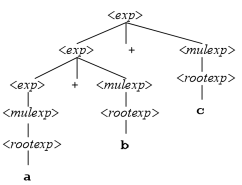


Figure 3: desc

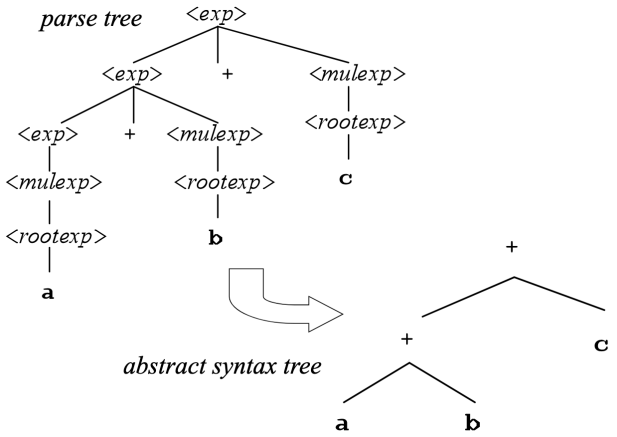
- Operator Associativity
  - Left-associative operators group left to right: `a+b+c+d = ((a+b)+c)+d`
  - Right-associative operators group right to left: `a+b+c+d = a+(b+(c+d)`
  - Most operators in languages are left-associative
- Associativity in Grammar
  - To fix the associativity problem, we modify the grammar to make trees of `+`s grow down to the left (and likewise for `*`s)
  - Change the previous grammar to:

```
<exp> ::= <exp> + <exp> | <mulexp>
<mulexp> ::= <mulexp> * <rootexp> | <rootexp>
<rootexp> ::= (<exp>)
                | a | b | c
```

  - So the syntax tree is now:



- EBNF and Parse Trees
  - We have that {x} means "zero or more repetitions of x" in EBNF
  - So `<exp> ::= <mulexp> {+ <mulexp>}` should mean a `<mulexp>` followed by zero or more repetitions of `+ <mulexp>`
  - But what is that associativity?
  - One approach to this is to use {} whenever possible
  - Another approach is to use the form for left associative operators: `<exp> ::= <mulexp> {+ <mulexp>}`, or use explicitly recursive rules for these unconventional situations: `<expa> ::= <expb> [ = <expa> ]`
- Full-Size Grammars
  - Any realistic language has many non-terminals
  - Extra non-terminals guide construction of unique parse tree
  - Once parse tree is found, such non-terminals are no longer of interest
- Abstract Syntax Tree
  - Language systems usually store an abreviated version of the parse tree, the AST
  - Usually, there is a node for every operation, with a subtree for every operand

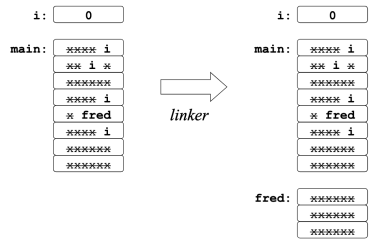


- Parsing Revisited
  - When a language system parses a program, it goes through all the steps necessary to find the parse tree
  - But it usually does not construct an explicit representation of the parse tree in memory
  - Most systems construct an AST instead
- Grammars define not just a set of legal programs, but a parse tree for each program

Chapter 4 - Language Systems

- Compiling
  - Compiler translates to assembly language (machine specific)
  - Each line represents either a piece of data, or a single machine-level instruction
- Assembling
  - Assembly language is still not directly executable, still in text format
  - Assembler converts each assembly-language instruction into the machine's binary format that is machine language

- Linking
  - Object file still not directly executable - missing parts and has some names, mostly machine language at this point
  - Linker collects and combines all the different parts of the program



- Loading
  - Executable file still has some names even through it is mostly machine language, the final step is now when the program is run, the loader loads it into memory and replaces names with addresses
  - Loader finds an address for every piece and replaces names with addresses
- Running
  - After loading, the program is entirely in machine language, where all names have been replaced with memory addresses

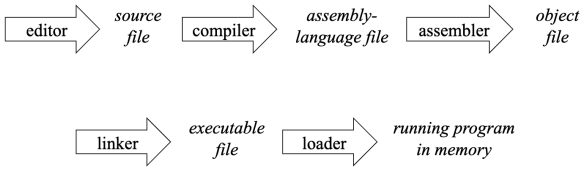


Figure 4: The Classical Sequence

- Optimization
  - Code generated by a compiler is usually optimized to make it faster, smaller, or both
  - Other optimizations may be done by the assembler, linker, and/or loader
  - Compilers can remove variables, remove code, add code, move code around, etc.
- Variation: Interpreters
  - To interpret a program is to carry out the steps it specifies, without first translating into a lower-level language
  - Interpreted languages like python are usually much slower, since compiling takes time up front, but fast at runtime
- Virtual Machines
  - A language system can produce code in a machine language for which there is no hardware: an intermediate code
  - Language system may do the whole classical sequence, but then interpret the resulting intermediate-code program
  - This allows for cross platform execution and heightened security
- Delayed linking
  - Delay linking step
  - Code for library functions is not included in the executable file of the calling program

