Levi D. Smith CS3411 Programming Language Concepts

CS 3411 * Programming Language Concepts * Summer 99

Welcome to the Summer offering of CS 3411! This course is a **comparative study** of important programming language concepts. We will study the history and development of **features** and **constructs** in a wide variety of programming languages. We will develop an appreciation for language design as a craft and hone our skill at selecting the appropriate language for the task at hand. Along the way we will learn how languages are specified (syntax and semantics) and implemented. In addition we will develop an appreciation for the interrelationship between programming methodologies, language features and hardware capabilities. You will be asked to prove your understanding by implementing three programs in the languages Perl, Prolog and C++. The summer term is accelerated and we will move quickly. I will make the programming assignments a bit easier and assign 3 instead of 4 homeworks to help out.

Details

Course: CS3411 Programming Language Concepts 3-0-3 (Section a)

Prerequisites: 2390 Modeling & Design (object-oriented programming)

2430 Control & Concurrency (C, procedural programming)

NOTE: 2360 Knowledge Representation and Processing is no longer a prerequisite for this class. We will spend one lecture on LISP and functional programming. Having taken 2360 will obviously be an advantage (usually about half of the students have 2360). Also, some familiarity with object-oriented concepts is required as a prerequisite. Students have successfully taken 2390 concurrently with 3411 but it is actually a prerequisite and I will assume you have knowledge of Smalltalk. Finally, knowledge of Java (1502) will be helpful but is not required.

Lectures: MW 4:00-5:45pm College of Computing Building CCB Room 101

Text: required: Sebesta *Concepts of Programming Languages* 4e 1999 Addison-Wesley

Exams:

midterm: Wed 14 July 7:30-9pm CCB 101 (MAKEUP Fri 16 July)

final: Tues 10 August 2:50-5:40pm (CCB 101)

NOTE: Since this section meets at an "irregular" time, we get to hold the exam during class time. I have selected Tuesday but need to confirm the availability of the room.

Homeworks: 3

Programs: 3 (Perl, C++, Prolog)

Grading: Exams 25% 25% (graduating seniors: 25%, 25%)

Homeworks 7% 7% 6%

Programs 10% 10% 10%

Instructor: Phillip W. Hutto

office:	119 College of Computing Building (CCB)
email:	pwh@cc.gatech.edu
phone:	h 404.892.7444 (Emory 404.727.0668)
hours:	Tu, Th 4-5 in the Student Commons CCB and by appointment
TA/Hours:	Chris Bradfield (MW 3-4 Commons) (ceejaybe@cc.gatech.edu)
Newsgroup:	git.cc.class.3411a (read it! this is a **requirement**!!)
Web Page:	http://www.cc.gatech.edu/classes/cs3411_99_summer

Exams

There will be a midterm and final exam. The midterm will be scheduled OUTSIDE of regular class time in the evening. You are responsible for clearing your schedule. If you have an insurmountable conflict, send email to the instructor (pwh@cc.gatech.edu) to schedule a MAKEUP. Exam times are:

Midterm: Wednesday 14 July 7:30-9pm (CCB 17 tentative)
MAKEUP Friday 16 July
Final: Tuesday 10 August 2:40-5:40pm (CCB 101)

Late Policy

Homeworks and projects may be turned in up to 3 school days late for a 10% penalty each day. Late penalties are not assessed over weekends or holidays. If a homework is due at 4pm on Wednesday, it is considered 1 day late up until 4pm the next school day (Thursday). No late submissions are accepted beyond the third school day.

Turn-in Instructions

Projects will be submitted using the turnin or workon system on acme. Individual instructions will be made available for each project specifying a file or directory and any supplemental information (such as README files) that must be submitted. Homeworks are due at the beginning of class and you must submit a legible hardcopy. Email submissions are not accepted for either homeworks or projects. Homeworks must be stapled and have your name, the homework number and your GT number clearly printed on the first page. A 5% penalty will be deducted for not complying with the turnin instructions.

Instructor

Phil spent his youth in the Masters and PhD programs at Tech after receiving an undergraduate degree in computer science from Brown University. Since then he has worked as a software developer in industry, as a research scientist at Georgia Tech and Emory and as an instructor at "the other place" in Athens and, for the last three years, at Emory University. This is Phil's fourth quarter back at Tech teaching and his fourth crack at 3411. Phil is also co-teaching 3431 with Vernard Martin. He will be continuing his "day job" at Emory where he will be teaching an introductory Java course in the mornings. Look for him on campus at Tech M-Th afternoon this quarter. (Email anytime!)

3411 Summer

1	JUN	m 21	ch1,2	Intro	
2		w 23	ch3	Syntax/Semantics	
3		m 28		PERL	P1, H1 OUT
4		w 30	ch4	Names etc.	
	JUL	m 5		HOLIDAY	
5		w 7	ch5	Data Types	
6		m 12	ch6	Expressions	P1, H1 DUE 4 pm / P2, H2 OUT
7		w 14	ch11	C++	MIDTERM 7:30-9:00 pm location TBA
		f 16			LAST DROP DATE / MIDTERM MAKEUPS
8		m 19	ch11	C++	
9		w 21	ch7	Control	
10		m 26	ch8	Subprograms	P2, H2 DUE 4 pm / P3, H3 OUT
11		w 28	ch9	Subprogram Imp	
12	AUG	m 2	ch15	PROLOG	
13		w 4	ch13,14	Exceptions/FP	P3, H3 DUE 4 pm
		t 10			FINAL EXAM 2:50-5:40 CCB 101

CS 3411 - Programming Languge Concepts June 21, 1999

~~Lect 1 - Programming Language Landscape~~
~~Lect 2 - History of Langugе Design~~
~~Lect 3 - Syntax and Semantics~~
~~Lect 4 - Names, Bindings, Type Checking, Scope~~
~~Lect 5 - Perl~~
~~Lect 6 - Data Types~~
~~Lect 7 - Expressions and the Assignment Statement~~
~~Lect 8 - Abstract Data Types~~
~~Lect 9 - Statement Level Control Structures (Sebesta Ch 7)~~
~~Lect 10 - Sub program~~
~~Lect 11 - Implementation~~

Website: http://www.cc.gatech.edu/classes/cs3411b-99-spring
Instructor: Phil Hutto

"Concepts" - languages at an abstract level
Book: White / Blue
- historical components, terms, need to know for exams.
- Ex question: Tell me the history of of x ...
- Specific Features, A lot of details.
- 3 small programs - C++ ~~and~~, Prolog, and Perl
- Prolog - logic language, for AI.
- Midterm, Final, and some homeworks

★ Final Exam Question - He gives 5 programming langs and you tell the most important features

Book: Concepts of Programming Langues
Mid - 25%-30%
Final - 25%-40%
Home works
Office 119 - holds office hours in commons

Newsgroup Required -

Syllabus - 13 lectures
Midterm will be held outside class
1. Ch 1-2: Intro, History (a lot of History)
2. Ch 3: Syntax / Semantics (formal theory)
3. PERL (Perl Assignment Following)
4. Ch 4 Naming, Scoping
5. Ch 5 Data Types
6. Ch6 Assignment, Expressions
(Strange Things / Tricks about C
7. C++
8. C++ (Assignment Follows)
9. Ch 7 Control Folow - if / loop / switch / goto
10. Ch 8 Subprograms (procedures, functions, subroutines) - parameter passing
11. Ch 9 - Subprograms - Implementation issues (Compiler Stuff)

Question - operational semantics - what does a new feature do?? foobar X
→ what you would read in a language reference manual - Pseudo code

parameters are checked at compile time; functions are more like jump statements

12. Prolog

13. Exception Handling - Functional Programming

- Explain, Define, Describe questions - Essay Questions on exam

---

Lecture One June 21

- Why Study Languages?
  - Find Features in a language that you can't do in other languages
    - Can't find a job with old languages (PL One)
    - Language can enable new types of programs
    - to be able to "say it well"
    - develop a craftsman's concern for his tools
    - REALLY understand all the features (constructs)
      - there are sometimes bad features - no perfect languages
    - increase your programming vocabulary
    - helps you learn new languages
    -
- History of Language Development
  - Some languages develop because of large entities (companies, military, etc)
- Language Evaluation Criteria
  - how do you decide if one way of doing something is better than another way.
  - "Big Four"
    1) Readability - trying to figure out what someone else has written. Can you read it yourself??
    2) Writability - if there is a feature in that lang that takes less lines to write, then that lang is more writable

Mathematica, Maple - less lines of code to write math progs than C

Writable aka expressive
Readability can suffer
EX: Perl - one command to open a file, load it, and close it
This can limit what you can do

strtok

3.) Reliability - your program won't run the same on all computers. Language standard specifies how things are done
• pointers are dangerous (coredumps). Java has references

lose writeability & efficiency (types)

4.) Cost - how hard is it to learn. • compilation cost - waiting hours for a program to compile. - some langs just compile the parts needed.

"simplicity" "expressiveness" "safety"
"efficiency" "orthogonality"

Ada does a lot of type checking vs. Small Talk less Reliable

BASIC - Darthmouth - for liberal arts students
- first interactive interpretive language
• Biggest Cost in software is maintence

Pascal - easy language
Ada, C++ - big languages
Java - not big, but has a lot of libraries
PL1 - has a lot of features, big

orthogonality (perpendicular) design concept where you start with a small number of components, combining rules, then you can put those small components together to make something larger. No SPECIAL CASES. - good for learning - you don't have to remember special cases.

Assembly language - ADD, R1, R2, R3
ADD, R1, #10, #20
ADD, R1, *1000, #20

if you can mix these up in any way than your language is orthogonal.

June 21, 1999

you have to put { in Java try blocks (non orthogonal)

<u>Algol-68</u> - VERY orthogonal - wrote a lot of code (features) that aren't ever used

C - all parameters are passed by value except arrays

int f(int a, int [] b)

---

int f(int [5] list)

int g(struct { int a; int b }) not allowed, non orthogonal

~~Interpretors vs. Translators~~

- Procedural (Imperative) - list of commands
- Object Oriented - communication between languages - populate the world with objects.
- Functional - a sequence of function applications
- Logic - based on formal logic - predicate calculus. - you tell it what you want it to do. System proves the answer - it does a proof.

(Java Layout Managers)

Interpreters vs. Translator (compiler)

- Interpreted - SmallTalk, Perl, Basic
  - they are "semi" compiled, to an intermediate language
- Compiled - lacks flexibity
- Interpreted - a lot of things are done at run time

  input → program

  output → the output of the program

Compiled input → program
output → machine code

javac - Java bytecode, Java virtual machine

"java program" java is the interpreter - interpreter actuall executes the commands

★ CONCEPT - a hardware interpreter interprets the code for the CPU

PL1 - "kitchen sink language"

ALGOL-60 - First step towards sophistopicated

FORTRAN - used for scientific computations
- "the first compiled language"
LISP was developed at about the same time

FORTRAN 4, 77, ratFORTRA - translate FORTRAN to C.

Big Three - FORTRAN, COBOL, LISP

FORTRAN 90 - new one - bares hardly any resimblance to original FORTRAN

APL - an old language

Small Talk - 70's

Simula 67 - Introduced the idea of Objects - "gran-daddy of OO"

•1967 outdates our idea of types?

Zuse's - made Plankalkul - 1945
- hierarchial data types
- floating points (2s complement)
- no goto, for, continue, if (no else)
- assertions (Effel) check validity of a constraint
- wrote sophisticated programs on paper

His code:

```
  | A + 1  ⇒ A  ← array
V |          5  ← index
S |          1.n ← datatype
```

FORTRAN - Strongly based on a machine
John Backus
- FORTRAN is very efficient - not many features
  - no recursion
- FORTRAN was only supposed to run on one machine - Algol 60 introduced the idea of running of different machines
- new features added Strings, recursion (F90)

LISP - first function language
John McCarthy - MIT 1958
McCarthy was a good theoritician
LISP is big in lambda calculus

ALGOL 60

Algol 58 ideas - not implemented by 60
Algol 60 influenced many other languages.

ALGOL - Algorithmetic Language.
Based on IAL - Internation Algorithmic Language

formalized our notion of data type
identifiers
blocks
~~ex~~ infinite arrays
lower array bound (Ø vs. 1) debate
call by value, call by name
array that can be of any size
block structure~~d~~ - blocks inside blocks
allocation
inside blocks
BNF - notation system for describing
grammars. - context free grammar

Context Free Grammars - Norm Chomsky

Redd → PROLOG, C++, COBOL, PL1, BASIC, Ada

June 23, 1999

- Homework Posted Tonight
  www.cc.gatech.edu/classes/CS3411-99-summer
  notes on CS3411b-99-spring
- Algol 60 introduced BNF
- w/o formalism people have no way to say what was and was not acceptable
- formerly, anything accepted by a compiler was the syntax
- hard to describe the ~~exact~~ percise syntax of a lang
- context free grammar - almost the same thing as BNF
- Chomsky made the context free grammar - tried to do it with natural langs
- everyone uses BNF now

Def
- Syntax (form / structure)

Def
- Semantics (what does it actually mean)
- semantics of a programming lang (open problem)

```
int x;
x = 2 > 3 ? 100 : 200;
```
- valid

operational semantics - assume what someone unstands - transform something into a simpler piece of code (lower level)
- or state it as the outcome - "x gets 200"
- axiomatic and denotational semantics - use mathmatics to ~~pro~~ show semantics.
- compilers - [ANALYSIS | SYNTHESIS] → executable code
  source frontend backend

ANALYSIS - ~~co~~ Takes your code apart (error messages here)
parsing, lexical analysis, type checking

SYNTHESIS - code generation

code generation - assembly code
- a suite of compilers will often use the same backend (GNU)

Intermediate

1.) Symbol Table - all ~~of~~ of the names you used in the program - tells you about them

2.) Parse Tree -

Parse Tree uses the rules of the grammar.

(also front end) PT is a proof that it is in the language

2 * 3 → 4

→ binary operator

syntatically correct, looks right, but types are wrong

or 2 * 3.4 % 2.5 - checked in semantic analysis

2 * 3.4 @ 2.5 ⇐ checked in lexical analysis

@ is not a valid operator

parser breaks program into lexemes

3.1415 - you know its a floating point, not a number, dot, number

lexical analysis - breaks program into words

throws away everything in comments

" " spaces / tabs

lexan A - puts things into categories

lexeme — "value" IDENTIFIER ⇐ TOKEN

↑ category

PLUS-OPERATOR - the category itself determines it. No need for lexeme since there is only one word for it. lexeme is thrown away.

SCANNER / LEXER - produces an automata.

how do we express rules?? regular expressions

- three basic operations that you need: "star"
1) alternation 2) concatination 3) Kleene closure *
- there are also helper operators that you can use
- regular expressions are ~~some~~ at times hard to use
- ab concatination
  a|b
4) Parenthesis - operators have precedence:
  a* Kleene - closure
  post fix urinary operator - apply it to operator
  i++ urinary
  2 > 3 ? 100 : 200 ; tirnary
- can transfor a regex to a FSA - finite state automata
- a|b*c
  * kk
  concat
  Precedence
  a|((b*)c)
  one 'a' or multiple 'b's and a c or just a c
  ε - nothing aka a|ε
  a? post fix urinary - means "optional" - zero or one
  a+ one or more
  [ ] character class - you can put individual characters in there
  [aeiou] - matches any lowercase value
  [A-Z] - a range of capital letters
  [A-Z a-z] all letters
  [0-9] digits
  [^A-Z] - anything but a capital letter
  - . any single character
  - * any string of any length

$ - end of line ; g$ - g at end of line
^ - beginning of line
- equivalent to finite automata
- valid variable name in C
  - start with a letter
  - underscore,

digit = [0-9] NOT 0-9
digits = digit*
letter = [a-z A-Z]
letterordigit = letter | digit
variable = letter (letter | digit)*

- Input to a parser is a stream of token
- Ouput is a pase tree
- Sentence is an entire program
- Lang is a set of sentences
  main() {} — a sentence
  main() { x=1; x=1; (x=1;)x } infinite set
- to get a parse tree, a derivation is made
- parser is a lang recognizer
- context, regular grammars
- CFG - context free grammar
- notation for describing patters
- <assign> ← <variable> '=' <expression> production
  (<assign>: non terminal; '=': terminal)
- terminals appear in the actual program - tokens, lexemes
- non terminals "ideas"
  A ← V = E cap letters for nonterminals
- start symbol
  for C, <program> = <main>
  S
- regular expressions are limited

- 2 things regexp can't do
  1) nesting - any control statements
  2) $a^n b^n$ - no way to specify num of a's followed by an equivalent numbers of b's
- CFG - can do recursion
- can express a list
- (a) (a,a) (a,a,a,a)

$S \leftarrow L$ (isA)
$L \leftarrow a \mid a, L$ } infinite list of separated 'a's (right recursive)

$S \leftarrow L$
$L \leftarrow a \mid L, a$ } right recursive

a,a,a ← prove this ~~is in the language~~ described by the production

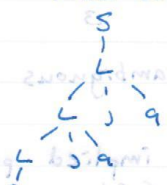
$S \Rightarrow L \Rightarrow a$

derives

$S \overset{①}{\Rightarrow} L \overset{③}{\Rightarrow} L, a \overset{③}{\Rightarrow} L, a, a \overset{②}{\Rightarrow} a, a, a$

① $S \leftarrow L$ ② $S \leftarrow a$
③ $S \leftarrow L, a$

Parse Tree



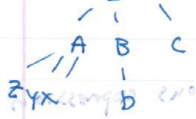
terminals - frontiers, leaves

a, a, a ← sentence (all terminals)
L, a ← sentential form
$S^* \Rightarrow a, a, a$ "S derives a,a,a"

$S \leftarrow a^{①} \mid ABC$ ②
$A \leftarrow zyx^{③} \mid zAx$ ④
$B \leftarrow b^{⑤} \mid d^{⑥}$

do a derivation

② $S \Rightarrow ABC \Rightarrow Abc \Rightarrow zyxbc$



if you always pick the left one, then you have a leftmost derivation

S ← <u>if</u> E <u>then</u> S <u>else</u> S <u>fi</u>

S ← <u>while</u> E <u>begin</u> S <u>end</u>

BNF - brackets, ::=
<expr>

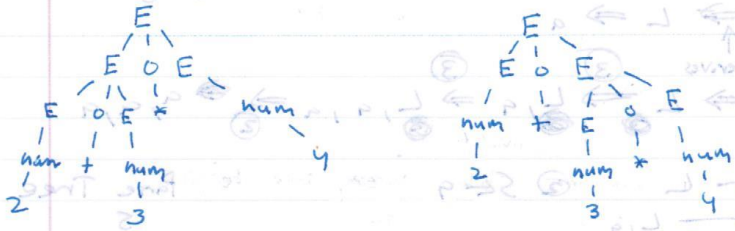
E ← E O E | (E) | <u>var</u> | <u>num</u>

lexeme → name lexeme → the actual num

O ← + | - | * | /

E

E ⇒ E O E ⇒ E O E O E ⇒ num O E O E ⇒ 2 O E O E



ambiguous grammar - two or more parse trees for a sentence

implied precedence

memorize this

Classic ETF grammar

E ← E+T | T

T ← T* F | F

F ← (E) | <u>var</u> | <u>num</u>

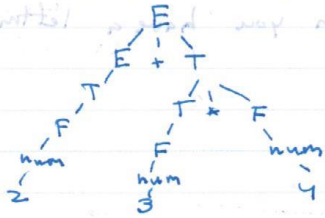
~~E ⇒ E+T ⇒ E+~~

E ⇒ E + T ⇒ T + T ⇒ F + T ⇒ num + T

2 + T ⇒ 2 + T * F ⇒ 2 + F * F ⇒

2 + num * F ⇒ 2 + 3 * F ⇒ 2 + 3 * F ⇒

2 + 3 * num ⇒ 2 + 3 * 4



unambiguous expression grammar

CS 11 Ø 1 Ø 1 Ø 1 Ø Ø 11

3411
- 2048
1363
- 1024
2339
- 256
83
- 64
19
- 16
3

2048	1024	512	256	128	64	32
1	1	Ø	1	Ø	1	Ø

16	8	4	2	1
1	Ø	Ø	1	1

June 28, 1999 CS3411

- Finish Up Chapter II
- Perl Lecture
- Make sure you put all info on homeworks

Semantics - Meaning Syntax - form
(BNF - CFG) - not much difference
context free grammar

Syntax (parser stuff)
- parser outputs a symbol table
- sentence - your program
- parser - front end
  code generation - backend

Semantics (everything else) - fuzzy line between Semantics & Syntax
- still researching on this

(approaches) attribute grammar
- operational semantics
- axiomatic semantics
- denotational semantics

You should know these four

attribute grammar
- types have to match
- function parameters
- assignment statements

must be String, not int or anything else

Java
String myString = "Hello World"

C
struct { } myStruct;
myStruct = 3; /* C complains */

Compiler figures the type of an expression, not the actual value.

