CS4610—Programming Languages—Exam 2

Spring 2017

March 30, 2017

- **Write your name and UVa ID on this exam.** Write your UVa ID on every page of this exam in case pages become separated. Pledge the exam before you turn it in.
- This exam has twelve (12) pages (including this one) and six (6) questions. If you get stuck on a question, move on and come back later.
- You have 1 hour and 15 minutes to complete this exam.
- The exam is closed book, but you may refer to your reference sheet. You are allowed two paper-sides of notes (either one page [front and back] or two fronts).
- Use of electronic devices (cell phones, tablets, laptops, etc.) is prohibited for the duration of the exam. Even vaguely looking at a device **is cheating**. Avoid questionable situations by turning all of these devices off and placing them underneath your chair.
- Please write your answers in the space provided on the exam, and clearly mark your solutions. You may use the backs of the exam pages as scratch paper. Please do not use any additional scratch paper.
- Solutions will be graded on correctness and clarity. Each question has a relatively straight-forward answer. Points may be deducted for both difficult-to-read answers and also solutions that are overly complicated.
- Partial credit will be awarded for partial answers. If you leave a non-extra-credit portion of the exam blank, **you will receive one-third of the points for that small portion (rounded down)**. If you wish to make an answer to a sub-question blank, draw an "X" through the portion you wish to be blank. **DO NOT** cross out portions of the exam until you near the end because this mark cannot be undone.
- The final two pages of this exam are the Cool Assembly Language reference sheet. You may detach them from the exam.

UVa ID: KEY

Name (print): KEY

UVa ID (again): ______________________

Problem	Max Points	Points
1—Definitions	15	
2—Earley Parsing	30	
3—Type Inference	15	
4—Typing Rules	5	
5—Code Generation	15	
6—Extending Cool	20	
Extra Credit	0	
TOTAL	100	

Honor Pledge:

How well do you think you did? ______________________________

1. Definitions (15 points)

For each of the following terms, briefly describe or define its meaning. Each definition is worth three points.

(a) Free Variable:

A variable is free in an expression if the expression contains an occurrence of a variable that refers to a declaration outside of the expression.

(b) Type Environment:

A type environment is a mapping from free variable identifiers to types

(c) Type Inference:

The process of filling in missing type information in a program

(d) Instruction Set Architecture (ISA):

Defines the interface with a processor (the set of valid instruction values)

(e) Liskov Substitution Principle:

If B is a subclass of A, then an object of class B can be used where an object of class A is expected.

2. Earley Parsing (30 points)

(a) (9 points) Briefly describe each of the three operations used for filling in an Earley parsing chart (parsing table). Do not exceed four sentences for each description.

i. Operation: Prediction
Description:

The symbol to the right of • is a non-terminal. Add rules to the set for the non-terminal to the right of the •, and set the position to the index of the current valid set.

ii. Operation: Scan
Description:

The symbol to the right of the • is a terminal. If the input matches, add the current item to the next valid set with • shifted right.

iii. Operation: Completion
Description:

There is nothing to the right of the •. Look for the parent item in the valid set indicated by the location given by the rule. Add the parent to the current valid set with • shifted right.

(b) (20 points) Complete the Earley parsing chart (parsing table) on the next page.

(c) (1 point) Does the input that you processed for the previous part have a valid parse given the grammar? (circle one)

YES / NO

<u>Grammar</u>

$$
\begin{aligned}
S &\to \texttt{id}\ A\ M \\
A &\to\ = E \mid \varepsilon \\
M &\to \texttt{+}\ E\ M \mid \texttt{-}\ E\ M \mid \varepsilon \\
E &\to \texttt{id} \mid \texttt{int}
\end{aligned}
$$

