Programming Languages

Sessions 11 & 12 – Main Theme
Declarative Programming
and
More on Scripting Languages

Dr. Jean-Claude Franchitti

New York University
Computer Science Department
Courant Institute of Mathematical Sciences

Agenda

1. Session Overview
2. Declarative Programming
3. More on Scripting Languages
4. Conclusion
What is the course about?

- Course description and syllabus:
  - [http://www.nyu.edu/classes/jcf/g22.2110-001](http://www.nyu.edu/classes/jcf/g22.2110-001)

- Textbook:
  - *Programming Language Pragmatics (3rd Edition)*
    - Michael L. Scott
    - Morgan Kaufmann

- Additional References:
  - Osinski, Lecture notes, Summer 2010
  - Grimm, Lecture notes, Spring 2010
  - Gottlieb, Lecture notes, Fall 2009
  - Barrett, Lecture notes, Fall 2008

Session Agenda

- Session Overview
- Declarative Programming
- More on Scripting Languages
- Conclusion
Session 9 & 10 Review

- Session Overview
- Control Abstractions
  - Generic Programming and Templates (C++, Java, ML, etc.)
  - Containers and Iteration
  - Exception Handling
  - Continuations
- Concurrency
  - Threads (Ada, Java, etc.)
  - Tasks synchronization
  - Communication
- Dynamic Allocation and Garbage Collection
  - Mark/sweep, copying, reference counting
- Conclusion
Agenda

1 Session Overview
2 Declarative Programming
3 More on Scripting Languages
4 Conclusion

Prolog

- Logic Programming
  - Theorem Proving
  - Program is a series of statements (axioms) and a goal
  - Execution attempts to prove goal
    - Reach goal from axioms
    - Uses inference
Horn Clauses

- A Horn clause is of the form
  - \( C \leftarrow B_1, B_2, B_3, \ldots, B_{n-1}, B_n \)
  - \( C \) is the **Head** or **Consequent**
  - \( B_i \) is a term of the **Body**
  - When \( B_i \) is true for all \( i = 0..n \), then \( C \) is true
- If \( C \leftarrow A, B \) and \( D \leftarrow C \) then \( D \leftarrow A, B \)
- Terms are usually parameterized predicates
  - \( \text{Rainy}(\text{Seattle}) \)
  - True iff it rains in Seattle
- If \( \text{flowers}(X) \leftarrow \text{rainy}(X) \) and \( \text{rainy}(\text{NY}) \) then \( \text{flowers}(\text{NY}) \)

Unification

- Process of substituting **expressions** for **variables**
  - A **constant** unifies only with itself
  - Two **structures** unify iff predicate is the same, the number of parameters is the same and corresponding parameters unify (recursively)
  - Variables unify with:
    - Values – variable instantiated with value
    - Other Variables – names alias each other
  - Unification also used in ML type inference
A Prolog Program

- A Program is a sequence of statements
  - rainy(seattle)
  - rainy(rochester)
  - cold(rochester)
  - snowy(X) :- rainy(X), cold(X)
- A Query is applied to the program
  - ?- snowy(C)
  - C = rochester
- Interpreter may return only the first or all values satisfying the query

Applications

- Limitations
  - Unification can take exponential time
  - Subset of first order logic
    - not is inability to prove, not false
- Can be used for
  - Problems for which efficient and straightforward procedural solutions are not available
  - Natural Language
  - AI and Expert Systems
  - Relationships (student of, ancestor of)
  - Relational Database-like applications
Instant Prolog

- Goal: use first-order logic as a programming language
- Data is set of axioms and rules of inference
- Desired output is theorem to be proved from the axioms
- Program is trace of proof.
- Completely declarative model: no sequencing, no control structures
- Unrealizable in practice: need to define operational semantics to guarantee termination, guide search, etc.
- Prolog is a plausible approximation to logic programming

What It Looks Like

- father (adam, abel). % an assertion
- father (adam, cain). % another one
- brother (X, Y) :- father (Z, X), father (Z, Y). % a rule
- ?:father (X, abel) . % query
- X = adam % response
- ?:father (adam, A). % bidirectional
- A = abel
- ?:brother (abel, cain). % nothing to instantiate
- yes
The Basics

- A **fact** is an assertion involving constants
- A **rule** is a universally quantified formula consisting of a head and a body.
- The **head** is a term involving variables
- The **body** is a conjunction of terms
- **Terms** consist of uninterpreted function names and their arguments, which are variables, constants, or terms (recursively)
- Intuitively: the head is true if there is an instantiation of the variable that makes the body true.
- For all X and Y: X is brother of Y iff there is a Z such that Z is father of X and Z is father of Y

Map Coloring in Prolog

- color (X) := member (X, [green, red, blue, yellow]).
- member (X, [X|_]).
- member (X, [_|Y]) :- member (X, Y)
- next (R1, R2) :- color (R1), color (R2), R1 \= R2.