- Expressions have data types

3L - cast 3 as long

int
3 + 4.5

long float

3L + 4.5f

C compiler will do an implicit cast here:

int double C compiler

3 + 4.5 $\longrightarrow$ ((double) 3) + 4.5

it does this at compile time.

```
double d;
d = 3+4.5;
```
} Okay with C

```
int i;
double d;
i = 3 + 4.5;
```
} Allowable in C - expr. is truncated
Not Allowable in Java

Type Checking involves traversing the tree - usually a "bottom up"

attribute grammar - specifying arbitrary contribution

- powerful mechanism
- invented by Knuth
- entire compiler is usu done w/ atribut grammar
- Syntax directive compilation

our grammar

S → a {code}
S → AB {code}
A → xy
B → z
B → w

} type this into YACC

called "symant. routines"

In attribute grammar:
take nonterminals + terminals, an associate
with each one, one or more variables

nonterminals:

$S \rightarrow a\,AB$

$A \rightarrow x\,y$ terminals

$B \rightarrow z$

$\rightarrow w$

associate $\boxed{\text{int value;}}$ with S

S.value

$L \rightarrow num, L$

$L \rightarrow num$

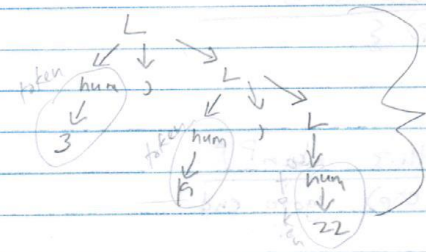
3

3, 5

3, 4, 3.159

3, 19, 22

Parse Tree



generated parse tree from our parse

- Sum the number as it's parsing

attributes L.sum (int)
num.value (int) // given

Base Case

{ L.sum = num.value } // to compute sum of one element list

Recursive Case

$L_1 \rightarrow num, L_1$

$\{ L_1.sum = num.value + L_2.sum \}$

Two Types of Attributes in AG (Attribute Grammar):

Synthesized - flows up (our example - "bottom up"

Inherited - flows down

Intrinsic attributes - the attributes types are just given to you

L.sum (int)
num.value (int) //given
{ L_1.sum = num.value }
{ L_1.sum = num.value + L_2.sum }

Operational Semantics

```
while (exp) {
  // body
}
```

what does this mean??

If this is easier to explain →

```
top: if (!exp) goto end
     //body
     goto top
end:
```

We're using Operational Semantics by explaining this in simpler terms

Mathmatical Semantics

axiomatic - preconditions / post conditions

assertions - mathmatical claims

- collection of claims

denotational Semantics - extremely mathmatizes / precise mathmatics

every construct has a mathmatical construct
everything is done on these mathmatical objects
- VERY COMPLICATED

What do we need Semantics for??

Trying to learn a new language
Trying to find a bug
→ learn by example code, manual, or mathematical proofs
- Mathematical proofs (denotational) are harder than the language itself

---

Perl is interpret
- strict reads 1 line, then does it
- Perl reads all lines, then executes chunks

- Java byte code is externalized
- Perl code is parsed an compiled each time

- Use perl 5 for this program
% perl -v // gives you version

"my" makes it local to the block - local scope
"$" at start of names -
- tells Perl something about the type

"$" - Scalar variable
int - float - boolean - etc. dynamic typing

Perl - highly efficient - processes lists and regexs quickly

Perl has a lot of predefined variable names

if (~~~) ← boolean context

"Semantic overloading" - meaning depends on context - a lot of this in Perl

Perl Visual Basic - Similar Capabilities
Python

---

modules - packages

CS 3411 June 30, 1999

- Assignment
- Chapter 4

@terms1 @terms2 @dict
%tf1 %tf2
@w1 @w2 ← parallel structure to the dictionary
- scale the raw frequency

%idf - Inverse Document Frequency
$nf1 $nf2 ← normalization

Chapter 4
Names Bindings, Type Checking, Scopes
git.cc.classes.CS3411 NOT 'A'

variables - symbolic name for an address in old times
- six attributes - name, address, value (set of bit), type, lifetime, scope

names (identifiers)

- length - BASIC - one letter, then one letter + number
- some langs have length limitations
- upper/lower - valid symbols
- case sensitivity -
- special words: key words, predefined, reserved

key word - you can't use it (unless you hack into it, bad idea)

predefined - (PAS, CA) var has a meaning, but can be overridden

INTEGER REAL (over writing predefined
REAL INTEGER variables)

reserved - Java "goto" - doesn't let one
attempt to use "goto"

binding

can appear on LHS of assignment ← L-value - something with an address in memory
R-value -

total = total + 1;
lvalue rvalue

these are different

CONSTANT - not an lvalue
FUNCTION - not an lvalue
NAME
FCTN PTR - can be an lvalue
ARRAY NAME - not an lvalue

alias - multiple names that refer to the
same address
- flexible
- pointers, references, parameter passing
unions

TEST QUESTION ☆ type - defines a set of operations and valid
range of values
- finite set of values and operations
due to range of values
- implicity specifies size and interpretation
of bits
C → sizeof(int)

value - contents, r - values

binding - association between attribute and entity
(variable and address)

- lang s and be distinguished by binding types

static $\xrightarrow{to}$ dynamic

language design time - bind symbol '+' to addition
when you decide to use that binding

lang implementation time -
int gets 4 bytes - compiler decisions

STATIC

compile time
when its compiled

link time
name to lib code - assigns address to lib call (strcat)
- lib is not bound till link time
- dynamic link - wait until you actually need it (you might not even use it)

load time
when you invoke the program - load from disk to memory
static variables don't exist on disk

DYNAMIC

run time
variables aren't bound till run time.
run time stack - where local variables live

type binding - declarations: explicit, implicit
FORTRAN (i, j, k) PERL ($, @, %)
dynamic type binding - Small Talk
- type can change throughout the program
- flexible, loss of error detection.

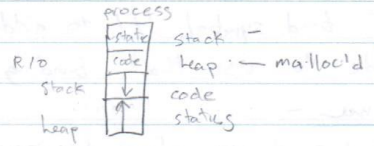
- ML - type inference - it figures out the data type for you. It tells you about strange data types - experimental feature

Storage Allocation -

lifetime - how long is storage bound to a var

- static (global, static) - entire program
- stack dynamic - local vars - done at runtime -
- explicit heap dynamic - malloc

- Three areas of memory



- implicit heap dynamic - not in C
  (Perl) list = (a, b, c, d)
  garbage collection collects leftovers

static type checking - compile time, safer
dynamic - runtime, more flexible
type error - caught by compiler

- Index out of bounds - sometimes caught at compile time int A[5]; A[6] = 5 ✓
  int A[var]; A[6] = 5 ? maybe caught

coercions - C, truncating floats when assigning to ints

strong typing

LESS ↓ MORE

Fortran, C (no type checking of parameters) old C
Pascal (variant records)
Ada, Modula-3 (possible to turn off type checking)
ML, Java

Strong typing & Static type checking is best
(at compile time)

strong typing - no undetected types.

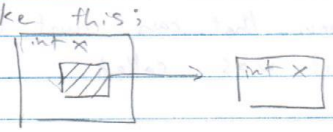
C has undefined code.
When you execute undefined code, your whole program becomes undefined

scope (visibility)

static (lexical) scoping - you can figure out where it lives.

- point where it is declared to innermost closing block.
- nested procedure declaration.
  - procedure can define scope
- nested class declarations - Java, C++
- static parent, static ancestor. -

- holes in the scope of a variable created like this;



x is "shadowed"

block scope

Algol 60

block - each block has its own space

compound statement - begin and end and list of executable statement

- Pascal does not have blocks

- keep the visibility of a variable as small as possible.

int increment(){ x++; }

C++, Java allow you to interleave var declarations.
(You can declare variables halfway through the block) - C, you have to put them at top.

dynamic scoping - the other alternative to static scoping - Present in Perl
Problems
f() calls g(): vars in f() are visible in g()
h() calls g(): vars in h() visible in g().
- depends on call sequence
- dynamic parent (caller), ancestor
- costly, hard to understand.

scope and lifetime
are related for local variables
static variables (class variable)
- write a function that can count how many times it was called.

type compatability -
do two object have the same data type

Chapter 5 July 5, 1999

- variant records aka Unions
- Perl - turnin
- Homework - hardcopy, turn in during class

- Data Types - evolved from floats and integers
- INTEGERS / REAL
- Ada - 1983 - User defined types
  - Abstract data types - beginning of classes
- Descriptor
  - thing that has information that describes something
  - all type information is dealt with at compiletime. All type information is lost at runtime
  - Java has a descriptor (to tell you the size of an array at runtime)
- Primitive data types reserve a certain number of bytes
- floating point numbers are just approximations
- never use == to a float
- IEEE has a floating point standard
- mantissa, exponent
- DECIMAL NUMBERS

RCD
| 9 | 8 |

- could calculate without roundoff ~~numbers~~ error (use with COBOL and money issues)

BOOLEAN - a bit, but usually takes a byte
C has bitfields
- true / false

ab concat
~~alt alternation~~
~~Ex Kleene-Closure~~

booleans improve readability
C - ∅ is false, everything else is true (7, 8, 9)
- all modern languages should have an explicit boolean
- C - if(x > y > z)
  return 1 BAD

Java - if (flag = true) // also BAD
should write it like this → if (flag)

Character Strings
- Should they be primitive, complex, library??
  C - an array of characters, operations are done by library calls

Random Number Generators
- library call or builtin
- what type of language is it?? Simulation languages should have RNG
- cost of adding the generator - makes the language larger
- garbage collection??
  C has a library call to do this
- gc increases reliability
- type checking with "new" in C++
  no type checking with malloc - you just get a void pointer
-

Pascal - packed arrays
Ada, FORTRAN, BASIC - somewhat primitive

Add & → Concat
C and C++ → library calls ~~Hard~~ to work with Strings
SNOBOL4 → a string manipulation language
Java → String class
Gosling wanted String class in Java
- String s = "HelloWorld";
  ↓
  String s = new String("Hello World");
  '+' is Java is overloaded
- literal strings are objects in Java
  "abc".length().
- How dynamic do you want strings to be in your language
- C / C++ limited allocation - once you get to the null terminator you can't expand anymore.

<u>CHARACTER DATATYPES</u> - EPSODIC, ASCII
↓ Shift Space ↓ 128, 256 Extended

UNICODE
↓
2 bytes, all known characters
64,000 characters

ASCII is a subset of UNICODE
UNICODE → Internationalizing language
CDC Cyber - 64 characters - only uppercase Chars

<u>ORDINAL TYPES</u> - user defined (scalar data types)
enumeration in C / C++
- names for numbers

enum { APPLE, ORANGE, KIWI } → APPLE = 0; ORANGE = 0; KIWI = 0;

You can also over ride the assignments
enum { APPLE = 100, ORANGE, KIWI }

C lets you assign bad ~~variables~~ values to enum. EX: 14 to a weekday, but 7 is max.

PASCAL - can compare ~~enums~~ CONSTANTS

ADA - const ORANGE can be used as fruit AND color.

- ORANGE is overloaded
- Ada does this: Color'ORANGE
  Fruit'ORANGE

C - unsigned int - subrange type

~~signed~~

unsigned - 1's complement only, 4 billion

signed - about 2 billion

Subranges - you might declare a new type for it

the ~~sa~~ subrange is still compatable with the base range in some cases

Ada gives two ways - derived or ~~base~~ sub type

not compatable ↔ compatable

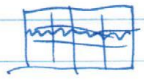
degree

°F °C - you would not want them to be added together

Arrays + Records

A + R are both collection
- A is an ordered collection - homogenous
- As are ordered - 1st position, 5th position, etc
- some langs let you set lower bound, ∅, 1, etc.
- range checking - out of bound references slows computations,

static, local, stack, implicit heap dynamic, explicit h dyn
- initialization
- you can only assign array elements in C
  - in Perl, you can assign the whole array
- array slicing



Slicing - assign this part of the array to another 1D array - better than writing for loops to do this - convinience

<u>Categories</u>

STATIC - before the program starts running

FIXED STACK DYNAMIC - local variable array

```
int f(){
    int list[100];
}
```

← CONSTANT EXPRESSION

this cannot be a variable has to be known at compile time

Perl allows arrays to grow and shrink

Java - arrays are objects
arrays must be instantiated
remembers the size: list.length

C / Perl fake multi dimensional arrays
- each element in an array can be an array

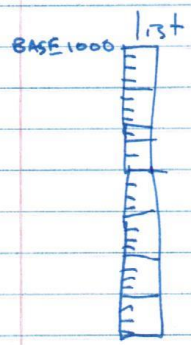
C/C++ → int stuff[] = {2, 4, 6, 8};
Ada → SCORE: array (1..14, 1..2) :=
(1 ⇒ (24, 10), 2 ⇒ (10, 7),
3 ⇒ (12, 30), others ⇒ (0, 0));

APL → the array language

list 1 = list2;
list 1 = list 2 * list 3;
you have to worry about semantic issues - length, etc;

Slicing - only if your language supports this notation
Ada - LIST (4..10)
FORTRAN 90 - MAT (1:4, 1) - the first col
MAT (2, 1:4) - the second row

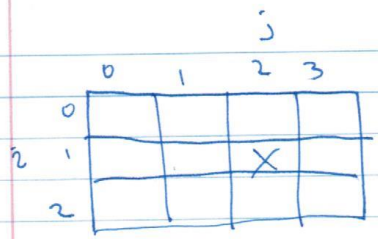
Scripting languages usually have hashes - associative arrays - Java has hashes as a library call.



