CS 230
Programming Languages

12 / 01 / 2022

Instructor: Michael Eckmann

Today's Topics

- Questions? / Comments?
- Chapter 11 & 12 --- Abstract Data Types / Encapsulation / Object Oriented Programming Language features
- Chapter 16 – Logic Programming Languages

Anyone know the key features required of a language to be considered Object-Oriented?

Abstraction/Encapsulation/OOP

- Chapters 11 & 12
  - Abstraction
  - Information Hiding
  - Object Oriented Programming Language features

Chapter 11 & 12

- Abstraction
  - Process abstraction
    - Making a method/function/subprogram to handle some task
  - Data abstraction
    - Records (data only) OR abstract data types (data and member methods)
    - Example: floating point

Chapter 11 & 12

- Information hiding
  - is a key concept of data abstraction
- User defined ADTs
  - representation of the type is hidden from rest of program that uses them = Information Hiding
  - declarations of the type and ops/methods contained in a single unit = Encapsulation
    - e.g. a class
    - one can then create variables of these types

Chapter 11 & 12

- Parameterized ADTs
  - the ability to create a class that can hold data of any appropriate type (to be specified when used)
  - e.g. create a stack class that can hold any kind of data (e.g. stack of ints, stack of strings, etc.)
  - In C++ this is handled via templates
  - In Java
    - Pre 5.0 – used Object as the type (need to cast to actual type to use)
    - Starting in 5.0 – uses generics (type specified in angle brackets, no casting needed), better design

Chapter 11 & 12

- Benefits of information hiding
  - Reliability
  - reduction of what the programmer needs to be aware of can change the implementation
    - (e.g. stack may have been implemented as a linked list but then say you need to change it due to memory problems --- can use a contiguous data structure, an array) and the clients of the stack do NOT need to update any of their code ...

Chapter 11 & 12

- A language that is an OOPL must contain support for
  - ADTs
  - Inheritance
  - Dynamic binding of method calls to methods

Chapter 11 & 12

- Inheritance
  - What do we know about inheritance?
- Dynamic binding of method calls to methods
  - What do we know about dynamic method binding?

Logic Languages

- Logic (or Declarative) languages
  - Express programs in a form of symbolic logic
  - Produce results via logical inferences
- Logic programs
  - specify the desired results
  - instead of creating the procedures (or functions) to produce them
- Since we are writing no procedures, the syntax of logic languages is quite different from imperative (e.g. C, C++, Java) or even Functional (e.g. LISP, Scheme)

Logic Languages

- Logic languages are based on formal logic
- Prolog is a Logic Programming Language
- Prolog was developed by people with interests in
  - Natural language processing
  - Automated theorem proving

Logic Languages

- Symbolic logic can be used for the basic needs of formal logic:
  - express propositions (defined on next slide)
  - express relationships between propositions
  - describe how new propositions can be inferred from other propositions
- Particular form of symbolic logic used for logic programming called predicate calculus

Logic Languages

- Some terminology
- Proposition --- logical stmt that may or may not be true
    - Represented by simple terms.
    - Terms are either constants or variables
    - A constant represents an object
    - A variable can represent different objects at different times
- An atomic proposition consists of compound terms
  - Compound term – one element of a mathematical relation --- looks like mathematical function notation with names and parentheses.
    - Two parts are:
      - Functor
      - Ordered list of parameters

Logic Languages

- Compound term
  - The compound terms are also called tuples. 1-tuples, 2-tuples, 3-tuples, 4-tuples ...
    - Depending on the number of parameters
- Compound term examples:

student(jon)

like(seth, OSX)

like(nick, windows)

like(jim, linux)

- The meaning of the relations (i.e. student and like) are up to the programmer.

Logic Languages

- The meaning of the relations are up to the programmer.
- For instance
  - like(jim, linux)
    - Could mean jim likes linux. Or it could mean jim is similar to linux. Or it could mean something else entirely.
- There is no intrinsic semantics.

- If the relations are named well though, the programs obviously become more readable.

Logical connectors

Name	Symbol	Example	Meaning
negation	$\neg$	$\neg$ a	not a
conjunction	$\cap$	a $\cap$ b	a and b
disjunction	$\cup$	a $\cup$ b	a or b
equivalence	$\equiv$	a $\equiv$ b	a is equivalent to b
implication	$\supset$ $\subset$	a $\supset$ b a $\subset$ b	a implies b b implies a

Logic Languages

- Logical operators
  - Conjunction (and) is like intersection among sets.
  - Disjunction (or) is like union among sets.
- Implication
  - a implies b can be read as "if a, then b"

Logic Languages

- Precedence:
  - Negation
  - Conjunction, Disjunction, Equivalence
  - Implication

Logic Languages

- So far we've seen only constants in these propositions, e.g. jim, linux, etc.
- To use a variable in a proposition we need to preface it with either “for all” or “there exists”
- These are quantifiers
  - For all is represented by an upside-down A
  - There exists is represented by a backwards E

- Predicate calculus consists of the logical connectors we just saw in addition to the two quantifiers above.