  - In windows functions for delayed linking are stored in .dll files
    * Load-time dynamic linking - Loader finds .dll files and links the program to functions it needs, just before running
    * Run-time linking - Running program makes explicit system calls to find .dll files and load specific functions
  - In Unix, Libraries of functions for delayed linking are stored in .so files: shared object
    * Shared libraries - Loader links the program to functions it needs before running
    * Dynamically loaded libraries - Running program makes explicit system calls to find library files and load specific functions
  - In Java, the JVM automatically loads and links classes when a program uses them
  - The Advantages of Delayed Linking
    * Multiple programs can share one copy of library functions
    * Library functions can be updated independently of programs
    * Can avoid loading unused code

- Profiling
  - The classical sequence runs twice
  - First run of the program collects statistics: parts most frequently executed
  - Second compilation uses this information to help generate better code

- Dynamic Compilation
  - Some compiling takes place after the program starts running
  - Many variation like compile each function only when called, or roughly compile then spend more time on frequently used pieces of code

- Binding
  - Binding means associating two things—especially, associating some property with an identifier from the program
  - Ex: What set of values is associated with int?
  - Different bindings take place at different times
  - Language Definition Time
    * Some properties are bound when the language is defined like meanings of keywords like `void`
  - Language Implementation Time
    * Some properties are bound when the language system is written
    * Range of values of type int in C or implementation limitations like max identifier, max number of array dimensions
  - Compile Time
    * Some properties are bound when the program is compiled or prepared for interpretation
    * Types of variables, in languages like C and ML that use static typing
    * Declaration that goes with a given use of a variable, in languages that use static scoping
  - Link Time
    * Some properties are bound when separately-compiled program parts are combined into one executable file by the linker
    * Object code for external function names
  - Load Time
    * Some properties are bound when the program is loaded into the computer's memory, but before it runs
    * Memory locations for code for functions and static variables
  - Run Time
    * Some properties are bound only when the code in question is executed:
    * Values of variables or types of variables in languages like lisp

- Runtime Support
  - Additional code the linker includes even if the program does not refer to it explicitly

- Includes things like Exception handling, memory management, allocating memory, and an OS interface

Chapter 5 - A First Look at ML

- Left Associative precesdence is `{+,-} < {*,/,div,mod} < {~}`.
- String concatination: ^ operator
- Ordering comparisons: `<, >, <=, >=`, apply to string, char, int and real
- Boolean operators: andalso, orelse, not.

```
- true orelse 1 div 0 = 0;
val it = true : bool
```

- andalso and orelse are short-circuiting operators: if the first operand of orelse is true, the second is not evaluated and likewise for andalso
- Function Associativity is left-associative, so So `f a b` means `(f a) b`
- Tuples
  - ML gives the type of a tuple using * as a type constructor
  - For example, int * bool is the type of pairs (x,y) where x is an int and y is a bool
- Lists
  - All elements of the list must be of the same type
  - The @ operator concatenates lists
  - The :: operator is right-associative
  - The explode function converts a string to a list of characters, and the implode function does the reverse
- Functions
  - Function Definition syntax:

    ```
    <fun-def> ::=  fun <function-name><parameter> = <expression> ;
    ```

  - ML gives the type of functions using -> as a type constructor
  - All ML functions take exactly one parameter To pass more than one thing, you can pass a tuple
- Combining Constructors
  - When combining constructors, list has higher precedence than *, and -> has lower precedence
  - Here is an example:

```
(* These two are the same *)
- int * bool list
- int * (bool list)

(* These two are the same *)
- int * bool list -> real
- (int * (bool list)) -> real
```

Chapter 6 - Types

- A type is a set of values plus a low-level representation plus a collection of operations that can be applied to those values

- Primitive vs Contructed Types
  - Any type that a program can use but cannot define for itself is a primitive type in the language
  - Some primitive types in ML: `int, real, char`
- Primitive Types
  - Some languages define the primitive types more strictly than others
  - Java for example defines them exactly, whereas languages like C and ML leave a little wiggle room
- Constructed Types
  - Additional types defined using the language, like uples, arrays, strings, lists, unions, subtypes, and function types
  - Enum types are supported by many languages, where you can explicitly state all the different potential values
  - A common representation is to treat the values of an enumeration as small integers
  - Can also make sets by tupling and combining two different types of values
  - C lets you do this with structs while ML and other languages have explicit tuples
  - Many languages also support union types like in C:

    ```
    union element {
        int i;
        float f;
    };
    ```

  - Or like in ML:

    ```
    datatype element =
        I of int |
        F of real;
    ```

- Strictly Typed Unions
  - In ML all you can do with a union is extract the contents and state what to do with each type of value in the union
- Making Subtypes
  - Some Languages support subtypes with more or less generality
  - Example of Lisp Types with predicates

    ```
    (declare (type integer x))
    (declare (type (or null cons) x))
    (declare (type (and number (not integer)) x))
    (declare (type (and integer (satisfies evenp)) x))
    ```

- Representing Subtype Values
  - Usually, we just use the same representation for the subtype as for the supertype
- Operations on Subtype Values
  - Usually, supports all the same operations that are supported on the supertype
  - And perhaps additional operations that would not make sense on the supertype: like `function toDigit(X: Digit): Char`;
  - A subtype is a subset of values, but it can support a superset of operations.
- Classes

  - In class-based object-oriented languages, a class can be a type: data and operations on that data, bundled together

- Functions
  - Most languages have some notion of function types, what is the domain and codomain?

