CS143 Spring 2023 – Written Assignment 3 – Solutions

This assignment covers semantic analysis, including scoping and type systems. You may discuss this assignment with other students and work on the problems together. However, your write-up should be your own individual work, and you should indicate in your submission who you worked with, if applicable. Assignments can be submitted electronically through Gradescope as a PDF by Tuesday, May 23, 2023 at 11:59 PM PDT. Please review the course policies for more information: `https://web.stanford.edu/class/cs143/policies/`. A LaTeX template for writing your solutions is available on the course website.

1. Consider the following Cool programs:

   (a)

```
1  class A {
2      x: A;
3      baz(): A {x ← new SELF_TYPE};
4      bar(): A {baz()};
5      foo(): String {"am"};
6  };
7  class B inherits A {
8      foo(): String {"I "};
9  };
10 class C inherits A {
11     baz(): A {{ new A; }};
12     foo(): String {"Therefore "};
13 };
14 class Main {
15     main(): Object {
16         let io: IO ← new IO,
17             b : B  ← new B,
18             c : C  ← new C
19         in {
20             io.out_string(c.bar().foo());
21             io.out_string(b.baz().foo());
22             io.out_string(b.bar().baz().foo());
23         }
24     };
25 };
```

   What does this code currently print? Modify lines 2–4 so that this program prints "Therefore I am".

   Answer: This code currently prints "`amI I `". Changing line 4 to "bar(): A {**new** C};" makes the program print "Therefore I am".

(b)

```
1  class Main {
2      main(): Object {
3          let io: IO ← new IO, x: Int ← 20 in {
4              io.out_int(x);
5              let x: Int ← 2 in {
6                  x ← (* YOUR CODE HERE *) ;
7                  io.out_int(x);
8              };
9              if x = 23 then
10                 io.out_string("x")
11             else
12                 io.out_int(x)
13             fi ;
14         }
15     };
16 };
```

Replace *(∗ YOUR CODE HERE ∗)* with a single expression that gets this code to print “2023x”. If it is not possible, explain why.

Answer: Impossible. There is a scope issue: no matter what is set in the inner **let**, when we hit line 9, the visible x is always 20, and thus we can never execute the *io*.out_string on line 10.

2. Type derivations are expressed as inductive proofs in the form of trees of logical expressions. For example, the following is the type derivation for $O[\text{Int}/y], M, C \vdash y + y : \text{Int}$:

$$\dfrac{\dfrac{O[\text{Int}/y](y) = \text{Int}}{O[\text{Int}/y], M, C \vdash y : \text{Int}}\,[\text{Var}] \qquad \dfrac{O[\text{Int}/y](y) = \text{Int}}{O[\text{Int}/y], M, C \vdash y : \text{Int}}\,[\text{Var}]}{O[\text{Int}/y], M, C \vdash y + y : \text{Int}}\,[\text{Arith}]$$

The [Var] and [Arith] labels refer to the corresponding inference rules in the Cool Reference Manual, section 12.2.[1]

Consider the following Cool program fragment:

```
class A {
    i: Int;
    b: Bool;
    s: String;
    o: SELF_TYPE;
    foo(): SELF_TYPE { o };
    bar(): Int { 2 * i + 1 };
};

class B inherits A {
    a: A;
    baz(x: Int, y: Int): Bool { x = y };
    test(): Object { (* PLACEHOLDER *) };
};
```

Note that the environments O and M at the start of the method test() are as follows:

$$O = \emptyset[\text{Int}/i][\text{Bool}/b][\text{String}/s][\text{SELF\_TYPE}_{\text{B}}/o][\text{A}/a][\text{SELF\_TYPE}_{\text{B}}/\mathit{self}],$$

$$\begin{aligned} M = \emptyset&[(\text{SELF\_TYPE})/(\text{A}, \text{foo})][(\text{Int})/(\text{A}, \text{bar})] \\ &[(\text{SELF\_TYPE})/(\text{B}, \text{foo})][(\text{Int})/(\text{B}, \text{bar})] \\ &[(\text{Int}, \text{Int}, \text{Bool})/(\text{B}, \text{baz})][(\text{Object})/(\text{B}, \text{test})]. \end{aligned}$$