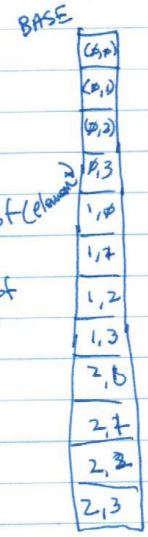
list [3]

$\underbrace{\text{BASE + sizeof (int) * 4}}_{\text{array access formula for C}}$

l-value - something you can apply & to in C



int list [3][4];



ROW MAJOR ORDER
1st row first

(only FORTRAN does it differently COLUMN MAJOR)

ROW MAJOR EQUATION

BASE + i * sizeof(ROW) + j * sizeof(element)

↓

sizeof(element) * sizeof cols

you have to know the number of columns

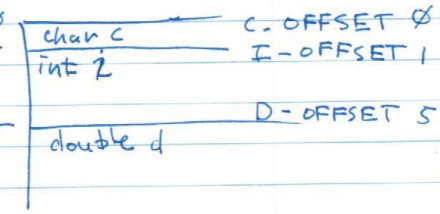
- column varies most rapidly in ROW MAJOR

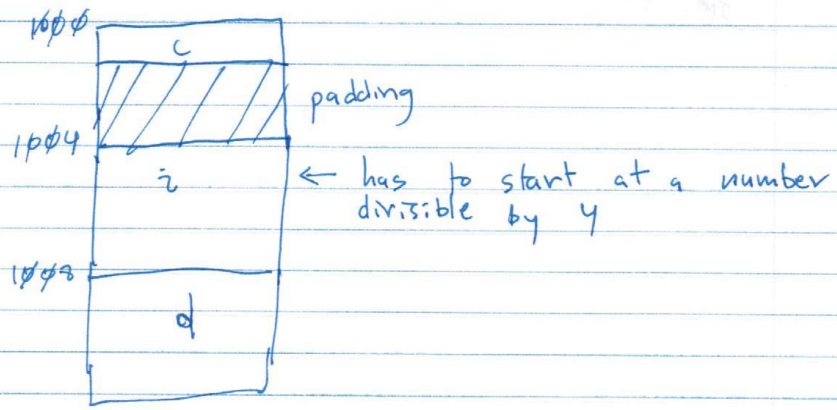
Records

Record - Standard Data Structure - heterogenous collection

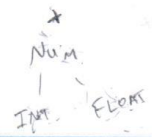
- Records have no order
- Name associated with an element is a field name
- COBOL was first to have records
- Records with Records are nested data structures
  R.f1.f2
  - . → record accessor operator
- how do you compute the address of an element

```
struct {
    char c; //1
    int i; //4
    double d; //8
}
```





- Some langs allow you to compare two structs
- Some langs allow you to use aggregate constants to initialize records


NUM
INT
FLOAT

Levi D. Smith
July 12, 1999
CS 3411

Midterm - probably everything including today
- Ch 1 - 5
- 1 ½ hour
- Rm 17 - 7:30 - 9:00

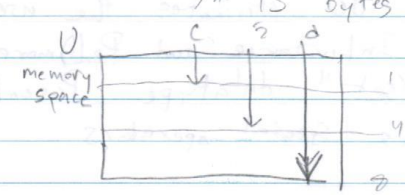
- History Question on MD
- 3 fundamental regex operation concat | ** |
- know definitions

Chapter 4
Variants - aka Unions
Pointers - Quick Overview

Union - C / C++ (variant Record)
- like a record, but the fields are disjoint
- was more important when saving data was important
- also saves time overhead (manipulating, copying, etc)

C: UNION { /* U can contain either an */
  int i; /* 4 bytes */ /* int, double, or char */
  double d; /* 8 bytes */
  char c; /* 1 byte */
} u;

/* 13 bytes without a union */



/* with union you */
/* only use 8 bytes */

u.i = 10;
u.c = z;
u.d = 3.1415926535 9;

tagged unions / tagged variants / "discriminated"
a small bit of information that keep tracks of the current value // you dont want to // print a char, etc

- Records can have variant parts.
- a field in a union can be a record.
- Unions introduced in Algol - did a runtime check to ensure safety

<u>Pascal</u>

```
type intreal =
    record tagg : Boolean of
      true : (bint : integer)
      false : (breal : real)
    end;
```

- it's up to the programmer to check the tag
- Ada requires tag to always be present
  - the correct way to do it is to have the tag automatically change whenever you assign something to the variant record
  - In Ada, you must specify the variant record type whenever you assign something to the variant record
- Subclassing in OO eliminates the use of unions — Inheritance and Polymorphism
- There is a "set" datatype - Pascal
  - Set manipulations - Special operators

Ch 5 Pointers

- pointer was just an address
- first appeared in PL I
- pointers began to have types
  - typing allows you to be able to point to that type (ex: only allows you to point at ints).
- type tells you how to interpret the bytes and how many bytes are supposed to be there
- (5) Pointers are good for linked structures
  - recursive definitions
  - dynamic
- (2) Pointers can fake "pass by reference"
- (3) Heap allocation, dynamic memory allocation.
- (4) bit level manipulations
- Java only supports (1) (2) and (3)
- in C, a pointer can point to anything stack, heap, code, etc.
- Java reference can only point to things on the heap
- Pascal, Ptrs can only point to C

Pointer operations

```
int n = 10;
int *ip;    /* read backwards */
            /* ip uses 4 bytes, but system dependent */
ip = &n;
~~ip = malloc(10);~~   /* can't do anything with void * pointers */
            /* also can't dereference void *ptrs */
ip = (int *)malloc(4);
*ip = 10;
```

3 Deadly Pointer Sins

① dereferencing NULL or INVALID pointer

```
ip = NULL;
*ip = 10;

ip = (int *) 10003; /* alignment error */
*ip = 10;
```

② memory leak

```
C - free();
C++ - delete();
```

③ dangling pointer

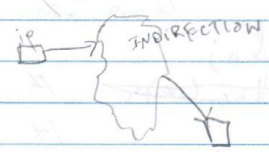
```
ip = (int*) malloc(4);   /* returning a pointer */
*ip = 10;                /* that has been freed */
free (ip);
*ip = 20;                /* this is BAD */
```

Prevention

1. Detection
   - Java, Null Pointer Exception
2. Dangling
   Taken care by garbage collection

Tombstones

Lock and key



takes care of some pointer problem
but it costs you space

C++ - References - limited
Java - Object References
- Java uses mark and sweep
- no dereferencing operator

In C++ you must do this
(*t).doSomething;

Java → t1 = t2; //changing the pointer

C++ Reference introduced
Strostrup hated pointer parameter passing.

```
C - f(int a, int b, int* min, int* max)
        if (a>b) {                          f(3, 15, &myMin, &myMax)
        *min = b; *max = a;
        } else {
        }
```

```
C++ - f(int a, int b, int & min, int &max)
    /* called like this:                    int myMin, myMax;
        f(3, 15, myMin, myMax) */

    min = max; // altering what min points to
               // you can't alter C++ references
               // you can only initalize references
               // in C++.
```

~~Lecture~~ Chapter 6

Aritmetic Expression

Operand order Evaluation

f() + g() * h() // what order do you
// call the functions.
// you don't know - it's
// undefined.

- does matter which order they're evaluated unless they have side effect
  - alter global variables
  - change parameters
  - print
- Users can overload their own operators in C++,
- Mixed mode expression.
  3 + 4.5 // Not allowed in Ada

operations of different data types
// converts 3 to a float

Ternary x = (a > b) ? 3 : 4;

- Parens are at the top of precedence list.

- APL many operations - mathmatical long
- SmallTalk evaluates from right to left

~~Operator precedence~~

x = a * b + c * d;

language doesn't specify anything about which one occurs first

$a * b * c * d$ ← when they're all at the same level, then you go left to right

$a + b + c + d$

associativity - mathmatic and computational

Max + Min + Max + Min

- Short Circuit Evaluation → precise operand evaluation rule

$f(g(), h(), i())$ → evaluated left to right in Java

→ no specification about this in C or C++

7:50

C++ Project tomorrow

July 14, 1999

Leftovers from Ch 6

Associativity - left to right

Non-Associativity - Ada exponential

Associativity - two operator

a ** b ** c → have to use parens to avoid compiler error

- Short Circuit Evaluation

usually property of boolean operators

Pascal does not have short circuit evaluation.

- sideeffects

if (a>=0 && list[a] == key

- if (e1 and e2) // no short circuits Ada

if (e1 and then e2) // short circuit Ada

or / or else

C / C++ & → bitwise AND

- sideeffects

changing global or out params functional - if (f()) * if f() returns something else then SE

refferencial transperancy has no sideeffects / increments

expressional -

referential transperancy - you get the same value from a function every time.

```
int i = 10;
i = i++;
printf("%d", i);
```

UNDefined - lang (C) doesn't specify which assignment comes first

sequence points - can't modify a value more than once.

~~type~~ conversion

explicit

implicit (coercions)

void f (int i) { ... } f(3.14) parameter passing

widening + narrowing

4 byte (int) — you can lose significant
2 byte (short) bits.

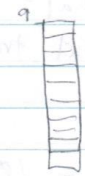
implicits conversions are generally widening

widening → promotions

float 4 bytes
int 4 bytes lose bits

Pointer Arithmatic → Defined For Arrays

```
int a[10];
int * p;
p = & a[10];
```




```
p = a
p + 2;
```

1000
+ 2 * sizeof(int)
1008

```
*(p+2) = 10;
```

```
a[2] = *(a+2)
```

C++

Bjarne Stroustrup

C with Classes

- Standardization for C++ took 10 years
- multi paradigm language
- C prototypes came from C++
- C is a subset of C++
  C programs can be type checked with C++ compilers
- tries to avoid preprocessor

Reliability

const int x = 4;

initializers can be functions

C++

```
int * a;
a = new int [20];
a[19] = 10;
delete[] a;
```

You can redefine new and delete to implement garbage collection.

bool data type

bool b = true;

C++ also allows if(7) instead of a boolean to allow backwards compatability

Strings are truly objects

```
String s = "Hello";
String t = "World";
s += t;
```

```
String s = "hello" + "world";   //this does
                                //pointer aritmatic
```

char* char*

C++ Standard library

strings

Standard Template ← (generic) component classes (lists / hashs) (generic algorithms)

I/O stream

Reliability References

clean call by reference

```
int a = 10, b = 20;
swap(a, b);
```

```
void swap (int &x, int &y) {
    int tmp = x;
    x = y;
    y = tmp;
}
```

```
int a = 16;
int &b = a;
b++;
print a
```

function template - ability to write generic code

```
int max(int a, int b){
    return a > b ? a : b;
}
double max( double a, double b) {
    return a > b ? a : b;
}
```

- you can do function overloading in C++
- Same function

```
print (max( 3.14 , 5.2)) // finds the correct function
```

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

```
max(4.5, 3.2)
```

- may be semantic restrict - the code must support the types that you send int.

---

class declaration and definition

```
struct Foo {
    int x;
    char y;
};
```

```
class Foo {
    int x;
    char y;
}
```

} member
// data
// function

// C++ is about the same thing
// as a C++ class
// C++ structs are more flexible
// that C structs

const means that the code in a function will not change the state of the object.

stack allocated → Point p(2, 3); if this is in a block then it is local

heap allocated → Point * pp = new Point(2, 3);

print(p.x());

print((*p).x());

—OR—

print(pp → x());

No polymorphism in stack allocated Objects

const Point ORIGIN(0, 0);
ORIGIN.translate(2,3); // you can't do this

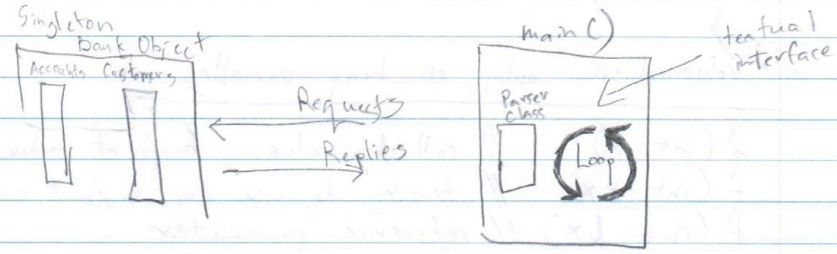
"cast away constness"
- static hashtable → const
  - what if you want to keep track of how many times something was accessed

Put definitions in a header file (*.h)
Put code in another file (*.cc)

- if you define something inline, ~~then it~~ then the compiler tries to optimize it.
- it is only a hint - the compiler doesn't have to do it.

Levi D. Smith
July 19, 1999
CS3411

P2 - Object Oriented Banking



Bank, Customer, Account } Must have these

Parser, Request, Reply } You could possibly have these

Wrap commands into a request object.
Must run on g++ GNU compiler.
C++ has no standard graphical library.
Don't use ACME machines

- a class can see private ~~methods~~ function / variable of other instances of the class.
- Accessor / Mutator.
  ↓
  use const

```
const Point p(1,2);
p.translate(3,4); // compiler error

int x = 10;
int x(10); // the same thing

x = 10; // assignment
int x = 10; // initialization
```

```
Point (int x, int y) {
    xx = x;
    yy = y;
}
```

- Polymorphism only on heap variables

```
f (int x)      // call by value - copy of value
f (int *x)     // trying to use an in/out param
f (int &x)     // reference parameter

f ( const String & s)
```

objects can be **BIG** - if you copy by value, then you have to copy the whole thing. If you make it const, then it won't be copied (const ref parameters)

User defined overloading

- small number of operators you can overload

```
Point operator+ (const Point &p1, const Point &p2) {      // C++ user
                                                          // operator overl
                                                          // loading
    return Point ((p1.x() + p2.x()), (p1.y() + p2.y()));
}
```

temporary object (anonymous object)

global function

```
Point &   operator += (   const Point & p2) {
        xx += p2.xx;
        yy += p2.yy;
        return *this;
}
// you can't overload operator precedence
// you can also do function overloading

# include <iostream>
int x;
cin >> x;          // stream extraction operator
cout << "The value of x is" << x << endl; //print it
                   // stream insertion operator

while (cin >> x)   // return false when fails
      cout << x;

String s;
while (cin >> s)   // skips whitespace
      cout << s;

cout               //knows how to print all types
(((cout << 123) << 456) << 789);

cout returns the modified stream
endl  // IO manipulator // newline + flush

cout setw(8) << x;
```

toString in C++:

```
Thing t;
cout << t;

outstream & operator << ( o stream & o, Point p){
  o << "(" << p.x() << "," << p.y() << ")";
  return o;
}
```

Class Templates

```
template <class T>
class Foo {
  int size;
  T list[n];
public:
  void add (T n){
    list[size++] = n;
  }

Foo <String> myList;
```

```
Vector <Customer> customers;

class Derived          // class Derived
: public Base          // extends Base
{ ... };
```

.h file

```
class Shape {
  virtual float area() const; //inline
}
```

use virtual for polymorphic functions

Polymorphism 3 things

① Parent / Child relationship

② Overriding - same function, same name, same parameters

③ upcasted child - Parent Object
Child c;
p = c;

④ Pointers (for C++); Parent *p;
Child *c = new Child();
p = c; //upcasting

p → f(); // child's f gets invoked

Polymorphism - dynamic binding of functions at runtime

- A pure virtual function.
virtual float area() const = ∅; // no definition
//defferred
// all children must implement this function.

- a class that has only pure functions is an abstract class.
- v, pv, abc → need to know these

- friend function -
friend f - function f can see all data
matrix class - you don't want to call a function everytime you access a piece of data

- multiple inheritance

```
class Foo : public Base1, public Base2 {
```

Foo.h

```
# ifndef FOO_H
# define FOO_H
    class Foo {
        }
}
# end if
```

// a header file included
// more than once

// the next time it is included
// it just sees that FOO_H is
// already defined

Four new cast operators

- you can identify the type of cast

```
static_cast<T>(x)          // int to float
dynamic_cast<T>(x)         // upcast, downcast
                           // during runtime
reinterpret_cast<T>(x)     // *char to int
const_cast<T>(x)           // casting away
                           // constness
```

CS 3411

July 21, 1999

- Ch 7 Control Structures
- C++

main.cc

```
int main(){
    Bank
    return 0;
}
```

#include "Bank.h" ← ABC order

Foo.h

Foo.cc

```
#include "Foo.h"
```

~~Foo~~ .h

Template (by itself)

```
g++ *.cc -o oob
```

Makefile

dependencies - only compile things when they need to be compiled

<u>Exceptions</u> - not required for our program

STL - String class is a template

~~(instantiation~~

Vector - dynamic array

```
vector<int> v;      //stack
vector<int> *v = new ? //heap
v.push_back(3);
v.push_back(4);
cout << v[0];  // overloaded brackets
```

iterators

:: → Name Space Resolution operator

```
                                    //nested class
for(vector<int>::iterator i = v.begin(); i != v.end(); i++)
    cout << *i;
```

Syntax Gotcha

Vector of Vectors of ints

```
vector< vector<int> > v;
```

space

>> is error

- 60's - GOTOs

people started inventing structured control flow statements

- first CS were machine statements like terinary operator

Bohm - Jacopin Result

All you need is:

- sequencing
- if statement
- while statement

3 or 4 loops for most programming languages

Categorize them by IF STATEMENT / LOOPS

IF	LOOP
conditional	iteration
selection	repition
	(recursion)

July 21, 1999

IF

1 WAY - if

2 WAY - if / else

~~3~~ WAY - switch / case or

```
if ()
  else if ()
  else if ()
  else if ()
```

FORTRAN →

```
if (x <= ) goto end
  s1;
  s2;
end
```

dangling else

```
if ... then
  if ... then
  else
```

← who does this goto??

C, Pascal : else goes to the nearest unmatched if

Ada, Modula 2, FORTRAN 77 : have to have an ending keyword for every if.

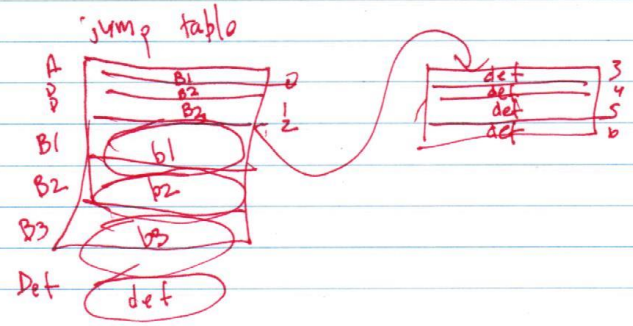
Algol W - came between 60 and 58

W invented case statement (and Quicksort)

```
switch (e) {
  case 1:
  case 2:
  case 3:
    s1;
    s2; break;
  case 4:
  case 5;
    s3; s4; break;
```

constant expression

- switch statement goes directly to the correct choice if the correct choice is available.



LOOPING

counting loops (for loops)

what type can the counter be??