- Operations on Functions
  - We have taken it for granted that other types of values could be passed as parameters, bound to variables, and so on
  - Not all languages let you pass in functions as parameters

- Type Annotations
  - Many languages require, or at least allow, type annotations on variables, functions
  - This is used to let the prgrammer specify static type information

- Extreme Type Inference
  - ML takes type inference to extremes - infers a static type for every expression and for every function, without annotations

- Simple Type Inference
  - Most languages require some simple kinds of type inference
  - Expressions may have static types, inferred from operators and types of operands

- Static Type Checking
  - Static type checking determines a type for everything before running the program: variables, functions, expressions, everything
  - Most modern languages are statically typed

- Dynamic Typing
  - In some languages, programs are not statically type-checked before being run
  - They are still dynamically type checked at runtime, the language system checks that operands are of suitable types for operators
  - Although dynamic typing does not type everything at compile time, it still uses types, even moreso than static languages, since it must store type information along with values in memory

- Static And Dynamic Typing
  - Statically typed languages often use some dynamic typing
  - Everything is typed at compile time, but compile-time type may have subtypes
  - At runtime, it may be necessary to check a value's membership in a subtype

- Type Equivalence
  - An important question for static and dynamic type checking is when types are the same?
  - Name equivalence: types are the same if and only if they have the same name
  - Structural equivalence: types are the same if and only if they are built from the same primitive types using the same type constructors in the same order
  - ML uses more structural equivalence, PASCAL more name equivalence

Chapter 7 - A Second Look at ML

- Constants as Patterns
  - Underscore can be used as a pattern, it matches anything, but doesn't bind a variable
  - Can explicitly list the values you want to match, but if you don't cover everything you will get a non-exhaustive warning at runtime

- Local Variable Definitions
  - When you use val at the top level to define a variable, it is visible from that point forward
  - There is a way to restrict the scope of definitions: the let expression `<let-exp> ::= let <definitions> in <expression> end`
  - Here, the variable is only allowed to be used within the `in <expression>`
- Example: Merge Sort
  - The halve function divides a list into two nearly-equal parts of a tuple

```
fun halve nil = (nil, nil)
|   halve [a] = ([a], nil)
|   halve (a::b::cs) =
        let
            val (x, y) = halve cs
        in
            (a::x, b::y)
        end;
```

  - Merge merges two sorted lists

```
fun merge (nil, ys) = ys
|   merge (xs, nil) = xs
|   merge (x::xs, y::ys) =
        if (x < y) then x :: merge(xs, y::ys)
        else y :: merge(x::xs, ys);
```

  - And now the full merge sort:

```
fun mergeSort nil = nil
|   mergeSort [a] = [a]
|   mergeSort theList =
        let
            val (x, y) = halve theList
        in
            merge(mergeSort x, mergeSort y)
        end;
```

Chapter 8 - Polymorphism

- Functions with the flexibility to take in many different types are known as polymorphic, and this takes on many forms
- Overloading
  - An overloaded function name or operator is one that has at least two definitions, all of different types
  - Many languages have overloaded operators, while some even let you define your own, like C++
  - C++ lets virtually all operators to be overloaded including: `+,-,*,/,%,^,&,|,~,!,=,<,>, +=,-=,=,*=,/=,%=,^=,&=,|=, <<,>>,>>=,<<=,==, !=,<=,>=,&&,||,++,--,->*„`
  - Compilers usually implement overloading in the same way:
    - Create a set of monomorphic functions, one for each definition
    - Invent a mangled name for each, encoding the type information
    - Have each reference use the appropriate mangled name, depending on the parameter types
- Coercion
  - A coercion is an implicit type conversion, supplied automatically even if the programmer leaves it out
  - An example in Java would be:

```
double x;
x = 3; // Casts 3 to double
```

- Parameter Coercion
  - When a language supports coercion of parameters on a function call, the resulting function (or

operator) is polymorphic

```
void f(double x) {
    ...
}
// This f can be called with any type of parameter Java is willing to coerce to
// type double

f((byte) 1);
f((short) 2);
f('a');
f(3);
f(4L);
f(5.6F);
```

- Defining Coercions
  - Languages usually have pages defining valid coercions, through languages like ML have none
  - This can get tricky however with complex function definitions
- Parametric Polymorphism
  - A function exhibits parametric polymorphism if it has a type that contains one or more type variables
  - A type with type variables is a polytype
  - In C++:

```
template<class X>
X max(X a, X b) {   return a>b ? a : b; }
```

  - In ML:

```
- fun identity x = x;
val identity = fn : 'a -> 'a
- identity 3;
val it = 3 : int
- identity "hello";
val it = "hello" : string
- fun reverse x =
=   if null x then nil
=   else (reverse (tl x)) @ [(hd x)];
val reverse = fn : 'a list -> 'a list
```

- Subtype Polymorphism
  - A function or operator exhibits subtype polymorphism if one or more of its parameter types have subtypes

