EECS 390 – Lecture 17

Modules and Logic Programming

Review: Modules

- An ADT defines an abstraction for a single type
- A ***module*** is an abstraction for a collection of types, variables, functions, etc.
- Often, a module defines a scope for the names contained within the module
- Examples:
  - `math` module in Python
  - `java.util` package in Java
  - `<string>` header in C++

Review: Python Modules

- A Python source file is called a ***module***
  - First unit of organization for interrelated entities
- A module is associated with a scope containing the names defined within it
- Names can be ***imported*** from another module

```
from math import sqrt

def quadratic_formula(a, b, c):
  return (-b + sqrt(b * b - 4 * a * c)) / (2 * a)

def main():
    import sys
    print(quadratic_formula(int(sys.argv[1]),
                            int(sys.argv[2]),
                            int(sys.argv[3])))

if __name__ == '__main__':
    main()
```

Import single name from a module

Import the name of a module into local scope

Use module name

Python Packages

- Python packages are a second level of organization, consisting of multiple modules in the same directory
- Packages can be nested

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
```

Denotes a package

Namespaces in C++

- A ***namespace*** defines a scope for names

```
namespace foo {
  struct A {};
  int x;
}

namespace foo {
  struct B : A {};
}

foo::A *a = new foo::A();

using foo::A;
using namespace foo;
```

Can have multiple namespace blocks in the same or different files

Can use a name from the same namespace without qualification

Use scope-resolution operator to access a name

Import a single name

Import all names

Global Namespace

- An entity defined outside of a namespace is actually part of the global namespace

```
int bar();

void baz() {
  std::cout << ::bar() << std::endl;
}
```

Qualified access to global namespace

- Java similarly places code without a package declaration into the anonymous package

Initialization

- Languages specify semantics for initialization of the contents of a class, module, or package
- In Java, a class is initialized the first time it is used
  - Generally when an instance is created or a static member is accessed for the first time
- In Python, a module's code is executed when it is imported
  - If a module is imported again from the same module, its code does not execute again

Circular Dependencies

- Circular dependencies between modules should be avoided
- Can require restructuring code
- Example:

```
$ python3 foo.py
Traceback (most recent call last):
  File "foo.py", line 1, in <module>
    import bar
  File "bar.py", line 1, in <module>
    import foo
  File "foo.py", line 9, in <module>
    print(func1())
  File "foo.py", line 4, in func1
    return bar.func3()
AttributeError: module 'bar' has no
attribute 'func3'
```

foo.py

```python
import bar

def func1():
    return bar.func3()

def func2():
    return 2

print(func1())
```

bar.py

```python
import foo

def func3():
    return foo.func2()
```

Initialization in C++

- C++ has a multi-step initialization process
  1. **Static initialization**: initialize compile-time constants to their values, and all other variables with static storage duration to zero
  2. **Dynamic initialization**: initialize static-storage variables using their specified initializers
     - Can be delayed until first use of the translation unit
- Within a translation unit, initialization is in program order, with some exceptions
- Order is undefined between translation units
  - Cannot rely on another translation unit being initialized first

Logic Programming

- Imperative programming: express computation as sequences of operations on the program state
- Functional programming: express computation as mappings between function inputs and outputs
- Logic programming: express computation as relations between pieces of data
- First-order predicate calculus is the foundation of logic programming

$$\forall X.\exists Y.\ P(X) \lor \neg Q(Y)$$
$$\forall X.\exists Y.\ Q(Y) \Rightarrow P(X)$$

Horn Clauses

- A logic program is expressed as a set of ***axioms*** that are assumed to be true
- An axiom takes the form of a ***Horn clause***, which specifies a reverse implication:

$$\texttt{H :- } \underbrace{\texttt{B}_1\texttt{, B}_2\texttt{, ..., B}_N}_{\textbf{\textit{Body} of clause}}$$

Individual *term*

***Head* of clause**

- This is equivalent to

$$(B_1 \wedge B_2 \wedge \ldots \wedge B_N) \Rightarrow H$$

with implicit quantifiers.

Queries

- A ***goal*** is a query that the system attempts to prove from the axioms

```
parent(P, C) :- mother(P, C).                    % rule 1
parent(P, C) :- father(P, C).                    % rule 2
sibling(A, B) :- parent(P, A), parent(P, B).     % rule 3

mother(molly, bill).                             % fact 1
mother(molly, charlie).                          % fact 2
```

- Possible reasoning:

Goal