For each of the following expressions replacing *(∗ PLACEHOLDER ∗)*, provide the inferred type of the expression, as well as its derivation as a proof tree.[2] For brevity, you may omit subtyping relations where the same type is on both sides (e.g., Bool $\leq$ Bool). You also do not need to label each step with the inference rule name like we did above.

[1] See `https://web.stanford.edu/class/cs143/materials/cool-manual.pdf`, pp. 18–22.

[2] To draw proof trees in LaTeX, consider using the `ebproof` package. You can also use the tree in the template as an example.

(We use "ST" as a shorthand for "SELF_TYPE".)

(a) $\{\ s \leftarrow$ `"world!"`; $b \leftarrow \textbf{self}.\mathrm{baz}(i,\ 1);\ \}$

Answer: The inferred type is Bool.

Lemma:

$$\dfrac{M(\mathrm{B},\mathrm{baz}) = (\mathrm{Int},\mathrm{Int},\mathrm{Bool}) \quad \dfrac{O(\textbf{self}) = \mathrm{ST_B}}{O, M, \mathrm{B} \vdash \textbf{self} : \mathrm{ST_B}} \quad \dfrac{O(i) = \mathrm{Int}}{O, M, \mathrm{B} \vdash i : \mathrm{Int}} \quad \dfrac{}{O, M, \mathrm{B} \vdash 1 : \mathrm{Int}}}{O, M, \mathrm{B} \vdash \textbf{self}.\mathrm{baz}(i, 1) : \mathrm{Bool}}$$

Main proof:

$$\dfrac{\dfrac{O(s) = \mathrm{String} \quad \dfrac{}{O, M, \mathrm{B} \vdash \text{“}\texttt{world!}\text{”} : \mathrm{String}}}{O, M, \mathrm{B} \vdash s \leftarrow \text{“}\texttt{world!}\text{”} : \mathrm{String}} \quad \dfrac{O(b) = \mathrm{Bool} \quad \dfrac{\text{See lemma}}{O, M, \mathrm{B} \vdash \textbf{self}.\mathrm{baz}(i, 1) : \mathrm{Bool}}}{O, M, \mathrm{B} \vdash b \leftarrow \textbf{self}.\mathrm{baz}(i, 1) : \mathrm{Bool}}}{O, M, \mathrm{B} \vdash \{\ s \leftarrow \text{“}\texttt{world!}\text{”}; b \leftarrow \textbf{self}.\mathrm{baz}(i, 1);\ \} : \mathrm{Bool}}$$

(b) $\textbf{let}\ c\colon \mathrm{A} \leftarrow \textbf{self}.\mathrm{foo}()\ \textbf{in}\ c.\mathrm{foo}()$

Answer: The inferred type is A.

$$\dfrac{\dfrac{\dfrac{O(\textbf{self}) = \mathrm{ST_B}}{O, M, \mathrm{B} \vdash \textbf{self} : \mathrm{ST_B}} \quad M(\mathrm{B}, \mathrm{foo}) = (\mathrm{ST})}{O, M, \mathrm{B} \vdash \textbf{self}.\mathrm{foo}() : \mathrm{ST_B}} \quad \mathrm{ST_B} \le \mathrm{B} \le \mathrm{A} \quad \dfrac{\dfrac{O[\mathrm{A}/c](c) = \mathrm{A}}{O[\mathrm{A}/c], M, \mathrm{B} \vdash c : \mathrm{A}} \quad M(\mathrm{A}, \mathrm{foo}) = (\mathrm{ST})}{O[\mathrm{A}/c], M, \mathrm{B} \vdash c.\mathrm{foo}() : \mathrm{A}}}{O, M, \mathrm{B} \vdash \textbf{let}\ c\colon \mathrm{A} \leftarrow \textbf{self}.\mathrm{foo}()\ \textbf{in}\ c.\mathrm{foo}() : \mathrm{A}}$$