Chapter 9 - A Third Look at ML

- Case expressions
  - The syntax for case expressions: `<case-expr> ::= case <expression> of <match>`
  - So in ml:

```
- case 1+1 of
=    3 => "three" |
=    2 => "two" |
=    _ => "hmm";
```

- Anonymous Functions
  - Functions in ML don't have names, rather you assign them to variables that do

```
(* Named function: *)
- fun f x = x + 2;
val f = fn : int -> int
```

```
(* Anonymous function: *)
- fn x => x + 2;
val it = fn : int -> int
```

- Higher-order functions
  - Every function has an order
  - A function that does not take any functions as parameters, and does not return a function value, has order 1
  - A function that takes in a function or returns a function has order n+1, where n is the highest-order parameter/returned val
- Currying
  - Functions in ML take one parameter, then for functions of multiple parameters, they are curried so a function that takes in two ints actually is a function that takes in an int and then that gives a function that takes in another int
  - Advantages is no tuples, and we get to specialize functions for particular initial parameters

Chapter 10 - Scope

- Definitions
  - When there are different variables with the same name, there are different possible bindings for that name
  - A definition is anything that establishes a possible binding for a name
- Scope
  - There may be more than one definition for a given name
  - An occurrence of a name is in the scope of a given definition of that name whenever that definition governs the binding for that occurrence
- Blocks
  - A block is any language construct that contains definitions, and also contains the region of the program where those definitions apply

```
let
    val
    val
in
    x+y
end
```

- Different ML Blocks
  - The `let` is just a block: no other purpose
  - A `fun` definition includes a block
  - All the matching alternatives are their own block
- Java Blocks
  - In Java and other C-like languages, you can combine statements into one compound statement using { and }
  - A compound statement also serves as a block
- Classic Block Scope Rule
  - he scope of a definition is the block containing that definition, minus the scopes of any redefinitions of the same name in interior blocks
  - Most statically scoped, block-structured languages use this or some minor variation
- Labeled Namespaces
  - A labeled namespace is any language construct that contains definitions and a region of the program where those definitions apply
- ML Structure
  - A little like a block: a can be used anywhere from definition to the end

```
structure Fred = struct
    val a = 1;
```

```
        fun f x = x + a;
    end;
```

- Namespace Advantages
  - Allow for simple definitions like `max` to be available multiple times in different namespaces
  - Can avoid name conflicts
- Dynamic Scoping
  - Each function has an environment of definitions
  - If a name that occurs in a function is not found in its environment, its caller's environment is searched
  - And if not found there, the search continues back through the chain of callers
- Static Vs. Dynamic
  - The scope rules are similar
  - Both have scope holes - places where a scope does not reach because of redefinition
  - static rule talks only about regions of program text, so it can be applied at compile time
  - The dynamic rule talks about runtime events
- Seperate Compilation
  - Parts are compiled separately, then linked together
  - Scope issues extend to the linker: it needs to connect references to definitions cross separate compilations
- C Approach - Compiler Side
  - Two different kinds of definitions:
    - Full definition
    - Name and type only: a declaration in C-talk
- C Approach, Linker Side
  - When the linker runs, it treats a declarationas a reference to a name defined in some other file
  - It expects to see exactly one full definition of that name

Chapter 11 - A Fourth Look At ML

- Type Definitions
  - Predefined, but not primitive in ML:

    ```
    datatype bool = true | false;
    ```

  - Type constructor for lists:

    ```
    datatype 'element list = nil |
        :: of 'element * 'element list
    ```

- Wrappers
  - You can add a parameter of any type to a data constructor, using the keyword `of`

    ```
    datatype exint = Value of int | PlusInf | MinusInf;
    ```

  - In effect, such a constructor is a wrapper that contains a data item of the given type
  - `Value` is a data constructor that takes a parameter: the value of the int to store
  - It looks like a function that takes an `int` and returns an `exint` containing that `int`
- To recover a data constructor's parameters, use pattern matching

```
- val (Value y) = x;
val y = 5 : int
```

- Type Constructors With Parameters
  - Type constructors can also use parameters:

```
datatype 'a option = NONE | SOME of 'a;
```

  - Type constuctor parameter comes before the type constructor name
  - We have types `'a option` and `'a list`

- Uses For option
  - Used by predefined functions when the result is not always defined

```
- fun optdiv a b =
=   if b = 0 then NONE else SOME (a div b);
val optdiv = fn : int -> int -> int option
```

Chapter 12 - Memory Locations For Variables

- A Binding Question
  - Variables are bound (dynamically) to values
  - How are variables bound to memory locations?
- Function Activations
  - Activation of a function: the lifetime from execution to return
  - Activation-specific variable: each activation has its own binding of a variable to a memory location
- Activation Specific variables
  - In most modern languages, activation-specific variables are the most common kind

```
fun days2ms days =
    let
        val hours = days * 24.0
        val minutes = hours * 60.0
        val seconds = minutes * 60.0
    in
        seconds * 1000.0   end;
```