Query:
- ?:- next (R1,R2), next (R1,R3), next (R1,R4), next (R2,R3), next (R2,R4), next (R3,R4)
- **Query embodies data**
**Basic Control Structure: Unification**

- Query matches a fact if there is an instantiation of the variables that can unify the query with the fact.
- `?:- father (X, abel) matches father(adam, abel)` with map `(X => adam)`
- If query matches the head of clause, must find unifier that satisfies all terms in the body.
- If several matches are possible, system will find one that works, if one exists: backtracking is built-in.
- If no match is found, query is not provable (not necessarily false).

**Operational Semantics**

- A rule is a procedure
- Several rules can have the same head (definition by cases)
  - `fact (N,F) :- fact (N, 1, F).`
  - `fact (N, L, F) :- N > 0, L1 is L*M, N1 is N-1, fact (N1, L1,F)`
  - `fact (0, F, F)`
- Each term in the body is a goal that must be matched
- Unification generalizes parameter passing
- Goals are attempted left-to-right
- The above is Prolog, not logic programming
### Prolog Example

- **Facts**
  - ?- [user].
  - shape(rectangle).
  - shape(triangle).
  - shape(circle).

- **Relations**
  - shape(square,X,Y) :-
  - shape(rectangle,X,Y), X=Y.
  - shape(rectangle,4,4).
  - | ? shape(square,4,4).
  - yes
  - | | ?- shape(X,4,4).
  - X = square ? ;
  - X = rectangle
  - yes
  - | | ?- shape(square,X,X).
  - X = 4

### Prolog Lists

- `[circle,square,rectangle,triangle]`
  - Head = circle
  - Tail = [square,rectangle,triangle]
  - |?- [X|Y] = [circle,square,rectangle,triangle].
  - X=circle
  - Y=[square,rectangle,triangle]
  - | ?-
  - [X,_,|Z]=[circle,square,rectangle,triangle].
  - X = circle
  - Z = [rectangle,triangle]
Suppose we know the average speeds of several automobiles on a particular racetrack and the amount of time they are on the track.

This basic information can be coded as facts, and the relationship between speed, time, and distance can be written as a rule.

speed (ford, 100).
speed (chevy, 105).
speed (dodge, 95).
speed (volvo, 80).
time (ford, 20).
time (chevy, 21).
time (dodge, 24).
time (volvo, 24).
distance (X, Y) :- speed (X, Speed),
time (X, Time),
Y is Speed * Time.

Queries can request the distance traveled by a particular car.
E.g., distance (chevy, Chevy_Distance).
would instantiate Chevy_Distance with the value 2205.

trace.
distance (chevy, Chevy_Distance).

(1) 1 Call : distance (chevy, _0) ?
(2) 2 Call : speed (chevy, _5) ?
(2) 2 Exit : speed (chevy, 105)
(3) 2 Call : time (chevy, _6) ?
(3) 2 Exit : time (chevy, 21)
(4) 2 Call : _0 is 105*21 ?
(4) 2 Exit : 2205 is 105 * 21
(1) 1 Exit : distance (chevy, 2205)
Appending to a List in Prolog

append ([ ], List , List ) .
append ([ Head | List_1 ] , List_2 , [ Head | List_3 ] ) :- append ( List_1 , List_2 , List_3 ) .

trace .
append ( [ bob , jo ] , [ jake , darcie ] , Family ) .

(1) 1 Call : append ( [ bob , jo ] , [ jake , darcie ] , _10 ) ?
(2) 2 Call : append ( [ jo ] , [ jake , darcie ] , _18 ) ?
(3) 3 Call : append ( [ ] , [ jake , darcie ] , _25 ) ?
(3) 3 Exit : append ( [ ] , [ jake , darcie ] , [ jake , darcie ] )
(2) 2 Exit : append ( [ jo ] , [ jake , darcie ] , [ jo , jake , darcie ] )
(1) 1 Exit : append ( [ bob , jo ] , [ jake , darcie ] , [ bob , jo , jake , darcie ] )
Family = [ bob , jo , jake , darcie ]
yes

Arithmetic

- Syntax: X is expression
  - X is then value of expression
  - Not: X=expression

| ?- 11 is 6+5.
  yes
| ?- X is 7/2.
  X = 3.5
  yes

- Area of a rectangle
  area(X,Y,A) :- rectangle(X,Y), A is X*Y.
  rectangle(2,3).

| ?- area(2,3,X).
  X = 6
More Definitions of Area

area(X, Y, A) :- rectangle(X, Y), A is X*Y.
area(X, Y, A) :- ellipse(X, Y), A is X*Y*3.14159.
ellipse(X, Y) :- circle(X, Y).
circle(X, Y) :- ellipse(X, Y), X = Y.
rectangle(2, 3).
circle(2, 2).

