CSCI 565 - Compiler Design

Spring 2013

Midterm Exam - Solution

March 6, 2013 at 3:30 PM in class (RTH 115)

Duration: 2h 30 min.

Problem 1: Regular Expressions and DFAs [20 points]

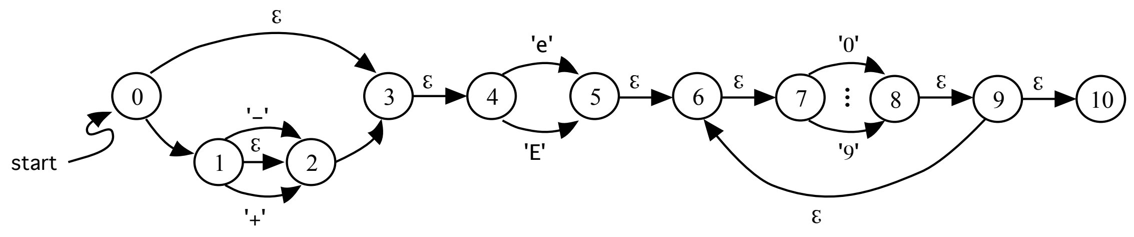
For the regular expression depicted below answer the following questions:

RE = (ε | - | +)? . (e | E) . (0-9)+

a) [05 points] Using the Thompson construction (or a simplification thereof) derive a NFA that recognizes the strings specified by this RE.
b) [10 points] Convert the NFA in a) to an equivalent DFA using the subset construction.
c) [05 points] Make sure the DFA M_1 found in b) above is minimal (you do not need to show that it is in fact minimal) and construct the DFA M_2 that accepts the complement of the regular language the first DFA accepts. Show that M_2 does accept the word w = "10e" which is not accepted by the original DFA.

Solution:

a) The figure below depicts the NFA with ε-transitions and where we have simplified the transition on numeric digits.



b) We use the subset construction by tracing for every set of states the other possible states the NFA could be in if it were to traverse ε-edges only. The list below depicts the results of this construction where we also describe the computation of edge DFA edge using the function DFA_edge and then computing the e-closure of the resulting intermediate set of states.

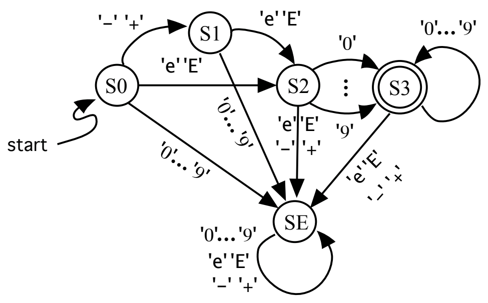
ε-closure({0}) = { 0, 1, 2, 3, 4} = S0
DFA_edge(S0,'-') = ε-closure({1}) = { 2, 3, 4} = S1
DFA_edge(S0,'+') = ε-closure({2}) = { 2, 3, 4} = S1
DFA_edge(S0,'e') = ε-closure({5}) = { 5, 6, 7} = S2
DFA_edge(S0,'E') = ε-closure({5}) = { 5, 6, 7} = S2
DFA_edge(S0,'<digit>') = ε-closure({}) = SE; error state

DFA_edge(S1,'-') = ε-closure({}) = SE; error state
DFA_edge(S1,'+') = ε-closure({}) = SE; error state
DFA_edge(S1,'e') = ε-closure({5}) = { 5, 6, 7 } = S2
DFA_edge(S1,'E') = ε-closure({5}) = { 5, 6 ,7 } = S2
DFA_edge(S1,'<digit>') = ε-closure({}) = SE; error state