(c) $\textbf{if}\ 1 \le i\ \textbf{then self}.\mathrm{foo}()\ \textbf{else}\ a.\mathrm{foo}()\ \textbf{fi}$

Answer: The inferred type is A.

$$\dfrac{\dfrac{\dfrac{}{O, M, \mathrm{B} \vdash 1 : \mathrm{Int}} \quad \dfrac{O(i) = \mathrm{Int}}{O, M, \mathrm{B} \vdash i : \mathrm{Int}}}{O, M, \mathrm{B} \vdash 1 \le i : \mathrm{Bool}} \quad \dfrac{\dfrac{O(\textbf{self}) = \mathrm{ST_B}}{O, M, \mathrm{B} \vdash \textbf{self} : \mathrm{ST_B}} \quad M(\mathrm{B}, \mathrm{foo}) = (\mathrm{ST})}{O, M, \mathrm{B} \vdash \textbf{self}.\mathrm{foo}() : \mathrm{ST_B}} \quad \dfrac{\dfrac{O(a) = \mathrm{A}}{O, M, \mathrm{B} \vdash a : \mathrm{A}} \quad M(\mathrm{A}, \mathrm{foo}) = (\mathrm{ST})}{O, M, \mathrm{B} \vdash a.\mathrm{foo}() : \mathrm{A}}}{O, M, \mathrm{B} \vdash \textbf{if}\ 1 \le i\ \textbf{then self}.\mathrm{foo}()\ \textbf{else}\ a.\mathrm{foo}()\ \textbf{fi} : \mathrm{A}}$$

Note that $\mathrm{SELF\_TYPE_B} \le \mathrm{B} \le \mathrm{A}$, so $\mathrm{A} \sqcup \mathrm{SELF\_TYPE_B} = \mathrm{A}$.

3. Consider the following Cool program:

```
class Main {
    b: B;
    main(): Object {{
        b ← new B;
        b.foo();
    }};
};
```

Now consider the following implementations of the classes A and B. Analyze each version of the classes to determine if the resulting program will pass type checking and, if it does, whether it will execute without runtime errors. Please include a brief (1–2 sentences) explanation along with your answer. Note it is not sufficient to simply copy the output of the reference Cool compiler: if it fails type checking, you must specify which hypotheses cannot be satisfied for which rules.

(a)

```
1  class A {
2      i: Int ← 1;
3      a: SELF_TYPE ←new A;
4      foo(): Int {i};
5  };
6
7  class B inherits A {
8      j: Int ← 1;
9      baz(): Int {i ← 2 + i};
10     foo(): Int {
11         j ← a.baz() + a.foo()
12     };
13 };
```

Answer: The program will not pass type checking. On line 3, we are initializing an attribute of type $\text{SELF\_TYPE}_\text{A}$ with a value of type A. However, $\text{A} \not\leq \text{SELF\_TYPE}_\text{A}$, so the third hypothesis of the [Attr-Init] rule fails.

(b)

```
class A {
    i: Int ← 1;
    a: SELF_TYPE;
    foo(): Int {i};
};

class B inherits A {
    j: Int ← 1;
    baz(): Int {i ← i + j};
    foo(): Int {{
        a ← new SELF_TYPE;
        j ← a.baz() + a@A.foo();
    }};
};
```

Answer: The program will pass type checking and execute correctly. Here's the sequence of actions that occur:

1. Upon initialization, the variable b initially contains $B(i = 1, a = \texttt{void}, j = 1)$.
2. b.foo() initializes a to be another B.
3. a.baz() sets a to be $B(i = 2, a = \texttt{void}, j = 1)$ and returns 2.
4. a@A.foo() calls A.foo(), which returns $a.i = 2$.
5. b.foo() finally sets $b.j$ to 4 and returns 4.
6. At the end, the main function returns Int(4). The final object b looks like

$$B(i = 1, a = B(i = 2, a = \texttt{void}, j = 1), j = 4).$$

4. Consider the following extensions to Cool:

   (a) Tuples.

$$
\begin{aligned}
\textit{expr} ::= \;& \ldots \\
\mid\;& \textbf{new} \;\langle\; \text{TYPE} \;[\![, \text{TYPE}]\!]^{*} \;\rangle \;[\; \textit{expr} \;[\![, \textit{expr}]\!]^{*} \;] \\
\mid\;& \textit{expr} \;[\; \text{INT} \;]
\end{aligned}
$$

A tuple is a fixed-size list of values of potentially different types. Empty tuples are not allowed. We define a new family of types called *tuple types* $\langle T_1, T_2, \ldots, T_n \rangle$, where $T_1, T_2, \ldots, T_n$ could be any type in Cool (including SELF_TYPE and other tuple types). Note that the entire hierarchy of tuple types still has Object as its topmost supertype. Additionally, the subtype relation between tuple types is defined as follows:

$$\langle T_1, T_2, \ldots, T_n \rangle \leq \langle T'_1, T'_2, \ldots, T'_n \rangle \quad \text{if and only if } T_i \leq T'_i \text{ for all } i.$$

A tuple object can be initialized with an expression similar to

$$\textit{my\_tuple} \colon \langle \text{Int}, \text{Object} \rangle \leftarrow \textbf{new}\ \langle \text{Int}, \text{String} \rangle [42, \text{“}\texttt{answer}\text{”}];$$

Thereafter, the i^{th} element in the tuple can be accessed as “$\textit{my\_tuple}[i]$”. Tuple elements are 0-indexed. The tuple index is an integer literal that is always known at compile time.

Provide new typing rules for Cool which handle the typing judgments for the two new forms of expressions. As an example, your type rules should ensure the following given the earlier declaration:

$$O, M, C \vdash \textit{my\_tuple}[0] : \text{Int} \qquad\qquad O, M, C \vdash \textit{my\_tuple}[1] : \text{Object}$$

Hint: See [New] in the Cool manual for an example that deals with SELF_TYPE in a way similar to how you will have to.

Answer:

$$
\frac{
\begin{array}{c}
m = n \\
O, M, C \vdash e_1 : S_1 \\
\vdots \\
O, M, C \vdash e_m : S_m \\
T'_i = \begin{cases} \text{SELF\_TYPE}_C & \text{if } T_i = \text{SELF\_TYPE}, \\ T_i & \text{otherwise} \end{cases} \qquad \forall i \in \{1, \ldots, n\} \\
S_i \leq T'_i \quad \forall i \in \{1, \ldots, n\}
\end{array}
}{
O, M, C \vdash \textbf{new}\ \langle T_1, T_2, \ldots, T_n \rangle\ [e_1, e_2, \ldots, e_m] : \langle T'_1, T'_2, \ldots, T'_n \rangle
} \text{ [Tuple-New]}
$$

$$
\frac{O, M, C \vdash e : \langle T_1, T_2, \ldots, T_n \rangle \qquad i \text{ is an integer constant} \qquad 0 \leq i \leq n-1}{O, M, C \vdash e[i] : T_{i+1}} \text{ [Tuple-Index]}
$$

(b) Permissive method overriding.

