Section 9: Compiling Constraints

In this section, we will practice compiling 401_{CP} programs into SAT.

401_{CP} is an extension of our 401 language. It has the following additional constructs which allow us to express program constraints:

- `assert(E);` // Add a constraint stating that E is true
- `choose();` // Resolves to a value that satisfies all program constraints

SAT is our language for expressing constraints. It has the following syntax:

- `(a Int)` // Variable declaration. Supported types are Int and Bool
- `(a + b)` // Arithmetic operators (+, -, * …)
- `(a = b)` // Equivalence operator. Not assignment!
- `(a=b ∨ ¬(a=10))` // Logical connectives (∨, ∧, ¬)
- `ct(a=b)` // Defines constraint that a must be equal to b
- `ite(a>b,a=1,a=2)` // Syntactic sugar for (a > b ∧ a=1) ∨ (¬(a > b) ∧ a=2)

Exercise 1: Polynomials

a) Using the translation rules from the last lecture, convert the following 401_{CP} code for solving the polynomial $\mathbf{x^2 + 2y + 3}$ to SAT.

```
def x;
def y;
def poly;
x = choose();
y = choose();
poly = x * x + 2 * y + 3;
assert(poly == 0);
```

Answer:

```
(x Int)
(y Int)
// We use choose to specify range of values we are interested in
ct( x <= 100 ∧ x >= 0 ∧
    y <= 100 ∧ y >= 0 ∧
    poly = x * x + 2 * y + 3 ∧
    poly = 0)
```

b) Here is another version of 401_{CP} code computing the same polynomial:

```
def x;
def y;
def poly;
X = choose();
y = choose();
poly = x * x;
poly = poly + 2 * y;
poly = poly + 3;
assert(poly == 0);
```

Explain briefly, why this code needs to be re-written before it can be translated to SAT?

```
Yes, because SAT does not have assignments. Poly can only be assigned
once. We must convert this to SSA form.
```

Re-write the code to eliminate the problem:

```
def x;
def y;
def poly;
X = choose();
y = choose();
poly = x * x;
poly1 = poly + 2 * y;
poly2 = poly1 + 3;
assert(poly2 == 0);
```

Exercise 2: Sorting

a) The following 401_{CP} code uses bubble sorting algorithm to sort the array **arr**.

```
lambda bubbleSort(array){
    // range(0,3) iterates 0 <= index < 3
    for(index in range(0,3) {
        if(arr[index] > arr[index + 1]){
            def temp = arr[index];
            arr[index] = arr[index + 1];
            arr[index + 1] = temp;
```

```
        }
    }
}

def arr = {};
arr[0] = 10;
arr[1] = 20;
arr[2] = 15;
arr[3] = 5;

bubbleSort(arr);
```

Re-write the above code such that it may be translated to SAT. *Hint: Generate a new variable for each index of the array.*

```
def arr0_0 = 10;
def arr0_1;
def arr1_0 = 20;
def arr1_1;
def arr1_2;
def arr2_0 = 15;
def arr2_1;
def arr2_2;
def arr3_0 = 5;
def arr3_1;

def index_0;
def index_1;
def index_2;

index_0 = 0; // Code for range inlined

if(index_0 != null){
    if(arr0_0 > arr1_0){
        def temp_0 = arr0_0;
        arr0_1 = arr1_0;
        arr1_1 = temp_0;
    }
    else{
        arr0_1 = arr0_0;
        arr1_1 = arr1_0;
    }
}
```

```
// Code for range inlined
if(index_0 < 2) { index_1 = index0 + 1; }
else { index_1 = null }

if(index_1 != null){
    if(arr1_1 > arr2_0){
        def temp_1 = arr1_1;
        arr1_2 = arr2_0;
        arr2_1 = temp_1;
    }
    else{
        arr1_2 = arr1_1;
        arr2_1 = arr2_0;
    }
}

// Code for range inlined
if(index_1 < 2) { index_2 = index1 + 1; }
else { index_2 = null }

if(index_2 != null){
    if(arr2_1 > arr3_0){
        def temp_2 = arr2_1;
        arr2_2 = arr3_0;
        arr3_1 = temp_2;
    }
    else{
        arr2_2 = arr2_1;
        arr3_1 = arr3_0;
    }
}

// The array now is basically a0_1, a1_2, a2_2 and a3_1
```

b) In this example, the length of **arr** is static. Explain briefly, how would you handle the case where the length of the array was dynamic / unknown?

```
Unroll a constant N number of times where N is the unroll bound. If
you look at the code above, any further unrolls will not be executed
as index will be null.
```

c) Translate the rewritten code from part (a) to SAT.

```
(arr0_0 Int)
(arr0_1 Int)
(arr1_0 Int)
// … declare remaining variables

ct(
    // Set values
    arr0_0 = 10 ∧
    arr1_0 = 20 ∧
    arr2_0 = 15 ∧
    arr3_0 = 5 ∧
    index_0 = 0 ∧
    // … and so on

    ite(index_0 = null,
        ite(arr0_0 > arr1_0,
            temp_0 = arr0_0 ∧
            arr0_1 = arr1_0 ∧
            arr1_1 = temp_0,
            arr0_1 = arr0_0 ∧
            arr1_1 = arr1_0),
        true) ∧

    ite(index_0 < 2,
        index_1 = index0 + 1,
        index_1 = null) ∧

    // … and so on for the remaining unrolls.
```

Exercise 3: Map Coloring

In this exercise we want to find out a way to color the map in such a way that any two countries on the map that share a border have a different color. A country can only be colored either red, green or blue. The following 401_{CP} code defines the constraints.

```
// 5 Countries on the map
def countries = {0=A, 1=B, 2=C, 3=D, 4=E}

// Dictionary mapping countries to the list of neighbours
def neighbours = {}
neighbours[A] = {0=B, 1=D}
neighbours[B] = {0=A, 1=D, 2=E}
neighbours[C] = {0=E}
neighbours[D] = {0=A, 1=B, 2=E}
neighbours[E] = {0=B, 1=C, 2=D}

// Colors of each country. Some unknown
def colors = {}
colors[A] = Red
colors[B] = Blue
colors[C] = Choose()
colors[D] = Choose()
colors[E] = Choose()

// Define constraints
for(country in countries){
    for(neighbour in neighbours[country]){
        assert(colors[country] != colors[neighbour]);
    }
}
```

Translate the code to SAT.

```
// Declare variables
(countries0 Int)
(countries1 Int)
(countries2 Int) // We can encode strings as integers
(neighboursA0 Int)
(neighboursA1 Int)
//... and so on
```

```
ct( // Set Values
    countries0 = 0 ∧   // where 0 means A
    countries1 = 1 ∧   // where 1 means B
    // ...
    colorsA = 0 ∧
    colorsB = 1 ∧
    // Add options where choose
    (colorsC = 1 ∨ colorsC = 2 ∨ colorsC = 3) ∧
    // …


    // first few iterations shown
    country0 = countries0 ∧
    neighbour0 = neighboursA0 ∧
    colorsA = colorsB ∧
    neighbour1 = neighboursA1 ∧
    colorsA = colorsD ∧

    country1 = countries1 ∧
    neighbour2 = neighboursB0 ∧
    colorsB = colorsA ∧
    neighbour3 = neighboursB1 ∧
    colorsB = colorsD ∧
    neighbour4 = neighboursB2 ∧
    colorsB = colorsE ∧

    // and so on
```