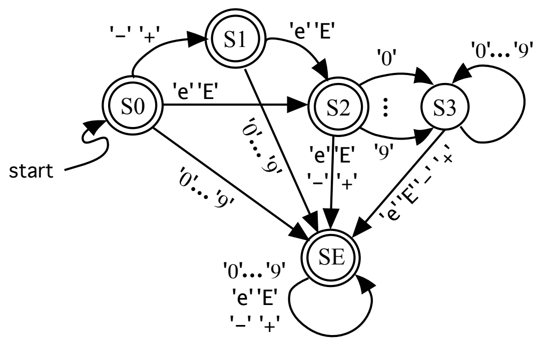
DFA_edge(S2,'-') = ε-closure({}) = SE; error state
DFA_edge(S2,'+') = ε-closure({}) = SE; error state
DFA_edge(S2,'e') = ε-closure({}) = SE; error state
DFA_edge(S2,'E') = ε-closure({}) = SE; error state
DFA_edge(S2,'<digit>') = ε-closure({8}) = { 6, 7, 8, 9, 10 } = S3;

DFA_edge(S3,'-') = ε-closure({}) = SE; error state
DFA_edge(S3,'+') = ε-closure({}) = SE; error state
DFA_edge(S3,'e') = ε-closure({}) = SE; error state
DFA_edge(S3,'E') = ε-closure({}) = SE; error state
DFA_edge(S3,'<digit>') = ε-closure({8}) = { 6, 7, 8, 9, 10 } = S3;

The figure below depicts the DFA resulting from the application of the subset construction. Notice that this is a fully specified DFA where each state includes transition for all the characters in the considered alphabet.



c) Let M_1 be the DFA found in b). This DGA is in fact minimal. Informal proof: All states are required. S0 is the start state. S1 captures the fact that we have observed a sign.. State S2 captures to fact we need to observe at least one digit. All the digits are then captured by state S2. State SE is the trap error state.
Let M_1 be this first DFA. We construct M_2 by making all accepting states of M_1 non-accepting states and all non-accepting state s of M_1 as accepting states of M_2. The state start is unchanged. This is a DFA as we only change the state of non-accepting states. The start state is still a singleton and all transitions are clearly identified in the newly created DFA as shown below.



In this newly created DFA the input string w = "10e" takes the DFA to state SE which is an accepting state n this "complement" machine.

Problem 2: Context-Free-Grammars and Parsing Algorithms [50 points]

Consider the CFG with non-terminal symbols N={S, E, A}, with start symbol S, terminal symbols T={ **id**, ';' , '=' } and the productions P listed below.

(1) <S> → <E> $
(2) <E> → <E> ';' <A>
(3) <E> → <A> ';'
(4) <A> → **id**
(5) <A> → <A> '=' <A>

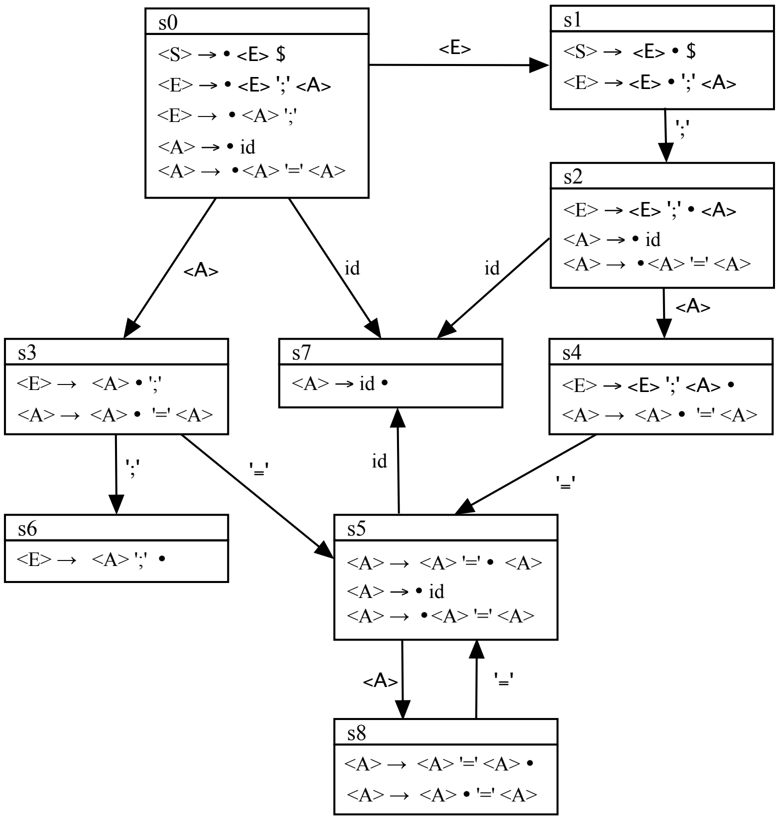
Questions:

