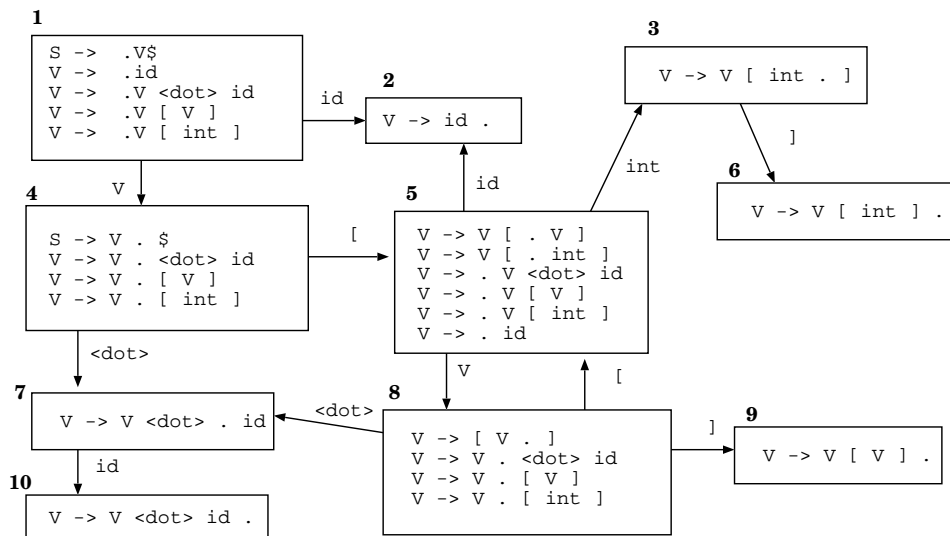


Computer Science 414
Spring 2010
Midterm 2 Practice Problems

1. Give the LR(0) states and transitions, as well as the LR(0) parse table, for the following grammars

(a)

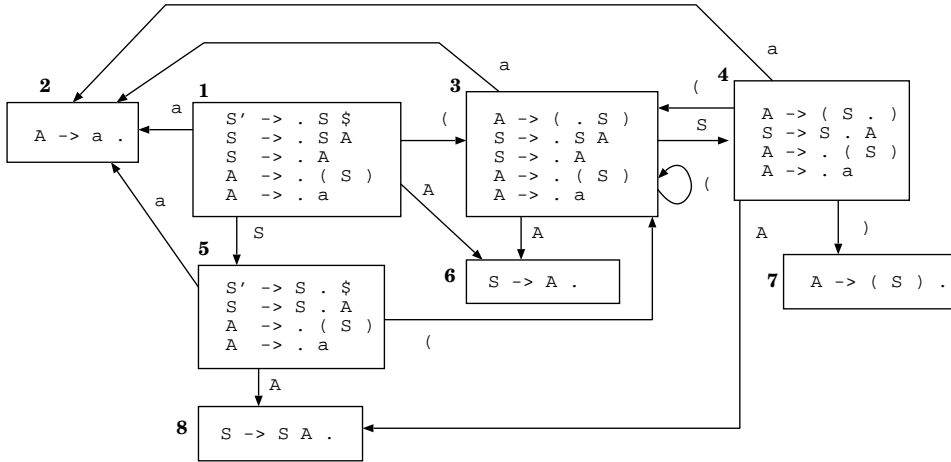
Terminals = {id, ., [,], int }
 Non-Terminals = {S, V}
 Rules = (0) $S \rightarrow V\$$
 = (1) $V \rightarrow id$
 = (2) $V \rightarrow V \cdot id$
 = (3) $V \rightarrow V [V]$
 = (4) $V \rightarrow V [int]$
 Start Symbol = S



	int	id	[]	<dot>	\$	V	S
1		s2					g4	
2	r(1)	r(1)	r(1)	r(1)	r(1)	r(1)		
3				s6				
4			s5		s7	accept		
5	s3	s2					g8	
6	r(4)	r(4)	r(4)	r(4)	r(4)	r(4)		
7		s10						
8			s5	s9	s7			
9	r(3)	r(3)	r(3)	r(3)	r(3)	r(3)		
10	r(2)	r(2)	r(2)	r(2)	r(2)	r(2)		

