Final practice solutions

Where and when?

Wednesday December 13, 8:30-11:30am Kresge Auditorium (in the Law School)
Open notes/closed book

1) The embedded action will introduce a shift-reduce conflict. Yacc created an anonymous non-terminal that reduces on epsilon for it. After putting the type and identifier on the parse stack and looking ahead to the '(', the parser can either shift ''(' for a function declaration or reduce the embedded action in the function definition. It's too early at this point to know whether we are processing a declaration or a definition, so we have a conflict. We can't distinguish the two until we have finished the parameter list and can look at the next token to see whether it is ';' or '{'. One way to resolve the conflict is to gather the parameters in a DeclList for both declaration and definition and delay pushing the scope for definition until after the closing ')'.

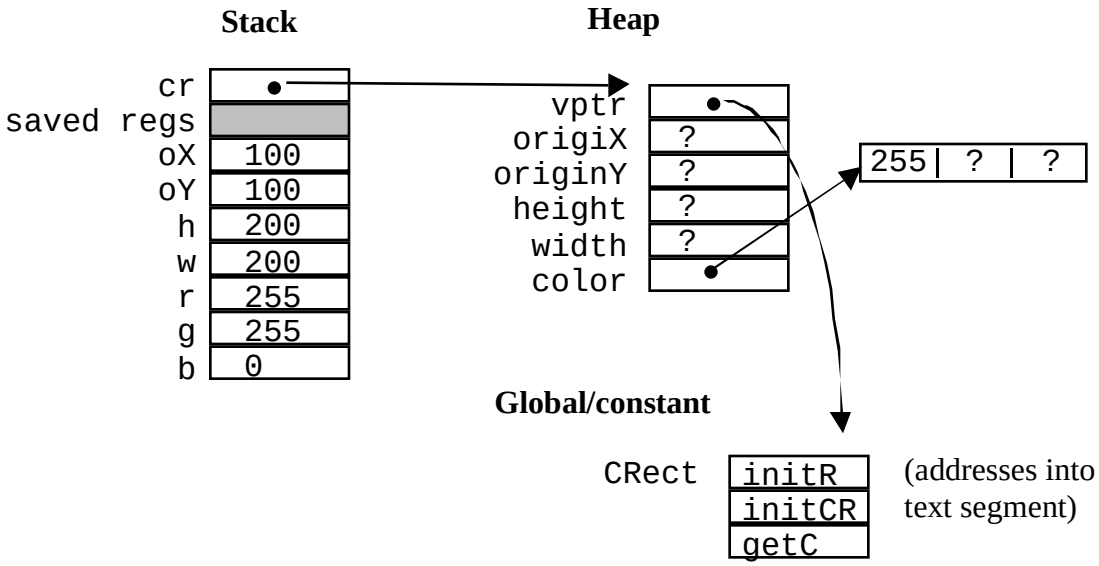
2) For example, Designator can expand to Expression[Expression]. The subscript expression is not limited to integer type, although that constraint must be enforced eventually. One could change it to Expression[IntegerExpression] and add a whole host of productions to work with IntegerExpression (arithmetic, relational, return values, etc.) trying to track expressions with integer type separately. This appears messy and complicated, but more importantly, it just won't work. The parser *cannot* be solely responsible for ensuring the array subscript is integer. For example, the variable name 'x' might refer to an integer in one context and not in another. A context-free grammar can't do both, i.e. it cannot allow IntegerExpression to expand to 'x' sometimes and not others (this violates the whole notion of context-free). So it must either always allow it or always disallow it, but there are programs where arr[x] is valid and others where it is not. Consider the alternative of having one check in the semantic analyzer to verify that the expression type was integer. It's simple, easy to get right, can handle the context-sensitivity, and allows the compiler to provide a target error message ("Array subscript not integer") when violated as opposed to the parser's vague syntax error.

3) Globals are particularly inappropriate when the state you are trying to record is not, in fact, global. Much of the information the compiler tracks is specific to the current scope. Given the possibility of nested scopes, it is a better idea to track that info in the scope stack, where it can be kept with the scope level. This allows the compiler to easily remove that state when that scope is closed and pick back up with whatever state was in place in the outer scope.

