Compiler Construction
Professor Jeremy Siek

Midterm Exam
October 19, 2009

Name: ____________________

This exam has 7 questions, for a total of 100 points.

1. $\boxed{\text{10 points}}$ Given the following token definitions and grammar:

```
TRUE = "True"
FALSE = "False"
PRINT = "print"
INPUT = "input"
IF = "if"
ELSE = "else"
LP = "("
RP = ")"

module ::= statement
statement ::= PRINT expression
expression ::= TRUE | FALSE | INPUT LP RP | expression IF expression ELSE expression
```

show the tokenization and parse tree for the following program. The precedence and associativity of the arithmetic expressions is the same as in Python.

```
print input() if input() else False
```

Solution:

```
Tokenization: PRINT INPUT LP RP IF INPUT LP RP ELSE FALSE

Parse tree: (drawing an actual tree is OK)
module
    statement
        PRINT
        expression
            expression
                INPUT
                LP
                RP
            IF
            expression
                INPUT
                LP
                RP
            ELSE
            expression
                FALSE
```

2. $\boxed{\text{10 points}}$ Draw the abstract syntax tree (AST) (or write down its textual representation) that Python's built-in parser would produce for the program in question 1. (You can leave out `None` values.)

Solution:

```
Module(Stmt([Printnl([IfExp(CallFunc(Name('input'), []),
                            CallFunc(Name('input'), []),
                            Name('False'))])]))
```

3. $\boxed{\text{10 points}}$ Given the input

```
1
0
```

what is the output of the program in question 1? Given the input

```
0
1
```

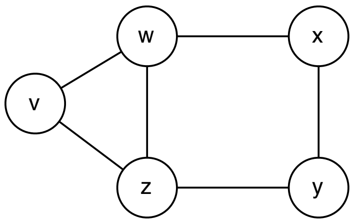
what is the output?

Solution:

```
given input:
1
0
produces output:
0

given input:
0
1
produces output:
False
```

4. $\boxed{\text{20 points}}$ Given the following interference graph that has a node for each variable, assign each variable to a location (a register or stack offset) using the greedy saturation-based algorithm. You are compiling for a computer architecture that has only two general purpose registers, %eax and %ebx, but is otherwise just like the x86. Write down the following information for each step in the algorithm: 1) how many available registers there are for each variable at the beginning of the step, 2) what variable is chosen for assignment in this step, and 3) which register or stack location is assigned to that variable. Please format your answer as a table with one row per step.



Solution: (The following is one of many correct solutions.)

v	w	x	y	z	chosen variable	chosen location
2	2	2	2	2	v	eax
-	1	2	2	1	w	ebx
-	-	1	2	0	z	-4(%ebp)
-	-	1	2	-	x	eax
-	-	-	1	-	y	ebx

5. $\boxed{\text{20 points}}$ Translate the AST from question 2 into monomorphic intermediate representation given by the following grammar. The only kinds of values in this subset of Python that you need to deal with are integers, booleans (no lists or dictionaries), and pyobj (which is the tagged union of integers and booleans). Recall that the tag for integers is 0 and the tag for Booleans is 1.

```
type ::= int | bool
boolean ::= True | False
integer ::= 0 | 1 | 2 | ...
identifier = [a-zA-Z0-9_]+
value ::= boolean | integer
op ::= "==" | "!="
expr ::= Name(identifier) | Input() | Const(value)
       | Let(identifier, expr, expr) | IfExp(expr, expr, expr)
       | InjectFrom(type, expr) | ProjectTo(type, expr) | GetTag(expr)
       | Compare(expr, op, expr)
stmt ::= Stmt([stmt,...,stmt]) | Printnl(expr)
module = Module(stmt)
```

The intermediate representation that you produce must conform to the following type checking rules. The symbol Γ is a dictionary, mapping variables to types.

$$\frac{}{\Gamma \vdash \texttt{Name}(x) : \Gamma[x]} \qquad \frac{}{\Gamma \vdash \texttt{Input}() : \texttt{pyobj}} \qquad \frac{b \in \texttt{boolean}}{\Gamma \vdash \texttt{Const}(b) : \texttt{bool}} \qquad \frac{n \in \texttt{integer}}{\Gamma \vdash \texttt{Const}(n) : \texttt{int}}$$

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash \texttt{Let}(x, e_1, e_2) : T_2} \qquad \frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : T \quad \Gamma \vdash e_3 : T}{\Gamma \vdash \texttt{IfExp}(e_1, e_2, e_3) : T}$$

$$\frac{\Gamma \vdash e : T \quad T \in \{\texttt{bool}, \texttt{int}\}}{\Gamma \vdash \texttt{InjectFrom}(T, e) : \texttt{pyobj}} \qquad \frac{\Gamma \vdash e : \texttt{pyobj} \quad T \in \{\texttt{bool}, \texttt{int}\}}{\Gamma \vdash \texttt{ProjectTo}(T, e) : T} \qquad \frac{\Gamma \vdash e : \texttt{pyobj}}{\Gamma \vdash \texttt{GetTag}(e) : \texttt{int}}$$

$$\frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T \quad T \in \{\texttt{bool}, \texttt{int}\}}{\Gamma \vdash \texttt{Compare}(e_1, op, e_2) : \texttt{bool}} \qquad \frac{\{\} \vdash e : \texttt{pyobj}}{\vdash \texttt{Printnl}(e)}$$

$$\frac{\vdash s_0 \quad \cdots \quad \vdash s_n}{\vdash \texttt{Stmt}([s_0, \ldots, s_n])} \qquad \frac{\vdash s}{\vdash \texttt{Module}(s)}$$