(b)

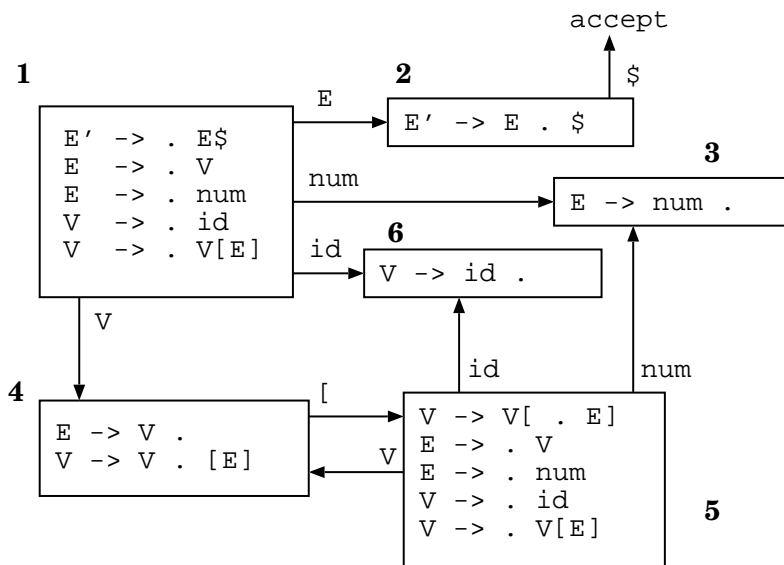
Terminals = {a, (,), \$}
 Non-Terminals = {S', S, A}
 Rules = (0) $S' \rightarrow S\$$
 = (1) $S \rightarrow SA$
 = (2) $S \rightarrow A$
 = (3) $A \rightarrow (S)$
 = (4) $A \rightarrow a$
 Start Symbol = S'



	a	()	\$	S	A
1	s2	s3			g5	g6
2	r(4)	r(4)	r(4)	r(4)		
3	s2	s3			g4	g6
4	s2	s3	s7			g8
5	s2	s3		accept		g8
6	r(2)	r(2)	r(2)	r(2)		
7	r(3)	r(3)	r(3)	r(3)		
8	r(1)	r(1)	r(1)	r(1)		

2. Create a set of LR(0) items and transitions for the following grammar. Show that the grammar is not LR(0). Give the SLR(1) parse table for the grammar.

- Terminals = {num, id, [,], \$}
 Non-Terminals = {E', E, V}
 Rules = (0) E' → E\$
 (1) E → V
 (2) E → num
 (3) V → id
 (4) V → V[E]
 Start Symbol = E'



The LR(0) parse table has duplicate entries:

	num	id	[]	\$	E	V
1	s3	s6				g2	g4
2					accept		
3	r(2)	r(2)	r(2)	r(2)	r(2)		
4	r(1)	r(1)	r(1),s5	r(1)	r(1)		
5	s3	s6					g4
6	r(3)	r(3)	r(3)	r(3)	r(3)		

First and follow sets:

Non-Terminal	First	Follow
E'	num, id	
E	num, id	\$,]
V	id	\$,], [

The SLR(1) parse table:

	num	id	[]	\$	E	V
1	s3	s6				g2	g4
2					accept		
3				r(2)	r(2)		
4			s5	r(1)	r(1)		
5	s3	s6					g4
6			r(3)	r(3)	r(3)		

3. When creating the Abstract Assembly for a function definition, Do we *need* to store the Stack Pointer on the stack? What about the Frame Pointer and Return Register? Explain.