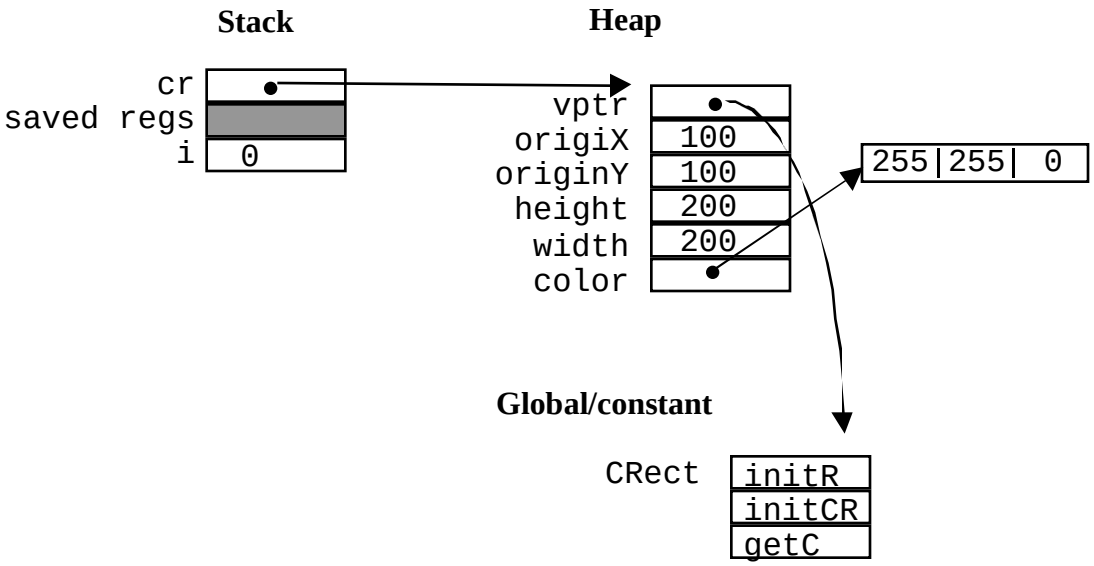
4) Synthesized attributes are those passed up a parse tree, so we start at the leaves and work our way upward, creating the left-side attribute from the right-sides:

```
B -> 0    {B.val = 0}
   | 1    {B.val = 1}

L -> LB   {L.val = L.val * 2 + B.val}
   | B    {L.val = B.val}

S -> L.L  {S.val = L1.val + L2.val / 2^(NumBinaryDigits(L2.val))
   | L    {S.val = L.val}
```

5) Midway through the call to initColorRect:



During the call to getColor:



How much detail should you know about the SOOP runtime memory structures? You should know where things are (what is in stack, what is in heap, and what is in global/constant), and the runtime layout for variables, arrays, and objects and their vtables. The order of data within a stack frame (parameters, locals, temporaries, saved registers) is not that interesting, but you should know the general structure of the stack and how each call has its own activation record.

6) There are other equivalent/correct TAC sequences as well:

```
    Var i;
    Var t0;
    t0 = 0;
    i = t0;
L0:
    Var t1;
    t1 = 12;
    Var t2;
    t2 = i < t2;
    ifZ t2 Goto L1;
    Arg i;
    PrintInteger();
    t3 = 1;
    i = i + t3;
    Goto L0;
L1:
    String L2 = "Done.";
    Arg L2;
    PrintString();
```

7a) The syntax we chose looks like this:

```
class Binky {
    void Constructor(int x, double d, bool b) { . . . }
}

void main(void)
 {
   Binky *b;
   b = New(Binky(3, 2.5, true));
 }
```

The parentheses after the classname inside New will not be optional. Even if the constructor takes no arguments, the empty parentheses will be required. When defining a constructor for the class, the method must have the name "Constructor" and the return type must be void. We decided constructors must always be public (trying to mark private will be error, okay to explicitly call public). We will not put the Constructor into the vtable since it is not inherited and cannot be overridden. When a New expression is parsed, the class name must be explicitly given, and we will look up and use that constructor as specified at compile-time.

b) We add the reserved word "Constructor" to the scanner with its own unique token type. (Alternatively we could check for identifier "Constructor" when processing a method definition to recognize the constructor, that would change some of the steps below).

c) Given Constructor is a reserved word, we add another expansion to ClassField to match a Constructor, since our previous method definition looks for a T_Identifier:

```
ClassField:     ...
          | OptPublic T_Void T_Constructor '(' ParamsOrVoid ')'
            CompoundStatement
```

The New production changes to:

```
Expression:      ...
```

```
| T_New '(' T_Identifier (' ArgumentList ')' ')'
```

d) When processing a constructor definition, all of the previous function checking needs to be done (no re-use of parameter name, no redefinition, definition matches any earlier declaration, etc.) along with a few additional constructor-specific checks:

1) the return type must be void. This is actually checked by the grammar we gave in part c. Alternatively the production could allow Type and then report a more helpful error about incorrect constructor return type than the parse error that will be given if specified as above.

2) the constructor must be public. The production from c) disallows use of private, but again, it might be more helpful for the parser to accept private anyway, and have the semantic analyzer provide a helpful message in this context.

In the action for a New expression, we check that the class name is valid (as before), and then do a look up of the identifier "Constructor" in that class's scope and re-use our existing function call checking to ensure the actual arguments used here match the formal arguments in number and type. We should not need to verify that the method return type is void, since we earlier disallowed any constructor to have a non-void return type.

In the action for function/method call, we should disallow calling the constructor manually. If "Constructor" is a reserved word, such an attempt won't parse at all. Alternatively, the semantic analyzer could print an error.

