CS 352 — Compilers: Principles and Practice
Midterm Examination II (Final), 11/29/12

Instructions: Read carefully through the whole exam first and plan your time. Note the relative weight of each question and part (as a percentage of the score for the whole exam). The total points is 100 (your grade will be the percentage of your answers that are correct).

This exam is **closed book, closed notes**. You may *not* refer to any book or other materials.

You have **75 minutes** to complete all **four** (4) questions. Write your answers on this paper (use both sides if necessary).

Name:

Student Number:

Signature:

1. (Parsing; 35%) Consider the following simple grammar and the language it describes:

$$S \rightarrow \varepsilon \mid a \mid bSd \mid bScSd$$

(a) (5%) Is this *grammar* LL(1)? Explain. [There is a simple argument.]
Answer:

No, the grammar is not LL(1) by inspection since the last two rules have a common prefix.

(b) (5%) Is this *language* LL(1)? If so, exhibit a simple grammar; if not, explain why not.
Answer:

Yes, the language is LL(1); by factoring the common prefix we get:

$$\begin{aligned} S &\rightarrow \varepsilon \mid a \mid bST \\ T &\rightarrow d \mid cSd \end{aligned}$$

(c) (5%) Is this *grammar* LR(0)? Explain. [Again, there is a simple argument.]
Answer:

No, the grammar is not LR(0) because there is a shift-reduce conflict between the first rule and the other three.

(d) (5%) Is this *grammar* SLR(1)? Explain. [You need not build the SLR(1) state machine; rather, you can argue from FOLLOW sets.]
Answer:

Yes, the grammar is SLR(1). FOLLOW(S) is $\{\$, c, d\}$ so SLR(1) will reduce $S \rightarrow \varepsilon$ using the lookaheads in FOLLOW(S), and a and b are not in FOLLOW(S). Thus, the LR(0) shift-reduce conflict is eliminated.

(e) (5%) Is this *grammar* LR(1)? [Hint: Consider your previous answer!]
Answer:

Yes, since all SLR(1) grammars are LR(1).

(f) (5%) Is this *language* LR(0)? Why or why not?
Answer:

Yes, the language is LR(0), because if a language is LR(k) for any k, it is also LR(0) (though the LR(0) grammar may not be intuitive or easy to construct).

(g) (5%) Is this *language* regular? Why or why not?
Answer:

No, the language is not regular because it involves bracketing, such as $b^n d^n$, known not to be regular because it requires an unbounded number of states and so cannot be recognized by a finite automaton.

2. (Type checking; 15%) In each of the following questions you must answer whether the given `mojo` program is type consistent according to the concept of *structural* typing. If the program is not type consistent, indicate any line that results in a compile-time type error.

   (a) (5%) Yes or no? If not, why not (indicate the type error)?

```
1  {
2    var a: struct { a: int; };
3    var b: struct { b: int; };
4    a := b;
5  }
```

   Answer:

   No.

   (b) (5%) Yes or no? If not, why not (indicate the type error)?

```
1  {
2    var a: class { a: int; };
3    var b: class { a: int; b: int; };
4    a := b;
5  }
```

   Answer:

   No.

   (c) (5%) Yes or no? If not, why not (indicate the type error)?

```
1  {
2    type A = class { a: int; };
3    type B = class extends A { b: int; };
4    var a: A;
5    var b: B;
6    var ab: class extends class { a: int; } { b: int; };
7    a := b;
8    a := ab;
9    b := ab;
10 }
```

   Answer:

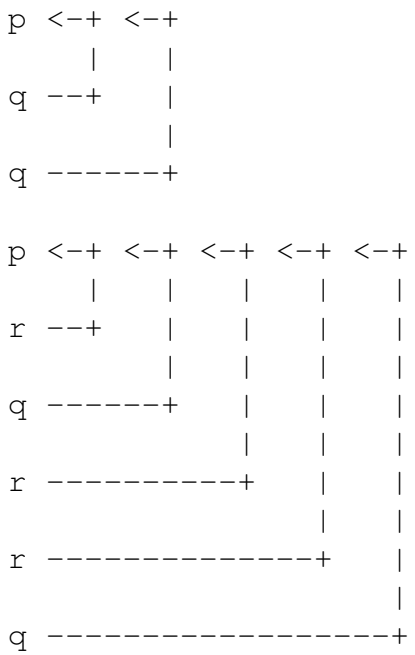
   Yes.