C

```
for (i=1; i<= 10; i++) {

}
/* i == 11 */
```

```
for i = a+b to c*d do
    c = 10                    /* what will this do*/
```

for i = 5, 3, 19, 22 Algol 60

```
for( ; ;)   infinite loop
for (i=1, j=10;          ; i++, j--)
```

, → forces left to right evaluation

value of expression is the value of rightmost expression

matrix [i, j] ← accepted by compiler (finds j)
matrix [i] [j] ← the correct way

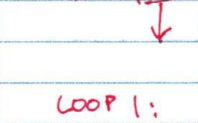
while (~) ← continuation
  S1;
  Ø or more

do
  S1;
  while (~) ← continuation
  1 or more

Pascal: repeat
  S1;
  S2;
  until (~) ← termination

User Defined Loop

```
while ( ) {
  if ( ) break;   /* termination */
}
```




```
LOOP1:
    LOOP2:
        LOOP3:  while (true)
                  if (weird error) break LOOP   //named //break
```

foreach $item @list → Perl: no iteration variable
don't have to worry
about array size

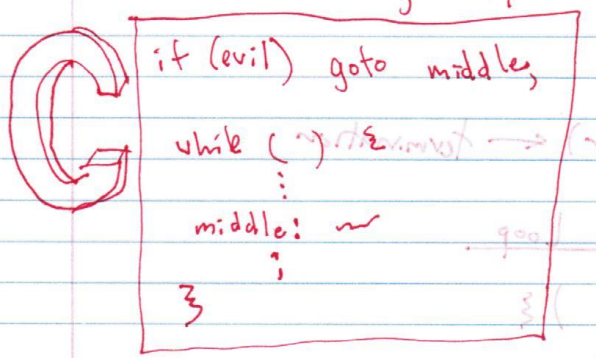
Arguments for GOTOS

- you can add extra variables to test in conditional part of loop

break statements can't go up

GOTOS can take you up

GOTOS can go anywhere



PL/I

```
var Label;
var = LABL2;
  goto var;

LABEL1:

LABEL2:
```

Levi D. Smith
July 26, 1999
CS 3411

7/26/99
M Ch 8 + Ch 9
W Prolog — P3 (Prolog) out
M Prolog — P2 due
W Exception / FP — HW3 due

Final 2:50 pm Tuesday

Subprograms

- Fundamental part of procedural programming
- Procedure is a form of abstraction (3rd fundament form of abstraction)
- Parameter passing methods is a big part of this chapter
- call by value, call by reference.
- subprogram has only a single entrance
- procedure is a helper
- "parameter transmission" - copying / referencing values to a method
- "activation record" aka stack frame
- C declaration (signature prototype) - puts the definition in a symbol table
- separated declaration
- Pascal has "forward"
- function prototypes improves compiler efficiency
- Java uses an extra pass through the source code once more to find all of the declarations

2 things: declaration • invocation • definition

- formal vs. actual parameters

```
int f(char c);
}
int f (char c) {
    printf ("%ch ", c);
}
```

```
f ('Z');
```

/* c = 'z' */

. Stroustroup - C++ has initialization semantics

f ('B' + ('a' - 'A')) // to lower

↑ actual parameter

int f (char a) {

↑ formal parameter

f (10, 'z', 3.4)

f (int i, double d)

Ada

you can mix those up

f(d ⇒ 3.4, i ⇒ 10, c ⇒ 'z')

downside, you have to remember the names of parameters

default parameters

C++

f (int i, double d = 3.14) {

f (10, 'z', 5.2);

or

f (10, 'z');

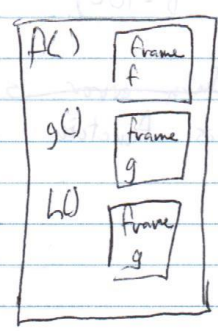
f(int a = 16, int b, int c = 26)
f(42, 43); //which two??

~~Dflt Pa Dflt~~ No
~~Dflt P~~
~~Pa Dflt Dflt~~ Yes

No way to pass just a and c

- old C and FORTRAN did not do parameter type checking
- you can turn parameter type checking off in C++
- some languages let you pass subprograms as parameters
- separate vs. independent compilation.
- independent compilation - doesn't look at the whole program (all files) when it compiles - only looks at the part that needs to be compiled
- Ada was first to have separate compilation (java has it too)
- runtime stack is there to support recursion
- activation record aka stack frame - keeps track of where we are in the program before we start the subprogram
  1. program counter + registers (where you are)
  2. local variables ~~+ constants~~
  3. parameters

old FORTRAN



no need for a runtime stack if you dont have recursion

these frames are static - they live throughout the whole program

There is a separate area of memory for statics

in, out, in/out - higher conceptual terms

implementation techniques

~~pass~~ ~~by~~

pass/call by - value ~~(in)~~
- reference
- result
- value/result
- name

- call ~~by~~ by value has a cost - large objects have to be copied entirely to the stack
- call by result (like an out parameter)

```
int x = 0;
    f(x, x)        // aliases
void f(int a, int b) {   // pass by result
    a = 5;
    b = 100;
}
// return error is undefined
// this function call is undefined
```

- pass by reference
  you passing an access path (pointer)
- A reference parameter is an alias

- pass by reference is more efficient
- access is slower

Pascal

f(a: integer; var b: integer){ —— };
↖ (in/out)

f (int * p) // pointer is copied to stack
f (int * *p) // if you want to modify the pointer
(no pointer copy)

<u>Algol 60 - Pass by Name</u>

MACRO - processing

int x=10, y=100;
f (x + y) // call it with an expression

void f(int z) {
~~z = 100 * z + 7;~~ // textual substitution
}

int temp;
temp z * 100 + ~~x~~ z / 3
// x+ y * 100 + ~~x~~ + y /3
}

~~Implement~~
Impossible to Implement Swap with pass by name

thunk - book description

- delayed or lazy binding
- call by reference - copy a pointer to the stack instead of a value

C

```
print (int matrix[10][20]) {
    for (i = 0; i < 10; i++)
        for (j = 0; j < 20; j++)
            printf(m[i][j]);
}
void print (int m[][20], int nrows)
```

<u>can't do this</u> `void print (int m[][], int nrows, int ncols)`

```
print (int *p, int irows, int icols)
    //you have to compute the
    //access formula yourself here
```

C has a function pointer

func(sqrt) actual

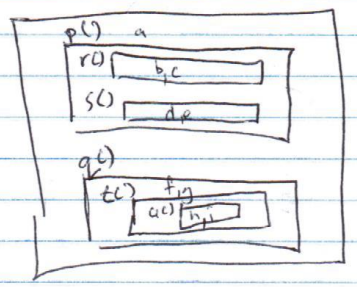
```
func( formal f ) {
    f(x); //like an alias in the
          //function
}
```

July 26, 1999

plot (data f)
map (list, fn) //LISP apply fn to every //element of the array

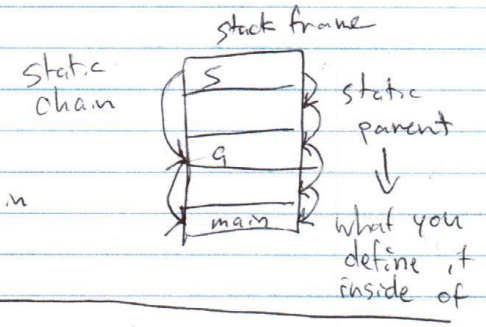
Chapter 9

Accessing non-local variables
PASCAL
nested procedure definitions



Algol
Ada
Modula
Pascal

s() - you can see d, e, a, and globals
a is a non-local



static link - another field in the stack

---

nest class definitions - C / Java

Levi D. Smith
July 28, 1999
CS3411

PROLOG

- declarative programming language - Prolog defines this others (imperative, OO, functional).
- cleaner than imperative (avoid side effects)
- in declarative, say what you want, not how to do it.
- computes the "answer" (solution) by formulating a "proof" - declarative
- ~~we lose~~ pure functionality may have to be sacrificed for efficiency.
- Applications: 1) databases, 2) AI (deductive) (expert systems, theorem provers) 3) Security (access control)

LOGIC

Symbolic Logic

Symbol → basic data type (apple, joe, wine) $\xrightarrow{\text{corresponds}}$ object

Propositions - estabish relationship between symbols

Predicate Calculus

man (jake)
like (bob, steak)

← tuples

"arity" (how many)

functor

symbols / variables

meaning (semantics)
up to you - extrinsic

- logical operators (like primitives)

$\neg$ NOT

$\cap$ AND / conjuction

$\cup$ OR / disjunction

$\equiv$ equals

$\supset$ implication $\quad a \supset b$ "implies"

if a then b

($\subset$ also implication $\quad b \subset a$

b follows a, b is implied by a, b if a

this is how Prolog likes it)

$a \cap \neg b \supset d$

<u>Quantifiers</u>

$\forall$ - "for all" (Universal Quantification)

$\exists$ - "there exists" (Existential Quantifiers)

$\forall x P$

$\exists x P$

cap letters represents variable

$\forall X$ (woman (X) $\supset$ human (X))

$\exists X$ (mother (mary, X) $\cap$ male (X))

Simplification $\rightarrow$ CLAUSAL FORM

"or" "if"

one of these are true if $\rightarrow$ $B_1 \cup B_2 \cup B_3 \cup \cdots \cup B_n \subset$

all of these are true. $\rightarrow$ $A_1 \cap A_2 \cap \cdots \cap A_m$

"and"

• consequence — antecedent (before)

$B_1 \cup B_2 \cup \cdots \cup B_n \subset A_1 \cap A_2 \cap \cdots \cap A_m$

likes (bob, trout) $\subset$ likes (bob, fish) $\cap$ fish (trout)

HORN CLAUSES

only one thing on LHS

$B_1 \subset A_1 \cap A_2$

$B_2 \subset A_3 \cap A_4 \cap A_5$

$B_3 \subset A_1 \cap A_3 \cap A_6$

head

if a rule has two heads, then there are two different ways to prove truth (aka OR)

PROLOG PROGRAM

A RULE: likes (bob, trout) :- likes (bob, fish), fish (trout).

Automated Proof Procedure (Algorithm)

RESOLUTION (the APP that Prolog uses)

$Q_1 \subset T$ ← can include variables

$T \subset P_2$

$? \; Q_1 \subset P_2$ "transitivity of implication"

$T_1 \cap T_2 \cap T_3 \cap T_4$

Unification – making two thing look the same

"structural equivalence"

man (jake)	man (bob)	NOT structural equivalent
man (jake)	man (jake)	YES " "
man (X)	man (jake)	maybe, depending on X

X = jake
"bindings of variables"

instanciation

- variables are more like variables in mathmatics, not like in regular programming languages.

headless Horn clause: fish(Ized)

file: db - contains facts and rules

```
female (shelly).
male (bill).
female (mary).
male (jake).
father (bill, jake).
father (bill, shelly).
mother (mary, shelly).
:
```

fish(X) - everything is a fish

```
$ prolog
?- [db].                    # consulting the database
?- yes.
?- male (jake).             # query (a question)
?- yes.                     # if not in database, then it
?- male (X, Y)              # will say no
?- no.                      # arity is also a part of the query
?- male (X)
    X = bill ;              # keep looking
    X = jake ;              # "back tracking"
    no.
?-
```

Query Mode

```
?- male(jake), female(shelly).        # compound query
?- male(X), female(Y).
        X = bill
        Y = shelly;
        X = bill
        Y = mary;
        X = jake
        Y = shelly;
        X = jake
        Y = mary;
?
        no.
?- male(X), female(Y), X = Y.
                       # unification.
```

db

```
parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).        } OR
grandparent(X, Z) :-
    parent(X, Y), parent(Y, Z).
sibling(X, Y) :-
        mother(M, X),
        mother(M, Y),
        father(F, X),
        father(F, Y).
```

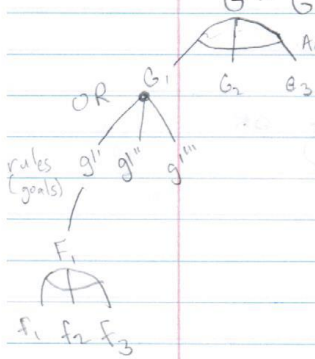
?- sibling (bill, mary). # satisfy the goal
sub goal mother (M, bill).

unify - satisfy a test (subgoal)

---

likes (X, Y)
likes (jake, Y)
likes (X, wine)
likes (jake, wine)

$G = G_1, G_2, G_3$



Proof Tree
AND / OR tree

depth - first search of Proof Tree

- it is possible to have infinite tree branches

"occurs check"

Levi D. Smith
August 21, 1999
CS 3411

¡Yo quiero dormir!

- Translation of symbolic object into predicate Calculus
- Resolution proof technique
- Unification - Structural Match
- Numbers are terms in Prolog
- operator "is"

X is Y + Z

unbound bound → this must be true, else it will fail

X is X + 1 → can't do this single assignment

X = (3*5) + Z

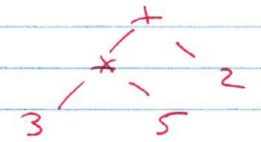
unification

3 + 5 = 8 → fails

3+5 → data structure → + (3, 5)
+ is the "functor"

"is" → actually evaluate the structure

+ (* (3, 5) , Z) → this is just a struct



~~book~~

book (moby-dick, author (herman, melville))

X = book (moby-dick, author (herman, melville))

bind ~~the~~ the value to X

[a , b] ⟹ list

[a] ⟹ list

[X, b] ⟹ list

[[a,b], [ ]] ⟹ list containing a list

first (X, L) :-

IN IN

SPLIT OPERATOR

L = [X | Y]

first (head) car

rest (tail) cdr

rule for first:

first (X, L) :-
L = [X | Y].

- OR -

first (X, [X | _]).

?- first (3, [3, 4, 5]).

_ (underscore) anonymous variable
you do need to name it

prepend (X, Y, [X | Y])
bound bound unbound

prepend (3, [4,5], L)
L = [3, 4, 5]

[Z, [a]] = [[a, b], [4,5]]
does this unify
:
Z = [x, b]

Sample Programs

member (X, [X | Xs]).
member (X, [Y | Ys]) :- member(X, Ys)

member (3, []) /* split operator will fail

member (3, 4) /* split operator will not split with just the element

prefix
- the empty list is a prefix of everything

membership
member2 (X, Xs) :- sublist([X,], Xs).

~~July~~ Aug 2 1999

append - two rules
range - sub string
* sum list - be sure to check this one
~~out~~

[a, b], [c, d]

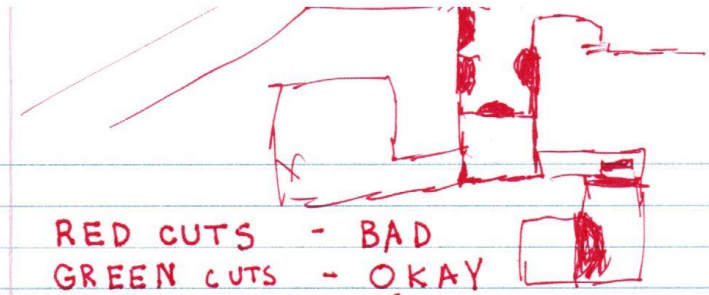
[a, |Zs]
[b + [c, d]]

pure knowlege doesn't not have precedence;
you chan change order of operations
! ~~goto~~ CUT - like goto or break:
always succeeds
dis allows backtracking
g :- a, b !, c, d.
X Y
and are now bound

- never backtrack - accross a cut
- lets you avoid redunant caltion

member(E, [E|_])
member(E, [_|L]) = member(E, L);
set membership

if the first rule succeeds, then
don't do second one - use a CUT
1.) member(E, ([E|_]) :- !.
2) member(E, [_|L] :- member(E, L).



RED CUTS - BAD
GREEN CUTS - OKAY

closed world a Sumption

negation
"failure as negation"
"I can't prove it"

CWA - It can only answer things that it knows about (what you tell it)

not (father (chelsea, bill))
- not operator works as long as CWA holds true

not (not (p(X))) ⟹ X comes out unbound

X = _12345 ⟹ displays an anonymous binding

---

6 rules to implement

some tree should be some tree

Tree is represented as a structure
tree(3, L, R)

tree (a, void, void)
the simpliest tree - void is an empty tree

tree (a, tree (b, void, void), void)

SESSION

> prolog
?- [tree].
?- tree 2 (A), // bind tree 2 to A
tree-element (e, A)

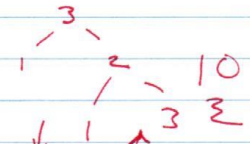
preorder rule - elements in tree as pre order - we have to modify it to do in order

✓ inorder (T, In)

✓✓ substitute (X, Y, TX, TY)

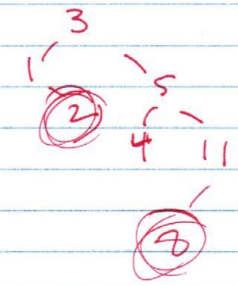
find all occurances of X in TX and replace them with Y
(make another tree from scratch)

✓ sum-tree

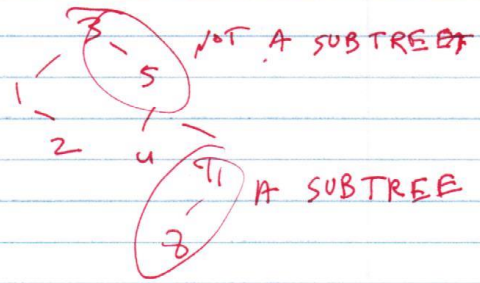


✓✓✓ tree-insert (X, Tree, Tree1)

BST



EXC
Subtree (S, T)
Ordered (T, T)




```
?- trace(member/2).
?- member(3,[3,4]).
```

Exception Handling

FINAL - T - HERE - 2:50 - Everything tested

C - UNIX - everything returns an error code

Can differentiate errors by numbers

```
void f() {
  result = fopen()
      if res < 0 {
        /* error stuff */
      }
  n = printf()
      if (n < 0) {
      }
```

Java Exception

```
try {                          • write normal code without
  : normal code                  error checking
} catch (BoundsException e) {
  : error case
} catch (FileNotFoundException e) {
  :
}
```

C++, Java, and Ada are ~~most~~ the major exception users

PL/I used exceptions, but didn't do a good job implementing it.

FORTRAN

READ (UNIT=5, FMT=1000, ERR=100, END=99) WHF

(error_routine)

(function pointer)

- can't access the context of the state
- ~~exceptions~~ error routine may repair, but do you don't know if an error has occurred
- Exception propagation. Error can be propagated up to the calling program. You can fix it there, and not have to fix it multiple times

- can you disable exceptions??

PL/I - First Ex H lang

ON condition [SNAP]

BEGIN;

...

END;

- You had to execute the exception handler to get exceptions in PL/I

- Difficult to write and read
- Ada - static exception declaration (like in C++ or Java)

- propagates upward until it can find something that can handle the exception

main() ← finally looks here, then error
g() ← then looks here

f() ← looks here

Java: can pass one exception handler type

JAVA

Throwable

Exception — Error → internal processing error
can't be handled (or continue)

user defined
↳ class myException extends Exception

Exceptions - two types
CHECKED — UNCHECKED
(Runtime exception)

Checked Exceptions
① handle it
② propagate it

```
try {
  String s = stdin.readline();
} catch (IOException e) {
};
```

-OR-

```
void f() throws IOException {
    String s = stdin.readline();
}
```

```
try {
} catch ( ) {
} catch ( ) {
} finally {
    //clean up
}
```

before propagates up or continues

```
try {
} catch(Exception e) {
    s1, s2, s3
    throw = new Exception(e);
}
```

rethrowing an exception
local eh, then global eh

- makes you think that errors are a normal part of coding

Functional, Imperative, Declarative, OO

NO SIDE EFFECTS - NO ERROR - CODE PURITY - TIED TO MATHEMATICAL REASONING - MATHEMATICAL SEMANTICS - CORRECTNESS - LISP WAS FIRST FUNCTIONAL PROGRAMMING LANGUAGE - LISP WAS #2 OR #3 AFTER FORTRAN - LISP HAD 7 BASIC FUNCTIONS - MCCARTHY ADDED ADDITIONAL FEATURES TO LISP SINCE PURE FUNCTIONAL PROGRAMMING WAS NOT IMPORTANT -
VON - NEUMAN ACHITECTURE - NO ASSIGNMENTS
LAMDA EXPRESSIONS

Higher Order Functions - aka Functional Forms

composition -
operator takes two function
Construction
apply operation to each element in the list
Apply-to-All aka Map

LISP - everything is a list
atoms + lists
nested lists too

(A B C) → programs and data are the same

LISP

SCHEME

- DEVELOPED AT MIT - IMPROVED LISP
- Modern, Small, Lexically Scoped LISP

COMMON LISP

'(A B C) → use literally
quote (A B C) → same thing

CAR, CDR, CONS → fundamental LIST operations

(CONS 'A '(B C)) ⟹ (A B C) *NEW* LIST (makes it functional)

(CONS (+32) '(B C)) ⟹ (5, B C)
(CONS '(+32) '(B C)) ⟹ ((+32) B C)

You can define names for function names

(IF (e) (s1) (s2)) special form in LISP
it doesn't evaluate all of the functions

```
DEFINE (adder list)
  (COND
  (NULL? list) 0)
  (ELSE EVAL (CONS * list)
))
```

COMMON LISP

ML

- type inferencing - tries to figure out the type for you
- fun cube(x: int) = x * x * x //like Pascal

HASKELL

- purely functional
- lazy evaluation

```
f (a b c)
if (a == 1)
    return b + 10
else
    return c + 10
```

positives = [0 ..]