In Cool a subtype can only override a method with a method with exactly the same formal parameters and return type. Or as judgements (with some abuse of notation to quantify over the elements in environments):

$$\frac{T_i = S_i \quad \forall i \in \{1, \dots, n+1\}}{(T_1, \dots, T_n, T_{n+1}) \leq (S_1, \dots, S_n, S_{n+1})} \text{[Method-Subtype]}$$

$$\frac{T_c = T_p}{M \vdash T_c \leq T_p} \text{[Class-Subtype1]}$$

$$\frac{\begin{array}{c} T_c \textbf{ inherits } T_p' \qquad M \vdash T_p' \leq T_p \\ M \vdash \forall m \in M(T_p)\colon M(T_c, m) \leq M(T_p, m) \end{array}}{M \vdash T_c \leq T_p} \text{[Class-Subtype2]}$$

The Method Subtype rule says that if a class X has a method f and class Y has a method g, to establish that f conforms to g (i.e., $M(X, f) \leq M(Y, g)$), we must show $M(X, f) = (T_1, \dots, T_n, T_{n+1}) = (S_1, \dots, S_n, S_{n+1}) = M(Y, g)$.

The two Class Subtype rules say that for a class T_c to be considered a subtype of a class T_p we must establish one of two things:

1. T_c must either be equal to T_p; or
2. (a) T_c must inherit from some class T_p' where T_p' is a subtype of T_p, and
   (b) For every method m on T_p, T_c must also have a method m such that the types of the methods are conforming (as defined by the Method Subtype rule). I.e., $M(T_c, m) \leq M(T_p, m)$.

The Method Subtype rule is more restrictive than necessary to ensure type safety. Rewrite it with new hypotheses so that T_i need not equal S_i. Note your solution should still ensure type safety without changing the rules for dispatch. Specifically, given $C \leq P$ with a method m if

$$out \leftarrow (p\colon P).m(e_1, e_2, \dots, e_n);$$

type checks then so should

$$out \leftarrow (c\colon C).m(e_1, e_2, \dots, e_n);$$

for the same arguments and output variable.

Answer:

$$\frac{S_i \leq T_i \quad \forall i \in \{1, \dots n\} \qquad T_{n+1} \leq S_{n+1}}{(T_1, \dots, T_n, T_{n+1}) \leq (S_1, \dots, S_n, S_{n+1})} \text{[Method-Subtype']}$$

This corresponds to allowing supertypes in arguments and subtype in the return.

A good way to understand this is considering functions of one argument. Suppose we have sets (types) A, B, X, Y where $A \subseteq B$ and $X \subseteq Y$, a function $f\colon B \to X$ is also a function $A \to Y$ as every element in A is mapped to an element Y by f. In other words functions $B \to X$ are a subtype of functions $A \to Y$.

It is tempting to use the rule $T_i \leq S_i \quad \forall i \in \{1, \dots, n, n+1\}$. However, this would lead to the same problems as allowing SELF_TYPE as parameter (see lecture 10 slide 23).

(c) Multiple inheritance.

Cool's type system allows single inheritance, where one class inherits from at most one other class. However, many programming languages[3] allow a class to inherit from multiple superclasses. This is especially useful for "interface"-like classes: a hypothetical `File` class can inherit from both `Reader` and `Writer`, while standard input only inherits from `Reader`:

```
class Reader {
    read(): String {""};                    -- to be overridden by subclass
};
class Writer {
    write(s: String): SELF_TYPE {self}; -- to be overridden by subclass
};
class File inherits Reader, Writer {
    read(): String { ... }
    write(s: String): SELF_TYPE {{ ...; self; }}
};
class Stdin inherits Reader {
    read(): String { ... }
};
```

Now, most Cool code would continue to work if Cool is extended to support multiple inheritance. However, one kind of Cool expressions would have undefined behavior without adjustments to its semantics. Identify which form of expression would be undefined and explain why it would be undefined.

