

CS 410 Final Exam
Spring 2002 [Bono]
April 30, 2002

Solution Sheet (12 pages)

There are 11 problems on the exam, with 105 points total available.

Problem 1 [5 pts.]

For each of the following language rules (numbered 1-5) for the programming language C, name the least powerful machine that can be used to check it. Give each answer as one of a, b, or c below:

- a. FSA
- b. PDA (the book calls this a stack automaton)
- c. Turing machine (this is equivalent to a program written in a general purpose language)

Language rules:

- 1. matching { } pairs
- 2. matching types of actual parameters to formal parameters
- 3. checking that string constants are properly terminated
- 4. checking which variable declaration applies for a particular variable reference
- 5. check that identifiers contain no illegal characters

Answer :

- 1. B
- 2. C
- 3. A
- 4. C
- 5. A

Problem 2 [10 pts.].

Circle all that are true about a bottom-up parser:

- (a) its actions correspond to a leftmost derivation
- (b) its actions correspond to a rightmost derivation in reverse
- (c) it is usually table-driven
- (d) it is usually recursive-descent
- (e) it is also called a predictive parser
- (f) it may be a LALR(1) parser
- (g) it may be a LL(1) parser
- (h) it Applies Thompson's construction
- (i) it is usually generated by another program
- (j) it usually takes a grammar as input

Answer:

- (b), (c), (f), (i)

Problem 3 [20 pts.]

Create the SLR parse tables for the following grammar for postfix expressions. Show your work. In particular, you must show (1) the augmented grammar, (2) the DFA for recognizing viable prefixes, including the set of items that each state corresponds to, and (3) the action and goto tables

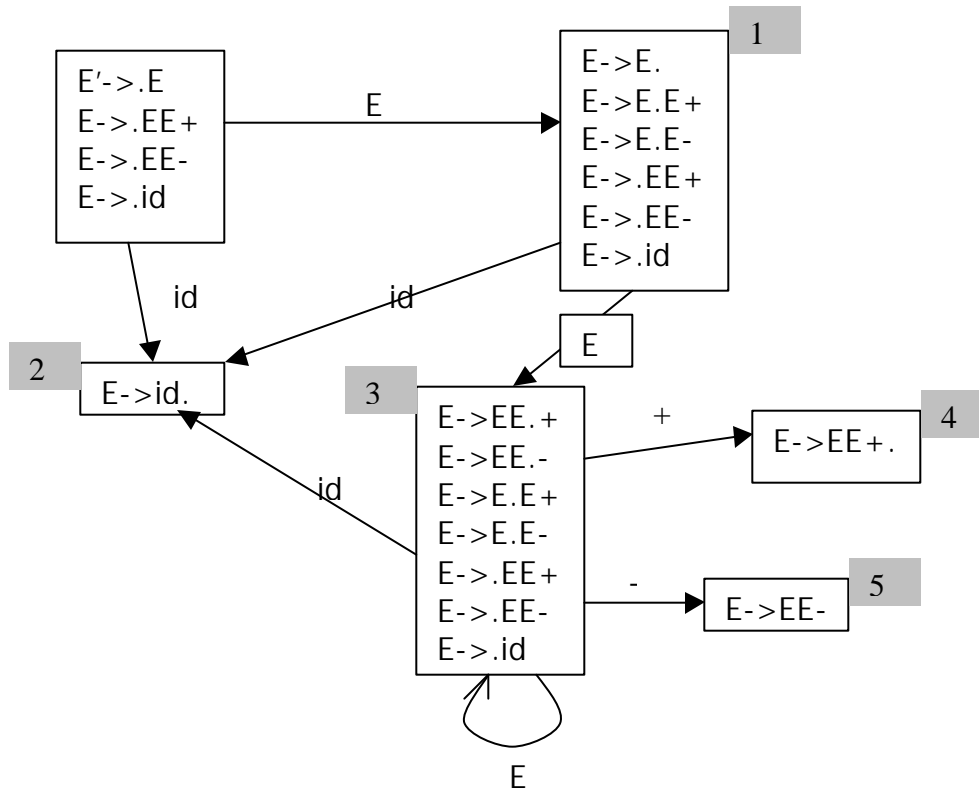
- (1) $E \rightarrow E E +$
- (2) $E \rightarrow E E -$
- (3) $E \rightarrow id$

Answer:

$$FOLLOW(E) = \{\$, ID, +, -\} = \{\$, ID, +, -\}$$

Augmented Grammar:

- (0) $E' \rightarrow E$
- (1) $E \rightarrow E E +$
- (2) $E \rightarrow E E -$
- (3) $E \rightarrow id$



action

goto

	+	-	ID	\$	E
0			s2		1
1			s2	acc	3
2	r 3	r 3	r 3	r 3	
3	s4	s 5	s2		3
4	r 1	r 1	r 1	r 1	
5	r 2	r 2	r 2	r 2	

Problem 4 [6 pts.]