?- area(2, 2, A).
A = 12.56636

- What if we add rectangle(2, 2) ?

Prolog Recursion and Arithmetic

- Fibonacci Sequence

  fibo(0, 0).
  fibo(1, 1).
  fibo(N, X) :- N > 1, N1 is N - 1, N2 is N - 2,
              fibo(N1, X1), fibo(N2, X2), X is X1 + X2.
Not

- \(+\)
  - \(+\) x succeeds when x fails
  - Not the same as logical negation
- flies(x):-bird(x).
- But what about penguins?
- We do not want to define a rule for each bird individually, but want to exclude penguins from the flies relation.

flies(X) :- bird(X), \(+\) penguin(X).
bird(X) :- penguin(X).
bird(X) :- eagle(X).
bird(X) :- seagull(X).
penguin(pete).
seagull(sally).
eagle(earl).

More on Not

- What if we change our previous definition to:
  flies(X) :- \(+\) penguin(X), bird(X).
- And try:
  |? flies(X).
- What is the result?
Review of Various Programming Paradigms

- What are the following Prolog concepts?
  - Fact
    - Relationship that is unconditionally true
  - Rule
    - Inference stating that for any binding of variables, if all the relationships on the RHS are true, so is the relationship on the LHS
  - Query
    - Request for all the values of unbound variables that make the queried relationship true

- Why would I write a logic program rather than a corresponding Java program?
  - The logic program does not require the writer to navigate any data structures. The logic program changes very little when I change which element of a relationship is unknown. It is easy to add or change rules and immediately see the effect on the output.

- What is a transaction?
  - A collection of actions on a database that are executed (1) "serializably" – meaning that all transactions execute in a total order with no appearance of interleaving; (2) "all-or-nothing", meaning that if the transaction aborts before finishing, nothing is done; (3) reliably with respect to machine failures.

- Why might we use a transaction language instead of Java synchronized methods?

- What is local-remote transparency? How might it be implemented?

---

Agenda

1. Session Overview
2. Declarative Programming
3. More on Scripting Languages
4. Conclusion
Scripting Languages

- Typically used for short programs to manage other programs
- Interpreted, dynamically typed, permissive semantics
- May include Garbage collection
- Usually minimal declarations
- Usually rich set of string operations (the ultimate untyped data)
- Easy interface to OS, file and directory manipulation
- Specialized control structures:
  - regular expressions (Perl) – A traditional scripting language
  - dictionaries (Python) – A modern hybrid for scripting & prototyping

Prototyping

- Prototyping:
  - Quickly putting together a program that does a task in order to be able to experiment with different designs.
- Some desirable characteristics of a prototyping language:
  - interactive (like LISP, ML, etc)
  - garbage-collected, no pointers (LISP, ML, etc.)
  - fast compilation: minimal translation to bytecode (like JAVA)
  - can be slow
  - dynamic typing (like LISP, SETL, APL)
  - higher-order functions (LISP)
  - built-in indexable structures (like SETL)
  - built-in associative structures (like SETL)
Perl Overview

- PERL stands for *Practical Extraction and Report Language*
- A very successful scripting language, originally for systems administration
- excellent string manipulation facilities
  - regular expressions for string matching and substitution
- combines and surpasses sh/sed/awk
- compact, often cryptic syntax
- dynamically typed (with some minor static typing features)
- scoping: static and dynamic (both kinds!)
- built-in arrays/lists and maps
- a vast array of libraries available
- strongly imperative, but
- has first-class functions
- and some higher-order functions: map, grep, sort

Perl History

- (1987) PERL 0-3: test versions
- (1991) PERL 4: first public version
- (1994) PERL 5: various new features:
  - references
  - function prototypes
  - statically scoped variables
  - first-class functions, map
  - OOP
- The Future:
  - PERL 6 is currently being designed and implemented
    - see http://dev.perl.org/perl6/
  - a prototype is being written in Haskell (!)
    - see http://www.pugscode.org/
Some Perl Code

```perl
my $greeting = "Hello"
my $num = 2;
print $greeting, $num, "\n"; # 'Hello2' + newline
print "$greeting $num\n"; # 'Hello 2' + newline
print '\$greeting $num\n'; # '\$greeting $num\n'
my @words = ($greeting, $num, "\n");
print @words; # 'Hello2' + newline
```