Logic Languages

- For Predicate Calculus, (that is using the logical connectors and the quantifiers) it should be obvious that there are many ways to state the same thing
- Anyone want to give an example?

Logic Languages

- For Predicate Calculus, (that is using the logical connectors and the quantifiers) it should be obvious that there are many ways to state the same thing
- Anyone want to give an example?
  - Any of DeMorgan's laws
  - e.g. not (a and b) = (not a) or (not b)
  - Let me write up the truth table showing this equivalence

Logic Languages

- Because there are many ways to state the same thing, a standard form for propositions was decided
- Clausal form:

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

means if all the As are true, then at least one B is true

- Antecedent: right side
- Consequent: left side

Logic Languages

- The A's and B's in the clausal form can be variables or constants.
- When they are variables, the “for all” quantifier is implied.
- Note: the meaning of variable here is different than the meaning we associate with variables in imperative languages.
- Clausal form can be used exclusively because:
  - All predicate calculus propositions can be algorithmically converted to clausal form. So, we do not lose generality by restriction to this form of propositions.

Logic Languages

- Some examples:

$C \subset A \cap B$

ForAll X.(mammal(X) $\subset$ dog(X))

ThereExists X.(father(bob,X) $\cap$ female(X))

  - Taken separately, read each of the above in English.

Logic Languages

- Some examples:

$C \subset A \cap B$

if both A and B are true, then C is true.

ForAll X.(mammal(X) $\subset$ dog(X))

All dogs are mammals

ThereExists X.(father(bob,X) $\cap$ female(X))

bob has a daughter

Predicate Calc and proving theorems

- One use of propositions is to discover new theorems that can be inferred from known theorems
- Resolution: an inference principle that allows inferred propositions to be computed from given propositions
- A simple example:

older(joanne, jake) $\subset$ mother(joanne, jake)

wiser(joanne, jake) $\subset$ older(joanne, jake)

New propositions can be inferred from resolution:

wiser(joanne, jake) $\subset$ mother(joanne, jake)

Predicate Calc and proving theorems

- The presence of variables in propositions requires resolution to find values for those variables that allow the matching process to succeed. This is called unification.
- Instantiation is the temporary assigning of values to variables.
- During some resolutions, instantiating some temporary values might cause the matching process to fail, at which time, backtracking must occur to instantiate a different value to the variable.

Predicate Calc and proving theorems

- One possible way to prove a theorem is by contradiction.
- Make a proposition that is the theorem negated.
- The theorem is then attempted to be proved by finding an inconsistency.

- This is a theoretically useful way to prove theorems, but in practice the time it takes might be too long.
- Theorem proving is the basis for logic programming so we need a faster way for it to be able to do it.

Predicate Calc and proving theorems

- A restricted kind of clausal form is used when propositions are used for resolution. These special kinds of propositions are **Horn clauses**
  - Only 2 forms
  - A single atomic proposition on left side
  - Or
  - Empty left side
- Left side is called the *head*. If left side is not empty the Horn clause is called a *headed Horn clause*.
- Anyone recall what an atomic proposition is?

Predicate Calc and proving theorems

- An atomic propostion is a functor and a set of parameters in parentheses.
  - e.g. mother(joanne, jake)
- A headed Horn clause example:

  older(joanne, jake) $\subset$ mother(joanne, jake)
  - these state relationships among relations

- What do you think the purpose of headless Horn clauses might be? e.g. mother(joanne, jake)
- Not all propositions can be stated as/converted to Horn clauses, so there is a loss of generality.

Overview of Logic Programming

- Logical programming is nonprocedural
  - Programs do not state how a result is to be computed, but rather the form of the result
- We don't tell the computer how to get a result. Instead we state the form of the result and let the logic programming "engine" determine how to get that result.
- We provide the relevant information via the predicate calculus and the computer infers the results using resolution.

