CS 536

Answer Key

Practice Final Exam

Fall 2018

1. Type checking:
   (a) $\texttt{Expr}_1$ must be a valid Boolean scalar.
   (b) $\texttt{Expr}_1$ and $\texttt{Expr}_2$ must be valid scalar expressions with the same type.
   (c) The type of the conditional is that of $\texttt{Expr}_1$(or $\texttt{Expr}_2$). Kind is value.

   Generated code is very close to that of an ordinary if statement:

```
      {Evaluate Expr1 onto stack top}
      ifeq L1
      {Evaluate Expr2 onto stack}
      goto L2
    L1:
      { Evaluate Expr3 onto stack }

    L2:
```

```
void visit(conditionalExprNode n) {
    this.visit(n.expr1);
    elseLab = genLab();
    branchZ(elseLab);
    this.visit(n.expr2);
    endLab = genLab();
    branch(endLab);
    defineLab(elseLab);
    this.visit(n.expr3);
    defineLab(endLab);

}
```

   The conditional expression

```
(i != 0) ? j/i : 0)
```

   is valid if `i` and `j` are integer scalars.

```
Generated code is:

Push i
Ifeq  L1
Ldc 1
Goto L2
L1: ldc 0
L2: ifeq L3
Push j
Push i
Idiv
Goto L4
L3: ldc 0
L4:
```

2. (a) (15 points)

We do type checking in an unusual order. When a method definition without types for parameters is found, its checking is delayed. When a call is found with valid parameters, the parameter types are copied to the method declaration and it is then type checked.

(b) (10 points) There are at two reasonable approaches:

(i) The first call found sets parameter types. Later calls must conform.

(ii) Subsequent calls (with different parameter types) force an overloading. The method definition is revisited with different types and rechecked.

3. (a) (5 points)
   We look at the exprNode that is the if's control expression. If it is a trueNode or a falseNode we code generate only the then part or else part of the if.

   (b) (10 points)
   We look at the exprNode that is the if's control expression. If it is a nameNode with a null subscriptVal we look at the idinfo link of the varName and see if it is a const (kind == val) with a literal initializer. If so, we code generate only the then part or the else part.

   (c) (10 points)
   Before we code-generate a control expression we call a new method evaluate() that walks an expression tree. If it is at a leaf and it matches the requirements of part a or part b above, we reset the adr field to literal and intval to 0 or 1. For Boolean operators we check if operands are set as literal. If so, we evaluate the expression immediately and mark the operator as a literal. If the control expression becomes marked as a literal, we code generate only the then or else part.

4. Since sum() is an instance function, we pass its object (new K()) as an invisible parameter.

   A frame for sum is pushed. The parameter is copied into the frame. The frame pointer is updated, and the old value is saved as the dynamic link. The return address is saved in the frame. sum() executes and then returns by putting the return value on the caller's stack. Frame is popped and frame pointer is reset. Return address is used to return to the caller.

5. (a) Consider the following context free grammar:

S → Label id ( ) ;
S → Label id = id ;
Label → id :
Label → λ

Is this grammar LL(1)? Why? Is this grammar LALR(1)? Why?

Not LL(1): id predicts productions 1 and 2.

Not LALR(1): Shift reduce conflict with

Label → . id :
Label → λ .

(b) Consider the following context free grammar:

S → Label id ( ) ;
S → Label id = id ;
Label → intlit :
Label → λ

Is this grammar LL(1)? Why? Is this grammar LALR(1)? Why?

Not LL(1): intlit predicts productions 1 and 2.

Is LALR(1) because

Label → . intlit :
Label → λ .

are resolved (id follows Label).

(c) Consider the following context free grammar:

S → Label id ( ) ;
S → Label id (Arg) ;
Label → intlit :
Label → λ
Arg → id
Arg → λ

Is this grammar LL(1)? Why? Is this grammar LALR(1)? Why?
Neither, because the grammar is ambiguous (productions 1 and 2 if Arg → λ).