- scalar variable names prefixed by $ (but this is not the whole story)
- list variable names prefixed by @
- variable interpolation within double-quoted strings, but not within single-quoted strings
- can declare variables using my
- print takes a list of arguments

A Simple Type Mode

- Atomic types: numbers and strings. Not much distinction between them.
- Standard operations use value semantics.

```perl
$count = 15;
$count++; # now $count == 16
$fullname = $first . $last; # concatenation
$fullname = "$first$last"; # string interpolation
$count = "count"; # dynamic typing.
$count = "15";
$count++; # now $count == "16"
```
Scoping

- to declare a *statically* scoped variable:
  
  ```perl
  my $a = 5;
  ```

- to “declare” a new *dynamically* scoped variable, just assign to it:
  
  ```perl
  $b = 6;
  ```

- to save the old value of a dynamically scoped variable and restore it at the end of the current scope:
  
  ```perl
  local $b = "hi";  # save old value of global $b, 
  # assign "hi" to it, 
  # and restore old value upon 
  # leaving scope
  ```

### Scoping Example

<table>
<thead>
<tr>
<th>Code</th>
<th>Output</th>
</tr>
</thead>
</table>
| ```perl
  sub f {
  my $a = "in f";
  print "f1: a = $a; b = $b\n";
  g();
  print "f2: a = $a; b = $b\n";
  }

  sub g {
  print "g1: a = $a; b = $b\n";
  my $a = "in g";
  $b = "in g";
  print "g2: a = $a; b = $b\n";
  }

  f();
  ```

  | f1: a = in f; b =  |
  | g1: a = ; b =      |
  | g2: a = in g; b = in g |
  | f2: a = in f; b = in g |
Arrays / Lists

- PERL has a data structure that subsumes arrays and lists
  - how to recognize a list variable: @a
  - how to write a list literal: ("a", 1, 2, "hello")
  - how to index a list: $a[4] (index starts at 0)
  - index of the last element: $#a
  - conversion to scalar:

    ```perl
    $numElems = @a;  # $numElems gets the number
                   # of elements in @a
    ```

- lists are automatically flattened.

  ```perl
  @XS = (1, 2, 3);
  @YS = (10, 20, 30);
  @ZS = ("a", @XS, "b", @YS);
  # @ZS is ("a", 1, 2, 3, "b", 10, 20, 30)
  ```

Hashes (Maps)

- PERL has an associative data structure called a hash.
  - how to recognize a hash variable: %w
  - how to write a hash literal:

    ```perl
    (cat => 5, dog => 22, parrot => 1)
    ```

  - how to look up a key in a hash: $w{dog}
  - how to add/modify a hash element: $w{dog}++
  - conversion to list:

    ```perl
    %w = (cat => 5, dog => 22, parrot => 1);
    @a = %w;  # @a gets ("cat", 5,
              "parrot", 1,
              "dog", 22)
    # key, val pairs in some order
    ```

  - conversion from a list: ("=>" is just an alias for comma)
  - hashes are flat – no hashes or lists as values in the hash
PERL 5 introduced a new kind of scalar: the reference.

- reference to an existing variable:
  - \$var (reference to a scalar)
  - @var (reference to an array)
  - %var (reference to a hash)
- a bit like & operator in C/C++
- creating a reference to a new value:
  - reference to a list: [1, 2, 3]
  - reference to a hash: { dog => 22, cat => 1, mouse => 0 }
Reference Derefencing

- if $v$ is a reference to a scalar, $$v$$ is the value of that scalar
- if $v$ is a reference to an array, @$v$ is the value of that array
- if $v$ is a reference to a hash, %$v$ is the value of that hash

- Accessing part of the aggregate:
  - if $v$ is a reference to an array, $v->[3]$ is the third element
  - if $v$ is a reference to a hash, $v->{dog}$ is the value associated with "dog"

Reference Testing

- ref $var$ is a string that describes the type of $var$.

```
my $i = 1;          # number (scalar)
my $rs = \$i;       # ref to a scalar
my $rr = \$i;       # ref to a ref to a scalar
my $ra = [ 1, 2 ];  # ref to an array
my $rh = { };      # ref to a hash

print ref $rs, "\n"; # SCALAR
print ref $rr, "\n"; # REF
print ref $ra, "\n"; # ARRAY
print ref $rh, "\n"; # HASH
```
Nested Data Structures

- PERL 4 only supported “flat” data structures:
  - lists of scalars
  - hashes of scalars
- References allow us to have nested data structures, e.g. lists of lists of hashes.