Answer: There are three kinds of expressions that we allowed as answer:

i. **case** expressions. From the Cool manual §7.9, given an expression with dynamic type C, a **case** expression always selects "the branch with the least type <typek> such that $C \leq$ <typek>." In other words, a **case** expression would try to find the most specific branch that still matches the input expression. However, if multiple inheritance is allowed, we could have a situation where two branches have types that are incomparable, or equally specific.
As a concrete example, consider the following expression:

```
case new File of
    r : Reader => ...;
    w : Writer => ...;
esac
```

It is undefined which branch should execute, since File $\leq$ Reader and File $\leq$ Writer, yet neither Reader nor Writer is "less than" (or more specific than) the other.

ii. **if** expressions. Suppose there exists a class File2 that also inherits from Reader and Writer:

```
class File2 inherits Reader, Writer {
    read(): String { ... }
    write(s: String): SELF_TYPE {{ ...; self; }}
};
```

and additionally there's code like

[3] Examples include C++, Go, and Python.

```
if ... then
    new File
else
    new File2
fi
```

The type of this expression is not well-defined, since both Reader and Writer are possible least upper bounds of File and File2.

iii. Dispatch expressions. Suppose there are two base classes that both define the same method in conflicting ways:

```
class IsTrue {
    test(): Bool { true };
};
class IsFalse {
    test(): Bool { false };
};
class Chimera inherits IsTrue, IsFalse {};
```

It's unclear which definition would be used for "(**new** Chimera).test()".

As an aside, it's interesting to see how real programming languages solve these problems.

i. For **case**, Go has a feature analogous to **case** called the type switch statement. Python 3.10 introduced the "pattern matching statement" with similar functionality. And a pattern matching syntax is proposed for C++ as well.
In all three languages, the match statement chooses not the "least" (or "best") branch, but instead the first branch that matches. So in our example, all three languages would choose the Reader branch.
For more details, refer to the specification of each of those languages:
   A. Go Programming Language Specification, `https://go.dev/ref/spec#Type_switches`;
   B. C++ P1371, §7.3 First Match rather than Best Match, `https://wg21.link/p1371r3#page=21`; and
   C. Python PEP-622, `https://peps.python.org/pep-0622/#match-semantics`.

ii. For **if**, Go does not have an equivalent expression type, while Python does not conduct static typing. C++ requires the two alternatives to be somewhat compatible in type, so our test case above would result in a type error. See `https://godbolt.org/z/vzqYce51e`.

iii. For dispatch, Python uses the order of inheritance to decide which parent class "wins". C++ and Go, on the other hand, forbid ambiguous calls to inherited methods. See the following "playground" links:
   A. C++, `https://godbolt.org/z/6WffGY864`
   B. Go, `https://go.dev/play/p/IyhGKmTOli0`

5. Consider the following assembly language used to program a stack (r, r_1, and r_2 denote arbitrary registers):

   - **push** r: copies the value of r and pushes it onto the stack.
   - **top** r: copies the value at the top of the stack into r. This command does not modify the stack.
   - **pop**: discards the value at the top of the stack.
   - **swap**: swaps the value at top of the stack with the value right beneath it. E.g., if the stack was $\langle \$, \ldots, 5, 2\rangle$ **swap** would change the stack to be $\langle \$, \ldots, 2, 5\rangle$
   - r_1 $*=$ r_2: multiplies r_1 and r_2 and saves the result in r_1. r_1 may be the same as r_2.
   - r_1 $/=$ r_2: divides r_1 with r_2 and saves the result in r_1. r_1 may be the same as r_2. Remainders are discarded (e.g., $5/2 = 2$).
   - r_1 $+=$ r_2: adds r_1 and r_2 and saves the result in r_1. r_1 may be the same as r_2.
   - r_1 $-=$ r_2: subtracts r_2 from r_1 and saves the result in r_1. r_1 may be the same as r_2.
   - **jump** r: jumps to the line number in r and resumes execution.
   - **print** r: prints the value in r to the console.

   The machine has two registers available to the program: **reg1**, and **reg2**. The stack is permitted to grow to a finite, but very large, size. If an invalid line number is invoked, a number is divided by zero, **top** or **pop** is executed on an empty stack, **swap** is executed on stack with less than 2 elements, or the maximum stack size is exceeded, the machine crashes.

