Department of Computer Science Columbia University

Sample Final Solutions – COMS W4115

Programming Languages and Translators

Monday, April 29, 2013 2:40-3:55pm

Closed book, no aids. Do questions1-5. Each question is worth 20 points. Question 6 is extra credit, 10 points.

1. Briefly describe

a) a C compiler and a Java compiler by drawing a diagram of each
C compiler: ALSU, Fig. 1.5, p. 4 (Note C has a preprocessor.)
Java compiler: ALSU, Fig. 1.4, p. 3.

b) applicative-order evaluation and normal-order evaluation
In applicative order, the leftmost innermost redex is always reduced. In normal order, the leftmost outermost redex is always reduced.

c) a function abstraction and function application in lambda calculus
Function abstraction is a lambda expression of the form `λx.expr` that defines a function. Here `x` is a variable that is the formal parameter of the function and `expr` is lambda expression that is the body of the function. Function application is a lambda expression of the form `λx.expr1 expr2`. This expression represents the application of the function `λx.expr1` to the argument `expr2`. It asks us to replace all (free) occurrences of `x` in `expr1` by the argument `expr2`.

d) all the redexes in the lambda expression `(λx.1)((λx.x x)(λx.x x))`
There are two redexes: the entire expression is a redex and the subexpression `((λx.x x)(λx.x x))` is a redex.

2. True or false? Briefly justify your answer.

a) The set of viable prefixes of an SLR(1) grammar is always a regular language.
True. The sets of LR(0) items for an SLR(1) grammar define the states of a DFA that recognizes exactly the viable prefixes of the grammar.

b) Removing left recursion from an SLR(1) grammar always produces an LL(1) grammar.
False. Consider the left-recursive SLR(1) grammar

```
S → Sa | ab | ac
```

Eliminating the left recursion produces the grammar

```
S  → abS′ | acS′
S′ → aS′ | ε
```

which is not LL(1).

3. Consider the following C while-statement assuming `s` and `t` are strings, which in C are pointers to arrays of characters terminated by the null character '\0'.

```
while (*s++ = *t++) ;
```

Translate this statement into three-address code. Explain what your code does and how it works. State all your assumptions.

The while-statement copies the string t to the string s. The while-statement terminates when the value of the condition becomes zero, so the characters are copied from t to s, up to and including the terminating null character. The value of *t++ is the character that t pointed to before t was incremented; that character is stored in the old s position and then s is incremented. The value of the assignment statement, and hence, that of the condition, is the value of the left-hand side. Note that the body of the while-loop is empty – all the action takes place in the condition. We assume s and t are pointers to strings available on entry to the computation and s is big enough to hold t.

```
begin: t1 = *t
       t  = t + 1
       *s = t1
       s  = s + 1
       if t1 != 0 goto begin
```