```perl
my $t = { vowels => [ "a", "e", "i", "o", "u" ],
          consonants => ... };
my @xs = (1..4);
my @ys = ("a", [ @xs, "b");
# @ys = ("a", [ 1, 2, 3, 4 ], "b")
```

File Reading

- To open a file:
  ```perl
  open (HANDEL, "water-music");
  ```
- To close a file handle:
  ```perl
  close HANDEL;
  ```
- To read one or more lines from a file handle:
  ```perl
  $v = <HANDEL>;  # scalar context: read one line
  @vs = <HANDEL>;  # list context: read all lines
  ```
  ```perl
  while (<HANDEL>) {
      # next line automatically read into $_
      ...
  }
```
Selection

if (condition1) {
    statement;
    ...
} elseif (condition2) {
    ...
} else {
    ...
}

- also has unless:
  unless (condition) {
    ...
}

- and there are the post-condition variants:

  statement if condition;
  statement unless condition;

Loops

my @a = (3, 5, 10, 2, 0, 12, 8);
while ($i < @a) {  # length of array
    print "a[$i] = @a[$i]\n";
    $i++;
    last if $a[$i] == 0;  # exit loop if element is 0
}

my $sum = 0;
foreach (@a) {  # each element gets assigned to $a
    $sum += $a;
}

for ($i = 0; $i < 10; $i++) {  # just like C
    ...
}
Functions

- arguments are passed by reference
- all arguments are passed as a single flat list of values, placed in global
- variable @_
- return value of the function is value of the last expression evaluated, or return expr can be used instead

```perl
sub findEq ($@) {   # optional 'prototype'
    # one scalar and one list argument
    # need to extract arguments manually:
    my ($a, @b) = @_;  
    my @result = (); 
    foreach (@b) {      
        push (@result, $_) if $a == $_;
    }
    return @result;
}

my @matches = findEq (34, 21, 13, 0, 20..40, 66);
```

Passing Functions

- Can also pass a reference to a function:

```perl
sub findEq ($@) {   # optional 'prototype'
    # one scalar and one list argument
    # need to extract arguments manually:
    my ($a, @b) = @_;  
    my @result = (); 
    foreach (@b) {      
        push (@result, $_) if &Sf($_);
    }
    return @result;
}

my @matches = findEq { sub { 34 == $_[0]; }, 21, 13, 0, 20..40, 66};

# we could use built-in grep instead:
my @matches = grep { 34 == $_ } (21, 13, 0, 20..40, 66);
```
### String Matching

- search for pattern in variable $v$:
  ```perl
  $v = m//pattern/;
  ```
- we can omit the variable:
  ```perl
  m//pattern/;  # defaults to $._
  ```
- we can also omit the m:
  ```perl
  /pattern/;  # same as above
  ```
- In all cases, returns true if the match succeeds.

```perl
my $text = "one word after another";
$text = m/r/;  # matches 'r' in "after"
$text = m/o.*n/;  # matches 'one word after an'
$text = m/...r$/;  # matches 'ther'
```

### Patterns (I)

- **Single characters:**
  - "ordinary" characters match themselves
  - *metacharacters* can be *escaped* by prefixing them with a backslash to make them ordinary; e.g. `\` matches the character \metacharacters:
    ```perl
    \{ } () {} * + . ^ $ | ?
    ```
  - `.` matches any single character
  - `[abc0-9]` matches any of the characters a, b, c, or any digit
  - `[^abc0-9]` matches any single character except a, b, c, or any digit
  - `\s` matches any space character, `\S` matches any non-space character

- **Alternation and sequencing:**
  - `abc|def|hij` matches any of abc, def, or hij
  - `p*` matches zero or more repetitions of p
  - `p+` matches one or more repetitions of p
  - `p{0,3}` matches from i to j repetitions of p
Patterns (II)

- Anchoring:
  - `^p` matches `p`, but only at the beginning of the line
  - `p$` matches `p`, but only at the end of the line

- Capturing:
  - `(p)` matches `p`, but also remembers the match for possible later use
  - `\3` matches the 3rd parenthesized pattern (which should precede this backreference)

Substitutions

```plaintext
$v =~ s/pattern/substitution/;
```
- searches for pattern in variable `$v` and replaces it with substitution

- substitution can be either a (possibly empty) plain string, or it could contain variables; in particular, `$1`, `$2`, ...
- These are the 1st, 2nd, ... captured matches.

Examples