Overview of Logic Programming

- Describe the characteristics of a sorted list, not the process of rearranging a list

sort(old_list, new_list) $\subset$
permute (old_list, new_list) $\cap$ sorted (new_list)

sorted (list) $\subset$ ForAll j such that $1 \leq j < n$, list(j) $\leq$ list (j+1)

- This says that if a list is in non-decreasing order, it is sorted. Also, if the old_list is a permutation of the new_list and the new_list is sorted, then that's what it means to sort an old_list to be the new_list

Prolog

- Our text uses the Edinburgh Syntax version of Prolog
- Term: a constant, variable, or structure
- Constant: an atom or an integer
- Atom: symbolic value of Prolog
- Atom consists of either:
  - a string of letters, digits, and underscores beginning with a lowercase letter
  - a string of printable ASCII characters delimited by apostrophes

Prolog

- Variable: any string of letters, digits, and underscores beginning with an uppercase letter
- Instantiation: binding of a variable to a value
  - Lasts only as long as it takes to satisfy one complete goal
- Structure: represents atomic proposition

functor(parameter list)

Prolog

- Fact statements
  - Headless Horn clauses

```
student(jonathan).
brother(tyler, cj).
```

- Rule statements
  - Headed Horn clauses
  - Right side: antecedent (***if*** part)
    - May be single term or conjunction
  - Left side: consequent (***then*** part)
    - Must be single term
- Conjunction: multiple terms separated by logical AND operations (implied)

Prolog

```
parent(kim,kathy):- mother(kim,kathy).
```

- Can use variables (universal objects) to generalize meaning:

```
parent(X,Y):- mother(X,Y).
sibling(X,Y):- mother(M,X),
    mother(M,Y),
    father(F,X),
    father(F,Y).
```

Prolog

- GOAL STATEMENTS
- For theorem proving, a theorem is in the form of a proposition that we want the system to prove or disprove
- Same format as headless Horn

```
student(james)
```

  - this asks if james is a student, it either fails (james is not a student) or it doesn't (james is indeed a student)

- Conjunctive propositions and propositions with variables are also legal goals

```
father(X,joe)
```

  - this asks for all fathers of joe, if any, otherwise it fails.

Prolog

- INFERENCING
- Queries are called goals
- If a goal is a compound proposition, each of the facts is a subgoal
- To prove a goal is true, the system must find a chain of inference rules and/or facts. For goal Q:

```
B :- A
C :- B
...
Q :- P
```

- Process of proving a subgoal is called **matching, satisfying,** or **resolution**

Prolog

- POSSIBLE WAYS TO INFER
- Bottom-up resolution, forward chaining
  - Begin with facts and rules of database and attempt to find sequence that leads to goal
- Top-down resolution, backward chaining
  - begin with goal and attempt to find sequence that leads to set of facts in database
- Prolog implementations use **backward chaining**

Prolog

- POSSIBLE WAYS TO INFER
- When goal has more than one subgoal, can use either
  - *Depth-first search*: find a complete proof for the first subgoal before working on others
  - *Breadth-first search*: work on all subgoals in parallel
- Prolog uses **depth-first search**
  - Can be done with fewer resources so it is preferred for use in Prolog

Prolog

- Backtrack during inference
- For a goal with multiple subgoals, if fail to show truth of one of subgoals, reconsider previous subgoal to find an alternative solution: backtracking
- Begin search where previous search left off
- In worst case, can take lots of time and space because may find all possible proofs to every subgoal

Prolog

- Recap: When trying to prove a goal, Prolog uses
  - backward chaining (starting at goal) to try to find facts that satisfy it
  - depth first search when satisfying a chain of subgoals
  - backtracking during inference to possibly get an alternative solution to an earlier subgoal

Prolog

- Prolog also supports integer variables and integer arithmetic
- ***is*** operator: takes an arithmetic expression as right operand and variable as left operand
- A is B / 10 + C
- Not the same as an assignment statement
- B & C must have been instantiated and A must not have been instantiated for that expression to have A take on the value of the expression.
- But if A is instantiated prior to this expression or B or C are not instantiated, then the clause is not satisfied and no instantiation can take place.
- Example: Sum is Sum + Number. Is this possible?

Prolog

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

- Here, the facts are the first six statements and the last is a rule stating the relationship between time, speed and distance.
- Example query: distance(chevy, Chevy_Distance)
- Chevy_Distance is instantiated with the value 2205

Prolog

- TRACE
- Tracing model of execution - four events:
  - Call (beginning of attempt to satisfy goal)
  - Exit (when a goal has been satisfied)
  - Redo (when backtrack occurs)
  - Fail (when goal fails)

Prolog

```
likes(jake,chocolate).
likes(jake,apricots).
likes(darcie,licorice).
likes(darcie,apricots).

trace.
likes(jake,X),
likes(darcie,X).
```

This is asking, what is X such that both jake and darcie like X

Let's trace it on the board.