We do not need to save the old value of the stack pointer on the stack. Since we are subtracting constant value from the stack pointer at the beginning of the function, we could just add that same value to the stack pointer at the end of the function. Our buildTree function for a function definition would be something like:

```
functionDefinition(body, framesize, start, end)
```

```
  Create tree as follows:
```

```
    start label
    code to save old FP
    code to save old Return Register
    FP <- SP
    SP <- SP - framesize
    body
    end label
    code to restore old FP from stack
    code to restore old Return register from stack
    SP <- SP + framesize
```

We do need to save the old frame pointer. Since we don't know who called us (and how big their activation record is), we need to save the old value of the FP, we can't recalculate it

If we are a "sub" function (that is, a function that does not call any other functions), then we do *not* need to save the Return register. However, if we are not a stub function (that is, we do call at least one other function), then we *do* need to save the Return register.

4. Given the following class definitions, and the local variable declarations in the function foo:

```

class c1 {
    int w;
    boolean z;
}

class c2 {
    c1 y[];
    int z;
}

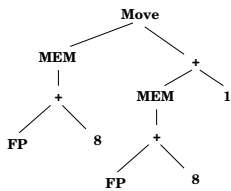
int foo(int x,y) {
    boolean a;
    boolean b[];
    c1 C;
    c2 D[];

    /* Body of foo */
}

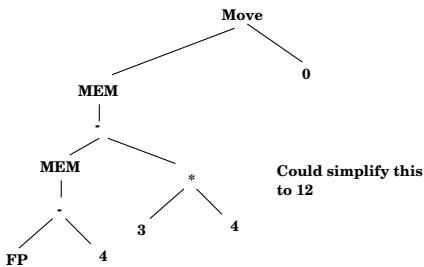
```

Give the abstract assembly tree for each of the following statements, if they appeared in the body of foo:

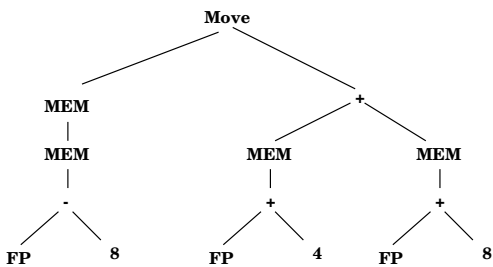
(a) `y++;`



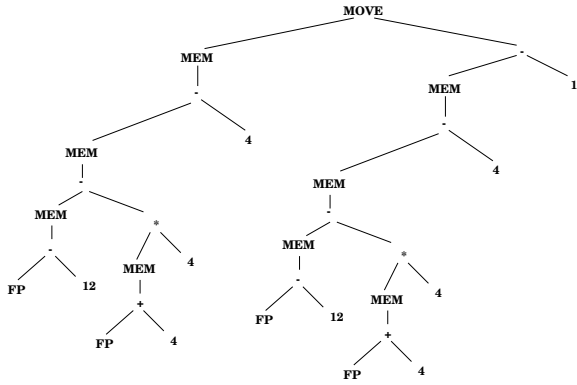
(b) `b[3] = false;`



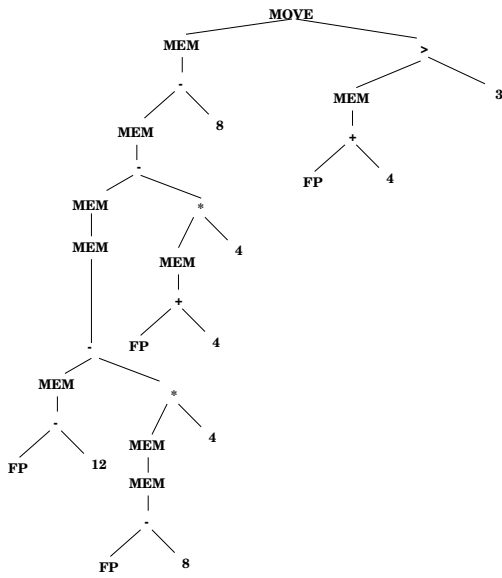
(c) `C.w = x + y;`



(d) `D[x].z--;`



(e) $D[C.w].y[x].z = x > 3;$



(f) $\text{while } (x < 10) \ x = x + 3;$