- Most imperative languages have a way to declare a variable that is bound to a single memory location for the entire runtime
- Object-oriented languages use variables whose lifetimes are associated with object lifetimes
- Scope And Lifetime Differ
  - In most modern languages, variables with local scope have activation-specific lifetimes,
- Block Activation Records
  - When a block is entered, space must be found for the local variables of that block
  - Can do this by either preallocating the containing function's activation record, extend the functions activation record, or allocate seperate block activation records
- Static Allocation
  - The simplest approach is to allocate one activation record for every function statically
  - This is simple and fast and used by old languages like Cobol and Fortran
  - However, each function only gets one record and can only have on activation alive at a time, which is broken for recursion or multithreading
- Stacks Of Activation Records
  - To support recursion: need new activation record for each activation
  - Dynamic allocation: activation record allocated when function is called
- Current Activation Record
  - Location of current activation record is not known until runtime
  - A function must know how to find the address of its current activation record
- Nesting Functions
  - Function definitions can be nested inside other function definitions like in ML, Python, Pascal, etc
  - An inner function needs to be able to find the address of the most recent activation for the outer function
- Functions As Parameters

  - What happens when you pass in a function as a parameter?
  - Functional languages allow many more kinds of operations on function-values
  - Function-values include both parts: code to call, and nesting link to use when calling it

Chapter 13 - A First Look At Java

- Java Terminology
  - Each point is an object
  - Each includes three fields
  - Each has three methods
  - Each is an instance of the same class
- Constructed Types
  - Constructed types are all reference types: they are references to objects
  - Includes any class, interface, or array type
- Strings
  - Predefined but not primitive: a class `String`
  - The + operator has special overloading and coercion behavior for the class String
- Operators With Side Effects
  - An operator has a side effect if it changes something in the program environment, like the value of a variable or array element
  - Assignment is an important part of what makes a language imperative
- Rvalues and Lvalues
  - Why does a=1 make sense, but not 1=a?
  - Expressions on the right must have a value, while expressions on the left must have a memory location
  - In most languages, the context decides whether the language will use the rvalue or the lvalue of an expression
  - Side-effecting expressions have both a value and a side effect
  - Value of `x=y` is the value of `y`; side-effect is to change `x` to have that value
- Class Method Calls
  - Class methods define things the class itself knows how to do, not objects of the class
- Method Call Syntax:
  - Normal instance method call
    `<method-call> ::= <reference-expression>.<method-name> (<parameter-list>)`
  - Normal class method call
    `<method-call> ::= <class-name>.<method-name> (<parameter-list>)`
  - Either kind, from within another method of the same class
    `<method-call> ::= <method-name>(<parameter-list>)`
- Object Creation Expressions
  - Objects are created with `new`
  - Objects are never explicitly destroyed, rather Java's garbage collection does that
- General Operator Info
  - All left-associative, except for assignments
  - 15 precedence levels, use parentheses
- References
  - A reference is a value that uniquely identifies a particular object
    ```
    public IntList(ConsCell s) {
        start = s;
    }
    ```
  - What gets passed to the IntList constructor is not an object — it is a reference to an object
  - Objects in Java are effectively pointers
  - Java variable cannot hold an object, only a reference to an object

Chapter 14 - Dynamic Memory Allocation

- Lots of things like objects and activation records require memory at runtime
- Language systems provide a runtime hidden memory management
- Declaring An Array
  - A Java array declaration:

    ```
    int[] = null
    ```

  - Array types are reference types - an array is really an object
  - Right now, we essentially declared an integer pointer, which can point to an int array
- Creating an Array:

  ```
  int[] a = new int[100];
  ```

- Stack of activation Records
  - For almost all languages, activation records must be allocated dynamically
  - These are statically allocated variables, that are pushed onto the stack and then freed after they are done
  - For the stack, the order of variables must be constant and known, what happens if allocations and deallocations happen out of order
- The Heap Problem and First Fit
  - A heap is a pool of blocks of memory, with an interface for unordered runtime memory allocation and deallocation
  - First fit is a linked list of free blocks, to allocate, can just search free list for first adequate block, and allocate however much memory is needed, returning the rest to the rest of the heap
- Quick Lists
  - Small blocks tend to be allocated and deallocated much more frequently, so a common optimization is to keep separate free lists for popular block sizes, so here blocks are one constant size
- Fragmentation
  - Note the heap may have a problem of fragmentation where there are a lot of small unused pieces of memory.
- Current Heap Links
  - A current heap link is a memory location where a value is stored that the running program will use as a heap address
- Heap Compaction
  - Manager can move allocated blocks to new locations and update all links on that block
  - So it can get rid of fragmentation by moving all the allocated blocks to one end
- Garbage Collection Approaches
  - **Mark and Sweep**
    - Uses current heap links in a 2 stage process
    - Mark - find the live link heaps and mark all the heap blocks linked to them
    - Sweep - make a pass over the heap and return unmarked blocks to the free pool
  - **Copying**
    - A copying collector divides memory in half, and uses only one half at a time
    - When one half becomes full, find live heap links, and copy live blocks to the other half
  - **Reference Counting**

        * Each block has a counter of heap links to it
        * When counter goes to zero, block is garbage and can be freed
        * Has a problem with cycles of garbage, and in general performance since the overhead is high

- Garbage collection
  - Generational Collection
    * Have generational counters that divide blocks according to age
    * Garbage collect in the younger generations more often
  - Incremental Collection
    * Collect garbage a little at a time
    * Avoid having spikes in overhead caused by the mark and sweep operations