a) [05 points] As specified, can this grammar be parsed using a predictive LL algorithm? Why or why not?
b) [15 points] Compute the DFA that recognizes the LR(0) sets of items for this grammar and construct the corresponding LR(0) parsing table. Comment on the nature of the conflicts, if any.
c) [05 points] How different would the SLR table look like? If there were conflicts in the original table will this table construction algorithm resolve them?
d) [10 points] Can you use operator precedence or associativity to eliminate conflicts? Explain your rationale.
e) [15 points] Explain how you would recover from the input w = "`id  = id id ; $`". Refer to the table in b) and indicate what the set of recovery non-terminals and synchronizing terminals would be.

Solution:

a) As this is a left-recursive grammar it does not have the LL(1) property and as such cannot be parsed using a predictive top-down LL parsing algorithm.

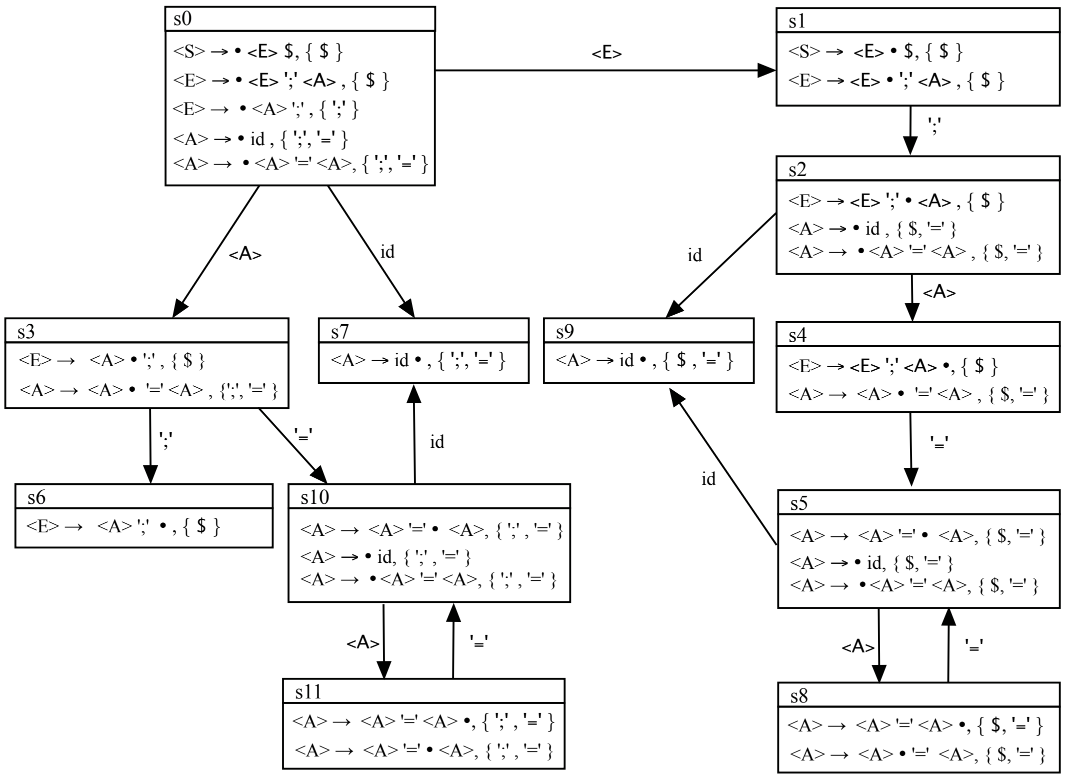
b) The figure below depicts the set of LR(0) items and the corresponding DFA.



c) As can be seen there are two shift/reduce conflicts respectively in states S4 and S8 both on '='. This is because we have the continuous assignments where we can chain multiple assignment in a single statement. At every identifier we could reduce but we can also try to shift the upcoming '=' terminal.