4. Consider the function

```
int fact(int n) {
   if (n < 1) return 1;
   else return (n * fact(n – 1));
}
```

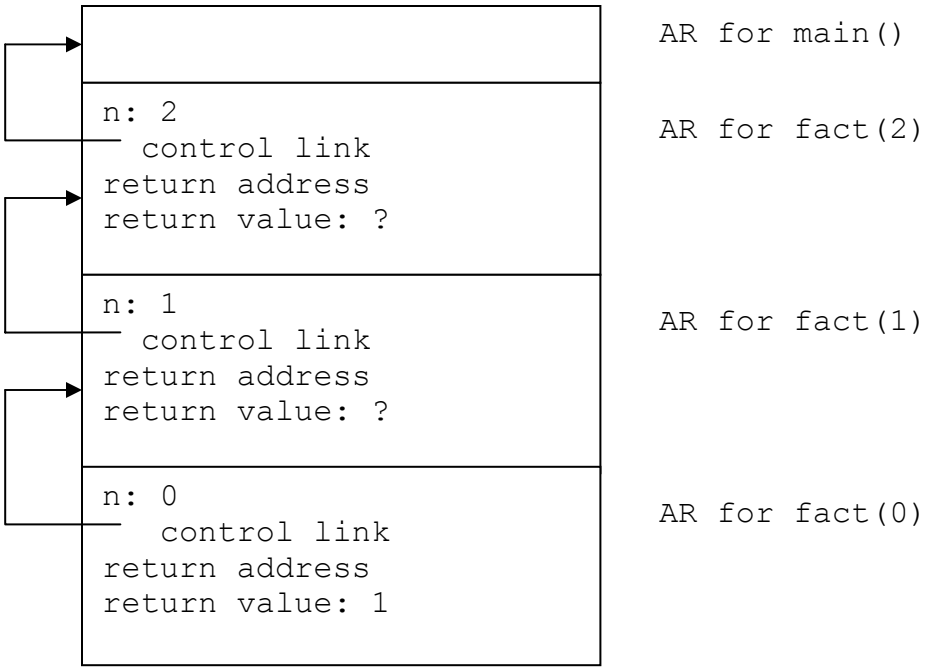
Explain in high-level terms how a machine-language program for this function would execute `fact(2)`. Show the contents of the run-time stack before, during, and after each procedure call. Explain all your assumptions.

The activation tree for fact(2) has fact(2) calling fact(1) calling fact(0). During a function call, the caller evaluates the parameters of the callee, creates a new activation record for the callee, and stores the parameters, control link (the frame pointer of the caller), return address, and other status information in the AR. It then jumps to the code for the callee, which allocates and initializes its local data and temporaries on the stack, adjusting the stack pointer as necessary. When the callee exits, it copies the frame pointer of its AR into the stack pointer, loads the control link of its AR into the frame pointer, jumps to the return address, and changes the stack pointer to pop the parameters of the callee off the stack.

Below is a picture of the run-time stack when the following procedure calls have taken place: main( ) → fact(2) → fact(1) → fact(0) and fact(0) is about to return. Initially only the AR for main( ) was on the run-time stack. When main( ) called fact(2), it put the AR for fact(2) on the stack. On each call of fact, a new AR was placed on the stack.

When fact(0) returns, it removes its AR and passes its return value to fact(1) which updates it to 1. When fact(1) returns, it removes its AR and passes its return value to fact(2) which updates it to 2. When fact(2) returns, it removes its AR and passes its return value to main( ). At this point only the AR for main( ) remains on the run-time stack.

The stack grows downward.



5. Live-variable analysis

a) Define what it means for a variable to be live at a program point.
A variable *x* is live at point *p*, if the value of *x* at *p* could be used along some path in the flow graph starting at *p*.

b) State two possible ways a compiler might use live-variable information.
Live-variable information can be used for register allocation and detecting uninitialized variables.

c) For the basic block below determine the sets of variables that are live at each program point in the basic block. You can assume that no variables are live on exit from the basic block.

The sets of live variables can be computed bottom-up, assuming the set of variables live on exit is empty.

```
     p0
a = 1
     p1
b = 2
     p2
c = 3
     p3
a = a + b
     p4
c = c - a
     p5
print a
     p6
print c
     p7
```

*p*0: { }

*p*1: { a }

*p*2: { a,b }

*p*3: { a,b,c }

*p*4: { a,c }

*p*5: { a,c }

*p*6: { c }

*p*7: { }

6. [Extra credit] Can every regular set be generated by an LL(1) grammar? Prove your answer.

Yes. Let D = (S, Σ, move, s_0, F) be a deterministic finite automaton. Let G be the grammar with the set of nonterminals S, alphabet Σ, start symbol s_0, and the following productions:

A → aB if move (A,a) = b
A → ε if A is in F

G generates the same language that is recognized by D. G is LL(1) since if A → aB | bC are two A-productions, a and b must be distinct symbols because D is deterministic. If A → ε is a production, then $ is the only symbol in FOLLOW(A).