CS2PLD: Programming Language Design Paradigms

Exam May 2025

Question 1:

1. Programming languages can be classified as those which ask the programmer to allocate and free memory manually and those that provide some form of automated memory management.

   - Give an example of a programming language in each group.
   - Give one advantage and one disadvantage of including automatic memory management in a programming language.

   —

   Java provides automatic memory management with a garbage collector.

   C requires programmers to manage memory.

   Advantage of including automatic memory management: Manual memory management is an error-prone process. We may free space too early (and be left with a dangling pointer) or too late (and leak space). Automated memory management helps prevent errors.

   Disadvantage: the runtime system must include a garbage collector, which complicates the implementation of the language. Another disadvantage is that programmers do not have control over the way the garbage collector runs.

2. The following grammar defines the abstract syntax for a simplified version of Solidity (a programming language used to write smart contracts for the Ethereum blockchain). Below, id, n, b and a denote tokens representing an identifier, a number, a boolean and an address, respectively.

$$
\begin{array}{lll}
S & ::= & C \mid C \cdot S \\
C & ::= & \texttt{contract}\ id\ \{StList\} \\
StList & ::= & St \mid St\ ;\ StList \\
St & ::= & Method \mid StateDef \\
StateDef & ::= & Type\ id \mid Type\ id\ =\ E \\
Type & ::= & \texttt{uint} \mid \texttt{bool} \mid \texttt{addresss} \mid \texttt{address payable} \\
Method & ::= & Funct \mid Stmt \\
Funct & ::= & ... \\
Stmt & ::= & (Stmt\ ;\ Stmt) \mid \\
 & & \texttt{if}\ (E)\ \texttt{then}\ Stmt\ \texttt{else}\ Stmt \mid \\
 & & \texttt{while}\ (E)\ \texttt{do}\ Stmt \mid \\
 & & id = E \mid \\
 & & E.\texttt{transfer}(E) \mid \texttt{skip} \mid \ldots \\
E & ::= & id \mid n \mid b \mid a \mid E\ Op\ E \mid \texttt{msg.sender} \mid \texttt{msg.value} \\
Op & ::= & + \mid - \mid * \mid / \mid > \mid < \mid = \mid \texttt{or} \mid \texttt{and}
\end{array}
$$

   Taking into account the grammar given above, draw an abstract syntax tree to represent the following program:

`contract` $Card\{$

`uint` $u = 0$;

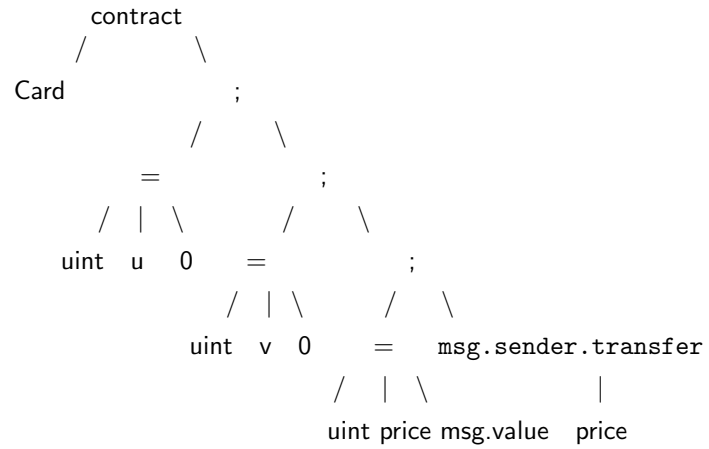
`uint` $v = 0$;

`uint` $price =$ `msg.value`;

`msg.sender.transfer`$(price)\}$

—

The program is represented by an abstract syntax tree with root contract and two subtrees: one with the name (Card), the other representing a list (StList) consisting of State Definitions (StateDef) and a statement (Stmt).



The three declarations (StateDef) associate to the right as indicated in the grammar.

3. Contracts in Solidity can use the local memory in the Ethereum Virtual Machine and also the blockchain memory. Accordingly, the operational semantics of Solidity is defined via configurations of the form $\langle c, \sigma \rangle$ where c is a program component and σ is a pair consisting of the local memory ρ and the blockchain storage μ. Both ρ and μ are partial functions from identifiers to values (as in SIMP).

