CSE 573 Problem Set 2

Answers due 10/28/08 8am

Preferably, turn in by email to both **daipeng@cs** & **weld@cs**

Please work on this problem set individually. If any problem doesn't contain enough information for you to answer it, feel free to make any assumptions necessary to find the answer, but state the assumptions clearly. You will be graded on your choice of assumptions as well as the clarity of your written answers.

We'll accept late problem sets under the following conditions:
Up to 1 hour late - no penalty
Up to one day late - 25% penalty
Up to two day late - 50% penalty
More than 2 days - no credit (please plan ahead!)

1. (15 points) R&N problem 7.4 parts, a, b and c.
   a. ($\Rightarrow$) Assume α is valid, then for every model *M*, *M*|=*α*. So every model which satisfies True must also satisfy α. That is, True|=α.
      ($\Leftarrow$) Assume True|=α. Then α is satisfied by every model which satisfies True. But True is valid, i.e. satisfied in every model, so α is also satisfied by every model. In other words, α is valid.
   b. Since False is bottom (unsatisfied), there is no model that satisfies False. Therefore, every model which does satisfy False also satisfies α, so False |=α.
   c. ($\Rightarrow$) Assume α |= β. Then every model satisfying *α* also satisfies β. Thus if a model, M, doesn't satisfy β (i.e. it does satisfy ¬β) it mustn't satisfy *α* (i.e. it does satisfy ¬ *α*). Thus every model satisfies (β ∨ ¬ *α*) and hence every model satisfies (*α*$\Rightarrow$β) which means that (*α*$\Rightarrow$β) is valid.
      ($\Leftarrow$) Assume (*α*$\Rightarrow$β) is valid. Then every model satisfies (β ∨ ¬ *α*) so every model which fails to satisfy ¬*α* must satisfy β. Equivalently, every model which satisfies *α* satisfies β. But this is the definition of α |= β.

2. (15 points) R&N problem 7.8 parts a, b, c and e.
   We abbreviate Smoke for S, Fire for R, and Heat for H in the truth tables.
   a. Valid

S	$S\Rightarrow S$
T	T
F	T

   b. Neither

S	R	$S\Rightarrow R$
T	T	T
T	F	F
F	T	T
F	F	T

c. Neither

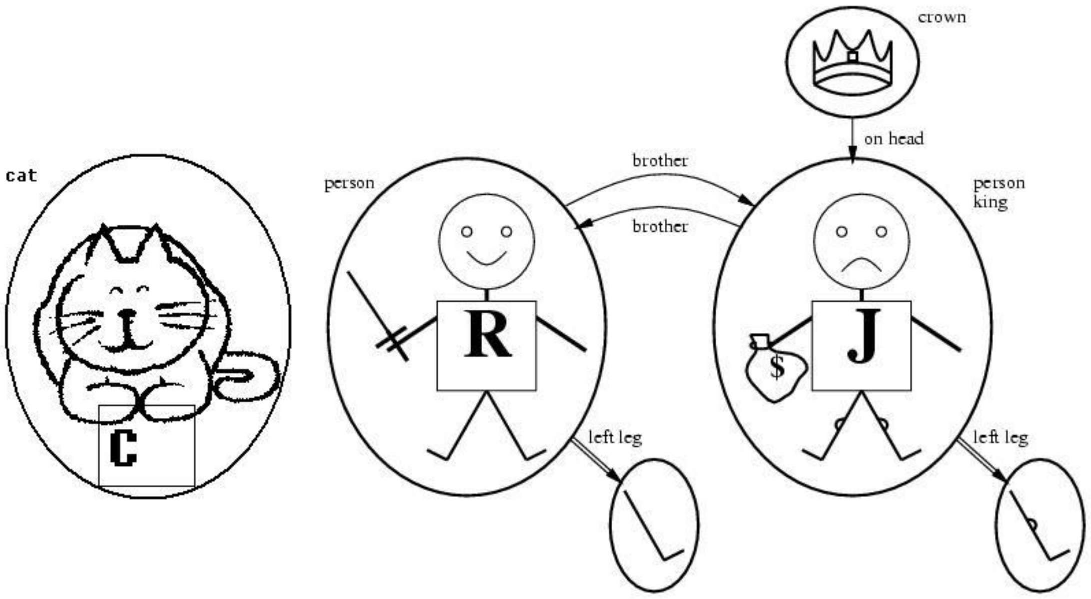
S	R	$S \Rightarrow R$	$\neg S \Rightarrow \neg R$	$(S \Rightarrow R) \Rightarrow (\neg S \Rightarrow \neg R)$
T	T	T	T	T
T	F	F	T	T
F	T	T	F	F
F	F	T	T	T