<u>Input</u>

`id = int + id - int`

Tokens: `id` | `=` | `int` | `+` | `id` | `-` | `int`

valid[0]	valid[1]	valid[2]	valid[3]	valid[4]	valid[5]	valid[6]	valid[7]
$S \to \bullet\ \texttt{id}\ A\ M\ \langle 0\rangle$	$S \to \texttt{id}\ \bullet\ A\ M\ \langle 0\rangle$ $A \to \bullet\ \texttt{=}\ E\ \langle 1\rangle$ $A \to \bullet\ \langle 1\rangle$ $S \to \texttt{id}\ A\ \bullet\ M\ \langle 0\rangle$ $M \to \bullet\ \texttt{+}\ E\ M\ \langle 1\rangle$ $M \to \bullet\ \texttt{-}\ E\ M\ \langle 1\rangle$ $M \to \bullet\ \langle 1\rangle$ $S \to \texttt{id}\ A\ M\ \bullet\ \langle 0\rangle$	$A \to \texttt{=}\ \bullet\ E\ \langle 1\rangle$ $E \to \bullet\ \texttt{id}\ \langle 2\rangle$ $E \to \bullet\ \texttt{int}\ \langle 2\rangle$	$E \to \texttt{int}\ \bullet\ \langle 2\rangle$ $A \to \texttt{=}\ E\ \bullet\ \langle 1\rangle$ $S \to \texttt{id}\ A\ \bullet\ M\ \langle 0\rangle$ $M \to \bullet\ \texttt{+}\ E\ M\ \langle 3\rangle$ $M \to \bullet\ \texttt{-}\ E\ M\ \langle 3\rangle$ $M \to \bullet\ \langle 3\rangle$ $S \to \texttt{id}\ A\ M\ \bullet\ \langle 0\rangle$	$M \to \texttt{+}\ \bullet\ E\ M\ \langle 3\rangle$ $E \to \bullet\ \texttt{id}\ \langle 4\rangle$ $E \to \bullet\ \texttt{int}\ \langle 4\rangle$	$E \to \texttt{id}\ \bullet\ \langle 4\rangle$ $M \to \texttt{+}\ E\ \bullet\ M\ \langle 3\rangle$ $M \to \bullet\ \texttt{+}\ E\ M\ \langle 5\rangle$ $M \to \bullet\ \texttt{-}\ E\ M\ \langle 5\rangle$ $M \to \bullet\ \langle 5\rangle$ $M \to \texttt{+}\ E\ M\ \bullet\ \langle 3\rangle$ $S \to \texttt{id}\ A\ M\ \bullet\ \langle 0\rangle$	$M \to \texttt{-}\ \bullet\ E\ M\ \langle 5\rangle$ $E \to \bullet\ \texttt{id}\ \langle 6\rangle$ $E \to \bullet\ \texttt{int}\ \langle 6\rangle$	$E \to \texttt{int}\ \bullet\ \langle 6\rangle$ $M \to \texttt{-}\ E\ \bullet\ M\ \langle 5\rangle$ $M \to \bullet\ \texttt{+}\ E\ M\ \langle 7\rangle$ $M \to \bullet\ \texttt{-}\ E\ M\ \langle 7\rangle$ $M \to \bullet\ \langle 7\rangle$ $M \to \texttt{-}\ E\ M\ \bullet\ \langle 5\rangle$ $M \to \texttt{+}\ E\ M\ \bullet\ \langle 3\rangle$ $S \to \texttt{id}\ A\ M\ \bullet\ \langle 0\rangle$

3. Type Inference (15 points)

Consider the following excerpt from the Cool syntax grammar and typing rules. Recall that the grammar specification for Cool is not in pure Backus-Naur Form (BNF); for convenience, we also use some regular expression notation. Specifically, A^* means zero or more A's in succession. Items in square brackets [...] are optional. Double brackets $[\![\,]\!]$ are not part of Cool; they are used in the grammar as a meta-symbol to show association of grammar symbols (e.g., $a[\![bc]\!]^+$ means a followed by one or more bc pairs).

$$\begin{array}{rcl} expr & \rightarrow & \texttt{let ID : TYPE in } expr \\ & | & \texttt{if } expr \texttt{ then } expr \texttt{ else } expr \texttt{ fi} \\ & | & expr\texttt{.ID(} [\ expr\ [\![, expr]\!]^*\ ]\ \texttt{)} \\ & | & \texttt{ID} \mid \text{integer} \\ & | & \text{true} \end{array}$$

Note that Γ maps free variables to associated types, M maps `(class name, function name)` pairs to method signatures (the type of each formal parameter followed by the return type), and C is the current class's name.

$$\frac{i \text{ is an integer constant}}{\Gamma, M, C \vdash i : Int} \text{ (Int)}$$

$$\frac{\Gamma(Id) = T}{\Gamma, M, C \vdash Id : T} \text{ (Identifier)}$$

$$\frac{T_0' = \begin{cases} \text{SELF_TYPE}_C & \text{if } T_0 = \text{SELF_TYPE} \\ T_0 & \text{otherwise} \end{cases} \quad \Gamma[T_0'/x], M, C \vdash e_1 : T_1}{\Gamma, M, C \vdash \texttt{let } x : T_0 \texttt{in } e_1 : T_1} \text{ (Let)}$$