Homework 1 - Sebesta 1-4, PERL

CS 3411 - Programming Language Concepts - Summer '99 (Hutto)

Due: 4pm (classtime) Monday 12 July

READ ME! Please turn in a legible hardcopy solution at the **beginning** of class. Staple, but do not fold, multiple pages. Print "HOMEWORK 1" and your name and GT number clearly on the front page. We will deduct 5 points if you do not follow these instructions!

LATE POLICY: Assignments may be turned in up to 3 school days late with a 10% penalty deducted for each day late (no deductions are exacted for holidays or weekends). Submissions later than 3 school days will not be accepted or graded under normal circumstances.

PROBLEMS vs. QUESTIONS: Your textbook has two sets of exercises at the end of each chapter titled "Review Questions" and "Problem Set". I will say "Question" when I mean "Review Questions" and "Problem" when I mean "Problem Set". I will also include the page number to help disambiguate but several students have done the wrong exercise in past quarters so beware!

Got Questions? Try office hours, email or the class newsgroup!

---

1) In your own words, what is the difference between an **interpreter** and a **compiler**?

2) Briefly **compare C and Smalltalk** in terms of the Big Four language design criteria: readability, writability, reliability and cost.

3) **Contrast Algol-60 and Algol-68** (how do they differ?).

Chapter 3: Describing Syntax and Semantics

4) Develop a **regular expression** matching strings expressing the **time of day** like:

```
9:17 am
12:30 PM
4 am
```

Notice that the "am/pm" part can be in either upper or lower case and must be preceeded by a single space. Your expression should disallow things like:

```
32:15 pm
9:99 am
```

You may find it useful to introduce **named regular expressions** like:

```
digit = 0|1|2|3|4|5|6|7|8|9
digits = digit+
```

5) Add synthesized attributes and semantic routines to the classic ETF grammar below (make it an **attribute grammar**) to **evaluate the expression** being parsed. Add an attribute named "value" to the expression non-terminal (E.value) that contains the value of the expression when the parse completes. You may assume that the lexer provides the value of an identifier (I) or number (N) as an intrinsic attribute (I.value, N.value).

```
E -> E + T
  -> E - T
  -> T
T -> T * F
  -> T / F
  -> F
F -> ( E )
  -> I
  -> N
```

Expr ::= Expr '+' Term | Expr '-' Term | Term

Term ::= Term '*' Factor | Term '/' Factor | Factor

Factor ::= '(' Expr ')' | Identifier | Num

6) Problem 5, page 152

7) Problem 7, page 152

8) Problem 8, page 152

Chapter 4: Names, Bindings, Type Checking, Scopes

9) Consider the increment operator (++) in languages like C and Java. What type of operand does ++ require, an **l-value**? an **r-value**? both? What type of result does ++ yield?

10) What is **dynamic scoping**? **dynamic binding**? **dynamic type-checking**?

11) Your text describes **four categories of scalar variables** based on the lifetime of the variable: static, stack-dynamic, explicit heap-dynamic, implicit heap-dynamic. Which categories does C support? Give an example of each category of variable in C, if possible.

12) C++ and Java allow variables to be declared and initialized intermixed with executable statements. Initializers can also be arbitrary expressions instead of "constant expressions". For example the following is a valid C++ program but not a valid C program:

```
int x = 10;
x++;
int y = f(x);     // syntax error in C! OK in C++ or Java
y++;
```

Most other languages only allow declarations at the beginning of blocks or procedures. **Discuss the advantages and disadvantages of intermixing declarations** with executable statements.

109/120

Levi D. Smith
gte187k
July 12, 1999
CS3411
HOMEWORK 1

C

1) Interpreters act as a virtual machine for a language. The input to an interpreter is the source program, then sometimes the interpreter compiles the program to an intermediate code, and then the interpreter reads and processes the intermediate code (or the source program if no intermediate code was generated) and outputs the output of the program.

Compilers lack the flexibility that interpreters have. The source program is the input for the compiler, and the output is machine code. Compilers are almost always faster than interpreters.

2) Readability

C's readability sometimes suffers due to multiple ways to perform an operation. For example, there are four ways to increment a variable. C also lacks orthogonality, for reasons such as a function being able to return a record but not an array, disallowing voids and functions from being array elements. C lacks the boolean type which reduces readability, although a boolean can be simulated by using enums. Statement groups always end with a "}" which can reduce readability.

no operator overloading is ST

SmallTalk's readabilty can be complex if a user decides to improperly overload operators. Also, there are a large number of classes and methods in SmallTalk so programmers may only learn a subset of the language. If programmers learn different subsets of the language, then the language becomes less readable. SmallTalk does have the boolean type which enhances readability.

Writability

Since C is usually less readable than Smalltalk, it is also less writable, for the same reasons it is not very readable. C does have some shorthand operations such as "++" which SmallTalk does not have which improves writability.

-2

SmallTalk is more abstract than C (due to the availability of objects which delegate the responsibilities of other larger objects), therefore improving writability.

Reliability

C can be unreliable do to the lack of strong type checking. Also, C has no bounds checking or exception handling which reduces reliability. Aliasing, or having the ability to have two names for the same memory cell, is also present in C. Aliasing also reduces reliability.

SmallTalk has some exception handling, such as exception blocks for when an element is missing from an array. There is no aliasing in SmallTalk.

Cost

Since many programmers already know C, the cost for training them is low. Both the C and SmallTalk compilers are free, so cost is not increased due to compiler cost. The cost of maintaining C programs are most likely higher than SmallTalk since C programs are generally less readable.

-2

Levi D. Smith
gte187k
July 12, 1999
CS3411
HOMEWORK 1

SmallTalk is not as commonly known as C, so the cost of training programmers will be higher than C. Since SmallTalk runs much slower than C, cost is increased.

3) Algol 60 introduced many concepts that would be used in the making of many other languages. Algol 60 introduced BNF (Backus-Naur Form) which is a way of describing the syntax of languages. Also introduced were: block structure (including block scopes for variables), pass by value and pass by name for parameters, recursion for imperative languages, and arrays which could be indexed by variables.

Algol 68's main purpose was to be orthogonal. User defined data types were included in Algol 68, which could be created from combinations of any of the simple components. Also included were dynamic arrays, which do not require bounds. Memory for dynamic arrays is automatically allocated.

4)

<time> --> <timewithminutes> | <timewithoutminutes>

<timewithminutes> --> <hours> ":" <minutes> " " <ampm>

<timewithoutminutes> --> <hours> " " <ampm>

-5 ambiguous

<ampm> --> am | AM | pm | PM

<hours> --> <positivedigit> | 10 | 11 | 12

<minutes> --> <tensminute> <onesminute>

<tensminute> --> 0 | 1 | 2 | 3 | 4 | 5

<onesminute> --> <positivedigit> | 0

<positivedigit> --> 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

5)
```
E -> E + T {0}
E -> E - T {1}
E -> T     {2}

T -> T * F {3}
T -> T / F {4}
T -> F     {5}

F -> ( E ) {6}
F -> I     {7}
F -> N     {8}

Rule 0:
  E1.value = E2.value + T.value
Rule 1:
  E1.value = E2.value - T.value
Rule 2:
```

-5

Levi D. Smith
gte187k
July 12, 1999
CS3411
HOMEWORK 1

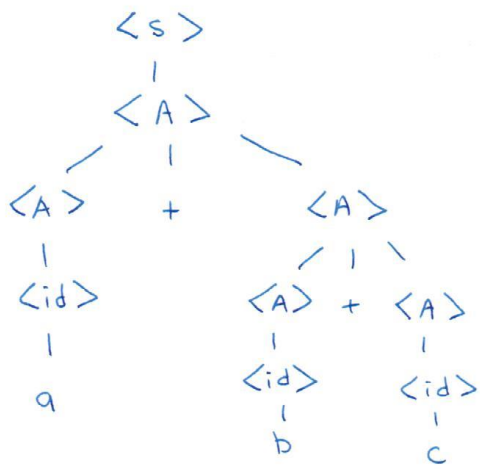
```
  E.value = T.value
Rule 3:
  T1.value = T2.value * F.value
Rule 4:
  T1.value = T2.value / F.value
Rule 5:
  T.value = F.value
Rule 6:
  F.value = E.value
Rule 7:
  F.value = I.value
Rule 8:
  F.value = N.value
```

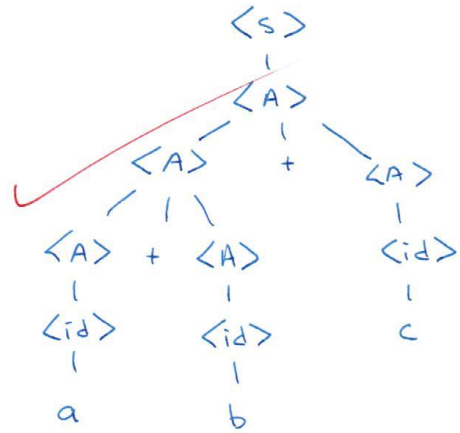
Note: I used the web examples as a guide while writing my attribute grammar so my structure is similar to the example.

Problem 5, page 152 Levi D. Smith

To prove that the grammar is ambiguous, I will show that the grammar can produce two different parse trees for the sentence

a + b + c





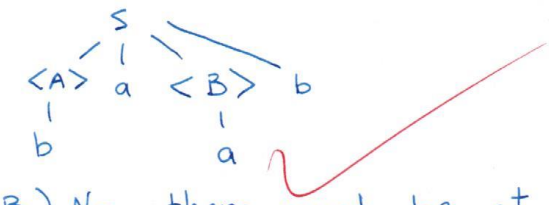
Levi D. Smith
gte187k
July 12, 1999
CS3411
HOMEWORK 1

7) S is a character String consisting of any number of 'a's followed by any number of 'b's followed by any number of 'c's. (When I write "any number" I mean any integer greater than zero).

Problem 8, page 152

Levi D. Smith

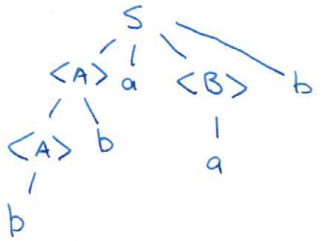
A.) Yes



B.) No, there must be at least 2 'a's in the sentence.

C.) No, all sentences must end with a 'b'.

D.) Yes



Sentence A. and sentence D. are generated by the grammar.

Levi D. Smith
gte187k
July 12, 1999
CS3411
HOMEWORK 1

-4 needs r-value too

9) (++) requires an l-value to store the result of the increment operation. (++) takes the value of the l-value and increments it (the integer) making an r-value.

10) Dynamic scoping is when the scope (or the visible range of a variable) can only be determined at run-time. Languages such as APL, Perl, and SNOBOL4 have dynamic scoping. Dynamic scoping occurs when a function calls another function and all of the variables in the caller function can be seen by the called function. When this occurs the variable's scope can only be determined at run-time.

Dynamic binding is when a variable is bound (setting associations of a variable) during run-time because bindings change while the program is running.

Dynamic type checking is when operands of an operator are checked during run-time to make certain that the operands are of compatible type.

11) C supports static, stack-dynamic, and explicit heap-dynamic.

```
static example:
  static int i; /* i's lifetime is the life of the program */

stack-dynamic example:
  int i; /* i's lifetime is the local block */

explicit heap-dynamic example:
  int *i;
  if ((i = malloc(1024)) == NULL) /* life time is determined   */
    printf("Out of memory");      /* by the programmer         */
                                  /* lifetime begins with      */
                                  /* malloc and ends when it   */
                                  /* is free()'d               */
```

12) Having all declarations at the top of the block allows the programmer to quickly find the location where a variable was declared. There is no need hunt through the whole block to find the variable's declaration.

On the other hand, with intermixing the programmer declare a variable exactly when it is needed. The scope of the variable is also limited to the location where the variable is declared to the end of the block. For example, the loop incrementer variable could be declared before the loop that it will be used, instead of at the beginning of the block. This would restrict other statements above the declaration from modifying the variable.

-4

Levi D. Smith
gte187k
July 26, 1999
CS3411
HOMEWORK 2

108/120

[p1]

Pointer arithmetic is the usage of the plus (+) or minus (-) operator on a pointer along with an integer telling the pointer how many bytes to move forward or backward in memory. The pointer will move by the integer (specified by the programmer) times the size of the type of the pointer. For example:

6/12

```
int *intPointer;  //an integer pointer
intPointer + 2;   //the usage of pointer arithmetic here will
                  //move the pointer forward 8 bytes in
                  //memory (assuming that the size of an
                  //int is 4 bytes on this machine)
```

Array notation and pointer notation are two ways to represent the same thing, a memory location. The square brackets signify that the pointer is dereferenced and moved forward by the number in the square brackets times the size of the type of pointer. For instance, p[42] is the same as *(p+42). Also, p is equivalent to &p[0], and *p is the same as p[0].

[p2]

A C++ reference is basically a name for a pointer to an object. A reference will not be copied to the function stack when it is used as a parameter in a function, allowing the ability to bypass call by value parameter passing. C++ references eliminate many of the complexities of C pointers, such as the dereferencing and address-of operators.

8/12

A Java object reference hides all usage of pointers from the user. Object references are passed to a function by value. If you change an object reference in a function, it will only effect the local objects to that function.

[p3]

BASE +
(i * (number of j elements) * (number of k elements) * ESIZE) +
(j * (number of k elements) * ESIZE) +
(k * ESIZE)

12/12

[p4]
The result on the cc compiler:
sum1 == 46
sum2 == 48

The results on the gcc compiler:
sum1 == 48
sum2 == 48

12/12

The reason the two compilers had different outputs is because the ANSI C standard does not guarantee the order in which the expressions are evaluated in an addition operation. The cc compiler evaluates the first function/expression and adds the result to the result of the second expression, and so on until it gets to the last function/expression in the right hand side. On the other hand, the gcc compiler executed all functions first, then added the results together. When pointer values or global variables are modified in functions producing side effects, unreliable results, such as in example above will be produced.

[p5]

There are ten operators:
= , , + () = / ++ - ||

12/12

a is assigned to c
    a = c
    a now equals 3

Everything in parenthesis is done next:
  d is divided by b and assigned to c
  c = 4 / 2
  c now equals 2
    b is incremented
    b now equals 3

b is added to the expression in parenthesis, and a is subtracted
    b + 2 - a
    3 + 2 - 1
    4

logical OR is taken between 4 and b
    4 || b
    4 || 3
    1

comma is applied to the three expressions
    3, 3, 1
    1 (overall result)

a == 3
b == 3
c == 2
d == 4

[p6]

The non-associative operator in Ada means that the programmer must explicitly place parenthesis to signify the precedence of the exponentiation operator. If the programmer does not place parenthesis in the expression an error will be generated.

```
ONE ** TWO ** THREE  /* ERROR  */
(ONE ** TWO) ** THREE /* This is okay */
ONE ** (TWO ** THREE) /* This is also okay */
```

12/12

Most other imperative languages use right to left associativity when evaluating expressions with multiple exponentiation operators.

[p7]

```
==================== BEGIN PRETEST LOOP =======================
#include <stdio.h>

#define SUCCESS 0
#define TERMINATE 0

void getNumber() {
  int iSum;
  int iInput;

  iInput = 1; /* assign a dummy value so it will loop
                 at least once */
  iSum = 0;
  while (iInput != 0) {
    printf("Enter a positive number (0 to quit):");
    scanf("%d", &iInput);
    iSum += iInput;
    if (iInput != 0) {
      printf("Running sum is %d\n\n", iSum);
    }
  }
}

int main(void) {
  getNumber();
  return SUCCESS;
}
```

12/12

```
==================== BEGIN POSTTEST LOOP ========================
#include <stdio.h>

#define SUCCESS 0
#define TERMINATE 0

void getNumber() {
  int iSum;
  int iInput;

  iSum = 0;
  do {
    printf("Enter a positive number (0 to quit):");
    scanf("%d", &iInput);
    iSum += iInput;
    if (iInput != 0) {
      printf("Running sum is %d\n\n", iSum);
    }
  } while (iInput != 0);
}

int main(void) {
  getNumber();
  return SUCCESS;
}
================ BEGIN USER CONTROLLED LOOP ====================
#include <stdio.h>

#define SUCCESS 0
#define TERMINATE 0

void getNumber() {
  int iSum;
  int iInput;

  iSum = 0;
  while (!TERMINATE) {
    printf("Enter a positive number (0 to quit):");
    scanf("%d", &iInput);
    if (iInput == TERMINATE) {
      break;
    }
    iSum += iInput;
      printf("Running sum is %d\n\n", iSum);
  }
}

int main(void) {
  getNumber();
  return SUCCESS;
}
```

========== BEGIN DISCUSSION ==========

The user controlled loop is the "easiest" since it allows the function to be broken immediately when the terminating condition is reached. The user controlled loop allows multiple exits for one loop, which may be considered bad style (reduces readability). The posttest loop would be better than the pretest loop since the loop will always be run once (eliminating the need to assign the input variable a dummy value).