```plaintext
my $t = "The bear ate the cat";
$t =~ s/(\w+) ate the (\w+)/$2 was eaten by the $1/;
print "$t\n";  # prints "The cat was eaten by the bear"

my $x = "abddddef";
$x =~ s/(.)/1/$1/g;  # now $x is "abddef"
```
## Agenda

<table>
<thead>
<tr>
<th></th>
<th>Session Overview</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Declarative Programming</td>
</tr>
<tr>
<td>3</td>
<td>More on Scripting Languages</td>
</tr>
<tr>
<td>4</td>
<td>Conclusion</td>
</tr>
</tbody>
</table>

## Assignments & Readings

- **Readings**
  - See “handouts” on the course Web site
- **Assignment #6**
  - See Programming Assignment #4 posted under “handouts” on the course Web site
  - Ongoing - Due on May 19, 2011
Course Logistics

- The final will be held on Monday, December 20th, 2010 in this room.
- Open book – any kinds of written or printed notes or books permitted. No computers.

The Final Exam

- Not deep
- Not like programming projects
- Anyone who understands all the classes and could do homework problems should get full marks
- Short answers
  » What does this program output?
  » Where is the bug in this program? Fix it.
  » What is the most general type of this ML function?
  » Small amount of programming
- Short discussions – sentence/paragraph/code excerpt
  » Why is this feature useful? Why is it potentially dangerous?
  » How do you implement this feature?
Languages in General
Language Syntax
The First High-Level Programming Language
Imperative Languages
Applicative Languages (LISP, Scheme, ML)
Simulating Objects with Closures
ML Type Inference Problem
Types
Object-Oriented Programming
Mechanisms for Object-Oriented Programming
Generics
Concurrency
Other Programming Paradigms
One word, two different meanings – the word “serialization”

Lexical and dynamic scoping
Memory management:
  » Structure of activation records on the control stack.
  » Procedure calling and returning protocols
  » Locating variables and objects at run time.
  » Garbage collection.
Parameter passing: Pass by value, pass by reference, pass by value-restore.
Macros and inline compilation.
Types (details):
  » Static vs. dynamic typing.
  » Base types and constructors.
  » Variant records
  » User defined types
  » Subtype relation
  » Casting, coercion, conversion, overloading.
  » Implementation of various types: records, arrays,
  » objects, pointers.
Dangling pointer problem.
Object-oriented programming (details)
  » Method inheritance
  » Static and dynamic dispatching. Viables.
  » Abstract methods and interfaces.
Programming in the large. Modules (packages).
Information hiding. Abstract data types.
Concurrency and synchronization. (no questions about the implementation.)
Generics/templates and type polymorphism.
Exception handling. (I will not ask about the implementation.)
Passing functions as parameters in C++, Scheme, ML.
Functional programming in Scheme and ML.
Functions as first-order objects.
Recursive programs
Lists in Scheme and ML.
Languages in General

- What benefits do high level languages achieve?
- What are the criteria for “good” decomposition into modules?
- What is the difference between an implementation and a specification?
- What is performance transparency? How is it good and how is it bad?
- What are name spaces? What are defining and applied occurrences of names? Illustrate in various languages. Be able to say for a given language what the scope of a name in it is.

Language Syntax

- Context-free grammars and BNF:
  - what are
    - Tokens
      - strings that are the “units” of parsing: identifiers, literals, punctuation
    - Terminals
      - nodes in the grammar that accept tokens and do not reduce
    - Non-terminals
      - nodes in the grammar that have rules reducing them to other nodes
    - Syntax rules
      - definitions of how a grammar node reduces to another node
    - Syntax tree
      - the result of parsing, grouping all nodes according to the reductions specified in the syntax rules
  - Given a BNF grammar and a string, be able to say whether the language accepts the string, and if it does, what the (concrete) syntax tree is.
  - Be able to define an “abstract” representation of a program
    - By eliminating redundant productions, and unnecessary punctuation
  - What does it mean for a context-free grammar to be ambiguous?
    - There exists a string of tokens that has more than one legal syntax tree

- Regular expressions:
  - Be able to state whether a RE accepts a string, or how to modify an RE to accept a certain kind of string
The First High-Level Programming Language

- What are Fortran’s types?
  - (Fixed, float, arrays)

- What are Fortran’s name spaces
  - Variables are local to the procedure; common blocks and subprogram names are global; all are static.

- How does Fortran pass arguments?
  - By reference

- What is a “secure” language? In what ways is Fortran not secure?
  - Every program either (a) raises a compile-time error; (b) compiles and produces an answer; (c) compiles and produces a valid run-time error.
  - Fortran is not secure because of lack of bounds checking, and lack of type checking of COMMON and EQUIVALENCE, that can cause storing values of the wrong type, and because by-reference passing can cause bizarre behavior such as overriding of constants.

