Compiler: Review Sets

ThanhVu H. Nguyen

Contents

1 Regular Languages and Finite Automata

This Review Set asks you questions on regular languages and finite automata. Each of the questions has a short answer.

Note: For the following, unless specified otherwise, the alphabet is $\{a, b\}$.

1. Create DFA's that recognize the following languages.

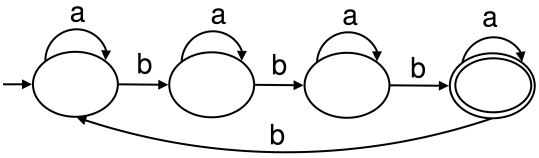
   (a) L_1: strings that contain at least one a and an even of b's follow the last a

   (b) L_2: strings that start and end with the same symbol

   (c) L_3: strings that contain the string **aab** as a substring

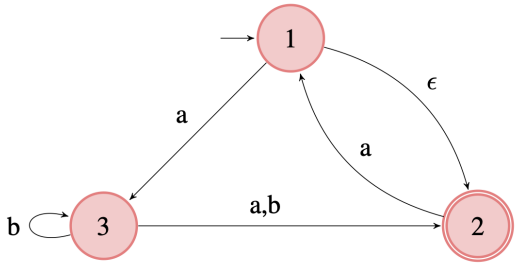
   (d) L_4: strings with lengths at most 3

2. For the following DFA, give a one-sentence description of the language recognized by the DFA. Write a regular expression for the same language.



3. Create an NFA that recognizes the language L consisting of strings that contain a b in the 3rd position from the end (e.g., $aabaa \in L$ but $aabb \notin L$).

4. Convert the following NFA to an equivalent DFA



5. Determine whether or not the following languages are regular. If yes, create a DFA for it. If no, explain in a couple of sentences.

   (a) L_1 is all strings over the alphabet $\{(,)\}$ where the parentheses are balanced. For example, $(()(())) \in L_1$ but $(() \notin L_1$.

   (b) L_2 is all binary representations (i.e., strings over the alphabet $\{0, 1\}$) of integers that divisible by 3.

   - Note that the right-most bit of w is the least-significant bit.
   - For example, the following strings are members of L: ε, 0, 11, 110, 1001, 1100, 1111, 10010, 10101, 00, 0011.

2 Context Free Grammar and Parsing

This Review Set asks you questions on regular languages and finite automata. Each of the questions has a short answer.

Note: For the following, unless specified otherwise, the alphabet is $\{a, b\}$.

1. Create the CFG's for the following languages.
   (a) G_1: over the alphabet $\{(,)\}$ that describe properly nested parentheses, e.g., $()(), ((())), (()())$
   (b) G_2: strings contain at least *three* b's
   (c) G_3: strings with **odd** length and the middle symbol is an a
   (d) G_4: strings that are palindrome
   (e) G_5: strings with more a's than b's
2. Which of the following grammars are ambiguous? For each ambiguous grammar, give an input string that can be parsed in two different ways, and draw both parse trees.
   (a) `S -> SS | a | b`
   (b) `E -> E + E | id`
   (c) `S -> Sa | Sb`
   (d)
   ```
   E -> E' | E' + E
   E' -> -E' | id | (E)
   ```
3. Create an unambiguous version of this ambiguous grammar `S -> SS | 0 | 1 | e`
4. Consider the grammar G

   ```
   S-> E
   E -> true | false
   E -> E or E | E and E
   E -> not E
   ```

   (a) Show that this grammar is ambiguous using the string `not false or true`.

   (a) Argue that this grammar is *left recursive*. Rewrite it to eliminate left recursion. That is, provide a grammar G' such that L(G) = L(G') but G' admits no derivation $X \longrightarrow_* Xa$

3 Type Checking and Operational Semantics

This Review Set asks you questions on type checking and operation semantics. Each of the questions has a short answer.

3.1 Basic Questions

1. The following typing judgments have one or more flaws. For each judgment, list the flaws and explain how they affect the judgment.

   - let-inint

```
          O |- e0 : T
          O |- T <= T0
          O |- e1 : T1
-------------------------------------  [let-init]
O[T0/x] |- let x : T0 <- e0 in e1 : T1
```

   - assign

```
   O(id) = T0
   O |- e1: T1
   T0 <= T1
-----------------  [assign]
 O|- id <- e1:T1
```

   - static-dispatch

```
  O,M,C |- e0:T0
       ...
  O,M,C |- en:Tn
      T0 <= T
  M(T0,f) = (T'1,...,T'n,T' n+1)
  T'n+1 # SELF_TYPE
  Ti <= T'i   1<=i<=n
-------------------------------- [static dispatch]
O,M,C |- e0@T.f(e1,...,en):T'n+1
```

2. Consider the following incorrect operational semantics rule for Cool let expressions.

```
so,E,S |- e1 :v1,S1
lnew = newloc(S1)
```

```
   so, E, S1 |- e2 : v2, S2
-----------------------------------  [let init]
so,E,S |- let id:T <- e in e2 :v2,S2
```

(a) Describe why this rule is incorrect.

(b) Write a corrected version of the operational semantics rule.