3. (Static links; 10%) Consider the following `mojo` program:

```
1 proc p() {
2   proc q() {
3     ...
4   }
5   proc r() {
6     ...
7   }
8   ...
9 }
```

Assume the following call sequence: `p`, `q`, `q`, `p`, `r`, `q`, `r`, `r`, `q`. Draw the call stack at the time of the last invocation of `q`, and show the static link for each invocation as a pointer from the stack frame to its outer scope frame.

Answer:



4. (Translation; 40%) Consider the following `mojo` source program:

```
1  proc printf(s: Text; i: int);
2  type Integer = class {
3    value: int;
4    init(i: int): Integer := Init;
5    mul(i: int): int := Mul;
6  };
7  proc Init(this: Integer; i: int): Integer {
8    this.value := i;
9    return this;
10 }
11 proc Mul(this: Integer; i: int): int {
12   var result := 0;
13   if i > 0 {
14     result := this.value + this.mul(i-1);
15   }
16   return result;
17 }
18 {
19   printf("%d\n", new(Integer).init(9).mul(3));
20 }
```

(a) (5%) What is the output of this program?

Answer:

27

(b) (10%) The following listing shows IR tree statements produced when the `mojo` compiler translates the `Mul` function. [Note: This code is generated without null checks.]

```
MOVE(
 TEMP result,
 CONST 0),
BGT(
 TEMP i,
 CONST 0,
 L.2, L.3),
LABEL L.2,
MOVE(
 TEMP result,
 ADD(
  MEM(
   TEMP this,
   CONST 0, 8),
  CALL(
   ESEQ(
    MOVE(
     TEMP t.0,
     TEMP this),
    MEM(
     MEM(
      TEMP t.0,
      CONST -8, 8),
     CONST 0, 8)),
   CONST 0,
   TEMP t.0,
   SUB(
    TEMP i,
    CONST 1)))),
LABEL L.3,
MOVE(
 TEMP %rax,
 TEMP result),
JUMP(
 NAME L.0),
LABEL L.0
```

Rewrite these IR tree statements in canonical form (i.e., so that there are no SEQ or ESEQ statements), and so that the CALL expression is not a child of any other expression (i.e., it is the child only of a MOVE statement). You may write your answer to the right of the original non-canonical IR trees.

Answer:

```
MOVE(
 TEMP result,
 CONST 0)
BGT(
 TEMP i,
 CONST 0,
 L.2, L.3)
LABEL L.2
MOVE(
 TEMP t.0,
 TEMP this)
MOVE(
 TEMP t.1,
 CALL(
  MEM(
   MEM(
    TEMP t.0,
    CONST -8, 8),
   CONST 0, 8),
  CONST 0,
  TEMP t.0,
  SUB(
   TEMP i,
   CONST 1)))
MOVE(
 TEMP result,
 ADD(
  MEM(
   TEMP this,
   CONST 0, 8),
  TEMP t.1))
LABEL L.3
MOVE(
 TEMP %rax,
 TEMP result)
JUMP(
 NAME L.0)
LABEL L.0
```

(c) (10%) Identify basic blocks (i.e., every block begins with a LABEL and ends with a JUMP or CJUMP) in the canonical IR tree statements and schedule the basic blocks into a linear trace such that whenever possible every CJUMP (e.g., BGT) is followed immediately by its false target and every JUMP is followed immediately by its target.
Answer:

```
LABEL L.6
MOVE(
 TEMP result,
 CONST 0)
BGT(
 TEMP i,
 CONST 0,
 L.2, L.3)
LABEL L.3
MOVE(
 TEMP %rax,
 TEMP result)
JUMP(
 NAME L.0)
LABEL L.2
MOVE(
 TEMP t.0,
 TEMP this)
MOVE(
 TEMP t.1,
 CALL(
  MEM(
   MEM(
    TEMP t.0,
    CONST -8, 8),
   CONST 0, 8),
  CONST 0,
  TEMP t.0,
  SUB(
   TEMP i,
   CONST 1)))
MOVE(
 TEMP result,
 ADD(
  MEM(
   TEMP this,
   CONST 0, 8),
  TEMP t.1))
JUMP(
 NAME L.3)
LABEL L.0
```