[p8]

The compiler generates a jump table(constants list) when it finds a multiway selection statement. The jump table is more efficient than if/then/elseif because it can go directly ("jump") to the correct choice, if the correct choice is available. With if/then/else, each choice has to be tested until the correct choice is found. Also, multiway selection statements are usually much more readable than if/then/else.

8/12

[p9]

First the user must define a label. A label is defined by an identifier (character string) followed by a colon. Labels must be followed by a loop statement. Multiple nested loops can now be exited by calling "break" from a loop followed by the name of the name of the label of the loop you wish to break. The use of break statements eliminates the need for boolean variables which would report error conditions. Here is my example:

12/12

========== BEGIN JAVA CODE ==========

```
class P9 {

public P9() {
  loopTest("THISISINUPPERCASE", 2);
  loopTest("ABORTlowercaseUPPERCASE", 2);
}

/**
  * loopTest - prints all characters in strTest iTimes times.  the
  *            process is aborted when it encounters a non uppercase
  *            character
  *
  **/
public void loopTest(String strTest, int iTimes) {
  int i, j;

  i = 0;
/**
  * //This isn't allowed
  *
  * if (iTimes == 0) {
  *   break EXIT;
  *  }
  *
```

```
  **/

  firstloop:
  while(i != strTest.length()) {

    secondloop:
    for (j = 0; j < iTimes; j++) {

                  //Test to see if character is uppercase
      if (!(Character.isUpperCase( strTest.charAt(i) ))) {
        break firstloop; //break out of both loops
                         //do not print any more characters
      }
      System.out.print(strTest.charAt(i));
    }
    i++;
  }
  System.out.print('\n');

//  EXIT:     //this is also an error
              //a label must be followed by a loop
}

public static void main(String Argv[]) {
  P9 myP9 = new P9();
}

}
==================== END JAVA CODE ========================
```

[p10]

friend function: allows other classes and functions to access this function's private data members

virtual function: tells the compiler that this object's type needs to be looked up at run-time, not compile-type. A function has to be declared "virtual" in order for the function to use the abilities of polymorphism.

pure virtual function: this function must be implemented by the subclass. pure virtual functions are declared by adding "= 0" to the declaration of a virtual function.

const member function: the data in a const member function cannot be changed (easily).

copy constructor: is called when a new class it created. it copies the values specified by the initialization parameters into the correct member data variables.

templated function: allows the programmer to define one template to use on multiple objects/types (generic code). the code inside of the templated function must semantically support the types that you send to the function.

initializer list: comes before the body of the constructor. it is basically a list of variables and the values to which they will be set.

12
12

Homework 3 - Sebesta 8, 9, 13-15, Prolog

CS 3411 - Programming Language Concepts - Summer '99 (Hutto)

Due: 4pm (classtime) Wednesday 4 August

READ ME! Please turn in a legible hardcopy solution at the **beginning** of class. Staple, but do not fold, multiple pages. Print "HOMEWORK 3" and your name and GT number clearly on the front page. We will deduct 5 points if you do not follow these instructions!

LATE POLICY: No late homeworks will be accepted for this assignment.

PROBLEMS vs. QUESTIONS: Your textbook has two sets of exercises at the end of each chapter titled "Review Questions" and "Problem Set". I will say "Question" when I mean "Review Questions" and "Problem" when I mean "Problem Set". I will also include the page number to help disambiguate but several students have done the wrong exercise in past quarters so beware!

Got Questions? Try office hours, email
or the class newsgroup git.cc.class.3411 (news.cc.gatech.edu)

Chapter 8: Subprograms

1) In C there is a **difference between the following function prototypes**:

```
int f();
int f(void);
```

Explain the difference. How would you achieve the same effect in C++? (Hint: void is allowed but unnecessary in C++. It actually means something in C.)

2) Problem 9, page 375

3) Write a **"mapping" function in C** that accepts an array of doubles as its first argument, the number of elements in the array as its second argument, and, as its third argument, a pointer to a function that returns a double and accepts a double as a parameter. The mapping function should apply the parameter function to the elements of the array. (The array is an in-out parameter.) Here is an example invocation of the function:

```
double a[] = { 1.2, 0.5, 3.8, 199.45, 3e-2 };
int n = sizeof(a) / sizeof(double);
...
map( a, n, sqrt );
```

4) What's a **thunk**?

5) Describe how **default parameters work in C++**. What's the relationship between **default parameters** and function **overloading**?

6) In "The Java Programming Language 2e", Gosling says: "**All parameters to Java methods are 'pass by value'**". How can that be? Can't you alter an object passed as a parameter to a Java method? (**Hint**: See Gosling's rant about the meaning of 'pass by reference'.) What does it mean to add the **keyword final** to an object parameter in Java? For example:

```
void changeTheString( final StringBuffer sb ) { ... }
```

Chapter 13: Exception Handling

7) What are the advantages of a language-based exception handling mechanism (like Java's) over simply returning numeric "error codes" (like UNIX library routines)? What are the disadvantages?

8) What is the difference between a **checked and unchecked exception in Java**? Under what circumstances is the **optional `finally` clause** in Java executed?

Chapter 14: Functional Programming

9) What does the following **Haskell code** do? (Explain the syntax in detail.)

factors n = [i | i <- [1..n **div** 2], n **mod** i == 0]

10) What do the supporters of the functional programming paradigm claim as the advantages of this approach over imperative programming?

Program 1 -- Computing Document Similarities in Perl

CS 3411 (Hutto) - Programming Language Concepts - Summer '99

DUE: 4 pm Monday 12 July (class time)

Got Questions? Try office hours, email or the newsgroup git.cc.class.3411a

Computing Document Similarities

Vector Space Method
looks at commonly occuring words
tells how many times each term appears in a file.

For your **first programming assignment** this quarter, you will complete a **Perl code framework** for computing the similarity between two text documents (files) expressed as a real number between 0 and 1.0. Two identical documents will report a similarity factor of 1.0. Two completely distinct documents will have a similarity factor of 0.0. So how do we assign a **similarity factor**? The technique used by most Web search engines looks at the number of **common "terms"** appearing in both documents. Documents with more common terms receive a higher similarity rating. This approach assumes that two documents talking about the same thing will use the same vocabulary. This works most of the time but fails badly when people use completely distinct but synonymous terminology. For example, one document might describe flashlights, while a British version of the same document might talk about "torches". The term frequency approach also fails when the same word has two distinct meanings and will relate documents describing gasoline drums and snare drums. Still, the longer the document, the less likely that ALL terms will fall into one of these two bad cases.

Perl is an excellent language for writing a document similarity program. It has sophisticated text manipulation capabilities and powerful operations on dynamic lists (arrays) and associative arrays (hashtables or, in Perl lingo, "hashes") that relate a key and a value.

Terms

To compare two documents, we begin by assembling a list of terms appearing in each document. Consider terms to be character sequences separated by whitespace. We will "clean up" the terms by removing funny punctuation and digits and lowercasing everything. (This is a bit extreme but works well for our simple version of the program. Real code removes commonly occuring terms, and prefixes and suffixes in a process called "stemming".)

Term Frequencies

Along with a list of terms appearing in each document, we will count the **frequency of occurrence** of each term. This is per document. Some terms appear many times in one document and not at all in another document. We can easily store term frequencies in an associative array or hash where the key is the term and the value is the count. We will maintain two term frequency hashes, one for each document.

Dictionary

Once we have term frequency hashes for each document, we can easily build a single list containing all terms. This list, called the **dictionary**, will contain a single occurrence (no duplicates) of each term appearing in either document. We will keep the dictionary in ascending sorted order.

Term Frequency "Vectors"

- two docs are similar if they use the same term
- fails with synonyms
- specific term - oil drums / snare drums
- throws away WS, numbers, punctuations

@words @words 2 @dictionary (all terms / sorted)
term list
hash structure to hold freq count / %tf 1 %tf 2

This approach to document similarity is called the **vector space model** because it constructs a "vector" for each document that contains the frequency of occurrence of each word in the dictionary (the entire document collection) in that particular document. These are **"vectors"** in a **multi-dimensional "term space"** where there are as many dimensions as there are terms. We can compute a similarity factor by **computing the angle or distance** between these vectors. Before we do that, however, we will **weigh** and **normalize** these term vectors in various ways to produce more desirable results.

Inverse Document Frequencies

Common terms appearing in a small number of documents are weighted more highly than terms appearing in all documents. The "document frequency" (df) is the number of documents in which a term appears once or more. Since we are working with a two document collection, this is a simple value. If the term appears in both documents its df is 2, otherwise it is 1 (it must appear in at least one document to even appear in the dictionary). We can use a hash to calculate this number for each word in the dictionary. Once we have the document frequencies, we just invert the value. Salton et al. suggest scaling these values as follows where N is the number of documents in the collection (2 in our case):

$$idf_i = \log (1 + (N / df_i))$$

This scaling factor seems to work well empirically.

Normalization Factors

If we don't "normalize" the term frequencies, long documents will have higher similarity than short documents. We need to compute a normalization factor for each document. When we compute the weighted term vectors, we will divide each weighted frequency by the normalization factor. This ensures that the resulting vectors are unit vectors in the multi-dimensional term space. To normalize a two-dimensional vector (x,y), just divide each component by the length of the vector. That's just the square root of the sum of
the squares of the components by a simple application of the Pythagorian Theorem:

$$nf = sqrt(x*x + y*y)$$

The components of the term frequency vectors are the individual term factors (tf) multiplied by the inverse document frequency (idf) of each term:

$$tf_i * idf_i$$

so the corresponding equation is:

$$nf = sqrt(SUM_i((tf_i * idf_i)^2))$$

where SUM_i means the sum over all terms in the document.

Weighted Term Vectors

The heart of the algorithm is producing the weighted term vectors for each document. We will represent each vector as a list of real numbers. Each value in this list is associated with the corresponding term in

the dictionary (they are "parallel arrays"). If a word doesn't appear in the document, its weight is 0. If it appears in the document, then the weighted value is:

$$w_i = (tf_i * idf_i) / nf$$

Since these values are normalized, they should be between 0 and 1.0. If you have properly normalized the vectors, compute the sum of the squares of all the elements. The square root of this value should be 1.0 by the Pythagorian theorem.

Computing Similarity

Once we have the two weighted term frequency vectors (one for each document) computing document similarity is very easy. We just compute the **dot product** of the two vectors. Recall that the dot product is the sum of the product of related elements (multiply the first two elements, then multiply the second two and add the result and so forth.) Since the vectors have been normalized (they are unit vectors), the result is guaranteed to be between 0 and 1.0.

If you want to read more about this approach, check out the CACM article by Salton, Allan and Buckley.

How Do I Start?

Copy the code framework `sim.pl` to a system that has a Perl 5 interpreter (type "perl -v" to check). Perl is very portable and you should be able to develop your program under Windows or Linux but we have Perl 5 installed on all the CoC machines and on acme. Under UNIX, you need to make this file executable:

```
chmod +x sim.pl
```

The framework requires two filenames as commandline arguments so you can invoke it like this:

```
sim.pl file1 file2
```

The framework has all the data structures declared and calls the appropriate subroutines but does nothing so it will print out some debugging information and report a similarity of 0.0 for any files you supply.

The framework is heavily commented. You should begin by examining the contents carefully. Here is the beginning of the file showing the "main" part of the script.

```
###
# MAIN --
# Perl programs don't have an explicit "main". Perl will execute all code in
# the file outside of subroutine definitions.
###

   # require two filenames as cmd line args; remember the names
   @ARGV == 2 || die "Usage: sim <fn1> <fn2>\n";
   ( $f1, $f2 ) = @ARGV;

   getTerms();
   computeTFs();
```

```
buildDict();
computeIDFs();
computeNFs();
computeWeights();
$sim = computeSimilarity();
  ....
```

This is a great program to develop INCREMENTALLY. Begin with getTerms(), the subroutine that opens the two files, and creates term lists. Once you get that working, start on computeTFs(), the routine that computes term frequencies.

Perl Stuff

You will probably end up loving and hating Perl at the same time. Larry Wall, the designer of Perl, is an old time UNIX hacker and one of his mantras is "There's More Than One Way To Do It". At first you will be overwhelmed by all the capabilities of Perl. Once you learn some tricks you will start appreciating how powerful and expressive the language is. Then you will spend hours debugging a strange problem involving a special case of some strange feature and you will start hating Perl again.

Perl is semi-compiled. Each time you run a script, the interpreter scans the entire file and compiles it into a fast intermediate representation that is then interpreted. Under UNIX, the file must be executable and begin with a special line indicating the location of a perl5 interpreter. On the CoC machines that would be:

```
#!/usr/local/perl5/bin/perl
```

A Perl file consists of executable statements and subroutine definitions. Subroutine definitions cannot be nested (like C, unlike Pascal). There is no explicit main() in Perl. The interpreter simply executes all the statements in the file.

Variables

Perl does not require variables to be declared. This is convenient but dangerous. I will ask you to "declare" all variables using the special function "my". To declare and initialize a variable you can say:

```
my $total = 0;
```

You can declare several at once like this:

```
my ( $term, $i, $nterms );
```

Types

Perl has a strange and flexible notion of data types. All variable names begin with a special character that indicates the "type" of the variable:

$scalar	(number or string or boolean)
@list	(array)
%hash	(associative array: key-value pairs)

14 17

Lists and hashes are dynamic and grow and shrink on demand up to the size of available memory. Perl follows the 0-1-infinity rule everywhere. The various "types" have separate namespaces so $name and @name are different things and may both exist in the same program.

Command line arguments are made available to the program through a special array called @ARGV.

Lists / Arrays

Perl has a notion of "contexts", places where either a scalar or a list or a hash is expected. Sometimes the meaning of a variable depends on the context. For example, an array name like @list, returns the number of elements in the list when used in a "scalar context". For example:

```
my $n = @list;   # scalar context so @list returns list size
```

In other situations, it refers to the entire list:

```
@list = ( 1, 2, 3, 4 );          # a list context
@list = ( 1, 2, 3, $n, 5, 6 );   # memory is dynamically allocated
```

Lists can also be viewed as arrays and stacks when accessed appropriately:

```
push( @list, 10 );
my $value = pop( @list );
```

You can select array elements using the standard square bracket notation but Perl requires that you use $ before the array name when selecting an element (this emphasizes that the result of selecting an element is a scalar, not a list):

```
my $third = $list[2];   # array subscripts begin at 0
```

There are lots and lots of powerful array functions and operations:

```
my @sorted = sort( @list );
```

Perl lets you leave off parentheses in function calls if you like. (It can do this because variables begin with special symbols so the parser can figure it out.)

```
my @sorted = sort @list;
```

Assigning to an undefined element implicitly grows the list. If you assign way past the end, the elements between the new one and the last one have the special value undef. Touching an undef value will generate a warning. You can test whether a value is defined using the function:

undefined = ∅ false "" coerced to whatever you need

```
if ( defined( $list[100] ) ) { ... }
```

Printing

There are two ways to print in Perl:

```
print "hello, world\n";
```

```
printf "%s %s\n", $word1, $word2;
```

The print command has an interesting capability called "variable interpolation" that evaluates variable references inside quoted strings:

```
$whatever = "world!";
print "hello, $whatever\n";   # produces "hello, world!"
```

There is a string concatenation operator (.) but you don't need it very often.

Regular Expressions / Substitute

Perl has a wide variety of capabilities for using regular expressions to manipulate text. Regular expressions (res) are delimited by forward slashes so:

```
/abc/
```

matches the string "abc" (using the implicit concatenation operator). Alternation (|) and Kleene closure (*) are also available along with the extensions we discussed: ^ $. + (). Perl has several predefined "character class abbreviations". For example: \d is the same as [0-9], \s matches a single whitespace character and \w matches a letter, digit or underscore. There are negated versions of these classes so \D matches any character that is NOT a digit and so forth. The special abbreviation \b matches a "word boundary" something like ^ and $ match a "line boundary". Comparing a string with a re returns true or false and is done with the binary "match" operator (=~) like this:

```
if ( $name   =~ /\b[Pp]hil/ ) {
   # do something
}
```

required

A special notation is useful for matching and then replacing or substituting. A similar syntax can be used in most text editors:

```
$which = "this is a test";
$which =~ s/test/quiz/;   # replace test with quiz in $which
```

There are zillions of other re capabilities in Perl. Checkout one of the Perl texts or the recent O'Reilly book "Mastering Regular Expressions" for details.

Split / Join / Translate

The split function takes a regular expression and a string. Parts of the string that DO NOT match the regular expression are returned in sequence as a list of values. split can be used to "tokenize" strings. You can breakup a tab-delimited string into pieces like this:

```
$s = "a\tb\tc\t\d";
@pieces = split( /\t/, $s );
```

separator

\s+

join ↔ makes array into a string

Split behaves a bit strangely if you split a string with leading whitespace so either read the manual carefully or be sure to trim any leading whitespace before you split.

The join function is the companion to split. You give it a "glue string" and an array and it returns one big string created by gluing the individual elements together. To put the string above back together we could write:

```
$s2 = join( "\t", @pieces );
```

Finally, there is an operator called tr (named after the UNIX tr command) that allows you to replace or remove individual characters in a string. You provide two lists of characters. Characters in the first list are replaced by characters in the second list. Syntactically tr looks like the substitute command:

```
$name = tr/a-z/A-Z/;   # translate lowercase letters to uppercase   only letters
$name = tr/aeiou//d;   # delete vowels
```

delete characters

Notice that you use an assignment operator and not the matching operator (=~). Also, tr does not support general regular expressions. About the only special character you can have between the forward slashes is the dash for defining ranges.

Subroutines / Parameters / Locals / Globals

Defining a subroutine is simple, just say:

```
sub mySub {
   statements;
}

$result = mySub( $one, $two, $three );
```

Paremeter passing is a bit unusual. Every sub implicitly receives a single parameter with the name @_ which is a list of the parameters provided to the subroutine (it's kind of like @ARGV). It is common to assign these parameters to named local variables using a statement like this:

```
my ( $p1, $p2, $p3 ) = @_;
```

This is actually just a special case of "array assignment". For example:

```
( $f1, $f2 ) = @ARGV;
```

accesses and assigns the first two elements of the list @ARGV to the specified variables. If there are additional elements in @ARGV, they are ignored. If there are not enough, the unlucky variables take on the value undef.

Variables "declared" with my inside a subroutine are local to that function. Variables declared outside any subroutine are global.