- What can interfere with invariant checking in Fortran? How are later languages better? worse?
  - GOTO statements make it hard to judge all the ways control can reach a program point
  - If I call a procedure, it might not only update my parameters, by also any COMMON storage.
  - If I am a called procedure, it is possible that two of my parameters are aliased.
  - Later languages have structured control flow, declare constants/in/out/inout
  - But, they have pointers, multi-tasking, inner procedures, which add additional opportunities for killing invariants.

Imperative Languages

- What are:
  - Stack
  - Heap
  - Static Storage
  - Static Bounds
  - Dynamic Bounds
  - What’s the difference between an array with dynamic bounds and a varying-sized array?

- What’s the difference between a local variable and an own variable?

- Understand the difference between
  - Call by reference
  - Call by value
  - Call by name
  - Call by value-result
Imperative Languages (continued)

- What are invariants? How can I analyze a program at compile-time to check them?
- What are global variables? Why are they harmful? What’s a better way?
- What are nested blocks and procedures? What benefit do they provide? What risks do they have?
- What does it mean for a language to support first-class procedure variables? What is an example of how it’s useful?
- What is a closure? What is an example of using a closure? Under what circumstances can using a closure in some imperative languages be unsafe? How is it represented at runtime?
- What’s the difference between C struct types and higher-level language records? Give examples of each.
- What are exceptions? Why are they preferable to using return codes?

Applicative Languages (Lisp, Scheme, ML)

- What is an applicative language?
- What is a lambda expression?
- What is the difference between dynamic versus lexical scope of binding? Be able to understand a case where the two would lead to different results.
- Why did the applicative community realize that lexical scoping was better?
- What are let and letrec?
- Be able to translate a simple imperative program to applicative:
  - Use lambda binding instead of assignments
  - Use recursion instead of loops
- Be able to translate a simple program into continuation-passing style – e.g. to avoid multiple outputs, or to avoid exceptions
- Be able to “simulate” objects with closures
- Be able to manipulate LISP’s favorite data type – the list
- Understand functionals like mapcar
- What are macros and what is referential transparency?
- What is eval? When might you want to use it?
- What is garbage collection? Why is it needed? How is it done? Why does it work in Lisp or Scheme? Why does it work in Java? Why can’t it work in C?
- What is parametric polymorphism? Let polymorphism?
- Be able to give the most general type for an ML function.
Simulating Objects with Closures

- Simple case – an object with one method:
  
  ```lisp
  (define make-closure (lambda (lst)
    (letrec (l lst)
      (getBefore (lambda (l e)
          (cond (eq (cadr l) e)
               (car l) (getBefore (cdr l) e))))
              (lambda(e) (getBefore l e))))
  ```

- Fancier case: use continuations:
  
  ```lisp
  (define make-closure (lambda (lst cont)
    (letrec (l lst)
      (getBefore (lambda (l e)
          (cond (eq (cadr l) e)
               (car l) (getBefore (cdr l) e))))
              (theMethod (lambda(e)
                  (getBefore l e))
              (anotherMethod (lambda...
                              ...))
                (cont theMethod anotherMethod) ))
  ```

ML Type Inference Problem

- Type inference:
  
  ```lisp
  fun zip f nil nil = nil |
  zip f (h::t) (i::s) = f(h,i)::zip f t s;
  ```
  
  What does it do?
  Takes two lists of equal length, and applies a binary operator to the corresponding members of each, to produce a new list.
  
  What is its most general type?
  ```lisp
  ('a * 'b) -> 'c * 'a list * 'b list -> 'c list
  ```

- Consider these two expressions
  
  (1) fun f g = g 7 + g false;
  (2) let val g = fn (x) => 20 in g 7 + g false;
  
  Why is (1) incorrectly typed and (2) correctly typed?
  Because int->int can’t be unified with int->bool in (1). The type of g has to be resolved knowing only the expression using g.
  But in (2), we know the type of g also from the binding.
  
  What is the type of g in (2)?
  It is 'a -> int
Types

- What is an enumeration type? What is the advantage of using that over an integer?
  » A type with a fixed set of named values.
  » You can't mix enums from one flavor with enums from another; you can document their type; you can iterate over the set.
- What is the difference between name equivalence and structural equivalence?
- What is a union type? How did Algol make them safe? How does Ada make them safe? How would you do them in Java?
  » A value can be from one of a fixed set of types
  » In Algol, you can only access a variant case in a case conformity clause.
  » In Ada, the case is a discriminant. Unconstrained variants: only whole-record assignment is possible, and case is tested; constrained variants: case is known.
  » In Java, the union can be a superclass, each case a subclass; casting and instanceof check case
- What are Ada discriminants?
- What pointer arithmetic do C and C++ allow?