State	**Action**				**Goto**	
	id	'='	';'	$	<E>	<A>
s0	shift s7				goto s1	goto s3
s1			shift s2	accept		
s2	shift s7					goto s4
s3		shift s5	shift s6			
s4	reduce (2)	shift s5 reduce (2)	reduce (2)	reduce (2)		
s5						goto s8
s6	reduce (3)	reduce (3)	reduce (3)	reduce (3)		
s7	reduce (4)	reduce (4)	reduce (4)	reduce (4)		
s8	reduce (5)	shift s5 reduce (5)	reduce (5)	reduce (5)		

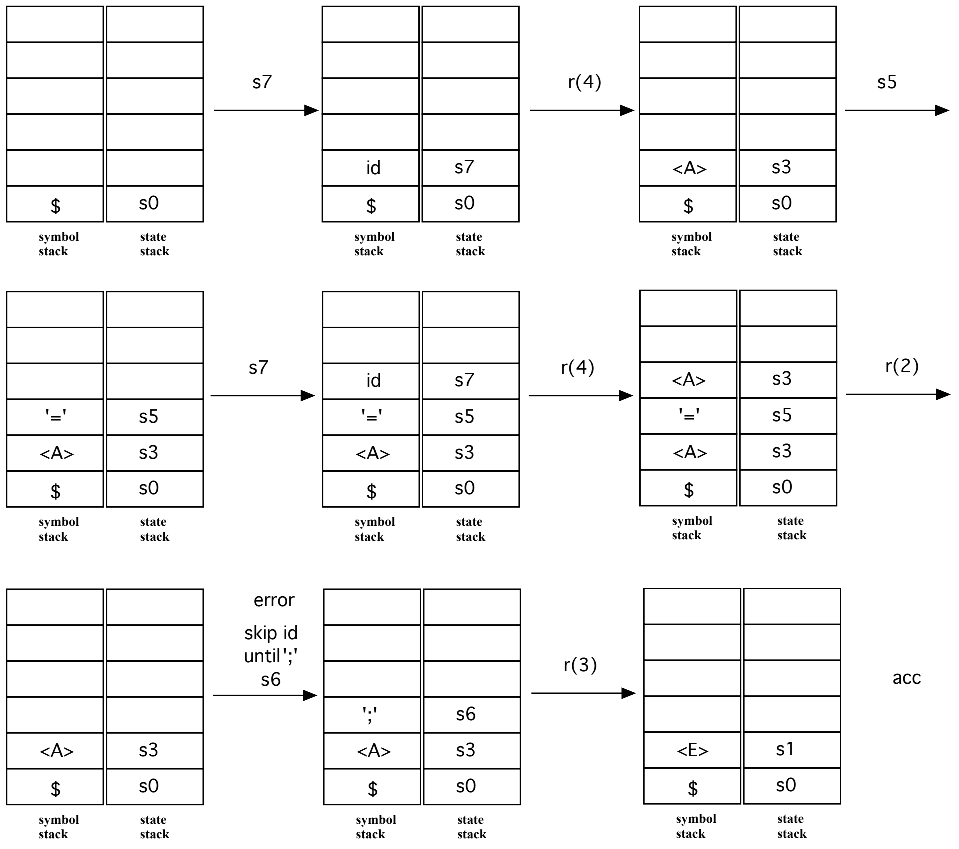
The use of the Follow(A) = { '=', ';', $ } and Follow(E) = { ';' , $} will help eliminate the conflict inn state S4 but not in state S8. As such this grammar is not SLR parseable.