$$\frac{}{\Gamma, M, C \vdash true : Bool} \text{ (True)}$$

$$\frac{\begin{array}{l}\Gamma, M, C \vdash e_1 : Bool \\ \Gamma, M, C \vdash e_2 : T_2 \\ \Gamma, M, C \vdash e_3 : T_3\end{array}}{\Gamma, M, C \vdash \texttt{if } e_1 \texttt{then } e_2 \texttt{else } e_3 \texttt{fi} : T_2 \sqcup T_3} \text{ (If)}$$

$$\frac{\begin{array}{l}\Gamma, M, C \vdash e_0 : T_0 \\ \Gamma, M, C \vdash e_1 : T_1 \\ \vdots \\ \Gamma, M, C \vdash e_n : T_n \\ T_0' = \begin{cases} C & \text{if } T_0 = \text{SELF_TYPE}_C \\ T_0 & \text{otherwise} \end{cases} \\ M(T_0', f) = (T_1', \ldots, T_n', T_{n+1}') \\ T_i \leqslant T_i' \quad 1 \leqslant i \leqslant n \\ T_{n+1} = \begin{cases} T_0 & \text{if } T_{n+1}' = \text{SELF_TYPE} \\ T_{n+1}' & \text{otherwise} \end{cases}\end{array}}{\Gamma, M, C \vdash e_0.f(e_1, \ldots, e_n) : T_{n+1}} \text{ (Dispatch)}$$

Consider a Cool program in which two classes, `A` and `B` are defined such that:

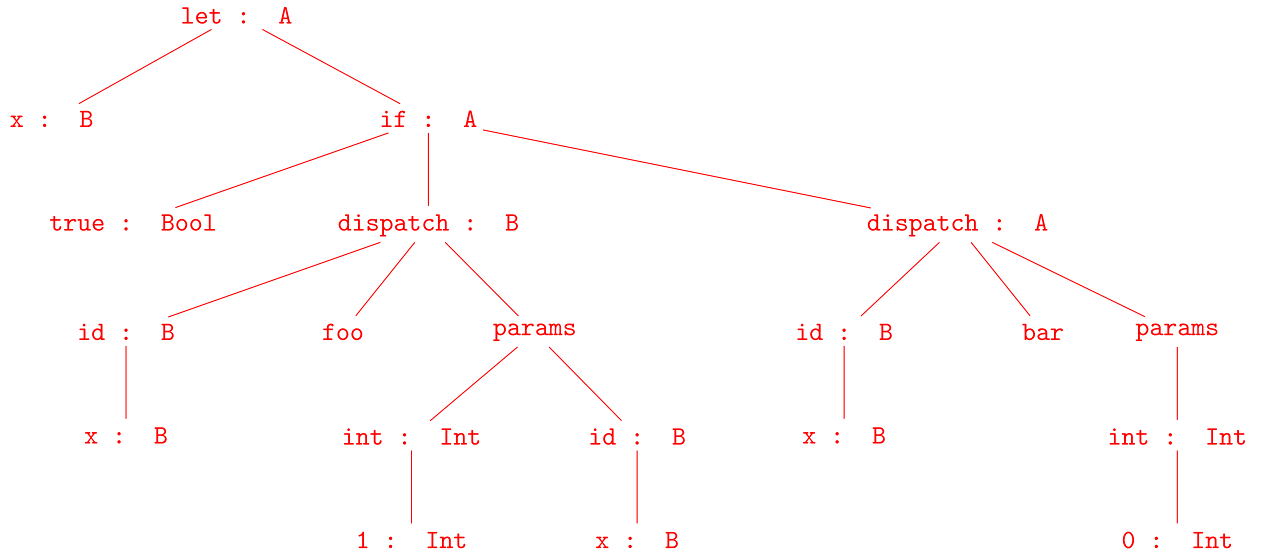
- `B inherits A`
- $M(B, foo) = (Int, A, B)$
- $M(B, bar) = (Int, A)$

(a) (7 points) In the space on the page, draw the AST for the follow Cool expression:

```
1 let x : B in
    if true then x.foo(1, x) else x.bar(0) fi
```

Let expressions should have two children (the binding identifier node, and contained expression node); *dispatches* should have three children (the expression node, the name of the function, and a node containing the list of arguments [with a variable number of children nodes]); *if* expressions should have three children; *identifier* and *integer* nodes each have one child; and *true* nodes have no children.

(b) (8 points) Annotate each node in the AST you drew above to indicate the appropriate type given the typing rules and assumptions about `A` and `B` on the previous page.