Chapter 15 - A Second Look at Java

- Interfaces
  - A method prototype just gives the method name and type—no method body
  - An interface is just a collection of method prototypes

```
public interface Drawable {
    void show(int xPos, int yPos);
    void hide();
}
```

- Implementing Interfaces
  - A class can declare that it implements a particular interface
  - Then it must have public functions that match the interface
- Benefits of an Interface
  - Interface can be implemented by many classes
  - Interface leads to use of polymorphism
- Polymorphism
  - One class can be derived from another, using the keyword `extends`
  - Classes that are derived inherit all the parent methods and fields
- Inheritance Chains
  - All classes but one are derived from some class
  - If you do not give an `extends` clause, Java supplies one: `extends` Object
  - Object is the root class
- The Object Class
  - All classes inherit methods from object like:
    * toString, for converting to a String
    * equals, for comparing with other objects
    * hashcode, for computing an int hash code
- Extending And Implementing
  - Classes can use `extends` and `implements` togethe
  - For every class, Java keeps track of what interfaces it implements, the methods it has to define, the methods defined for it, and the fields defined for it
  - An `implements` affects the first two, but an `extends` affects all four of those options
- Abstract Classes

  - Classes can get out of implementing all their methods by making them abstract classes
  - An abstract class is used only as a base class, no objects for that class will be created

- Collision Problem and Diamond Problem
  - When you have multiple classes you inherit from, might have conflicts when two parent classes have a method with the same name
  - A language that supports multiple inheritance must have mechanisms for handling these problems
  - Java designers that this additional power was not worth the language complexity

- Generics
  - Weakensses are no compile-type type checking on element types and primitives must be stored in their corresponding object type first, so `int` would actually be `Integer`
  - You can now use generics with the distinct angle bracket notation

```
interface Worklist<T> {
    void add(T item); booleanhasMore();
    T remove();
}

Worklist<String> w;
...
w.add("Hello");
String s = w.remove();
```

- Using Generic Classes

```
Stack<String> s1 = new Stack<String>();
Stack<Integer> s2 = new Stack<Integer>();
s1.add("hello");
String s = s1.remove();
s2.add(1);
inti = s2.remove();
```

  - Notice the coercions: `int` to `Integer`

Chapter 18 - Parameters

- By Value
  - The formal parameter is just like a local variable in the activation record of the called method. The only difference is that it is initialized using the value fot he corresponding actual parameter before the method executes
  - Simplest method that is widely used, and the only method in real Java
  - Note that pointers will still be able to be modified outside the scope of the function. So objects in Java can be modified even outside the method since they are only references
- By Result
  - Also called ***copy-out***, where the formal parameter is just like an uninitialized local variable, where after the method finishes executing, the final value of the formal parameter is assigned to the actual parameter
  - The actual parameter must have an lvalue
- By Value-Result
  - The formal parameter is just like a local variable in the activation record of the called method.
  - It is initialized using the value of the actual parameter before the method execute
  - It is initialized using the value of the actual parameter before the method executes
  - After the method finishes executing, the final value of the formal parameter is assigned to the actual parameter

- By Reference
  - The lvalue of the actual parameter is computed before the called method executes. Inside the method, that lvalue is used as the lvalue of the formal parameter.
  - In effect, the formal parameter is an alias for the acutal parameter
  - When two expressions have the same lvalue, they are aliases of each other
- By Macro Expansion
  - The body of the macro is evaluated in the caller's context. Each actual parameter is evaluated on every use of the corresponding formal parameter, in the context of that occurrence of that formal parameter
  - Done by the preprocessor
- By Need
  - Each actual parameter is evaluated in the caller's context, on first use of the corresponding formal parameter. The value of the actual parameter is then cached.
  - Avoids wasteful recomputations of by-name, used by lanugages like Haskell

Chapter 19 - A First Look At Prolog

- Terms
  - Everything in Prolog is built from terms
  - There are three kinds of terms: Constants: integers, real numbers, atoms, Variables, and Compound terms
- Constants
  - Integer and real constants: 1 vs 1.23
  - Atoms: A lowercase letter followed by any number of additional letters, digits or underscores, a sequence of non-alphanumeric characters plus a few special atoms: like []
- Atoms Are Not Variables
  - An atom can look like an ML or Java variable
  - Atoms are more like string constants
- Variables
  - Most of the variables you write will start with an uppercase letter
  - Those starting with an underscore, including _, get special treatment
- Compound Terms
  - An atom followed by a parenthesized comma-seperated list of items

    ```
    x(y,z), +(1,2), .(1,[]),
    ```

  - Think of them as structured data
- Terms
  - All Prolog programs are built on terms

    ```
    <term> ::= <constant>|<variable> |<compound-term>
    <constant> ::= <integer> |<real number> | <atom>
    <compound-term> ::= <atom> ( <termlist> )
    <termlist> ::= <term>|<term> , <termlist>
    ```

- Unification
  - Two terms unify if there is some way of binding their variables that makes them identical
  - For instance, `parent(adam,Child)` and `parent(adam,seth)` unify by binding the variable `Child` to the atom `seth`