s0
<S> → • <E> $, { $ }
<E> → • <E> ';' <A> , { $ }
<E> → • <A> ';' , { ';' }
<A> → • id , { ';', '=' }
<A> → • <A> '=' <A>, { ';', '=' }
<E>
s1
<S> → <E> • $, { $ }
<E> → <E> • ';' <A> , { $ }
';'
s2
<E> → <E> ';' • <A> , { $ }
<A> → • id , { $, '=' }
<A> → • <A> '=' <A> , { $, '=' }
<A>
id
id
s3
<E> → <A> • ';' , { $ }
<A> → <A> • '=' <A> , {';', '=' }
s7
<A> → id • , { ';', '=' }
s9
<A> → id • , { $, '=' }
s4
<E> → <E> ';' <A> •, { $ }
<A> → <A> • '=' <A> , { $, '=' }
';'
'='
id
id
'='
s6
<E> → <A> ';' • , { $ }
s10
<A> → <A> '=' • <A>, { ';' , '=' }
<A> → • id, { ';', '=' }
<A> → • <A> '=' <A>, { ';' , '=' }
s5
<A> → <A> '=' • <A>, { $, '=' }
<A> → • id, { $, '=' }
<A> → • <A> '=' <A>, { $, '=' }
<A>
'='
<A>
'='
s11
<A> → <A> '=' <A> •, { ';' , '=' }
<A> → <A> '=' • <A>, { ';' , '=' }
s8
<A> → <A> '=' <A> •, { $, '=' }
<A> → <A> • '=' <A>, { $, '=' }

The diagram depicts the DFA that recognizes the set of LR(1) items for this grammar. While the shift/reduce conflict is eliminated for state s4 (as the reduction only occurs on the $ terminal) for the state s8 there is still a shift/reduce conflict on the '=' terminal symbol. The grammar is thus not even LR(1) parseable.

d) There are really here no issues with the operators so precedence of operators does not really apply. Still we can have right associativity of the multiple assignments in a single statement thus giving priority to the shift operation. Thus if we give higher priority to the shift that will mean that we are favoring the continued parsing of more identifiers into a single statement of the form of <A> '=' <A> '=' <A>.

e) A possible scenario is as shown below. When the parser has seen <A> '=' <A> it will reduce using rule 2. At this point its DFA is left in state S3. For the 'id' input there is not entry defined in the table. As the only non-terminal in the table is <A> we can opt to look at the input for a terminal symbol in the Follow(A), say for instance ';' as in this state 3 there is a production that has as its RHS the sentential form <A> ';'. As such we skip the identifier and shift the ';' onto the stack and resume parsing. The sequence of stack states illustrates the sequence of action by this parser.



This recovery scenario is distinct from the one studied in class. Here we skip the input until we find some terminal symbol in the input for with the current state has a "shift" transition, rather looking into the stack and finding a state that has a goto transition on the current terminal symbol (as in HW2).

Problem 3: Syntax-Directed Translation [30 points]

In this problem you are asked to develop a SDD scheme for a simplified register allocation for the computation of expressions in assignment statements for a block of statements. You need to use the grammar provided and develop the set of attributes to be associated with each of the non-terminal and terminal symbols of the provided grammar. We can assume the input is a syntactically correct program building a well-formed abstract syntax tree (AST).

