ZHANG Xinyu 1155091989

1. Provide example code and necessary elaborations for demonstrating the advantages of Dynamic Scoping in using Perl to implement the simplified Monopoly game as compared to the corresponding codes in Python.

Perl package Land— function buyLand()

```
        local $Player::due = 1000;
        local $Player::handling_fee_rate = 0.1;
        $owner->payDue();

        $self->{owner} = $owner;
```

Python class Land — function buyLand()

```
tmp_due = Player.due
tmp_handling_fee_rate = Player.handling_fee_rate

if (cur_player.money < self.land_price):
    print("You do not have enough money to buy the land!")
    Player.due = 0
else:
    Player.due = self.land_price
    Player.handling_fee_rate = 0.1
    self.owner = cur_player

cur_player.payDue()
Player.due = tmp_due
Player.handling_fee_rate = tmp_handling_fee_rate
```

Perl package Player— function payDue()

```
sub payDue {
    my $self = shift;
    $self->{money} += $income * (1 - $tax_rate);
    $self->{money} -= $due * (1 + $handling_fee_rate);
}
```

Advantage 1:

In Perl, the syntax “local” denote dynamically scoped variables. Variables like “due” “handling_fee_rate” are package (“Player”) variables with temporality property. That is, the value 1000 and 0.1 assigned to “due” and “handling_fee_rate” seperately in buyLand() of package “land” are only available in the scope of the subprogram.

In program execution time, variables mentioned above which was declared with “local” hides those existing variable “Player::due” and “Player::handling_fee_rate” by masking it with temporary value 1000 and 0.1 respectively as its lexical scope exists, that is, the end of function buyLand(). Outside the scope, we can use the payDue() statement in other subprograms that contains references to package variables “due = 200” “handling_fee_rate = 0”.

In conclusion, when we use dynamic scoping in Perl, the correct attributes of nonlocal variables “due”, “handling_fee_rate” in Player package visible to the program statement in “Land” package cannot be determined statically.

However, in Python which only supports statistically scoped variables, after we change the class variables outside class “Player”, they are modified permanently. Hence, we have to store class variables “due”, “handling_fee_rate” in use temporarily and recover them up to avoid the problem.

Advantage 2:

Variables like “due” “handling_fee_rate” declared with syntax “local” are visible in called subroutine “payDue()” in Perl. That is, the called subprogram “payDue()” in package player calculate player’s money with variable “due = 1000” and “handling_fee_rate = 0.1” and they are implicitly visible.

Overall, we does not need to pass those variables by parameters across subroutines in Perl.

On the other hand, in Python, if we declare “due” “handling_fee_rate” as instance variables instead of class variables, the local variables should be passed from one subprogram “buyLand()” to “payDue()” in class “Player” are expected to be define by caller.

ZHANG Xinyu 1155091989

2. Discuss the keyword local in Perl (e.g. its origin, its role in Perl, and real practical applications of it) and giving your own opinions.

In Perl, they keyword local follows dynamic scoping.

It gives the following usages.

a) Variables declared with local are package variables with temporality property. Variables prefixed with local syntax hides the existing variables by masking them with temporary values as long as their lexical scope exist.

b) We do not need to pass parameters from one subprogram to another because dynamically scoped variables are implicitly visible to subroutines.

In my opinion, despite above convenience, “local” should be avoid as much as possible based on the following reasons.

a) Dynamic scoping results in less reliable programs than static scoping. During the time span beginning when a subprogram begins its execution and ending when that execution ends, the local variables of the subprogram are all visible to any other executing subprogram, regardless of its textual proximity or how execution got to the currently executing subprogram. There is no way to protect local variables from this accessibility. Subprograms are always executed in the environment of all previously called subprograms that have not yet completed their executions.

b) The second problem with dynamic scoping is the inability to type check references to nonlocals statically. This problem results from the inability to statically find the declaration for a variable referenced as a nonlocal.

c) Dynamic scoping also makes programs much more difficult to read, because the calling sequence of subprograms must be known to determine the meaning of references to nonlocal variables. This task can be virtually impossible for a human reader.

d) Finally, accesses to nonlocal variables in dynamic-scoped languages take far longer than accesses to nonlocals when static scoping is used.