Solution:

```
Module(Stmt([Printnl(
  IfExp(Let(tmp, Input(),
          IfExp(Compare(GetTag(Name('tmp')), '==', Const(0)),
                Compare(Const(0),'!=', ProjectTo(int, Name('tmp'))),
                Compare(Const(False),'!=', ProjectTo(bool, Name('tmp'))))),
      Input(),
      InjectFrom(bool, Const(False))))]))
```

6. $\boxed{\text{10 points}}$ A *simple expression* is a `Name` or a `Const`. All other expressions are *complex expressions*. Flatten the result of the previous question into an equivalent program that does not contain complex expressions nested inside of other expressions by introducing temporary variables. The output should be in the intermediate language described in the previous question, with the addition of `Assign` and `If` statements.

```
stmt ::= ... | Assign(identifier, expr) | If(expr, stmt, stmt)
```

Solution:

```
Module([Stmt([
Assign(0_tmp, Input()),
Assign(1_tmp, GetTag(0_tmp)),
Assign(2_tmp, Compare(1_tmp, '==', 0)),
If(2_tmp,
  Stmt([Assign(3_tmp, ProjectTo(int, 0_tmp)),
        Assign(4_tmp, Compare(Const(0), "!=", 3_tmp)),
        Assign(11_tmp, 4_tmp)]),
  Stmt([Assign(7_tmp, ProjectTo(bool, 0_tmp)),
        Assign(8_tmp, Compare(Const(False), "!=", 7_tmp)),
        Assign(11_tmp, 8_tmp)])),
If(11_tmp,
   Stmt([Assign(12_tmp, Input()),
         Assign(14_tmp, 12_tmp)]),
   Stmt([Assign(13_tmp, InjectFrom(bool, Const(False))),
         Assign(14_tmp, 13_tmp)])),
Println(14_tmp)
])])
```

7. $\boxed{\text{20 points}}$ Perform instruction selection, transforming the result of the previous question into pseudo-x86 assembly, that is, x86 assembly except that you should:

- use variables instead of registers and stack locations. (You don't need to do register allocation to answer this question.) You may still need to use a few registers, for example, you'll need to mention %eax to get the return value from a function call and %esp to move the stack pointer after a function call.
- use "if" statements instead of jumps instructions. An "if" statement should have the following form:

  ```
  if var:
     instructions
  else:
     instructions
  ```

  where *var* is a variable that is treated as 32-bit Boolean value: if it is 0 the else branch is taken, otherwise the then branch is taken.

Put comments in your assembly code that says which statement each sub-sequence of instructions comes from. You may make calls to the following C functions but no others.

```
void print_any(pyobj p);
pyobj input_int();
```

Recall that a pyobj is a 32 bit value, where the right-most two bits contain the tag.

Solution:

```
.globl main
main:
        pushl %ebp
        movl %esp, %ebp
        subl $8, %esp
        call input_int          # Assign(0_tmp, Input())
        movl %eax, 0_tmp
        movl 0_tmp, 1_tmp       # Assign(1_tmp, GetTag(0_tmp))
        andl $3, 1_tmp
        cmpl $0, 1_tmp          # Assign(2_tmp, Compare(1_tmp, '==', 0))
        sete %al
        movzbl %al, 2_tmp
        if 2_tmp:               # If(2_tmp, ...)
           movl 0_tmp, 3_tmp    # Assign(3_tmp, ProjectTo(int, 0_tmp))
           sarl $2, 3_tmp
           cmpl $0, 3_tmp       # Assign(4_tmp, Compare(Const(0), "!=", 3_tmp))
           setne %al
           movzbl %al, 4_tmp
           movl 4_tmp, 11_tmp   # Assign(11_tmp, 4_tmp)
        else:
           movl 0_tmp, 7_tmp    # Assign(7_tmp, ProjectTo(bool, 0_tmp))
           sarl $2, 7_tmp
           cmpl $0, 7_tmp       # Assign(8_tmp, Compare(Const(False), "!=", 7_tmp))
           setne %al
           movzbl %al, 8_tmp
           movl 8_tmp, 11_tmp   # Assign(11_tmp, 8_tmp)
       if 11_tmp:               # If(11_tmp, ...)
           call input_int       # Assign(12_tmp, Input())
           movl %eax, 12_tmp
           movl 12_tmp, 14_tmp  # Assign(14_tmp, 12_tmp)
       else:
           movl $0, 13_tmp      # Assign(13_tmp, InjectFrom(bool, Const(False)))
           sall $2, 13_tmp
           orl $1, 13_tmp
           movl 13_tmp, 14_tmp  # Assign(14_tmp, 13_tmp)
       pushl 14_tmp             # Printnl(14_tmp)
       call print_any
       addl $4, %esp

       movl $0, %eax
       leave
       ret
```