e. Valid

S	H	R	$S \wedge H$	$(S \wedge H) \Rightarrow R$	$S \Rightarrow R$	$H \Rightarrow R$	$(S \Rightarrow R) \vee (H \Rightarrow R)$	Formula
T	T	T	T	T	T	T	T	T
T	T	F	T	F	F	F	F	T
T	F	T	F	T	T	T	T	T
T	F	F	F	T	F	T	T	T
F	T	T	F	T	T	T	T	T
F	T	F	F	T	T	T	T	T
F	F	T	F	T	T	T	T	T
F	F	F	F	T	T	T	T	T

3. (5 points) R&N problem 7.16
We assume that the KB is consistent. In proving, we need to add $\neg\alpha$ to the KB. Since KB is consistent, it does not contain any empty clause. Then DPLL scans and finds all pure symbols and eliminate them. This scan has linear cost in the number of clauses. The next step, DPLL looks for any unit clause. We ignore the processing time after a unit clause is found (i.e. the time to update the clauses that contain the literal of the unit clause and its negation). Since α is also a unit clause, it takes DPLL linear time to find it. A contradiction is detected immediately afterwards.

4. (5 points) R&N problem 8.2
$(P(a) \wedge P(b))$ does ***not*** entail $\forall x\ P(x)$. To see this, let's revisit the definition of first-order entailment. We say $\Sigma \models \Phi$ when every ***interpretation*** satisfying Σ also satisfies Φ. So, to show that entailment doesn't hold, we have to present a counter example. Since an interpretation is a function mapping from constant, function and relation ***symbols*** in the syntax to actual objects, functions and relations which hold in a model, the first step is picking a model. Many suffice, but let's take the one depicted below, which we'll call M.



Now we need to define an interpretation, so let's say I(a) = R and I(b)=J. Furthermore, let's define I(P) to be the person relation. Now, clearly P(a) and P(b) are both satisfied by I, since I(P(a)) is "person(R)" which holds in M and likewise for P(b). But ∀*x P(x)* isn't satisfied by I since *x* ranges over all constants in M and in particular over the cat, C, which isn't a person. For that matter, we don't need to introduce the cat, since the crown isn't a person and neither is Richard's left leg.

Now an acceptable answer doesn't need to go into all this detail, it suffices to say that "(P(a) ∧ P(b)) does not entail ∀*x P(x)* because there exists an interpretation to a model where some new constant exists where P doesn't hold." But I wanted to work it out in detail, because people are often confused about first-order semantics.

Finally, when compiling FO logic to propositional logic we often assume a "closed universe" ie we assume away all models with objects which are not named in Σ. With such an assumption (which is ***not*** part of "real" FO logic, then the implication *does* hold.

5. (5 points) R&N problem 8.7 using Nationality(p, c) to say that p is from country c and the constant G to denote Germany.

   $\exists l \forall x\ Nationality(x,G) \rightarrow Speaks(x,l)$ or any other equivalent.

6. (5 points) R&N problem 8.8

   $\forall x \forall y\ Male(x) \wedge Spouse(x,y) -> Female(y)$ or any other reasonable ones.

7. (5 points) R&N problem 8.13
   The problem is that the axioms presented can't ever tell when a symbol is *not* an element of a string. This can be fixed by adding an axiom:

   $\forall s\ s \notin \{\}$ (Any symbol is not in the empty string)

8. (5 points) R&N problem 9.4
   a. *{x/A, y/B, z/B}*
   b. Does not unify
   c. *{x/John, y/John}*
   d. Does not unify

9. (30 points) Using the programming language of your choice, implement a function which performs unification. You may accept input in any convenient manner and use any syntax for variables and formulae. Hand in the code for your algorithm and examples showing its correctness, including the ones presented in class (slides of 10/14) as an explanation of the function, plus at least two others.