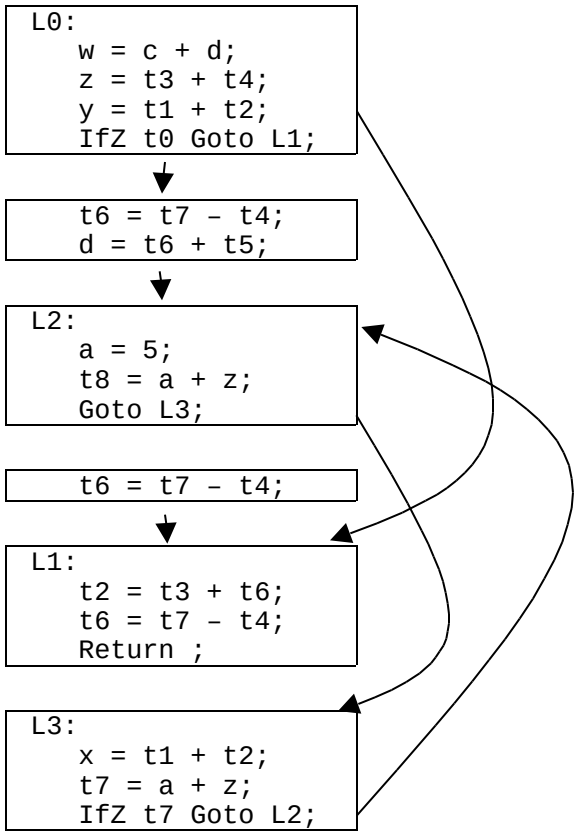
We also need to verify that each class does define a constructor (the rules state each class must define exactly one). At the end of processing ClassDefinition, there must be an additional semantic check to look up "Constructor" in the class scope and verify there was such a method defined, otherwise we emit an error about the need for the constructor.

Although we didn't ask you to consider inheritance, you should not to copy the Constructor declaration from parent scope to child scope (similar to the way we skip "this" when copying the fields from the parent scope).

e) The code generated for a constructor definition (accessing params, setting up locals, using this, return, etc.) will be exactly the same as for other method definitions, so no changes are required when generating code for the constructor's definition.

Generating code for a New expression will involve all the previous pp5 set-up for using the built-in New operation to allocate the memory and set the vptr. Then we will need to generate code to call the constructor. This should be able to re-use our existing code for function calls. We need to evaluate each expression in the argument list, add each parameter to the stack, and then jump to the target method. In this case, we can look the label up from the class scope and directly jump there, we do not need to make an indirection through the vtable, since we know exactly which class's constructor we are trying to invoke.

8) (That little block in the middle is unreachable, by the way.)



9) Instructions removed due to DCE are crossed out. Instructions that didn't change in the pass are grayed out. Although the last instruction in the final result is pointless, removing that redundancy is not a common optimization made by a compiler.

	CP (+ DCE)	CF	CP (again)	AI
t0 = 0;	~~t0 = 0;~~			
i = t0;	~~i = 0;~~			
t2 = 4;	~~t2 = 4;~~			
t3 = t2 * i;	t3 = 4 * 0;	t3 = 0;	~~t3 = 0;~~	
t1 = a + t3;	t1 = a + t3;	t1 = a + t3;	t1 = a + 0;	t1 = a;
t4 = 1;	~~t4 = 1;~~			
*t1 = t4;	*t1 = 1;	*t1 = 1;	*t1 = 1;	*t1 = 1;
t5 = 1;	~~t5 = 1;~~			
t6 = i + t5;	t6 = 0 + 1;	t6 = 1	~~t6 = 1~~	
i = t6;	i = t6;	i = t6;	~~i = 1~~	
t8 = 4;	~~t8 = 4;~~			
t9 = t8 * i;	t9 = 4 * i;	t9 = 4 * i;	t9 = 4 * 1;	t9 = 4;
t7 = a + t9;	t7 = a + t9;	t7 = a + t9;	t7 = a + t9;	t7 = a + t9;
t10 = 2;	~~t10 = 2;~~			
*t7 = t10;	*t7 = 2;	*t7 = 2;	*t7 = 2;	*t7 = 2;
t11 = 1;	~~t11 = 1;~~			
t13 = 4;	~~t13 = 4;~~			
t14 = t13*t11;	t14 = 4 * 1;	t14 = 4;	~~t14 = 4;~~	
t12 = a + t14;	t12 = a + t14;	t12 = a + t14;	t12 = a + 4;	t12 = a + 4;
t16 = 4;	~~t16 = 4;~~			
t17 = t16 * i;	t17 = 4 * i;	t17 = 4 * i;	t17 = 4 * 1;	t17 = 4;
t15 = a + t17;	t15 = a + t17;	t15 = a + t17;	t15 = a + t17;	t15 = a + t17;
*t12 = *t15	*t12 = *t15	*t12 = *t15	*t12 = *t15	*t12 = *t15

CP (again)	CSE	CPP	Final
t1 = a;	t1 = a;	~~t1 = a;~~	
*t1 = 1;	*t1 = 1;	*a = 1;	*a = 1;
~~t9 = 4;~~			
t7 = a + 4	t7 = a + 4	t7 = a + 4	t7 = a + 4;
*t7 = 2;	*t7 = 2;	*t7 = 2;	*t7 = 2;
t12 = a + 4;	~~t12 = a + 4;~~		
~~t17 = 4;~~			
t15 = a + 4;	~~t15 = a + 4;~~		
*t12 = *t15	*t7 = *t7	*t7 = *t7	*t7 = *t7;