4. Typing Rules (5 points)

Recall that static dispatch provides a way of accessing methods of parent classes that have been hidden by redefinitions in child classes. Instead of using the class of the leftmost expression to determine the method, the method of the class that is explicitly specified is used. For example, `e@B.f()` invokes the method `f` in class `B` on the object that is the value of `e`. Consider the following *incorrect* rule for static dispatch:

$$\frac{\begin{array}{c} \Gamma, M, C \vdash e_0 : T_0 \\ \Gamma, M, C \vdash e_1 : T_1 \\ \vdots \\ \Gamma, M, C \vdash e_n : T_n \\ M(T, f) = (T'_1, \ldots, T'_n, T'_{n+1}) \\ T_i \leqslant T'_i \quad 1 \leqslant i \leqslant n \\ T_{n+1} = \begin{cases} T_0 & \text{if } T'_{n+1} = \text{SELF_TYPE} \\ T'_{n+1} & \text{otherwise} \end{cases} \end{array}}{\Gamma, M, C \vdash e_0@T.f(e_1, \ldots, e_n) : T_{n+1}} \text{(Static-Dispatch)}$$

This typing rule is *unsound*: it allows programs that will lead to run-time errors. Describe the problem in one or two sentences, and provide an additional hypothesis that will make this rule sound.

$T_0 \leqslant T$ so that static dispatch can only be called on parent classes.

5. Code Generation (15 points)

This question refers to the following syntax for a simple integer arithmetic and function language:

$$
\begin{aligned}
P &\rightarrow D; P \mid D \\
D &\rightarrow \texttt{def id}(ARGS) = E; \\
ARGS &\rightarrow \texttt{id}, ARGS \mid \texttt{id} \\
E &\rightarrow \texttt{int} \mid \texttt{id} \mid \texttt{if}(E_1 = E_2)\,\{E_3\}\,\texttt{else}\,\{E_4\} \\
&\quad \mid E_1 + E_2 \mid E_1 - E_2 \mid \texttt{id}(E_1, \ldots, E_n)
\end{aligned}
$$

(a) (9 points) Suppose we decided to implement looping functionality in the simple language shown above. We choose to do this by adding three additional expressions to the grammar to support variable assignment, sequencing of expressions, and loops:

$$
\begin{aligned}
E &\rightarrow \texttt{id} \Leftarrow E \\
&\quad \mid \{E_1; E_2\} \\
&\quad \mid \texttt{while}(E_1 = E_2)\ E_3
\end{aligned}
$$

The loop functions by evaluating E_1 and E_2. If they are equal, then E_3 is executed and the process repeats. Otherwise, no further computation is performed for this expression. The value returned by this expression is undefined (i.e., we do not care what value this expression returns). Write pseudocode for a code generation routine that emits Cool assembly for $\texttt{while}(e_1 = e_2)\ e_3$. Assume that results are stored in register `r0`. (A Cool Assembly reference sheet is provided at the end of the exam).

```
cgen( while(e1 = e2) e3 ) =
    comp:
        cgen( e1 )
        push r0
        cgen( e2 )
        pop r5
        beq r0 r5 body
        jmp loop_end
    body:
        cgen( e3 )
        jmp comp
    loop_end:
```

(b) (6 points) Describe **one** inefficiency in the following *calling sequence* (the code executed by the caller immediately before and after a subroutine) and the *prologue* and *epilogue* (the code executed at the beginning and end) of a subroutine. How might you change these portions of code generation to avoid the inefficiency you identified? Limit your answer to four sentences. (Hint: there are at least two inefficiencies related to memory operations and at least one inefficiency related to generated code size.)

```
cgen( f(e_1, ..., e_n) ) =
    push fp
    cgen( e_1 )
    push r0
    ...
    cgen( e_n )
    push r0
    call f_entry
    sp <- add sp n
    pop fp
```

```
cgen( def f(x_1, ..., x_n) = e ) =
  f_entry:
    mov fp <- sp
    push ra
    cgen( e )
    pop ra
    return
```

(a) `sp <- add sp` n

is generated once per function call. This could me moved to the epilogue of the function definition.

(b) No space is allocated for intermediate values

(c) Rather than pushing everything to the stack, we could potentially use more registers. This would require some analysis to determine which registers to backup before performing computation.