3. Cool only supports less-than (<) and less-than-or-equal (<=) operations. Suppose we wish to add support to greater-than (>) comparisons. Write one or more operational semantics rules to describe the evaluation of > greater-than evaluations (i.e. given so, E, S, return appropriate values and stores for e1 > e2). You may not use boolean logic (such as AND, OR, or NOT) or mathematical greater-than (or greater-than-or-equal) in your rules.

4. 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) Draw the AST for the following Cool expression

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

(b) Annotate each node in the AST you drew above to indicate the appropriate type given the typing rules and assumptions about A and B as given.

Hints: you might find the grammar and type checking rules of Cool useful

3.2 Extending Cool

3.2.1 For loop

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 interpretation stages of the Cool compiler, we choose only to modify the `front-end` (lexer, parser) and the 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 lexer, parser, and type-checker.

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
    };
};
```

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
```

1. 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.

2. 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.

3. 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 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:

(a) Describe how you will support execution of Cool with for loops with the existing back-end. In other words, how do you transform a `for` loop `for ( id : Int <- expr1 ; expr2 ; expr3 ) do expr4 od` into an existing `while` loop?

(b) Propose a formal typing rule for for loop expressions.

3.2.2 Array

Consider an extension of Cool to support arrays of objects. We introduce an Array class that inherits from Object. Other classes cannot inherit from the Array class. We introduce four new expressions for manipulating Cool arrays:

```
e ::= new Array[e]
   | e1[e2]
   | e1[e2] <- e3
   | for each vi, ve in e1 do e2
```

Subexpressions are evaluated left-to-right (e.g., e1 before e2). The first expression form creates a new array of size `e`. The array initially holds e separate copies of new Object. The size must be non-negative at runtime to avoid an exception. The second expression form reads from array e1 at index e2, returning the object stored there. The third writes to array e1 at index e2 the value e3 (and returns e3). For reads and writes, the index must be between 1 and the size of the array at runtime to avoid an exception. The final expression executes e2 for every element in array e1 with variable name vi bound to the that element's index and variable name ve bound to that element's value. Each element is considered in ascending order starting from 1. For example, this code:

```
let arr : Array <- new Array[3] in
arr[3] <- 5309;
arr[1] <- 867;
arr[2] <- "unicorn";
foreach i, elt in arr do {
  out_string("element ") ;
  out_int(i) ;
  out_string(" is ");
  case elt of
    n : Int => out_int(n) ;
    s : String => out_string(s) ;
```

```
  esac;
  out_string("\n");
};
```

Produces:

```
element 1 is 867
element 2 is unicorn
element 3 is 5309
```

Give typing rules for the four new array expressions. Be as permissive as possible without permitting any unsafe programs.

4 Optimizations and Garbage Collection

This Review Set asks you questions on type checking and operation semantics. Each of the questions has a short answer.

4.1 Optimizations

1. In the code below (** indicates exponentiation), use only a single optimization at a time, and indicate which optimization you used. Stop when there are no more optimizations to perform. Use the following optimizations we learned in class: *Algebraic simplification, Constant folding, Common subexpression elimination, Copy propagation, Dead-code elimination.*

```
a <- x ** 0
b <- a ** 2
c <- x
d <- b * d
e <- a + c
f <- b + c
g <- e + f
return g
```

2. Draw a control-flow graph for the following code. Each node in your control-flow graph should be a basic block. Every statement should appear somewhere in your control-flow drawing.

```
     i = 1
L2:  j = 1
L3:  t1 = 10 * i
     t2 = t1 + j
     t3 = 8 * t2
     t4 = t3 - 88
     a[t4] = 0.0
     j = j + 1
     if j <= 10 goto L3
     i = i + 1
     if i <= 10 goto L2
L13: i = 1
     t5 = i - 1
```

```
t6 = 88 * t5
a[t6] = 1.0
i = i + 1
if i <= 10 goto L13
```

3. Fill in the sets of live variables at all program points in the basic block below. As you compute the live variables, cross out the dead instructions and do not consider them in the computation of the remaining live variable sets. The set of live variables at exit of the block is {s, e}.

```
b := x * 0
a := a + 1
n := b
g := a + n
u := a + n
a := b ** 2
g := b * b
e := a + n
s := 2 + 3
                    {s,e}
```

4.2 Garbage Collection

1. Describe whether you should or should not implement Stop and Copy for a langue like C/C++.

2. Research (online resources/books) and describe which GC strategies are used in the following languages. For each, give a brief description on how it works for that language. Give proper citations on where you get the information.

   (a) Python
   (b) Java
   (c) Rust
   (d) Javascript
   (e) Ocaml
   (f) Another language that you find interested that is not listed above

3. Name two specific disadvantages of Mark and Sweep. (Be specific. Just saying that it it slow, for example, is not adequate.)

4. Consider the following program:

```
while not_done() {
  ptr = malloc(100 * MEGABYTE); do_work(ptr);
  /* done with ptr */
}
```

   If you run this program with 4 gigabytes of physical memory and want to use automatic memory management, would you choose Stop and Copy or Mark and Sweep? Why?

4.3 Miscs

1. List 1 thing/advice you wish you were told at the beginning of class
2. List 3 things you would advice to a student taking this class in the future