Sample solution (courtesy of Prasang Upadhyaya unifier.cpp)

```
#include<iostream>
#include<string>
#include<vector>

using namespace std;

class treeNode {
   vector<treeNode> children;
   string val;
   bool isVar;

public:
   treeNode(string s){
      //      cout << "Creating new treeNode with string " << s << endl;
      int nextop = s.find('(');

      // if you didn't encounter any bracket: you are a leaf.
      if (nextop == string::npos) {
         val = s;
         if (s.at(0) == '?') isVar = true;
         else isVar = false;
      }

      // find the first open bracket: string till that point is the val.
      else {
         val = s.substr(0,nextop);
         isVar = false;
         s = s.substr(nextop + 1); // we removed the part till the first bracket.
         s = s.substr(0, s.length() - 1); // we removed the last bracket.
         //          cout << s << endl;

         int nextcom;
         while(s.length() > 0) {
      // cout << s << endl;

      // find the next open bracket and its closing bracket: that is the first child.
      nextop = s.find('(');
      nextcom = s.find(',');
      //     cout << nextcom << " " << nextop << " " << s.length() << endl;
      if (nextop == string::npos && nextcom == string::npos){
         treeNode temp(s);
         children.push_back(temp);
         s = "";
      }
      else if (nextop == string::npos || (nextcom != string::npos && nextcom < nextop)) {
         treeNode temp(s.substr(0,nextcom));
         children.push_back(temp);
         s = s.substr(nextcom+1);
      }
      else {
         int count = 1;
         nextop++;
         for (; count > 0 ; nextop++){
            if (s[nextop] == '(') count++;
            else if (s[nextop] == ')') count--;
         }
         treeNode temp(s.substr(0,nextop));
         children.push_back(temp);
         if (s.length() > nextop) {
            s = s.substr(nextop + 1);
         }
         else s = "";
      }
         }
      }
      // if you encounter a comma instead, that is the first child.
      // if you encounter the closing bracket, whatever was inside is the only child.

   }

   string value(){
      return val;
   }
```

```
int noOfChildren(){
   return children.size();
}

treeNode getChild(int i){
   return children.at(i);
}

bool isVariable(){
   return isVar;
}

string toString(){
   string s = "" ;
   s.append(val);
   if (noOfChildren() > 0){
      s += "(";
      s += getChild(0).toString();
      for (int i = 1; i < noOfChildren(); i++) {
   s += ",";
   s += getChild(i).toString();
      }
      s += ")";
   }
   return s;
}

bool occurCheck(treeNode var){
   if (noOfChildren() > 0) {
      for (int i = 0; i < noOfChildren(); i++){
   if(getChild(i).occurCheck(var)) return true;
      }
      return false;
   }
   else if (!isVariable()) return false;
   else if (value() == var.value()) return true;
   else return false;
}

void findAllVars(vector<string> &v){
   if (isVariable()) {
      for (int i = 0; i < v.size() ; i++)
   if (v.at(i) == toString()) return;
```

```
            v.push_back(toString());
        }

        else {
            for (int i = 0; i < noOfChildren(); i++){
        getChild(i).findAllVars(v);
            }
        }
    }

};

bool unifiable(treeNode, treeNode, vector<treeNode> &, vector<treeNode> &);
bool unifyVar(treeNode, treeNode, vector<treeNode> &, vector<treeNode> &);
bool cycleCheck(const vector<treeNode> &, const vector<treeNode> &);

bool unifiable(treeNode t1, treeNode t2, vector<treeNode> &subVar, vector<treeNode> &subTree){
    if (t1.toString() == t2.toString()) return true;
    else if (t1.isVariable()) return unifyVar(t1, t2, subVar, subTree);
    else if (t2.isVariable()) return unifyVar(t2, t1, subVar, subTree);
    else {
        if (t1.value() == t2.value()) {
            if (t1.noOfChildren() == t2.noOfChildren()) {
        for (int i = 0; i < t1.noOfChildren(); i++){
            if (!unifiable(t1.getChild(i), t2.getChild(i), subVar, subTree)) return false;
        }
        return true;
            }
            else return false;
        }
        else return false;
    }
}

bool unifyVar(treeNode var, treeNode x, vector<treeNode> &subVar, vector<treeNode> &subTree){
    int pos = -1;
    for (int i = 0; i < subVar.size(); i++){
        if (subVar.at(i).toString() == var.toString()) {
            pos = i;
            break;
        }
    }
    if (pos != -1) {
        return unifiable(subTree.at(pos), x, subVar, subTree);
    }
    else {
        for (int i = 0; i < subVar.size(); i++){
            if (subVar.at(i).toString() == x.toString()) {
        pos = i;
        break;
            }
        }
        if (pos != -1){
            return unifiable(var, subTree.at(pos), subVar, subTree);
        }
        else if (x.occurCheck(var)) return false;
        else {
            subVar.push_back(var);