Part A. Construct a short, but complete, C++ program that includes one call to `foo` such that using pass *by reference* with formal parameter `x` produces different output than using pass *by value-return* with `x` (i.e., for the same call in the same program). Write your answer on the lower part of this page, starting with the code given there.

Part B. Show the output of the program you gave for Part A assuming each of the two parameter-passing techniques:

- a. Output using call by reference:
 - b. Output using call by value-return:
-

Answer:

(There can be many possible solutions to this question. One of the answers:)

Part A:

```
int y = 100;

void foo (int x)
{
    x = 10;
    cout << y;
}

main ()
{
    foo(y)
}
```

Part B:

- a. 10
- b. 100

Problem 7 [5 pts.]

Assume we have the following C-style array declaration,

XType foo[10][20];

and furthermore suppose that the size of an **XType** is **XSize** bytes, and that two-dimensional arrays are stored in **column-major** order.

Give an expression for the correct byte offset to access **foo[i][j]** (note: this doesn't have to be written using 3AC, just express it mathematically):

Answer:

$10*j*XSize + i*XSize$

Problem 8 [9 pts.]

For each of the following data structures, say whether it exists at compile-time (C) or run-time (R):

Answer:

(Marked at the beginning of each subpart)

R (a) call stack

C (b) grammar stack

C (c) semantic stack

C (d) symbol table

R (e) dispatch table

C (f) abstract syntax tree

R (g) activation record

R (h) heap

R (i) display

Problem 9 [12 pts.]

Consider the following for loop:

```
for (int i = 0; i < n; i += 2) {  
    A[i] = B[i] + C[i];  
}
```

Suppose we generated the following unoptimized 3AC (three-address code) for the loop:

```
    i := 0;  
loop:  
    if (i >= n) goto after;  
    t1 := i * 4;  
    t2 := b[t1];  
    t3 := i * 4;  
    t4 := c[t3];  
    t5 := t2 + t4;  
    t6 := i * 4;  
    a[t6] := t5;  
    i := i + 2;  
    goto loop;  
after:
```

This problem concerns optimizing this code. Note: the optimizations you apply will be "by hand", rather than using compiler algorithms (e.g., no DAG creation necessary).

Part A. If possible, apply *common-subexpression elimination* to the 3AC above, or explain why we can't apply that optimization.

Part B. If possible, apply the *loop version of reduction in strength* to the most recent version of the 3AC (i.e., the code for the previous part if you gave some, otherwise the original code), or explain why we can't apply that optimization.

Part C. If possible, apply *code motion* to the most recent version of the 3AC, or explain why we can't apply that optimization.

Part D. If possible, apply *loop unrolling* to the most recent version of the 3AC, or explain why we can't apply that optimization.

(space for your answers is provided on the next page)

Answer:

Part A :

common-subexpression elimination

```
    i := 0;
loop:
    if (i >= n) goto after;
    t1 := i * 4;
    t2 := b[t1];
    t4 := c[t1];
    t5 := t2 + t4;
    a[t1] := t5;
    i := i + 2;
    goto loop;
after:
```

Part B :

loop version of reduction in strength

Proceeding from above:

```
    i := 0;
    t1 = 0;
loop:
    if (i >= n) goto after;
    t2 := b[t1];
    t4 := c[t1];
    t5 := t2 + t4;
    a[t1] := t5;
    i := i + 2;
    t1 := t1 + 8;
    goto loop;
after:
```

Part C :

code motion

It cannot be applied here. There are no invariant expressions within the loop.

Part D :

loop unrolling

It cannot be applied here. 'n' is not a constant and thus there is no constant number of loop iterations.

Problem 10 [14 pts.]

Suppose we are designing an activation record for a language whose functions take the following form, shown by example:

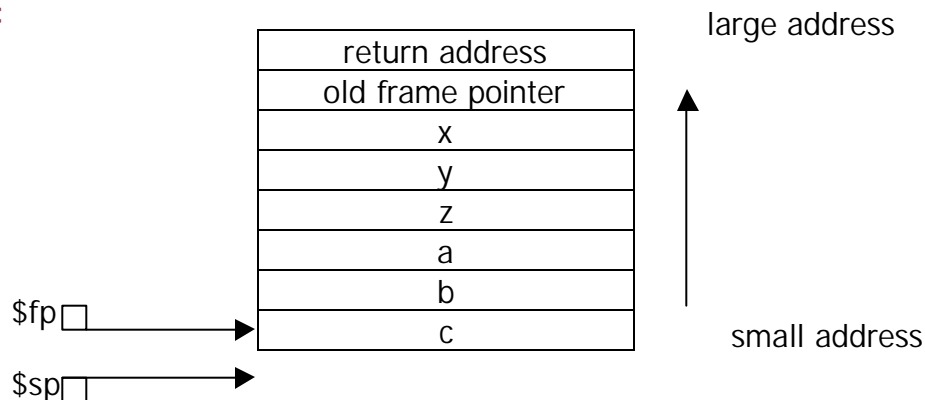
```
def foo(x, y, z) = {var a, b, c; funcBodyExpr }
```

More details about what this function means:

- all values are integers (that's why there are no types on the variable declarations; also, no objects in this language). Integers are 4 bytes.
- x , y , and z are parameters and a , b , and c are local variables
- expressions do not introduce any new variables
- expressions may be function calls, including recursive calls
- We'll be using a stack machine: see additional handout for details on this and pseudo-MIPS.