It is possible to return values from a subroutine. If no explicit return is included, the value of the last expression evaluated in the subroutine is used as the subroutine value (like LISP). It is possible to return multiple values from a subroutine as a list:

```
( $min, $max ) = minmax( $a, $b );
```

Input / Filehandles

Perl has an abstraction of a UNIX file pointer called a filehandle. They look like variable names in all caps. Two are provided by default: STDIN and STDOUT. To read a line as a string from STDIN, just say:

```
$line = <STDIN>;
```

$line will contain a terminating NEWLINE which can be removed by the chomp function:

```
chomp $line;
```

Evaluating <STDIN> by itself places a copy of the line in the special variable $_. This special variable is the default for all sorts of Perl commands, like print. To echo a file, removing all the newlines, you can just say:

```
while ( <STDIN> ) {
  chomp;   # using $_
  print;   # using $_
}
```

It is possible to associate a filehandle with a file or process using open(..):

```
open( MYFILE, "pathname" );
```

The <> operators have a special meaning when used in a "list context". If you say:

```
@lines = <MYFILE>;
```

← takes in all lines of MYFILE

Perl will create an array containing ALL the lines from the specified file!

Hashes

One of the more powerful features of Perl are its dynamic hash tables. Variables begining with % are hashes. They can be explicitly initialized by key, value pairs. Or you can simply add elements one at a time.

```
my %hash = ();
```

To add or update an element in a hash say:

```
$hash{$key} = $value;
```

To delete an element say:

```
delete $hash{$key};
```

If you request a key that is not in the hash, the value undef is returned:

```
if ( !defined( $hash{$key} ) ) {
```

```
    $hash{$key} = 0;
  }
```

Actually there is a special rule for hashes that allows you to simplify the above code when you are counting things. The expression:

```
  $hash{$key}++;
```

will increment the value associated with $key if defined. If you try to increment a key that is not in the hash, Perl will initialize the element to 1 for you!

The keys function returns a list of the keys of a hash:

```
  @keys = keys %hash;
```

The values function returns a list of the values:

```
  @values = values %hash;
```

These are returned in an unpredicatable order but the elements of the two lists correspond. It is possible to iterate through the elements of a hash with the each function:

```
  while ( ($key,$value) = each(%hash) ) {
    print "$value $key";
  }
```

Looping

Perl has standard for, while and do-while loops from C along with a do-until loop that allows you to specifiy a termination condition for a post-test loop. In addition, Perl has a very useful foreach statement that allows you to iterate through the elements of a list:

```
  my $element;
  foreach $element (@list) {
    print "$element\n";
  }
```

Strange Flow Control (||, &&, if, unless)

There is no switch statement in Perl but it is possible to build an equivalent structure several different ways. In addition, Perl has a few unusual control flow techniques. The short-circuited boolean operators are commonly used for flow control. Here is an example using the die() function that prints an error message and terminates the progam:

```
  open( MYFILE, $filename ) || die "Can't open file $filename\n";
```

If the open() succeeds then the die() function is never called.

Perl also allows the programmer to add conditionals like if and unless to the END of a statement. This occasionally increases the clarity of code:

```
$n = $list[$i] if defined( $list[$i] );
$n = sqrt( $x ) unless $x<0;
```

Helpful Advice

There are LOTS of books on Perl. I like the O'Reilly books (strong supporters of Perl; they maintain the Perl archive at their Web site). Here are three good books:

"Learning Perl 2e" // gentle introduction, first 10 chapters on electronic reserve
Schwartz & Christiansen

"Perl Cookbook" // zillions of examples
Christiansen & Torkington - Example Book

"Programming Perl" // a tough read but lets you get inside Larry Wall's head
Wall & Christiansen & Schwartz

Check out the official Perl Web page at:

http://www.perl.com

There is a lot to learn but this is a really cool assignment and Perl is a very interesting language, so hopefully, you won't mind jumping in the deep end! We aren't out to kill people with this assignment. We just want you to have some fun and learn something about Perl. Feel free to talk to your classmates but everyone must get the program working individually. Use office hours and the newsgroups.

Good luck! Start Early! Use Office Hours Often!

* remember to remove WS from beginning of line

```
#!/usr/local/perl5/bin/perl -w

###
# FILE: sim.pl --
# USAGE: sim.pl fn1 fn2 (sim.pl must be executable)
# --------------------
# Perl framework for document similarity computation using the vector space
# model. This technique computes a normalized similarity "factor" [0..1.0]
# between two documents based on the relative frequency of common terms
# found in both documents. This is essentially how Web search engines work.
# Normally the "document collection" consists of thousands of individual
# documents. The "query" is just considered a document that is then compared
# to the document collection. For this assignment, we will just compare
# two documents.
#
# For details on this approach, see the CACM article:
#
#   "Automatic Structuring and Retrieval of Large Text Files"
#   Salton, Allan, Buckley
#   http://www.acm.org/pubs/citations/journals/cacm/1994-37-2/p97-salton
#
# Phil Hutto
# CS 3411 Summer 99 * Programming Language Concepts
# College of Computing / Georgia Tech
# 18 April 1999
###

# requires that you "declare" all variables using my; helps avoid typos
use strict;

###
# GLOBALS --
# Perl provides references (like pointers) for implementing call-by-reference
# parameter passing and return but they are a bit tricky and the syntax
# is painful. For this assignment, we will compute and save intermediate
# results in global data structures. We will still use "my" declarations
# within subs to keep variables that need not be global local.
#
# We could, of course, pass lists and hashes by value but that would be
# very inefficient for large input files.
#
###

    my( $f1, $f2 );              # filenames
    my( @terms1, @terms2 );      # term lists
    my( $nterms1, $nterms2 );    # term counts
    my( %tf1, %tf2 );            # term frequency hashes
    my( @dict );                 # global term dictionary
    my( %idf );                  # inverse document frequencies
    my( $nf1, $nf2 );            # normalization factor for weights
    my( @w1, @w2 );              # weighted term vectors
    my $sim;                     # normalized document similarity

###
# MAIN --
# Perl programs don't have an explicit "main". Perl will execute all code in
# the file outside of subroutine definitions.
###

    # require two filenames as cmd line args; remember the names
    @ARGV == 2 || die "Usage: sim <fn1> <fn2>\n";
    ( $f1, $f2 ) = @ARGV;

    getTerms();
    computeTFs();
    buildDict();
    computeIDFs();
    computeNFs();
    computeWeights();
    $sim = computeSimilarity();

    # output intermediate results to verify correctness during debugging

    print "\nSORTED TERM FREQUENCIES:\n\n";
    my $key;
    print "$f1:\n";
    foreach $key (sort keys %tf1) {
        print "$key $tf1{$key}\n";
    }
```

```
    print "\n$f2:\n";
    foreach $key (sort keys %tf2) {
        print "$key $tf2{$key}\n";
    }
    print "\n";

    print "\nDICTIONARY:\n";
    my $term; foreach $term (@dict) { print "$term\n"; } print "\n";

    print "\nSORTED INVERSE DOCUMENT FREQUENCIES:\n";
    ## what goes here?
    print "\n";

    print "\nWEIGHTS:\n\n";
    my $i;
    print "$f1:\n";
    ## what goes here?
    print "\n$f2:\n";
    ## what goes here?
    print "\n";

    # the final answer

    print "SIMILARITY: $sim\n";

    exit(1);

###
# SUBROUTINE: getTerms --
# --------------------
# INPUTS: $f1 $f2
# OUTPUTS: @terms1 @terms2 $nterms1 $nterms2
#
# Open specified files. Create "tokenized" lists of terms from each.
# Count terms in each. Tokens are maximal non-whitespace character sequences
# separated by whitespace. Clean-up the input by removing everything but
# letters from the tokens (get rid of digits, punctuation, etc.).
# This is pretty extreme, but a simple rule. Things like "Hello," will be
# converted to "hello". Things like "What's123This?" will be converted to
# the single token "whatsthis" (bad characters are REMOVED, not converted
# to whitespace). Lowercase the tokens.
#
# Real systems eliminate common terms (a, the, and, of) found in a "stoplist".
# In addition they perform "stemming" which removes prefixes and suffixes.
#
# HINT: This sub can be amazingly short if you take advantage of all the
# powerful features of Perl. For example you can read all the lines of a file
# into an array with a single statement like this:
#
#   @array = <IN>;
#
# Look at the split, join, translate (tr) and substitute (s///) commands.
###
sub getTerms() {

    open( F1, $f1 ) || die "Can't open file: $f1\n";
    open( F2, $f2 ) || die "Can't open file: $f2\n";

    ## what goes here?

    close( F1 ) || die "Can't close file: $f1\n";
    close( F2 ) || die "Can't close file: $f2\n";
}

###
# SUBROUTINE: computeTFs --
# ----------------------
# INPUTS: @terms1 @terms2
# OUTPUTS: %tf1 %tf2
# Build two hashes to count (canonicalized) term frequencies in each document.
###
sub computeTFs() {
    %tf1 = %tf2 = (); # for now
}

###
# SUBROUTINE: buildDict --
# ---------------------
# INPUTS: @terms1 @terms2
```

term lists

size of text

produces terms in order of file

term frequencies

count how many times term occur

```
# OUTPUTS: @dict
# Build a "dictionary" (an array) containing all words found in either
# document. The dictionary should have no duplicates and should be in
# sorted ascending order.
#
# HINT: The dictionary needs to be a list, not a hash, but you can use
# an auxiliary hash to help you construct it. To sort a list in Perl,
# just say:
#
#   @sorted_list = sort $list;
#
###
sub buildDict() {
    @dict = (); # for now
}
```

contain one occurence of each term that is in either document

use Perl features for → for each

```
###
# SUBROUTINE: computeIDFs --
# ------------------------
# INPUTS: %tf1 %tf2
# OUTPUTS: %idf
# Build a single hash containing an entry for each word in the dictionary.
# If the a term appears in only one file, the associated entry is 1. If it
# appears in both files, the associated entry should be 2.
# Once we have "document frequency" (df) values, compute a "scaled inverse
# document frequency" like this:
#
#   idf = log( 1 + (N / df) )
#
# In this equation, N is the number of documents in the collection and log is
# used as a scaling factor. (This seems like a bit of black magic but has
# been proven to work well empirically by Salton et al.)
#
# (This number is important for large document collections. It allows less
# frequently occuring common terms to be weighted more heavily than those
# that appear in all documents.)
###
sub computeIDFs() {
    %idf = (); # for now
}
```

inverse document frequency

df ← how many documents in the file collection does the term occur in (1 or 2 for our program)

log (1 + (1/2))

log (1 + (2/2))

for every word in dict, this hash has a number

```
###
# SUBROUTINE: computeNFs --
# ----------------------
# INPUTS: %tf1 %tf2 %idf
# OUTPUTS: $nf1 $nf2
# Compute a "normalization factor" for each document so that each
# (multi-dimensional) term frequency vector will have length 1.0.
# This will ensure that identical documents have a similarity factor of 1.0
# and completely disimilar documents will have a similarity factor of 0.0.
#
# For each term frequency vector, multiply each term frequency (tf) by
# the inverse document frequency (idf) for that term. Square the result.
# Do this for every term in the vector and sum the results. The norm factor
# (nf) is the square root of the resulting sum.
#
#   nf = sqrt( SUM of all terms( square( tf*idf ) ) )
#
###
sub computeNFs(){
    $nf1 = $nf2 = 0.0; # for now
}
```

IMPORTANT

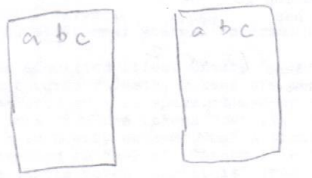
1 to ∞ — 1 or 2

```
###
# SUBROUTINE: computeWeights --
# ---------------------------
# INPUTS: @dict %tf1 %tf2 %idf $nf1 $nf2
# OUTPUTS: @w1 @w2
# This is the heart of the algorithm. Everything else is just preparation
# for building the weighted term frequency vectors.
#
# Build two lists of floating-point numbers. There will be one entry in each
# list for each term in the dictionary (document collection). These weights
# correspond to the terms in the dictionary (they are "parallel arrays").
# If a particular term does not appear in the document, the corresponding
# entry will be 0. Otherwise, the weight is the term frequency (tf) multiplied
# by the inverse document frequency (idf), divided by the normalization
# factor (nf) for that document.
#
```

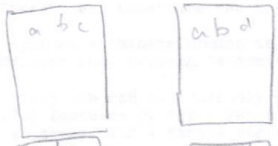
```
#   w = ( tf * idf ) / nf
#
###
sub computeWeights(){
    @w1 = @w2 = (); # for now
}

###
# SUBROUTINE: computeSimilarity --
# ------------------------------
# INPUTS: @w1 @w2
# OUTPUTS: $sim
# Just compute the "inner product" of the two weighted term vectors.
# This will effectively calculate the angle or "distance" between the
# two vectors in the multi-dimensional term space.
#
# Recall that the inner product is just the sum of the corresponding
# vector elements multiplied together.
###
sub computeSimilarity() {
    return 0.0; # for now
}
```

dot product

MUST RETURN 1 for two exactly same docs



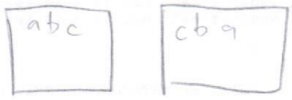
return 1.0



return < 1.0

return < 2nd

return 0.0



return 1.0

Program 2 -- OO Banking with C++

CS 3411 (Hutto) - Programming Language Concepts - Summer '99
DUE: 4 pm Monday 2 August (class time)

Got Questions? Try office hours, email or the newsgroup git.cc.class.3411

Object-Oriented Banking

For your **second programming assignment** this quarter, you will develop a banking transaction system in C++ with a simple textual interface. You have probably implemented or seen a similar application. The user enters lines of text to perform various banking transactions to open and close accounts, make deposits or withdrawls, and request current balances. The system supports multiple customers and account types. Customers can have one or more accounts. Different account types have varying restrictions and yield different rates of interest. Getting all of this working requires a certain attention to detail but it is straightforward.

Our challenge with this assignment will be deciding **how to best use the object-oriented capabilities of C++** to produce a clean, flexible and robust implementation. The familiarity of the problem domain will allow us to spend more time thinking about the architecture of our solution and how the programming language supports (or hinders!) our design.

Problem Description

The bank maintains a list of customers and accounts. There will be just one bank in this problem. Customers are identified by an arbitrary string containing no embedded whitespace. Valid names include: phil_hutto, joe123, ChrisBradfield, etc. Names like "phil hutto" are invalid. There are three types of accounts: checking, saving and moneymarket. Notice that there is no space in moneymarket and that saving is not plural. The bank is case-insensitive.

Customer and Account Numbers

When an account is first created for a named customer, the bank generates a unique customer number associated with that name. Customer numbers are unique digit strings. We also begin each customer number with the letter 'c' to make them easy to identify. Account numbers follow a similar convention but begin with the letter 'a'. The customer and account "namespaces" are distinct so there can be a customer c1 and an account a1 with no conflict. A cusomter can have many accounts. A single customer can even have two or more accounts of the same type. Customer names are used only to open accounts and to report account balances. All other transactions identify customers and accounts by number.

Account Types, Interest, Day Clock

Accounts earn interest, compounded daily. Checking accounts earn 1.5%,saving accounts earn 3% and moneymarket accounts earn 6%. To simplify date handling, assume the system maintains a "day clock" that starts at 1. Users manually advance the clock to simulate the passage of time. This simple scheme allows interest calculations to be performed each time the clock is advanced, based on the

ccount Restrictions

Different account types have different restrictions. Account balances are never allowed to drop below zero (overdrafts are not allowed). A $20 fee is charged for each withdrawl from a moneymarket account. Saving account withdrawls are limited to once every 7 days. This requires the system to record the last withdrawl date for saving accounts.

No Historical Data

To simplify the system, no historical data needs to be recorded other than the last withdrawl date. (Extensions might record all transactions, print statements, allow queries on historical data, etc.) The bank, does, however, need to keep a list of all active customers and accounts and maintain the current balance of each account (along with the day clock).

Input / Output

We will use a simple line-oriented textual interface for entering transactions. This portion of the system should
be designed in a modular way that minimizes dependencies and allows alternate transaction interfaces. For
example, transactions could be messages generated by an ATM or Web browser. We will not address security issues
in this assignment. Transactions requests will be entered via standard input. Input is case-insensitive and consists of whitespace-separated tokens. Requests begin with a single verb describing the action to be performed. Valid transactions include:

```
open
close
quit
withdraw
deposit
transfer
help
advance
```

balance

Most transactions require additional arguments. For example, deposit requires an account id and an amount (in that order). We assume each input line contains at most one transaction. Transactions cannot span lines. Blank lines are ignored. Amounts are specified as floating-point numbers. Assume the standard C++ format for floating-point values. All floating-point values should be rounded to the nearest penny for output.

Interest Calculation

Account balances are updated based on interest earned each time the clock is advanced using the following formula:

```
newBalance = oldBalance * ( 1 + interestRate / 365 * daysSinceLastUpdate )
```

Sample Session

The following is a sample run demonstrating a variety of commands and error conditions. Your output does not have to be identical but should provide the same information.

```
% oob

Welcome to First Polymorphism Savings & Loan!

cmd> help
>>     advance <n-days>
>>     balance <acct#> | <cust#>
>>     close <acct#> | <cust#>
>>     deposit <acct#> <amt>
>>     help
>>     open <acct-type> <cust-name> <init-deposit>
>>     quit
>>     transfer <from-acct#> <to-acct#> <amt>
>>     withdraw <acct#> <amt>
>> <acct-type> = checking | saving | moneymarket
>> <date> = <month>/<day>/<year>

cmd> open checking phil_hutto 450.50
>> c1 (phil_hutto) a1 (checking 1.5%) $450.50

cmd> open moneymarket phil_hutto 5000.00
>> c1 (phil_hutto) a2 (moneymarket 6%) $5000.00

cmd> open saving phil_hutto 800.00
>> c1 (phil_hutto) a3 (savings 3%) $800.00

cmd> open checking nick_black 100000.01
>> c2 (nick_black) a4 (checking 1.5%) $100,000.01

cmd> balance c1
>> c1 (phil_hutto) a1 (checking 1.5%) $450.50
>> c1 (phil_hutto) a2 (moneymarket 6%) $5000.00
>> c1 (phil_hutto) a3 (savings 3%) $800.00

cmd> advance 10
cmd> withdraw a1 210.25
>> c1 (phil_hutto) a1 (checking 1.5%) $240.44          /* 450.50 + 0.19 (interest) - 210.25 */

cmd> advance 12
cmd> transfer a4 a1 50000
>> c2 (nick_black) a4 (checking 1.5%) $50,090.42       /* 100000.01 + 90.42 - 50000 */
>> c1 (phil_hutto) a1 (checking 1.5%) $50,240.56       /* 240.44 + 0.12 + 50000 */

cmd> this is a stick up!
```

http://reality.sgi.com/austern/std-c++/faq.html#C8

STRING to INT
int n = atoi(s.c_str());

STRING to NUMBER
#include <strstream>
istrstream Iss(~~str~~ astring.c_str());
~~Iss >> Token;~~
float myFloat;
istrstream >> myFloat;

NUMBER to STRING (g++)
#include <strstream>
int i = 14;
ostrstream oss;
oss << i << ends;
cout << oss.str();

```
>> *invalid cmd: this is a stick up!

cmd> withdraw a9 123.45.67
>> *invalid acct: a9

cmd> withdraw a1 123.45.67
>> *invalid amt: 123.45.67

cmd> withdraw a1 10000.00
>> *insufficient funds

cmd> quit

%
```

Classes

Think of the bank as some sort of server that responds to transaction requests. Requests involve customers and accounts. There can be many customers and accounts so the bank needs to maintain a list (database) of customers and of accounts. Given a customer we need to be able to find the associated accounts. Given an account we need to find the associated customer.