            subTree.push_back(x);
            return true;
        }
    }
}

bool cycleCheck(vector<treeNode> &subVar, vector<treeNode> &subTree){

    vector<vector<int> > graph; // This 2D vectore represents the graph.
    string s; // This string is used temporarily for different purposes in this function
    for(int i = 0; i < subTree.size(); i++){
        vector<int> temp; // This temp vector will become the list of out-neighbours of node i.
        vector<string> vs; // This vector gives the actual outneighbours in string form (?x types).
        (subTree.at(i)).findAllVars(vs); // Populate vs. This is a call be reference.
        //      cout << "{" << subVar.at(i).toString() << "\\" << subTree.at(i).toString() << "} -> " << endl;
        for (int j = 0; j < vs.size(); j++){
            //          cout << vs.at(j) << endl;
            s = vs.at(j); // Get the string at position j.
            for (int k = 0; k < subVar.size() ; k++){
        if (s == subVar.at(k).toString()) {
            temp.push_back(k); // Find the index of string s in subVar.
            break;
        }
            }
        }
        graph.push_back(temp); // Add the outneighbours of node i.
    }

    // This part of the function checks for cylces by running a Depth First Search.
    vector<int> stack; // This vector is used to simulate a stack.
```

```
  int status[graph.size()]; // This array of the size |V| is used internally.
  for (int i = 0; i < graph.size(); i++){
    status[i] = 0; // 0: not seen, 1 on fringe, 2 explored.
  }

  bool foundCycle = false;
  bool seenAll = false;

  while(!seenAll && !foundCycle){

    // Find an 'unseen' node. Add it to the stack (which should be empty).
    // Change the status of the node to 'seen'.
    // Since we haven't seen all the nodes we'll definitely find such a node.
    for (int i = 0; i < graph.size(); i++)
      if (status[i] == 0) {
    status[i] = 1;
    stack.push_back(i);
    break;
      }

    // Actually perform DFS.
    while(stack.size() > 0){
      int node = stack.at(stack.size() - 1); // Get the topmost node.
      stack.pop_back(); // Remove it.
      if (status[node] == 2 || status[node] == 3) continue; //If the node has been explored already either in
this iteration (indicated by 2) or a previous iteration (indiacted by 3) skip the node. Exploring it won't lead to
a cycle.
      else if (status[node] == 1) { // The node is on the fringe and unexplored.
    for (int k = 0; k < graph.at(node).size() ; k++) {
      int neighbour = graph.at(node).at(k); // Get each of its neighbours.
      stack.push_back(neighbour); // Push the neighbour on the stack.
      if (status[neighbour] == 0) status[neighbour] = 1; // If the neighbour was unseen before, make it
'seen'.
      else if (status[neighbour] == 1) continue; // If it was already on the fringe do nothing.
      else if (status[neighbour] == 2) foundCycle = true; // If it was already explored in the current iteration
then we have found a cycle.
      else cout << "Something wrong" << endl;
    }
    status[node] = 2; // Change the status of the current node to 'explored'
      }
      else { cout << "Something wrong" << endl; }
    }

    seenAll = true;
```

```
        for(int i = 0; i < graph.size(); i++) {
            if (status[i] == 2) status[i] == 3; // Prepare for the next iteration.
            if (status[i] == 0) seenAll = false; // Check if free nodes are available.
        }
    }

    return foundCycle;
}

void unify(treeNode t1, treeNode t2){
    vector<treeNode> subVar, subTree;

    if (unifiable(t1,t2,subVar, subTree)) {
        if (!cycleCheck(subVar, subTree)){
            // if (true){
            string mgu;
            mgu += "{";
            for (int i = 0; i < subVar.size(); i++){
        mgu += subVar.at(i).toString();
        mgu += "\\";
        mgu += subTree.at(i).toString();
        mgu += ", ";
            }
            if (subVar.size() > 0) mgu = mgu.substr(0,mgu.length() - 2);
            mgu += "}";
            cout<< "UNIFY(" << t1.toString() << ","<< t2.toString() <<    ") = " << mgu << endl;
        }
        else {
            cout<< "UNIFY(" << t1.toString() << ","<< t2.toString() <<    ") = " << "The two sentences can not be
unified" << endl;
        }
    }
    else {
        cout<< "UNIFY(" << t1.toString() << ","<< t2.toString() <<    ") = " << "The two sentences can not be
unified" << endl;
    }
}

int main(){
    string s1, s2;

    /*
        cout << "Please input the two formulae to be unified in two different lines"<< endl;
        cout << "Sentence 1: ";
        cin >> s1;

        treeNode t1(s1);

        cout << "Sentence 2: ";
        cin >> s2;
        treeNode t2(s2);
        cout << s1 << " : " << s2 << endl;
    */
    while (cin >> s1){
        cin >> s2;
        treeNode t1(s1);
        treeNode t2(s2);
        unify(t1,t2);
    }
}
```