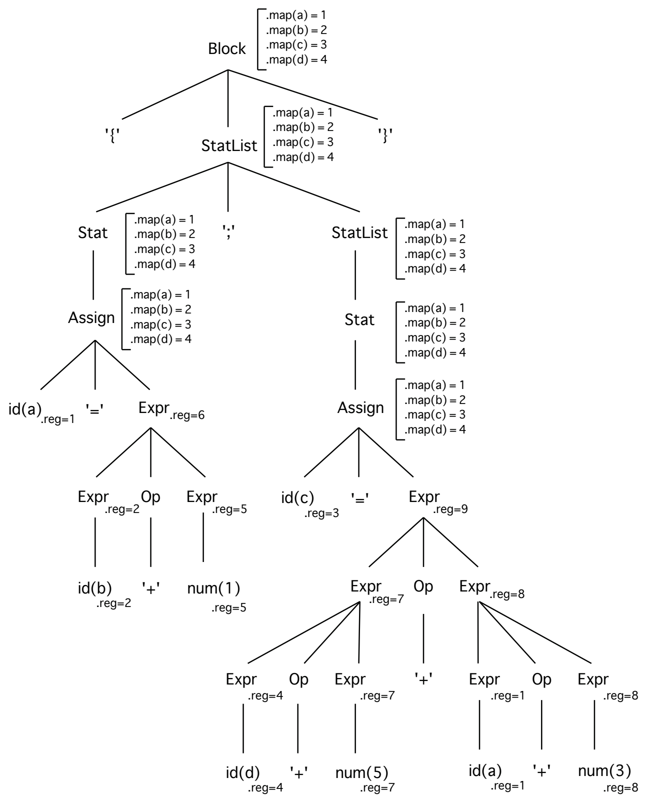
The output of the SDD scheme is an indication for each expression and sub-expression, which of the N registers will hold the results of the corresponding computation at run-time, say `reg = k`. Note that you are not being asked to generate code - that would be a secondary code generation phase. The assignment of registers to each of the variables is done via an attribute associated with the terminal **id** symbol, also named `reg` but whose lifetime spans the entire block of statements and are thus constant throughout the execution of the block of statements. You need to keep track of this mapping throughout your SDD, i.e., if there are `k` variables the temporary registers will begin at register number `k+1`. In this intermediate allocation of registers you are not supposed to minimize the number of uses temporaries, but in the end you need to determine the number of temporary registers your code block is using. The example below illustrates an example of the results of the SDD you need to develop for a sample input code alongside the corresponding AST.

For the specific grammar G below (left) answer the following:

a) [20 points] Define the set of attributes (synthesized and inherited) to define for each expression and sub-expression which of the N available register should be used to compute the sub-expression value. Assume that for identifiers the AST has been previously decorated with an attributed named register indicating the specific register holding the corresponding variable's value.
b) [05 points] Show your work for the block of statements whose AST is shown below (right).
c) [05 points] Can your solution be computed in a single pass of the AST? Why or why not?

CFG G = (Block, { Block, StatList, Stat, Assign, Expr, Op }, {'{', '}', ';' , '=', '+', '-' , id, num}, P) where the set of productions P is as shown below:

(1) Block → '{' StatList '}' $
(2) StatList → Stat ';' StatList
(3) StatList → Stat
(4) StatList → ε
(5) Stat → Assign
(6) Assign → **id** '=' Expr
(7) Expr → Expr Op Expr
(8) Expr → '(' Expr ')'
(9) Expr → **id**
(10) Expr → **num**
(11) Op → '+'
(12) Op → '-'


Block .map(a) = 1 .map(b) = 2 .map(c) = 3 .map(d) = 4
'{'
StatList .map(a) = 1 .map(b) = 2 .map(c) = 3 .map(d) = 4
'}'
Stat .map(a) = 1 .map(b) = 2 .map(c) = 3 .map(d) = 4
';'
StatList .map(a) = 1 .map(b) = 2 .map(c) = 3 .map(d) = 4
Assign .map(a) = 1 .map(b) = 2 .map(c) = 3 .map(d) = 4
Stat .map(a) = 1 .map(b) = 2 .map(c) = 3 .map(d) = 4
Assign .map(a) = 1 .map(b) = 2 .map(c) = 3 .map(d) = 4
id(a) .reg=1
'='
Expr .reg=6
Expr .reg=2
Op
Expr .reg=5
id(b) .reg=2
'+'
num(1) .reg=5
id(c) .reg=3
'='
Expr .reg=9
Expr .reg=7
Op
Expr .reg=8
Expr .reg=4
Op
Expr .reg=7
'+'
Expr .reg=1
Op
Expr .reg=8
id(d) .reg=4
'+'
num(5) .reg=7
id(a) .reg=1
'+'
num(3) .reg=8