   Write code to enumerate and print the factorials ($F_n = n \times F_{n-1}$ where $F_1 = 1$; e.g., $1, 2, 6, 24, \ldots$) starting at F_1. Assume that the code will be placed at line 100, and will be invoked by pushing 1, 1 onto the stack $\langle \$, \ldots, 1, 1\rangle$, storing 100 in reg1, and running **jump** reg1.

   Your code should use the **print** opcode to display numbers in the sequence. You may not hardcode constants nor use any other instructions besides the ones given above. There is no need to keep the number in memory after it has been printed out. Your code should not terminate (or crash) after any amount of time. Assume that registers and the stack can hold arbitrarily large integers so computation will never overflow.

   Hint: it may help to comment each line with a symbolic machine state and think about what the state the code should be in at the end. (You are not required to do this but it will help us give you partial credit if you do.) E.g.:

   // initial : *reg1*=100 *reg2*= *stack*=$\langle n, F_{n-1}\rangle$

   100 **top** reg2 *// reg1*=100 *reg2*=F_{n-1} *stack*=$\langle n, F_{n-1}\rangle$
   101 **pop** *// reg1*=100 *reg2*=F_{n-1} *stack*=$\langle n\rangle$

   // final : *???*

Answer:

	// *initial :*	*reg1*=100	*reg2*=	*stack*=$\langle n, F_{n-1}\rangle$
100	**top** reg2	// *reg1*=100	*reg2*=F_{n-1}	*stack*=$\langle n, F_{n-1}\rangle$
101	**pop**	// *reg1*=100	*reg2*=F_{n-1}	*stack*=$\langle n\rangle$
102	**push** reg1	// *reg1*=100	*reg2*=F_{n-1}	*stack*=$\langle n, 100\rangle$
103	**swap**	// *reg1*=100	*reg2*=F_{n-1}	*stack*=$\langle 100, n\rangle$
104	**top** reg1	// *reg1*=n	*reg2*=F_{n-1}	*stack*=$\langle 100, n\rangle$
105	**pop**	// *reg1*=n	*reg2*=F_{n-1}	*stack*=$\langle 100\rangle$
106	reg2 $*=$ reg1	// *reg1*=n	*reg2*=F_n	*stack*=$\langle 100\rangle$
107	**print** reg2	// *reg1*=n	*reg2*=F_n	*stack*=$\langle 100\rangle$
108	**push** reg2	// *reg1*=n	*reg2*=F_n	*stack*=$\langle 100, F_n\rangle$
109	**swap**	// *reg1*=n	*reg2*=F_n	*stack*=$\langle F_n, 100\rangle$
110	reg2 $/=$ reg2	// *reg1*=n	*reg2*=1	*stack*=$\langle F_n, 100\rangle$
111	reg1 $+=$ reg2	// *reg1*=$n+1$	*reg2*=1	*stack*=$\langle F_n, 100\rangle$
112	**push** reg1	// *reg1*=$n+1$	*reg2*=1	*stack*=$\langle F_n, 100, n+1\rangle$
113	**swap**	// *reg1*=$n+1$	*reg2*=1	*stack*=$\langle F_n, n+1, 100\rangle$
114	**top** reg1	// *reg1*=100	*reg2*=1	*stack*=$\langle F_n, n+1, 100\rangle$
115	**pop**	// *reg1*=100	*reg2*=1	*stack*=$\langle F_n, n+1\rangle$
116	**swap**	// *reg1*=100	*reg2*=1	*stack*=$\langle n+1, F_n\rangle$
117	**jump** reg1	// *reg1*=100	*reg2*=1	*stack*=$\langle n+1, F_n\rangle$
	// *final :*	*reg1*=100	*reg2*=	*stack*=$\langle n+1, F_n\rangle$