```
   sibling(bill, S)
-> parent(P, bill), parent(P, S)                (rule 3)
-> mother(P, bill), parent(P, S)                (rule 1)
-> mother(molly, bill), parent(molly, S)        (fact 1)
-> mother(molly, bill), mother(molly, S)        (rule 1)
-> mother(molly, bill), mother(molly, charlie)  (fact 2)
```

`S = bill` is also a valid solution given the axioms.

Prolog

- Prolog is the foundational language of logic programming and is the most widely used
- A Prolog program consists of a set of Horn clauses, using the syntax on the preceding slides
- A Horn clause has a head term and optional body terms
- A term may be atomic, compound, or a variable
  - ***Atomic***: atoms and numbers
    - ***Atom***: Scheme-like symbol or quoted string
    - If an atom starts with a letter, it must be lowercase

```
hello    =<    +    'logic programming'
```

  - ***Variables***: symbols that start with an uppercase letter

```
Hello    X
```

Compound Terms

- A compound term consists of a ***functor***, which is an atom, followed by a list of one or more argument terms

```
pair(1, 2)  wizard(harry)  writeln(hello(world))
```

- A compound term is interpreted as a ***predicate***, with a truth value, if it is a head term, a body term, or the goal

- Otherwise, the compound term is interpreted as data
  - e.g. `hello(world)` in `writeln(hello(world))`

Facts and Rules

- A Horn clause with no body is a ***fact***, since it is always true

```
mother(molly, bill).
mother(molly, charlie).
```

Period signifies end of clause

- A Horn clause with a body is a ***rule***

```
parent(P, C) :- mother(P, C).
sibling(A, B) :- parent(P, A), parent(P, B).
```

  - Meaning:
    - If `mother(P, C)` is true, then so is `parent(P, C)`
    - If `parent(P, A)` and `parent(P, B)` are true, then so is `sibling(A, B)`

- A program is a set of Horn clauses

Goals and Queries

- A ***goal*** is a predicate that the interpreter attempts to prove
- Loading the program from the previous slide and entering the goal `sibling(bill, S)` produces:

```
?- sibling(bill, S).
S = bill ;
S = charlie.
```

Ask for more solutions

- A semicolon asks for more solutions
- A period ends a query
  - Can be entered by the user
  - Can be produced by the interpreter, in which case it is certain no more solutions exist

Implementing Lists

- Compound terms can represent data structures
- Example: use `pair(A, B)` to represent a pair
  - This won't be a head or body term, so it will be treated as data
- Relations on pairs:

```
cons(A, B, pair(A, B)).
cdr(pair(_, B), B).
car(pair(A, _), A).
is_null(nil).
```

Relates a first and second item to a pair

Anonymous variable

```
?- cons(1, nil, X).
X = pair(1, nil).

?- car(pair(1, pair(2, nil)), X).
X = 1.

?- cdr(pair(1, pair(2, nil)), X).
X = pair(2, nil).

?- cdr(pair(1, pair(2, nil)), X),
   car(X, Y), cdr(X, Z).
X = pair(2, nil), Y = 2, Z = nil.

?- is_null(nil).
true.

?- is_null(pair(1, nil)).
false.
```

Singleton Variables

- A ***singleton variable*** is a variable that only appears once in an axiom
- Singleton variables can occur inadvertently as a result of a typo:

Oops

```
cons(First, Second, pair(Frist, Second)).
```

- To address this, the Prolog interpreter warns about the occurrence of a singleton variable
- We can inform the interpreter about an intentional singleton by using a name that begins with an underscore

Named, intentional singleton variable

```
cdr(pair(_First, Second), Second).
car(pair(First, _), First).
```

Anonymous variable – does not match any other occurrence of _

Prolog Lists

- Prolog also provides built-in linked lists, specified as elements between square brackets

```
[]    [1, a]    [b, 3, foo(bar)]
```

- The pipe symbol acts like a dot in Scheme, separating some elements from the rest of the list

```
?- writeln([1, 2 | [3, 4]]).
[1,2,3,4]
true.
```

- This allows us to write predicates like the following:

```
contains([Item|_], Item).
contains([_|Rest], Item) :-
  contains(Rest, Item).
```

Requires the first argument to be a list of at least one item