Solution:

a) There are many possible solutions to this SDT scheme. A simple solution involves the use of inherited and synthesized attributes. An inherited attribute consists in the mapping of the identifiers to the 'fixed' registers and has to be propagated downwards the tree so that when you make the assignment of register to expression you can determine what if any additional register is required. Also, inherited is the current starting assignment value of the registers which needs to be propagated downwards and then to the sibling nodes in a binary expression or in a statement list. Other attributes are synthesized and correspond to the register attribute use to propagate upwards when you are allocating registers to expressions.

As there are no requirements to reuse and thus minimize the number of registers the sematic rule just checks the indices of the registers uses and assign an additional register to hold the result of the current expression node. As such we are going to have two register tracking attributes, one inherited named 'reg_in' and another synthesized named 'reg_out' to be used throughout all the non-terminal symbols of the grammar. For the Expr non-terminals and both the **id** and **num** terminals we have an attributed, named 'reg' that indicates the register identifier that holds the corresponding value. This 'reg' attribute is not defined for the other non-terminal symbols, such as Block, Assing, Stat, StatList or Op. In addition we have a simple inherited 'map' attribute that is strictly 'read-only' to be propagated from the 'Block' non-terminal symbol downwards to the 'Expr' non-terminal. Associated with this 'map' attribute we also have a 'MapId' function that given the strings of an identifier returns the corresponding register mapping (assumed to always be valid). In addition we make use of a 'NumbersId' function that given a mapping computes the total number of identifiers currently mapped to the 'fix' set of registers. The solution below makes explicit all the attributes with the invariably long list of 'copy' rules.