(d) (15%) The compiler generates the following x86_64 instructions for the `Mul` function:

```
     movq %rdi, this    # this := %rdi
     movq %rsi, i       # i := %rsi
L.6: xorq t.2,t.2       # t.2 <-
     movq t.2, result   # result := t.2
     xorq t.3,t.3       # t.3 <-
     cmpq t.3,i         # <- t.3 i
     jg L.2             # <- : goto L.2 L.3
L.3: movq result, %rax  # %rax := result
     jmp L.0            # <- : goto L.0
L.2: movq this, t.0     # t.0 := this
     movq -8(t.0),t.4   # t.4 <- t.0
     movq 0(t.4),t.5    # t.5 <- t.4
     movq t.0, %rdi     # %rdi := t.0
     movq i, t.6        # t.6 := i
     subq $1, t.6       # t.6 <- t.6
     movq t.6, %rsi     # %rsi := t.6
     call *t.5          # %rdi %rsi %rdx %rcx %r8 %r9 %rax %r10 %r11 <- t.5 %rdi %rsi
     movq %rax, t.1     # t.1 := %rax
     movq 0(this),t.8   # t.8 <- this
     movq t.8, t.7      # t.7 := t.8
     addq t.1,t.7       # t.7 <- t.1 t.7
     movq t.7, result   # result := t.7
     jmp L.3            # <- : goto L.3
L.0:
```

The listing shows the uses and definitions for each instruction in the comments column on the right. The := annotation describes MOVE instructions that copy (*use*) one temporary/register to (*define*) another. The <- annotation describes an OPER instruction that *uses* temporaries/registers to the right of the the <- and *defines* temporaries/registers to the left. The `goto` annotation describes jump (JUMP and CJUMP) transfers.

Compute *liveness* information for each instruction in the `Mul` function, tracing uses backwards to definitions and annotating the (implicit) control flow edges between instructions with the temporaries/registers live on those edges. Write the temporaries/registers live on these control flow edges in the listing above. Be careful to propagate along both edges at a control flow merge point, by drawing explicit control flow edges from jumps to their targets as a reminder to propagate (backwards) along those edges as well.

Answer:

```
                                                        %rdi %rsi
     movq %rdi, this    # this := %rdi
                                                        %rsi this
     movq %rsi, i       # i := %rsi
                                                        this i
L.6: xorq t.2,t.2       # t.2 <-
                                                        this i t.2
     movq t.2, result   # result := t.2
                                                        this i result
     xorq t.3,t.3       # t.3 <-
                                                        this i result t.3
     cmpq t.3,i         # <- t.3 i
                                                        this i result
     jg L.2             # <- : goto L.2 L.3
                                                        this i result
L.3: movq result, %rax  # %rax := result
                                                        %rax
     jmp L.0            # <- : goto L.0
                                                        %rax
L.2: movq this, t.0     # t.0 := this
                                                        this i t.0
     movq -8(t.0),t.4   # t.4 <- t.0
                                                        this i t.0 t.4
     movq 0(t.4),t.5    # t.5 <- t.4
                                                        this i t.0 t.5
     movq t.0, %rdi     # %rdi := t.0
                                                        this i t.5 %rdi
     movq i, t.6        # t.6 := i
                                                        this t.5 %rdi t.6
     subq $1, t.6       # t.6 <- t.6
                                                        this t.5 %rdi t.6
     movq t.6, %rsi     # %rsi := t.6
                                                        this t.5 %rdi %rsi
     call *t.5          # %rdi %rsi %rdx %rcx %r8 %r9 %rax %r10 %r11
                        # <- t.5 %rdi %rsi
                                                        this %rax
     movq %rax, t.1     # t.1 := %rax
                                                        this t.1
     movq 0(this),t.8   # t.8 <- this
                                                        t.1 t.8
     movq t.8, t.7      # t.7 := t.8
                                                        t.1 t.7
     addq t.1,t.7       # t.7 <- t.1 t.7
                                                        t.7
     movq t.7, result   # result := t.7
                                                        result
     jmp L.3            # <- : goto L.3
                                                        result
L.0:
```