- What are dope vectors? Why are they used? What information would they contain?
- What do strongly typed languages use to escape strong typing? Why would that be needed? How can that be made safe?
- What are dangling references? How are they avoided?
  » Garbage collection
  » Tombstones
  » Keys/locks
  » Reference counts
  » Destructors
- What are destructors? Why are they needed?

Object-Oriented Programming

- How does object-oriented programming contribute to modularity as Parnas has described it?
- Define and distinguish:
  » Information hiding
  » Polymorphism
  » Inheritance
- Distinguish between classes and types (Java).
  » Note: This isn't always done consistently in the programming languages community. Ada types, for example, are more like classes.
Mechanisms for Object-Oriented Programming

- In Ada, there are three components:
  - Packages
  - Private parts of packages
  - Package bodies
- What goes in each?
- What needs to be compiled together with user in Ada? What doesn’t need to be compiled together with user?
- Distinguish between static dispatch and dynamic dispatch. When do you get each in: Ada, C++, Java?
- What’s the difference between a class extending another class and a class implementing an interface?
- Initialization, finalization, controlled. How does Ada get around the lack of initializers?

Generics

- Basic difference between Java, C++, and Ada styles
- Benefits and limitations of Java style
- When do you use bounded types in contracts – e.g., `<T extends X>`?
  - When the generic method needs to call a method in X
- When do I use wildcard types in interfaces e.g. `List<? extends Foo>`
  - When a list of any subclass of Foo will work; I’m reading Foo’s from them.
- What are the type-checking rules for parameterized types? Can I assign `List<ColoredPoint>` to variable of type `List<Point>`? Give reasons.
  - No. But you may assign `List<ColoredPoint>` to a variable of type `List<? extends Point>`
    - There may be, and in this case, there is, an operation add. You don’t want to add a Point to a variable of type `List<Point>` that actually has a value of class `List<ColoredPoint>`
- When do I use generic methods – e.g.,
  - `public <X> List<X> makeSingle(X element);` When there is a constraint between the types, in this case, the element type and the List type.
- What different things can I do with Ada generics?
Concurrency

- What are safe, regular, and atomic registers? What read results are possible for a given sequence of writes for each kind?
- What is a test-and-set or compare-and-swap instruction?
- What is a semaphore? What are the P and V operations? How can they be implemented using compare-and-swap?
- What is a critical region?
- How are critical regions used for safe programming?
- How are producer-consumer queues used for safe programming?

Concurrency (continued)

- What is a monitor?
- How is that implemented in C++, Java, Ada?
- Understand these concepts:
  » Synchronized method (Java)
  » Ada entry
  » Rendezvous call
  » Guarded select statement
- How are concurrent, conditionally available resources programmed in Java, Ada?
- Be able to write a guarded select in Ada, or a wait/notify in Java, and to spot and correct bugs in these programs
- What is a memory model? What are the advantages of a memory model with strong guarantees? What are the disadvantages?
- How can you assure thread safety by programming patterns? By compile-time enforcement?
Other Programming Paradigms

- What are the following Prolog concepts?
  - Fact
    - Relationship that is unconditionally true
  - Rule
    - Inference stating that for any binding of variables, if all the relationships on the RHS are true, so is the relationship on the LHS
  - Query
    - Request for all the values of unbound variables that make the queried relationship true
- Why would I write a logic program rather than a corresponding Java program?
  - The logic program does not require the writer to navigate any data structures. The logic program changes very little when I change which element of a relationship is unknown. It is easy to add or change rules and immediately see the effect on the output.
- What is a transaction?
  - A collection of actions on a database that are executed (1) "serially" — meaning that all transactions execute in a total order with no appearance of interleaving; (2) "all-or-nothing," meaning that if the transaction aborts before finishing, nothing is done; (3) reliably with respect to machine failures.
- Why might we use a transaction language instead of Java synchronized methods?
- What is local-remote transparency? How might it be implemented?

---

One Word, Two Different Meanings – The Word “Serialization”

- In databases:
  - The property of transactions that they appear as if they had executed in a serial order, that is, without overlap
  - Part of the requirement for atomicity
- In distributed RPC/RMI/message passing:
  - A technique for converting a message or argument list containing a set of typed objects on one machine into a bitstring that can be sent to another machine, parsed, and reconstructed as a corresponding message or argument list in another machine
  - Also called marshalling, and its counterpart demarshalling
Next Session: End of Course Material