10. In this problem, you are asked to execute SatPlan (a modern planning algorithm, developed by Henry Kautz and colleagues, which compiles problems to SAT) on a past International Planning Competition (IPC) domain.

    f. (10 points) Run problem 2. In how many time steps this problem can be solved? (check the MakeSpan info in the solution file) Write down (in English) which actions are executed in each time step. (To do this, you need to comprehend the domain file and the problem file first, and join them with the solution file)

       3 time steps:
       Step 0: Hoist0 goes out from depot0-1-1 to load area;
       Step 1: Hoist0 lifts Crate0 (in container0) to load area;
       Step 2: Hoist0 drops crate0 to depot0-1-1 from load area.

    g. (5 points) Could one switch the actions in time 0 and time 1, (i.e., the order of the first two actions in the solution) and still have a working plan? If not, what pre-condition(s) is/are violated if you perform the second action first?

       No, you cannot switch. If you try to do lifting at time 0, it violates the precondition that hoist0 must be at the load area.

    h. (25 points) Use the PDDL language to encode a domain and problem of your choice (mobile robots, getting a PhD, getting lunch, traveling to another city for a conference, dating, whatever) so a planner can solve it. Then use SatPlan to find a solution plan. Turn in your PDDL definitions of your domain, and at least one problem, the solution file for the first problem, and a qualitative description of the resulting plan (e.g., length number of actions, etc). Your grade will depend on the complexity of the domain and problem specifications with extra credit available.

Sample domain (courtesy of Hao Du)

Problem:

- There is a field, a robot, n boxes, and n target places (unit squares) for these boxes.
- The places in the field are 4-connected, except where the unit is a wall.
- At any time, a unit can be free, with the robot, or with a box.

Rule:

- The robot can move freely from a place to a neighboring place which is free in the field.
- The robot can push a box by moving into the place with the box, making the box move into the neighboring place, which has to be a free place, along the same direction. (i.e. The robot does not have the strength to push more than one box along the moving direction at one time)

Goal:

- Let the robot push all the boxes to the target places.

(Here I(Hao Du) found an example of this game,
http://push-it.freeonlinegames.com/)

- pushit_domain.pddl : the domain definition
- pushit_p02.pddl : the problem, which is the level three of the game *http://push-it.freeonlinegames.com/*
- pushit_p02.pddl.soln : the solution found by satplan.

It takes the robot 23 steps (actions) to finish this task.

The problem, (initial field)

```
; t r t .
; . b . .
; . b . .
; . .
; . .
; . .
```

where, t is a target place; b is a box; r is the robot; . is initial free places

Domain file (pushit_domain.pddl):

```
; Domain Pushit
; Authors: Hao Du

(define (domain Pushit)
(:requirements :typing)
(:types robot box area - object
         targetarea blankarea - area
)

(:predicates
                (atr ?x - robot ?a - area)
                (atb ?b - box ?a - area)
                (c2 ?a1 ?a2 - area)
                (c3 ?a1 ?a2 ?a3 - area)
                (clear ?a -area)
                (notclear ?a -area)
)

(:action move
  :parameters (?r - robot ?a1 ?a2 - area)
  :precondition (and (c2 ?a1 ?a2) (atr ?r ?a1) (clear ?a2))
  :effect (and (not (atr ?r ?a1)) (atr ?r ?a2)
                    ))

(:action push
  :parameters (?r -robot ?b -box ?a1 ?a2 ?a3 - area)
  :precondition (and (atr ?r ?a1) (atb ?b ?a2) (clear ?a3) (c3 ?a1 ?a2 ?a3))
  :effect (and (not (atr ?r ?a1)) (atr ?r ?a2) (not (atb ?b ?a2)) (atb ?b ?a3)
                      (not (clear ?a3)) (notclear ?a3) (clear ?a2) (not (notclear ?a2))
        ))
)
```