- The Prolog Database
  - A Prolog language system maintains a collection of facts and rules of inference
  - A Prolog program is just a set of data for this database, organized by facts
  - An example of a Prolog program is

```
parent(kim,holly).
parent(margaret,kim).
parent(margaret,kent).
parent(esther,margaret).
parent(herbert,margaret).
parent(herbert,jean).
```

- The `consult` Predicate
  - Predefined predicate to read a program from a file into the database
- Queries
  - A query asks the language system to prove something, return true or false
  - Some queries (like `consult`) are only executed for the side effects
- Conjunctions
  - A conjunctive query has a list of query terms separated by commas

```
?- parent(margaret,X), parent(X,holly).
X = kim .
```

  - The Prolog system tries prove them all
- The Need For Rules
  - A rule says how to prove something: to prove the head, prove the conditions

```
greatgrandparent(GGP,GGC) :-
parent(GGP,GP),
parent(GP,P),
parent(P,GGC).
```

  - To prove `greatgrandparent(GGP,GGC)`, find some `GP` and `P` for which you can prove `parent(GGP, GP)`, them `parent(GP, P)`, and then finally `parent(P, GGC)`
- A Program With The Rule

```
parent(kim,holly).
parent(margaret,kim).
parent(margaret,kent).
parent(esther,margaret).
parent(herbert,margaret).
parent(herbert,jean).
greatgrandparent(GGP,GGC) :-
    parent(GGP,GP), parent(GP,P), parent(P,GGC).
```

  - A program consists of a list of clauses
  - A clause is either a fact or a rule, and ends with a period
- Recursive Rules

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :-
```

```
parent(Z,Y),
ancestor(X,Z).
```

  - X is an ancestor of Y if:
    - Base case: X is a parent of Y
    - Recursive case: there is some Z such that Z is a parent of Y, and X is an ancestor of Z

- Core Prolog Syntax
  - The core syntax of Prolog can be boiled down to

```
<clause> ::=  <fact> |<rule>
<fact> ::=  <term> .
<rule> ::=  <term> :- <termlist> .
<termlist> ::=  <term>|<term> , <termlist>
```

- The Procedural Side
  - A Prolog program specifies proof procedures for queries
- The Declarative Side
  - A rule is a logical assertion
  - Just a formula – it doesn't say how to doanything – it just makes an assertion
- Declarative Languages
  - Each piece of the program corresponds to a simple mathematical abstraction
  - Many view it as the opposite of imperative
- Operators
  - An operator is just a predicate for which a special abbreviated syntax is supported
- The = Predicate
  - The goal =(X,Y) succeeds if and only if X and Y can be unified

```
?- parent(adam,seth)=parent(adam,X).
X = seth.
```

- Arithmetic Operators
  - Predicates +, -, * and / are operators too, with the usual precedence and associativity

```
?- X = +(1,*(2,3)).
X = 1+2*3.