   A statement of the form $S_1; S_2$ (see the grammar in part 2 above) is executed sequentially, as specified by the following rule ($SeqSol$).

$$\frac{\langle S_1, (\rho, \mu)\rangle \Downarrow \langle skip, (\rho', \mu')\rangle \quad \langle S_2, (\rho', \mu')\rangle \Downarrow \langle skip, (\rho'', \mu'')\rangle}{\langle S_1; S_2, (\rho, \mu)\rangle \Downarrow \langle skip, (\rho'', \mu'')\rangle}\, SeqSol$$

   (a) Define a notion of semantic equivalence for statements in Solidity. You can assume that all statements in Solidity terminate (this is due to the fact that there is an associated cost to running a contract, so contracts cannot "run forever").

   (b) Using the rule ($SeqSol$) given above in part 3 show that the statements $S_1; (S_2; S_3)$ and $(S_1; S_2); S_3$, where S_1, S_2, S_3 are arbitrary Solidity statements, are equivalent.

—

(a) Two statements are equivalent if they have the same semantics. In general, this would mean that they are both non-terminating, or they both terminate and produce the same results when executed in any given local memory and blockchain storage. Since all contracts terminate, we can simply say that statements are equivalent if they produce the same results in any given local memory and blockchain storage. Formally: S_1 and S_2 are equivalent if: $\langle S_1, \sigma\rangle \Downarrow \langle skip, \sigma'\rangle$ if and only if $\langle S_2, \sigma\rangle \Downarrow \langle skip, \sigma'\rangle$

(b) To show that the statements $S_1; (S_2; S_3)$ and $(S_1; S_2); S_3$ are equivalent, it is sufficient to show that they produce the same results when executed in any given local memory and blockchain storage. That is,
$\langle S_1; (S_2; S_3), (\rho, \mu)\rangle \Downarrow \langle skip, (\rho', \mu')\rangle$ if and only if
$\langle (S_1; S_2); S_3, (\rho, \mu)\rangle \Downarrow \langle skip, (\rho', \mu')\rangle$.
Using the rule above, we can see that in both cases S_1 is executed first, S_2 is executed in the state produced by S_1 and S_3 is executed in the state produced by S_2. Thus the end result is the same.
Formally

$$\frac{\langle S_1, (\rho, \mu)\rangle \Downarrow \langle skip, (\rho', \mu')\rangle \qquad \dfrac{\langle S_2, (\rho', \mu')\rangle \Downarrow \langle skip, (\rho'', \mu'')\rangle \quad \langle S_3, (\rho'', \mu'')\rangle \Downarrow \langle skip, (\rho''', \mu''')\rangle}{\langle S_2; S_3, (\rho', \mu')\rangle \Downarrow \langle skip, (\rho''', \mu''')\rangle}}{\langle S_1; (S_2; S_3), (\rho, \mu)\rangle \Downarrow \langle skip, (\rho''', \mu''')\rangle}$$

$$\frac{\dfrac{\langle S_1, (\rho, \mu)\rangle \Downarrow \langle skip, (\rho', \mu')\rangle \quad \langle S_2, (\rho', \mu')\rangle \Downarrow \langle skip, (\rho'', \mu'')\rangle}{\langle S_1; S_2, (\rho, \mu)\rangle \Downarrow \langle skip, (\rho'', \mu'')\rangle} \qquad \langle S_3, (\rho'', \mu'')\rangle \Downarrow \langle skip, (\rho''', \mu''')\rangle}{\langle (S_1; S_2); S_3, (\rho, \mu)\rangle \Downarrow \langle skip, (\rho''', \mu''')\rangle}$$

4. Solidity has a pre-test logically-controlled loop construct:

$$\texttt{while}\ (E)\ \texttt{do}\ Stmt$$

where E is a Boolean expression and $Stmt$ is a statement as defined by the grammar in part (2).

Give a formal definition of the semantics of a statement of the form `while` (E) `do` $Stmt$ in Solidity (you can give a big-step or a small-step semantics).

—

$$\frac{\langle E, (\rho, \mu)\rangle \Downarrow \langle False, (\rho', \mu')\rangle}{\langle while\ (E)\ do\ S, (\rho, \mu)\rangle \Downarrow \langle skip, (\rho', \mu')\rangle}$$

$$\frac{\langle E, (\rho, \mu)\rangle \Downarrow \langle True, (\rho', \mu')\rangle \ \langle S, (\rho', \mu')\rangle \Downarrow \langle skip, (\rho'', \mu'')\rangle \ \langle while\ (E)\ do\ S, (\rho'', \mu'')\rangle \Downarrow \langle skip, (\rho''', \mu''')\rangle}{\langle while\ (E)\ do\ S, (\rho, \mu)\rangle \Downarrow \langle skip, (\rho''', \mu''')\rangle}$$

Question 2:

1. Suppose the following function has been defined in Haskell:

```
pos x = if (x == 1) then True else (pos (x-1))
```

For each of the following expressions, briefly explain how a Haskell interpreter evaluates it, and what the resulting behavior is:

(a) `if (pos 0) then (pos 1) else (pos 1)`
(b) `if (pos 1) then (pos 0) else (pos 1)`
(c) `if (pos 1) then (pos 1) else (pos 0)`
(d) `if (pos 0) then (pos 1) else (pos 1)`

—

The following is a brief explanation. Evaluation of (pos 0) is not possible, due to an infinite recursion. Thus in the first and fourth expressions, a Haskell interpreter will hang and not produce any result. In the second expression, (pos 1) will evaluate to true, but then an interpreter will hang in trying to evaluate (pos 0). Only the third evaluates to a value, namely to True; in this case, since what comes after the *if* is true, an interpreter only evaluates what comes after the *then*.

2. Suppose the following definitions have been introduced in Haskell:

```
combine x y = x : y
add x y = x + y
funky x y = (3 * x) + y
twist f x y = f y x
```

(a) What is the type of the following expression: `combine`
(b) What is the type of the following expression: `(uncurry add)`
(c) What is the type of the following expression: `(combine 1)`
(d) What does the following expression evaluate to: `(funky 3) 2`
(e) What does the following expression evaluate to: `twist funky 3 2`

The following interaction with ghci may help you recall some notions from Haskell:

```
ghci> combine x y = x : y
ghci> combine 3 [4,5]
[3,4,5]
ghci> add x y = x + y
ghci> add 3 4
7
ghci> :t add
add :: Num a => a -> a -> a
ghci> :t curry
curry :: ((a, b) -> c) -> a -> b -> c
ghci> (uncurry add) (3, 4)
7
ghci> :t [1,2,3]
[1,2,3] :: Num a => [a]
```

—

(a) `combine :: a -> [a] -> [a]`

(b) `(uncurry add) :: Num c => (c, c) -> c`

(c) `(combine 1) :: Num a => [a] -> [a]`

(d) `11`

(e) `9`

3. For each of the following pairs of terms, written using Prolog notation, state whether or not they unify, and if they unify, give a most general unifier that unifies them.

(a) `p(X,Y,Z), p(Y,Z,X)`

(b) `p(X,Y,Z), p(Y,Z,f(X))`

(c) `q(X,[U,V]), q([V,W],X)`

(d) `q([H|T]), q([5,6])`

(e) `q([H|T]), q([[5,6]])`

—

(a) yes, mgu $\{X, Y \mapsto Z\}$

(b) no

(c) yes, mgu $\{U, V \mapsto W, X \mapsto [W, W]\}$

(d) yes, mgu $\{H \mapsto 5, T \mapsto [6]\}$

(e) yes, mgu $\{H \mapsto [5, 6], T \mapsto []\}$.

4. Consider the following Prolog program:

```
a(H,[],[H]).
a(X,[H|T],[H|U]) :- a(X,T,U).
r([H|T],V) :- a(H,T,V).
clo(X,X).
clo(X,Z) :- clo(X,Y),r(Y,Z).
```

Relative to this program, for each of the following queries give all answers to the query, in the order that Prolog will generate them. Assume the usual top-to-bottom, left-to-right evaluation strategy.

(a) `a(1,[],X).`

(b) `a(1,[3],X).`

(c) `r([1,2,3],Z).`

(d) `r(Z,[1,2,3]).`

(e) `clo(Y,[1,2,3]).`

—

(a) `X = [1]`

(b) `X = [3, 1]`

(c) `Z = [2, 3, 1]`

(d) `Z = [3, 1, 2]`

(e) `Y = [1, 2, 3] ; Y = [3, 1, 2] ; Y = [2, 3, 1]`

5. Throughout this question, `0` denotes the number zero.

   Consider the following Prolog program:

```
p(0,Y,Y).
p(s(U),V,s(X)) :- p(U,V,X).
m(0,Y,0).
m(s(A),B,C) :- m(A,B,D),p(D,B,C).
```

   Relative to this program, for the query

```
m(s(0),s(0),Z).
```

   draw the entire SLD-resolution tree for this query. Indicate all answers that are generated. Assume the usual top-to-bottom, left-to-right evaluation strategy.

   —

   Many examples of SLD-resolution trees were seen in class. In this case, there is only one branch. Briefly, starting from the query, after using the 4th and 3rd lines of the program (in sequence), one obtains `p(0,s(0),Z)`; then, the 1st line of the program can be applied, to obtain `Z -> s(0)` as the only answer.