Sample problem file (pushit_p02.pddl):

```
(define (problem Pushit)
(:domain Pushit)
(:objects
        r    - robot
        a12 a14 a21 a22 a23 a24 a32 a33 a34 a41 a42 a51 a52 a61 a62 - blankarea
        t11 t13 - targetarea
        b1 b2 - box
        )

(:init
        (c2 t11 a12) (c2 a12 t11) (c2 a12 t13) (c2 t13 a12) (c2 t13 a14) (c2 a14 t13)
        (c2 a21 a22) (c2 a22 a21) (c2 a22 a23) (c2 a23 a22) (c2 a23 a24) (c2 a24 a23)
        (c2 a32 a33) (c2 a33 a32) (c2 a33 a34) (c2 a34 a33)
        (c2 a41 a42) (c2 a42 a41)
        (c2 a51 a52) (c2 a52 a51)
        (c2 a61 a62) (c2 a62 a61)
        (c2 t11 a21) (c2 a21 t11) (c2 a41 a51) (c2 a51 a41) (c2 a51 a61) (c2 a61 a51)
        (c2 a12 a22) (c2 a22 a12) (c2 a22 a32) (c2 a32 a22) (c2 a32 a42) (c2 a42 a32) (c2 a42 a52) (c2 a52
a42) (c2 a52 a62) (c2 a62 a52)
        (c2 t13 a23) (c2 a23 t13) (c2 a23 a33) (c2 a33 a23)
        (c2 a14 a24) (c2 a24 a14) (c2 a24 a34) (c2 a34 a24)
        (c3 t11 a12 t13) (c3 a12 t13 a14) (c3 t13 a12 t11) (c3 a14 t13 a12)
        (c3 a21 a22 a23) (c3 a22 a23 a24) (c3 a23 a22 a21) (c3 a24 a23 a22)
        (c3 a32 a33 a34) (c3 a34 a33 a32)
        (c3 a41 a51 a61) (c3 a61 a51 a41)
        (c3 a12 a22 a32) (c3 a22 a32 a42) (c3 a32 a42 a52) (c3 a42 a52 a62) (c3 a32 a22 a12) (c3 a42 a32
a22) (c3 a52 a42 a32) (c3 a62 a52 a42)
        (c3 t13 a23 a33) (c3 a33 a23 t13)
        (c3 a14 a24 a34) (c3 a34 a24 a14)

        (atr r a12)
        (atb b1 a22)
        (atb b2 a32)
        (clear t11) (clear a12) (clear t13) (clear a14)
        (clear a21) (notclear a22) (clear a23) (clear a24)
        (notclear a32) (clear a33) (clear a34)
        (clear a41) (clear a42)
        (clear a51) (clear a52)
        (clear a61) (clear a62)
)

(:goal (and
        (notclear t11) (notclear t13)
        ))
)
```

Solution file (pushit_p02 .pddl.soln)

```
; Time 0.46
; ParsingTime       0.00
; MakeSpan 24
0: (MOVE R A12 T11) [1]
1: (MOVE R T11 A21) [1]
2: (PUSH R B1 A21 A22 A23) [1]
3: (PUSH R B2 A22 A32 A42) [1]
4: (PUSH R B2 A32 A42 A52) [1]
5: (MOVE R A42 A41) [1]
6: (MOVE R A41 A51) [1]
7: (MOVE R A51 A61) [1]
8: (MOVE R A61 A62) [1]
9: (PUSH R B2 A62 A52 A42) [1]
10: (PUSH R B2 A52 A42 A32) [1]
11: (PUSH R B2 A42 A32 A22) [1]
12: (PUSH R B2 A32 A22 A12) [1]
13: (MOVE R A22 A32) [1]
```

```
14: (MOVE R A32 A33) [1]
15: (MOVE R A33 A34) [1]
16: (MOVE R A34 A24) [1]
17: (MOVE R A24 A14) [1]
18: (MOVE R A14 T13) [1]
19: (PUSH R B2 T13 A12 T11) [1]
20: (MOVE R A12 A22) [1]
21: (MOVE R A22 A32) [1]
22: (MOVE R A32 A33) [1]
23: (PUSH R B1 A33 A23 T13) [1]
```