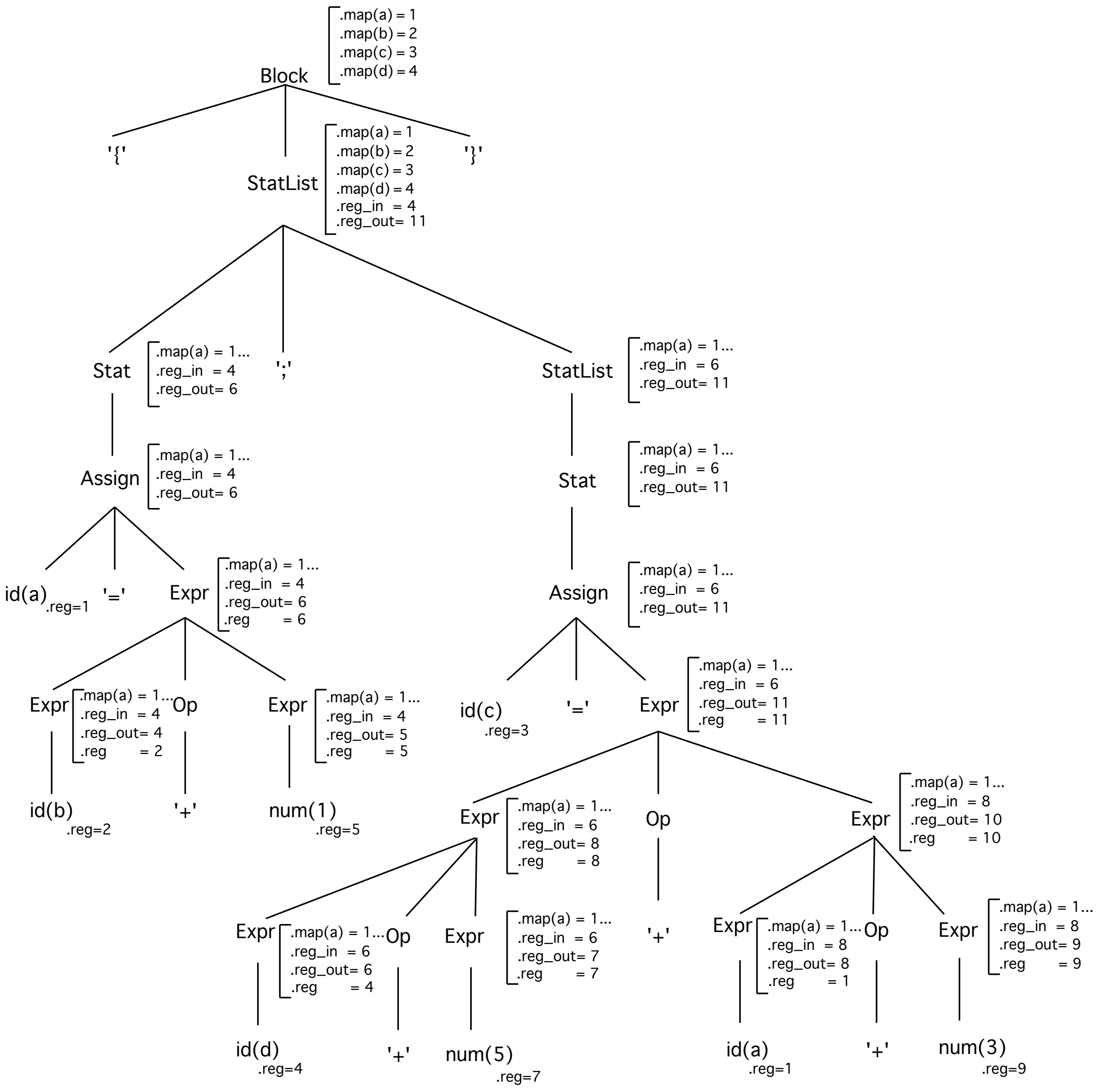
Block → '{' StatList '}' $	StatList.reg_in = NumberIds(Block.map);
	StatList.map = Block.map
StatList$_0$ → Stat ';' StatList$_1$	Stat.reg_in = StatList$_0$.reg_in;
	StatList$_1$.reg_in = Stat.reg_out;
	StatList$_0$.reg_out = StatList$_1$.reg_out
	Stat.map = StatList$_0$.map;
	StatList$_1$.map = StatList$_0$.map;
StatList → Stat	Stat.reg_in = StatList.reg_in;
	StatList.reg_out = Stat.reg_out;
	Stat.map = StatList.map;
StatList → ε	StatList.reg_out = StatList.reg_in;
Stat → Assign	Assign.reg_in = Stat.reg_in;
	Stat.reg_out = Assign.reg_out
	Assign.map = Stat.map;
Assign → **id** '=' Expr	Assign.reg_out = Expr.reg_out;
	Expr.reg_in = Assign_reg_in;
	Expr.map = Assign.map;
Expr$_0$ → Expr$_1$ Op Expr$_2$	Expr$_1$.reg_in = Expr$_0$.reg_in;
	Expr$_2$.reg_in = Expr$_1$.reg_out;
	Expr$_0$.reg_out = Expr$_2$.reg_out + 1;
	Expr$_1$.map = Expr$_0$.map;
	Expr$_2$.map = Expr$_0$.map;

$Expr_0$ → '(' $Expr_1$ ')'	$Expr_1$.reg_in = $Expr_0$.reg_in; $Expr_0$.reg_out = $Expr_1$.reg_out; $Expr_1$.map = $Expr_0$.map;

Expr → **id**	Expr.reg_out = Expr.reg_in;
	id.reg = MapId(Expr.map(id.text));

Expr → **num**	Expr.reg_out = Expr.reg_in + 1;
	num.reg = Expr.reg_in + 1;

b) The figure below depicts the results of the use of the semantics rule for the solution described in a).



c) Yes, the propagation of the inherited attributes up and down the AST can be done in a single depth-first-search traversal pattern. For the synthesized attributes this is trivially done. For the inherited attributes and given that the values flow from 'left-to-right' in the tree (making this a L-attributed grammar) the assignment can be done in a single pass.