6. Extending Cool (20 points)

We've become bored with `while` loops, and decide that we wish Cool had support for `for` loops. Our `for` loop will initialize an `Int` counter variable (the first part inside the parentheses), check a condition (the second part inside the parentheses, typically on the counter variable), and stop if the condition is false. If the condition is true, the `for` loop executes the body of the loop and then performs some additional operation (the third part inside the parentheses, typically to increment the counter variable). At this point, the process repeats from the condition check. Like `while` loops in Cool, the body of the `for` loop always has type `Object` during type checking. To avoid rewriting the code generation or interpretation stages of the Cool compiler, we choose only to modify the front-end (lexer, parser, and type-checker). Because we wish to provide rich error messages (and catch errors with the counter initialization), we will need to modify all three portions of the front-end.

Here is an example of functionality we would like to support:

```
class Main inherits IO {
    main() : Object {
        for ( i : Int <- 0 ; i < 6 ; i <- i + 1 ) do
            out_int(i);
        od
    };
    boringWhile(): Object {
        -- See? While loops are boring
        while false loop out_string( "nothing\n" ) pool
    };
};
```

This code should print out the numbers 0 through 5 in ascending order. Given the example above, note that `i` is in scope for the body expression, the comparison expression, and the additional expression. The generic form of our for loop is: `for ( id : Int <- expr ; expr ; expr ) do expr od`.

(a) (5 points) Describe the modifications you would make to the Cool lexer and associated data structures to support `for` loops. Limit your answer to at most five sentences.

We will need to add `for`, do, and od tokens to the token definitions. Additionally, we will need these tokens in the serialization format.

(b) (5 points) Describe the modifications you would make to the Cool parser and associated data structures to support `for` loops. Limit your answer to at most five sentences.

We will need to add a grammar rule for for:

$$expr \rightarrow \texttt{for (ID : TYPE <-}\ expr\ \texttt{;}\ expr\ \texttt{;}\ expr\ \texttt{) do}\ expr\ \texttt{od}$$

We will also need to create a custom serialization format for this structure.

(c) (10 points) Describe the modifications you would make to the Cool type checker and associated data structures to support `for` loops. Recall that we are not changing the code generator or interpreter. Therefore, *you should not modify the class map, parent map, implementation map, or annotated AST serialization formats*. Limit your answer to at most ten sentences. In your response, be sure to:

- (5 points) Describe how you will support execution of Cool with `for` loops with the existing back-end.
- (5 points) Propose a formal typing rule for `for` loop expressions.

Since we aren't changing the back-end, we'll have to transform the `for` loop into a `while` loop after performing type checks:

```
let i : Int <- e1 in
    while e2 loop
    {
        e4;
        e3;
    }
    pool
```

We will also need a typing rule to check that e_1 has type `Int` (in which we also just check the above conversion):

$$\frac{\begin{array}{c} T = \texttt{Int} \\ \Gamma, M, C \vdash \texttt{let}\ \ i\!:\!\texttt{Int} \leftarrow e_1\ in\ \texttt{while}\ \ e_2\ \ \texttt{loop}\ \ \{e_4;\ \ e_3;\}\ \ \texttt{pool} : \texttt{Object} \end{array}}{\Gamma, M, C \vdash \texttt{for(}\ \ i\!:\!T \leftarrow e_1\ \ \texttt{;}\ \ e_2\ \ \texttt{;}\ \ e_3\ \ \texttt{)}\ \ \texttt{do}\ \ e_4\ \ \texttt{od} : \texttt{Object}} \text{(For)}$$

If we don't want to be lazy, we might write:

$$\frac{\begin{array}{c} T = \texttt{Int} \\ \Gamma, M, C \vdash \texttt{let}\ \ e_1 : \texttt{Int} \\ \Gamma\,[\texttt{Int}/i]\,, M, C \vdash \texttt{let}\ \ e_2 : \texttt{Bool} \\ \Gamma\,[\texttt{Int}/i]\,, M, C \vdash \texttt{let}\ \ e_3 : T_3 \\ \Gamma\,[\texttt{Int}/i]\,, M, C \vdash \texttt{let}\ \ e_4 : T_4 \end{array}}{\Gamma, M, C \vdash \texttt{for(}\ \ i\!:\!T \leftarrow e_1\ \ \texttt{;}\ \ e_2\ \ \texttt{;}\ \ e_3\ \ \texttt{)}\ \ \texttt{do}\ \ e_4\ \ \texttt{od} : \texttt{Object}} \text{(For)}$$

Extra Credit (at most 5 points)

(a) (2 points max) Make an impassioned argument for either static type systems or dynamic type systems.

(b) (2 points max) If you could modify one thing about the Cool language, what would it be? Why?

(c) (1 point) Tell me one thing that is on your bucket list (something you would like to do before you die) or one obscure hobby you have.