I am writing below some of what you wrote but using quantifiers and removing superfluous parentheses etc. Also, I am writing slightly different versions.

canBuyInFuture ≜
e.cumulativePay( ) + e.pay)/2 ≥
(h.cumulativeValueSharesBought + (e.numOfSharesToBuy * s.price))

// Why not define as canBuyInFuture(e,h,s)
// why should the number of shares to buy be a property of e?
// Did you mean

canBuyInFuture(e,h,s,n) ≜
(e.cumulativePay( ) + e.pay)/2 ≥
(h.cumulativeValueSharesBought + (n * s.price))

sharesCost ≜ minimum(h.maxSharesCanBuy, e.numOfSharesToBuy) * s.price

// did you mean
sharesCost(h,e,s) ≜ minimum(h.maxSharesCanBuy, e.numOfSharesToBuy) * s.price

∀ e:Employee, ∃ t:BigTech, ∃s:Share, ∃ h.Handcuff
[t.handcuffs(e)
∧
(t.payDay(Now + 1) ∧ canBuyInFuture)
∨
t.isPayDay(Now) ∧ (s.price = s.history.lookup(Now-7)) ∧
(e.receivedMoney = e.pay - sharesCost) ∧
(e.receivedShares = minimum(h.maxSharesCanBuy, e.numOfSharesToBuy))
∨
¬ (t.isPayDay(Now) ∨ t.isPayDay(Now+1))
)
]

// I wonder whether the below is what you wanted

∀ e:Employee, ∀ t:BigTech, ∀s:Share, ∀ h.Handcuff
[t.handcuffs(e,h)
⟶
(t.payDay(Now + 1) ⟶ canBuyInFuture(e,h,s,??)) // how many shares?
∧
(t.isPayDay(Now) ⟶
s.price = s.history.lookup(Now-7) ∧
e.receivedMoney = e.pay – sharesCost(h,e,s) ∧
e.receivedShares = minimum(h.maxSharesCanBuy, e.numOfSharesToBuy))
)
]

I am not clear why the case

"t.payDay(Now + 1)"

is different from the t.payDay(Now + k)". I thought what you can buy in the future does not change. Do you mean that if you do not buy on the payday, then you increase your allowance for future purchases?

I am not clear what the below is expressing

"t.isPayDay(Now) ⟶
s.price = s.history.lookup(Now-7) ∧
e.receivedMoney = e.pay – sharesCost(h,e,s) ∧
e.receivedShares = minimum(h.maxSharesCanBuy, e.numOfSharesToBuy)) "

If it is a *sufficient* condition, then I think that is better expressed through a Hoare triple which says that if its payday, and the employee has a handcuff, and asks to be paid, and the different between his salary and the anount he is asking for is smaller than the shares he is entitiled to times their value 6 days ago ,

"t.isPayDay(Now) ∧ HasHancuff(t,e,h) ∧ t.salary(e) – amnt <=
h.numberOfMonthlyShare * t.shareprice(Now-6)
{ t.payMe(e,ammt) }
HasShares(t,e)= HasShares(t,e) $_{pre}$ + t.salary(e) – amnt /t.shareprice(Now-6)
∧
e.bankBalance = e.bankBalance$_{pre}$ + amnt

I will add some comments in the below in green

----------code

// how one deals with the cap of how many shares that can be bought and the price of shares at a given time are both error prone (and I have changed them more than once)
// I am assuming s:Share, e:Employee, t:BigTech, h:Handcuff - I know for actual code this is wrong, also I am using x' to mean the previous x.

```
Share{
Nat price //this either has the current share price or if it is within the 6 days before a pay day then it is share price as of 7 days before
Map<Date,Nat> history //from the start of the contract updated daily

Nat observeSharePrice( ){
post:  read( )  // what does this mean? That we read a value? From where?
}
void updateMap( ){   // arguments to this function?
pre: isEmpty(history) V last(history.date) = Now - 1)
post: history = append(history',<Now,observeSharePrice>)
}
Boolean isFrozen(Date d){  // why "freeze" the price? Instead you can just look up the
```

```
                                        // price from 6 days ago
post: exists offset. ((0 ≤ offset < 7) ^ t.isPayDay(d + offset) )
}
void: setPrice( ){
post: isFrozen(Now) V price = observeSharePrice( )
}
}

Employee{
Nat num //a unique number between 0 and the number of employees
Nat pay //current yearly pay/13 and is before share costs have been deducted
Nat cumulativePay = 0
// seems to me that cumulative pay can be specified
// holisically as I did for balance in the paper
Nat numOfSharesToBuy //this hasn't been capped by the maximum number of shares
that employee can buy
// what is the purpose of this field? Why is iot not an argument to the call when the
// employee wants to buy shares

void incrCumulativePay( ) {
post: (cumulativePay = cumulativePay' + receivedMoney( )) V (not(isPayDay(Now)) ^
(result = cumulativePay))
}
// is that an internal function?

void payMe(){
pre: isPayDay(Now)
post: receivedMoney( ) + minimum(numOfSharesToBuy, h.maxSharesCanBuy)*s.price
}
// how much money was received? And what is receivedMoney()? A function?

void setNumOfSharesToBuy( ){
post: numOfSharesToBuy = read( )
}
// who sets the number or shares to buy? Is there an argument? Is read() the argument?

Nat receivedMoney( ){
post: (e.pay - minimum(numOfSharesToBuy, h.maxSharesCanBuy)*s.price) V
(not(isPayDay(Now)) ^ (result = 0))
}
Nat receivedShares( ){
post:  minimum(numOfSharesToBuy, h.maxSharesCanBuy) V (not(isPayDay(Now)) ^
(result = 0))
}
}

BigTech{
BooleanArray[0..numOfEmployees] employees
```

```
Employee findEmployee(Nat num){
post: result.num = num
}

Employee raise(Nat num, Nat amount){
post: (result = findEmployee(num)) ^ (result.pay = result.pay +amount)
}
Employee leave(Nat num){
post: not employees[e.num]
}
void handcuffs(Employee e){
post: employees[e.num]
}
Employee releasedFromHandCuffs(Employee e){
pre: (Now = h.endDate) V not (employees[e.num])
post: not (employees[e.num])
}
Boolean isPayDay(Date d){
post: d mod 28 = 0
}
}

Contract Handcuff{
Date startDate = 0
Nat cumulativeValueSharesBought = 0

Date endDate(Date s) {
post: result = 1820 //13*28*5
}
Nat numOfPayDays( ) {
post: result = Now div 28
}
Nat maxSharesCanBuy( ) {
post: result = (e.cumulativePay/2 - cumulativeValueSharesBought) div s.price
}
}
```