Part A. Show a possible layout for the activation record for the example function shown above (there's more than one correct answer). In your answer, show where the stack pointer and frame pointer are pointing after constructing the activation record. Show the stack growing downward (i.e., towards the bottom of the page).

Answer:



Part B. Show the code that would be generated for a reference to c , from the example function shown, assuming you are using the activation record you gave in part A. (Your answer should consist of one or more complete MIPS or pseudo-MIPS instructions.)

Answer:

```
$a0 := 0($fp)
```

Part C Show the code that would be generated for a reference to y , from the example function shown, assuming you are using the activation record you gave in part A. (Your answer should consist of one or more complete MIPS or pseudo-MIPS instructions.)

Answer:

```
$a0 := 16($fp)
```

Problem 11 [16 pts.]

Consider the Java operator *instanceof* which tests if the run-time type of an object conforms to a particular class. It returns a boolean: true if the run-time type of the object *does* conform to the given type, and false otherwise. In Java this allows for safe downcasting. Here's an example of its use in Java (assume Circle is a subclass of Shape):

```
Shape s;  
...  
if (s instanceof Circle) {  
    return (Circle) s.getRadius(); // getRadius is not defined  
} // for other kinds of Shapes  
else {  
    return 0;  
}
```

Part A [4]. Write a Cool expression equivalent to the if-else statement above (assume the necessary definitions of Shape, Circle, s, and getRadius exist). One difference: instead of using "return", the *value* of expression you write be the same as what was *returned* in the original version. Note: your answer will *not* use instanceof, since Cool doesn't have this feature.

Answer:

```
case s of  
    c : Circle => c.getRadius();  
    s : Shape => 0;  
esac
```

Part B [4]. Consider adding **instanceof** to Cool (same semantics as described above). Here's the syntax:

expr -> expr **instanceof** Type

Using logic rules of inference, write a type-rule for instanceof. To remind you of the notation,

we'll give you the rule for let-with-init. (Hint: instanceof has a much simpler rule.)

$$O, M \vdash e1 : T1$$
$$T1 \leq T0$$
$$O[T0 / x], M \vdash e2 : T2$$

$$O, M \vdash \text{let } x : T0 \leftarrow e1 \text{ in } e2 \text{ end} : T2$$

Answer:

$$O, M \vdash e1 : T1$$
$$T1 \leq T$$

$$O, M \vdash e1 \text{ instance of } T : \text{Bool}$$

Problem 11 [cont.]

Part C [8]. Write a code-generation routine for **instanceof**. Here again is the description of what instanceof does: it tests if the run-time type of an object conforms to a particular class. It returns a boolean: true if the run-time type of the object *does* conform to the given type, and false otherwise.

More precisely, here is the operational semantic rule for **instanceof** in Cool:

$$s_0, E, S \mid - e : v, s_1$$
$$v = X(\dots)$$
$$v_1 = \begin{array}{l} \text{Bool (true), if } X \leq T \\ \text{Bool (false), otherwise} \end{array}$$

$$s_0, E, S \mid - e \text{ instanceof } T : v_1, s_1$$

Refer to the additional handout for details on the stack machine we'll be using and on pseudo-MIPS notation. Furthermore, you may assume:

- we already generated type tags for the classes by doing a *preorder* traversal on the inheritance hierarchy. **Hint:** In this scheme, **Object** has type tag 0; if the tag for type T has value v, then the tags for all types in the subtree rooted at T have values greater or equal to v. (*necessary but not sufficient*)
- the first word in every object is its type tag. *0(\$a0)*
- we'll show the input of the routine to be the source code itself; you can call cgen recursively on parts of the expression to generate code for those parts
- you may use the function newLabel() to generate unique labels (returns a char*).
- output your code by using << statements to cout

Answer: (please look at comments in blue italic above)

Assumptions:

- bool_const1 is the label of the const bool true.
- bool_const0 is the label of the const bool false.
- At compile time, T.tag contains tag for type T,
T.maxtag contains max tag in subtree rooted at T (i.e. max tag of types that conform to T)

```
cgen (e instanceof T) =
    cgen(e)
    string after = newLabel();
    string falseLab = newLabel();
    cout<<"$t0 := 0($a0);"           //e's tag
    cout<<"if ("<<T.tag<<">$t0) goto"<< falseLab;
    cout<<"$a0 := bool_const1";
    cout<< "goto"<<after;
    cout<<falseLab<<":";
    cout<<"$a0 := bool_const0;";
    cout<<after<<":";
```