?- X = 1+2*3.
X = 1+2*3.
```

- Not Evaluated
  - The term `+(1, *(2,3))` is not evaluated
  - There is a way to make Prolog evaluate such term
- Lists in Prolog
  - Similar to ML lists, the atom [] represents the empty list
  - A predicate . corresponds to ML's :: operator

ML Expression	Prolog Term
[]	[]
1::[]	.(1,[])

ML Expression	Prolog Term
1::2::3::[]	.(1,.(2,.(3,[])))

- The Anonymous Variable
  - The variable _ is an anonymous variable
  - Every occurrence is bound independently of every other occurrence
  - Matches any term without introducing bindings

    ```
    tailof([_|A],A).
    ```

  - This `tailof(X,Y)` succeeds when X is a non-empty list and Y is the tail of that list
- The not Predicate
  - For simple applications, it often works quite a bit logical negation

    ```
    ?- member(1,[1,2,3]).
    true .

    ?- not(member(4,[1,2,3])).
    false.
    ```

  - To prove `not(X)`, Prolog attempts to prove `X`
  - `not(X)` succeeds if `X` fails

Chapter 20 - A Second Look At Prolog

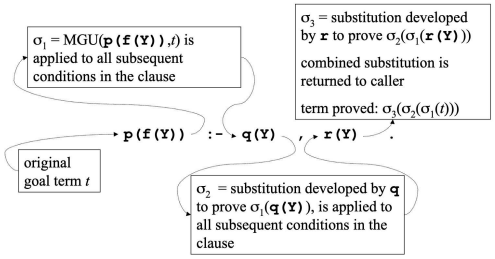
- Substitutions
  - A substitution is a function that maps variables to terms: σ = {`X`→`a`, `Y`→`f(a,b)`}
  - This σ maps `X` to `a` and `Y` to `f(a,b)`
  - The result of applying a substitution to a term is an instance of the term
  - σ(`g(X,Y)`) = `g(a,f(a,b))` so `g(a,f(a,b))` is an instance of `g(X,Y)`
- Unification
  - Terms t_1 and t_2 unify if there exists a σ such that $\sigma(t_1) = \sigma(t_2)$
  - Examples:
    - `a` and `b` do not unify
    - `f(X,b)` and `f(a,Y)` unify: a unifier is `{X→a, Y→b}`
    - `f(X,b)` and `g(X,b)` do not unify
    - `a(X,X,b)` and `a(b,X,X)` unify: a unifier is `{X→b}`
    - `a(X,X,b)` and `a(c,X,X)` do not unify
    - `a(X,f)` and `a(X,f)` do unify: a unifier is `{}`
- Most General Unifier
  - Term x_1 is more general than x_2 if x_2 is an instance of x_1 but not vice-versa, so `parent(fred,Y)` is more general then `parent(fred,mary)`
  - The most general unifier σ_1 of two terms t_1 and t_2 if there doesn't exist a unifier σ_2 that is more general
- The Occurs Check
  - Any variable `X` and term `t` unify with `{X→t}`
    - `X` and `f(a,g(b,c))` unify: an MGU is `{X→f(a,g(b,c))}`

    - `X` and `f(a,Y)` unify: an MGU is `{X→f(a,Y)}`
  - Unless `X` occurs in `t`:
    - `X` and `f(a,X)` do not unify, in particular not by `{X→f(a,X)}`

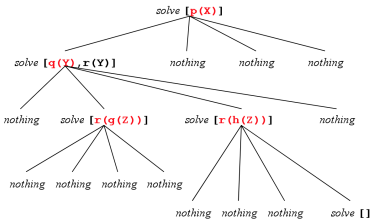
- A Procedural View
  - Each clause is a procedure for proving goals
  - Take for example `p :- q, r.` - to prove a goal, first unify the goal with p, then prove q, then prove r

- Backtracking
  - Prolog explores all possible targets of each call, until it finds as many successes as the caller requires or runs out of possibilities

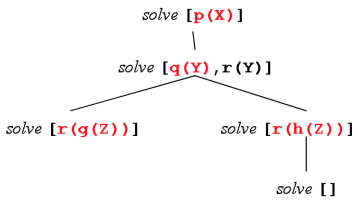
- Substitution



- Proof Trees
  - Proof trees capture the order of traces of prove, without the code
  - Root is original query
  - Nodes are lists of goal terms, with one child for each clause in the program



  - Children of a node represent clauses, in the order they appear in the program
  - nothing nodes, which represent clauses that do not apply to the first goal in the list can be eliminated
  - So the final, simplified tree wil look like this:



- Prolog Semantics
  - A Prolog language system must act in the order given by a depth-first, left to right traversal of the proof tree

- Variable Renaming
  - To avoid capture, use fresh variable names for each clause

  - The first application of `reverse` might be

```
reverse([Head1|Tail1],X1) :-
    reverse(Tail1,Y1),
    append(Y1,[Head1],X1).
```

  - The Next might be

```
reverse([Head2|Tail2],X2) :-
    reverse(Tail2,Y2),
    append(Y2,[Head2],X2).
```

- Quoted Atoms As Strings
  - Any string of chars enclosed by quotes is a term and treated as an atom
  - This allows for I/O:

```
?- write('Hello world').
Hello world
true.

?- read(X).
|: hello.
X = hello.
```

- The Cut
  - Written !, pronounced "cut", is a goal that always succeeds (like true), but when it succeeds, it eliminates some backtracking
  - If q1 through qj succeed, the cut does too and tells Prolog there's no going back, so there is no backtracking to look for other solutions for q1 through qj

```
p :- q1, q2, ..., qj, !.
```

  - the first solution found for a given goal using this rule will be the last solution found for that goal

Chapter 22 - A Third Look At Prolog

- Unevaluated Terms
  - Prolog operators allow terms to be written more concisely, but are not evaluated
  - These are all the same term:

```
+(1,*(2,3))
1+ *(2,3)
+(1,2*3)
(1+(2*3))
1+2*3
```

- Evaluating Expressions
  - The `is` predicate can be used to evaluate a term that is a numeric expression
  - `is(X,Y)` evaluates the term `Y` and unifies `X` with the resulting atom
  - Predicates must be defined at the time of operation

```
?- Y is X+2, X=1.
ERROR: is/2: Arguments are not sufficiently instantiated

?- X=1, Y is X+2.
X = 1, Y = 3.
```

  - For `X is Y`, the predicates that appear in `Y` have to be evaluable
- Real Values and Integers

  - There are two numeric types: integer and real.
  - Most of the evaluable predicates are overloaded for all combinations.
  - Prolog is dynamically typed and types are resolved at runtime
- Comparisons
  - Numeric comparison operators: `<, >, =<, >=, =:=, =\=`
  - Prolog evaluates both sides and then compares numerically
- Equalities
  - There are 3 types of equality in Prolog
  - `X is Y` evaluates `Y` and unifies the result with `X`
    - `3 is 1+2` succeeds, but `1+2 is 3` fails
  - `X = Y` unifies `X` and `Y`, with no evaluation: both
  - `X =:= Y` evaluates both and compares
  - Here is any example of `mylength`

```
mylength([],0).
mylength([_|Tail], Len) :-
    mylength(Tail, TailLen),
    Len is TailLen + 1.

?-mylength([a,b,c],X).X
X = 3.
```

- The `findall` Predicate
  - `findall(X,Goal,L)` finds all ways of proving `Goal` and for each applies to `X` the same substitution that made a provable instance of Goal and unigies `L` with all those `X`s