We need at least three classes: Bank, Customer, Account. There will be a single instance of the Bank class for this problem (this is the Singleton Design Pattern). There may be many Customer and Account instances. The Bank instance will contain lists of Customers and Accounts. Since accounts (and hence customers) can come and go, we need a collection that supports addition and deletion. We are solving a small instance of this problem so we don't need to worry too much about lookup efficiency. The C++ Vector class from the Standard Library is a good way to implement a list.

The Bank object should accept and respond to all requests. Some requests it will handle directly such as advancing the day clock or opening and closing accounts. Most other requests will be handled by "forwarding" or "dispatching" the request to the appropriate Account object.

If we wish to emphasize the client/server architecture and isolate the Bank object from the precise format of the incoming requests (such "firewalls" are generally a good idea and help anticipate change), we can construct simple Request and Reply classes. The main() routine can give control to a simple Parser object (another Singleton) that repeatedly prompts the user for a command, reads the standard input a line at a time, validates the input and produces Request objects corresponding to correctly formed transaction requests. The Parser object handles invalid input, shielding the Bank object. In essence, the Parser directs a stream of correctly formed transaction requests to the Bank in the same way that a lexical analyzer produces a stream of lexemes for a programming language parser. Some errors, such as "*insufficient funds", can only be detected by the Bank object (ultimately by the Account object).

In summary, here is a short list of suggested classes:

Parser (singleton)
Request
Reply
Bank (singleton)

Customer
Account

We can further refine our design by detailing the properties (member dtaa) and behaviors (member functions) of each class and determining class and object interactions.

Three Architectures

With a rough idea of the architecture of our banking system, things begin to get interesting. There are three different types of accounts and more could easily be added. A good OO design would allow flexibility in this part of the system. Accounts share much in common but different account types yield different interest rates and have slightly different restrictions. For example, withdrawls are allowed from savings accounts only once within a seven day period. Where should the "logic" that distinguishes Account types reside?

One common approach would form an **inheritance hierarchy** of Account types. If you have CheckingAccount, SavingAccount and MoneyMarketAccount should the parent class Account be abstract or is there implementation logic that can be "factored" into the parent class? Do you need polymorphism?

Another approach would implement an Interest class and allow the interest calculation logic to be "mixed-in" to the appropriate account type via **multiple inheritance**.

A third possibility uses **containment and delegation** to encapsulate and vary the interest calculation logic and account restrictions. Implement a Calculator class that has methods for computing a new balance for a given transaction. The Calculator class would form a simple inheritance hierarchy with a different "calculator" object for each account type. Now each Account object can simply contain a pointer or reference to an appropriate Calculator instance. When the Account objects receives a transaction request it simply delegates the account-specific logic to its (contained) Calculator object. This approach is very flexible and allows the logic to be replaced "dynamically" during program execution.

Your job is to consider these alternatives and select and implement the one you find most appealing.

Turnin Details / Grading Criteria

Create a directory called p2. You should create separate interface (.h) and implementation (.cc) files for each class you define (Java style). Your directory should contain a README file describing and justifying your architecture and design choices and describing any additional capabilities you wish to be considered for extra credit. We will supply a sample Makefile to automate compilation.

Submit the entire directory via turnin by the due date. Identify the project as "`cpp`". So:

```
Login to acme
Say "workon 3411"
Say "turnin cpp p2"
```

Verify that your submission worked by using "`getback`" (in a separate, empty directory).

We are looking for both a working program and a clean object-oriented design that utilizes the object-oriented facilities in C++.

```
compiles without warnings 40%
compiles and runs 50%
compiles + runs + produces correct output 70%
compiles + runs + correct output + good design 90%
compiles + runs + correct output + good design + good style 100%
extra credit: up to an additional 20%
```

Note: You may develop your program using any standard-compliant C++ compiler but your program must be submitted via workon and must compile and run properly in the CoC environment using g++.

C++ Details

You should develop your programs using the new g++ compiler available on all the CoC departmental machines. Do not use the acme compiler. It is out of date and does not support some of the more advanced features we wish to use. For example, even the string class does not work properly on acme.You can ask g++ what version it is like this:

```
{lennon:pwh:412} g++ --version
egcs-2.91.66
```

You will need to use the standard string class and you will need to be able to do simple i/o using the standard streams cin and cout. For example, you read a line of text into a string (discarding the NEWLINE) like this:

```
string line;
cin.getline( line );
```

You should probably use the Vector class from the standard template library. Vector is "templated" so you should understand the basics of how templates work and that you need to "instantiate" the template by specifying a base class like this:

```
Vector<string> list;
```

You certainly need to know how to define and instantiate simple classes with constructors, data members and member functions. Label member functions const where appropriate. Use function overloading where appropriate. See the examples from class for details.

A good solution will probably include virtual functions and maybe pure virtual functions and abstract classes (interfaces). With virtual functions you introduce the possibility of polymorphism which you may find useful. Remember polymorphism only occurs in C++ when objects are allocated on the heap and accessed by pointers or references.

There are all sorts of interesting data structures available in the standard template library such as lists, maps and trees. You are welcome to use these but we won't go into any more detail describing them. Vectors are sufficient for this assignment. You may find multiple inheritance useful. You may use exceptions in your solution but you are not required to. We will discuss C++'s exception handling

echanism in detail later in the course.

One of the problems with C++ is that there is no "official" online documentation for the standard library like the Java API. There are lots of resource available online but none are "definitive". I have placed a detailed chapter describing C++ on electronic reserve at the library. You might also want to pick up a good C++ book. There are zillions of them. The "C++ Primer 3e" by Lippmann is pretty good. The "official" book by Stroustrup is a hard read and is not recommended for the faint-hearted. If someone finds a good online resource, please share via the class newsgroup.

Good luck! Start Early! Use Office Hours Often!

Program 3 -- Binary Tree Module in Prolog

CS 3411 - Programming Language Concepts - Summer '99

Due: 2:50pm Tuesday 10 August (Exam Time)

Got Questions? Try office hours, email or the newsgroup git.cc.class.3411

NO LATE PROGRAMS ACCEPTED!
GRADUATING SENIORS ARE EXEMPT FROM THIS PROGRAM!

Recursive Data Structures

For your final programming assignment this quarter, you will write a collection of rules for representing and manipulating binary trees in Prolog. You will use Prolog's two primary data structuring mechanisms for this assignment: **symbolic structures** and **lists**. Prolog structures are something like records in imperative programming languages but they are more powerful. Prolog lists are similar to lists in Lisp. Both structures and lists are fundamentally recursive. Structures are the more basic mechanism because the Prolog list notation can be viewed as "syntactic sugar" for a more primitive structure-based representation of lists.

Here is a fact represented by a simple Prolog structure:

```
book( moby_dick, author( herman, melville ) ).
```

This fact can be visualized as a binary tree but structures can, in general, have any number of members ("arity"). As usual, things get more interesting when we allow structures to contain logical variables. Structures with self-similar components are easily manipulated by recursive rules so we call them "recursive structures." Binary trees can be easily represented in Prolog by recursive structures. Here is a simple rule that defines binary trees in Prolog:

```
/** binary_tree(Tree) :-
     Tree is a binary tree. */
binary_tree(void).
binary_tree(tree(Element,Left,Right)) :-
     binary_tree(Left), binary_tree(Right).
```

Notice there are no constraints on the first component Element, it can be anything, but that the second and third components must themselves be binary trees. As a base case, void is considered a binary tree.

Given our recursive tree definition, we can pretty easily write a rule that determines if a given element appears anywhere in a given binary tree:

```
/** tree_member(Element,Tree) :-
    Element is an element of the binary tree Tree. */
tree_member(X,tree(X,Left,Right)).
tree_member(X,tree(Y,Left,Right)) :- tree_member(X,Left).
tree_member(X,tree(Y,Left,Right)) :- tree_member(X,Right).
```

This rule precisely and concisely encodes the three possible cases: either the element in question is the root or it appears in the left subtree or it appears in the right subtree.

Here is another recursive rule that implements a preorder traversal of

```
/** preorder(Tree,Pre) :-
```

```
    Pre is a list of elements of Tree accumulated during a
    preorder traversal. */
preorder(tree(X,L,R),Xs) :-
    preorder(L,Ls), preorder(R,Rs), append([X|Ls],Rs,Xs).
preorder(void,[]).
```

This rule is a bit more complicated and uses the Prolog list notation ([a,b,c]). Prolog has a special operator (|) (sometimes called "split") that can be used to break a list apart into its head and tail. The split operator can also be used to put a list together like CONS in Lisp. We will talk more about the list notation in class and you can read about it in your text and the supplemental handout.

Here is the rule for append:

```
/** append(Xs,Ys,XsYs) :-
     XsYs is the result of appending the lists Xs and Ys.
*/
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

What Am I Supposed to Do?

Implement the following rules:

1. `inorder(Tree,In)` :- In is a list containing an inorder traversal of the elements of Tree. Assume Tree is bound on input. In may be bound or unbound. If it is bound initially then the query is asking Prolog to VERIFY that In is an inorder traversal. Otherwise the query asks Prolog to GENERATE an inorder traversal.

2. `substitute(X,Y,TreeX,TreeY)` :- TreeY is the result of replacing all occurrences of element X in the binary tree TreeX by element Y. Assume X, Y, and TreeX are bound on input.

3. **(EXTRA CREDIT)** `subtree(S,T)` :- S is a subtree of T. A subtree of S is the tree defined by the root or some internal node and all of its descendants to the leaf nodes. If S and T are identical then S is a subtree of T (we are not asking about "proper" subtrees.) Assume T is bound on input.

4. `sum_tree(TreeOfIntegers,Sum)` :- calculate or verify that Sum is the sum of the integer elements in TreeOfIntegers. (Hint: You will need to use the "is" operator.) Assume TreeOfIntegers is bound.

5. **(EXTRA CREDIT)** `ordered(TreeOfIntegers)` :- holds if TreeOfIntegers is a binary search tree without duplicates. That is, for each node in the tree with element e, all the elements in the left subtree are smaller than e and all the elements in the right subtree are larger than e. (Hint: Define two helper rules ordered_right(X, Tree) and ordered_left(X,Tree) that hold if Tree is ordered and X is larger (smaller) than the largest (smallest) node of Tree.)

6. `tree_insert(X,Tree,Tree1)` :- Succeeds if Tree1 is an ordered tree resulting from inserting X into the ordered tree Tree. If X already occurs in Tree, then Tree and Tree1 are identical. (Hint: Four rules are enough.)

Details

Create a single file called `tree` that contains all the facts and rules for the assignment. "Consult" this file each time you start prolog. Begin with VERY SIMPLE RULES and gradually add complexity and more rules. The "development environment" for our version of Prolog is hard to love. Do the best you can with it. Brian has prepared a nice Web page with helpful hints. Study the demo programs we have provided carefully. These will help you "think recursively" in Prolog.

You will need to use **one trick** to make life easier while developing your programs. Since typing in complex tree structures is tedious and error prone, you should define a few "named trees" like this at the beginning of your tree module:

```
/* Define a few trees to work with. */
tree1(tree(a,tree(b,void,void),tree(c,void,void))).
tree2(tree(d,tree(e,void,void),tree(f,void,void))).
tree3(
        tree(   a,
                tree(   b,
                        tree(e,void,void),
                        tree(   f,
                                tree(g,void,void),
                                void
                        )
                ),
                tree(c,void,void)
        )
).
```

Make sure you get the commas and parentheses correct. Once defined you can use these named trees to test your rules like this:

```
prolog
 ?- [tree].  /* consult the file containing the tree facts and rules */
 ?- tree2(A), tree_element(e,A). /* does e appear in tree2? */
```

Turnin

Create a directory called p3. Your directory should contain a single file named "`tree`" with all the required Prolog rules and facts. In addition create a file named README with a paragraph or two documenting your program. Describe any extra credit work you have done in the README file.

Submit the entire directory via turnin by the due date. Identify the project as "cpp". So:

```
Login to acme
Say "workon 3411"
Say "turnin prolog p3"
```

Verify that your submission worked by using "getback" (in a separate, empty directory).

Helpful Advice

It takes awhile for the "Prolog way of thinking" to sink in. Work steadily and incrementally. Spend a bit of time on a regular basis working with the Prolog interpreter, reading reference material and working through the assignment. Ask questions when strange things happen. Stay-tuned to the newsgroup for related assignment announcements.

Good luck! Start Early! Use Office Hours Often!

· Extended BNF — EBNF (convienence notation) June 23, 1999

[<flag>] - square brackets optional

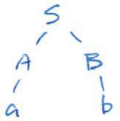
(+ | -) - choice

{ , <id> } - ∅ or more

{ , <id> }+ - 1 or more

L ⇐ a { , a }

Parse Trees vs. Derivations



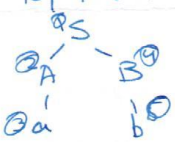
$S \Rightarrow AB \Rightarrow aB \Rightarrow ab$

$S \Rightarrow AB \Rightarrow Ab \Rightarrow ab$

Okay to have more than one derivation.
You're in trouble if you have more than one parse tree.

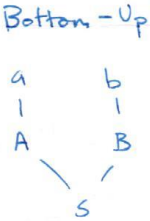
Parsing Styles - Top-Down
Bottom-Up

Top Down



apply production rules

Bottom-Up - complex



S ⇐ AB
A ⇐ a
B ⇐ b

YACC is a bottom-up parser

recursive descent parser - ~~write~~ each function deals with a nonterminal

```
void S() {
    A();
    B();

void A () {
  look for 'a'
     return normal
  else
     error;
}
void B() {
   look for 'b'
      return normal
  else
     error;
}
```

Algol 60

first for BNF

first for block structure

many other langs evolved from it

Naur wrote a journal about the development of Algol

* Machine independent
* 1st called IAL
* another version called JOVIAL

Backus - FORTRAN
McCarthy - LISP-58

Ada - Military
COBOL - Military

Perl - Walls
Java - Gosling
C++ - Stroustrup

* data types
* compound statement
* any number of array defs
* identifers with any num of chars
* recursion in imperative
* stack dynamic arrays
* confusing - BNF - NO I/O standard

CFG - Chomsky

E → + |
- |
T

T →

E → E+T | E-T | E

T → T*F | T/F | T

F → (E) | ID | NUM

2D ROW MAJOR EQ

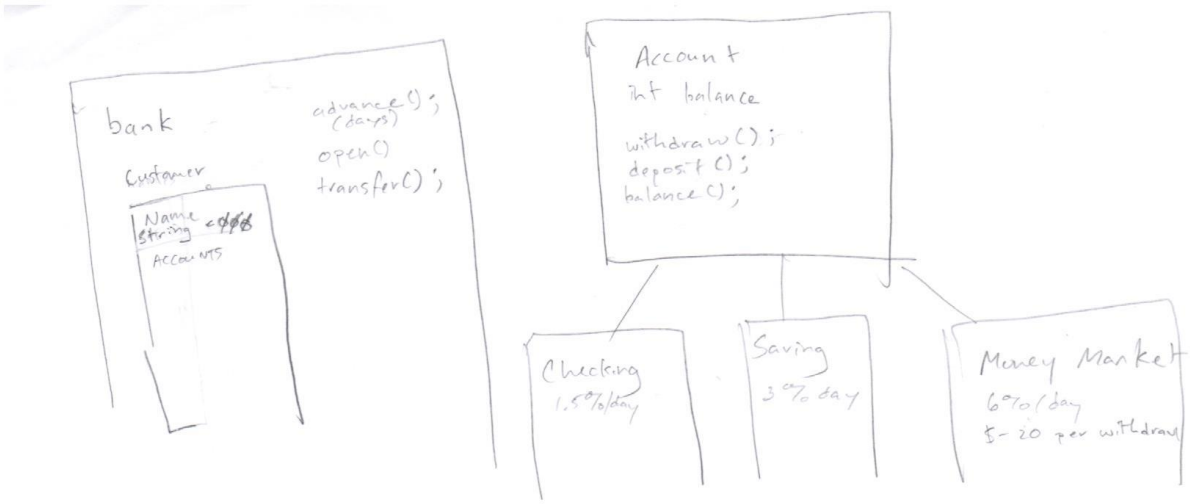
BASE + i * sizeof (ROW) + j * sizeof (element)
BASE + i * sizeof(element) * ROWS + j * sizeof(element)

1D BASE + i * sizeof (element)

Double Buffer

http://www.mrl.nyu.edu/perlin/java/BufferedApplet.html

三101
9001


bank
advance();
(days)
open()
transfer();
Customer
Name
string
ACCOUNTS
Account
int balance
withdraw();
deposit();
balance();
Checking
1.5%/day
Saving
3% day
Money Market
6%/day
$-20 per withdraw

4 3 2 1 3 6

b + (c = d / b++) - a || b

2 + (3 = 4 /

~~a = c, a, b +~~

b + (c = d / b++) - a || b

3 + (c = d / 3) - a || 3

3 + (c = 1) - a || 3

3 + 1 - a || 3

4 - a

1 || 3

3

a : 3
b : 3

c : 2
d : 4

0 43100 34026 3