CSCI 565 - Compiler Design

Spring 2012

Midterm Exam

Feb. 29, 2012 at 3.30 PM in class (RTH 115)

Duration: 2h 30 min.

Please label all pages you turn in with your name and student number.

Name: _____ **Number:** _____

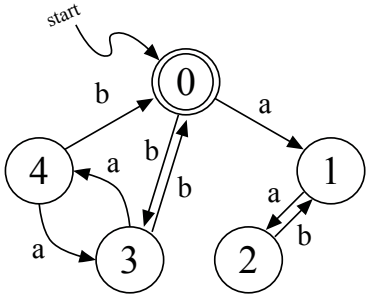
Grade:

Problem 1 [20 points]:
Problem 2 [30 points]:
Problem 3 [50 points]:

Total:

Problem 1: Regular Expressions and DFAs [20 points]

Consider the Finite Automaton (FA) below defined over the alphabet $\Sigma = \{a,b\}$.

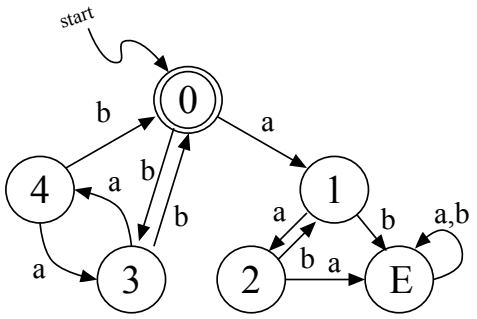


For this FA answer the following questions:

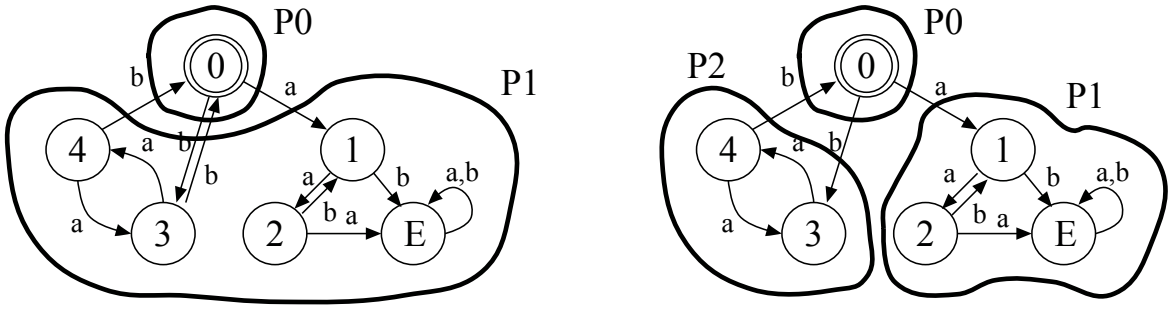
a) [15 points] Minimize the FA using the iterative refinement algorithm described in class. Make sure that you convert this FA to a DFA first.
b) [05 points] Describe in plain English the language accepted by this FA and represent it using a regular expression.

Solution:

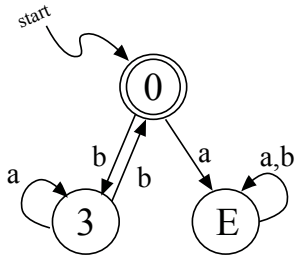
a) The only transitions missing are the transitions on 'b' from state 1 and transitions on 'a' from state 2 which will be omission lead to a "trap" or "error" state as shown below.



Using this DFA as the starting point for a minimization we get the following two DFA refinements below. The first refinement (on the left) corresponds to discriminate between accepting and non-accepting states whereas the second refinement (on the right) results from discrimination on the 'a' input character (a similar reasoning would hold for the 'b' character).



No more refinements are possible yielding the minimized DFA shown below:



b) This DFA as well as the original FA accept the language of string over the alphabet Σ ={a,b} that consist of zero or more occurrences of the pattern 'b' followed by any occurrence of the letter 'a' ending with the 'b' character.

In terms of a regular expression one can represent it as L(M) = (b.a*.b)*, thus yielding string such as "baaab", "babbb" or even the empty string.

Problem 2: Context-Free-Grammars and Parsing Algorithms [30 points]

Consider the CFG G = {NT = {E,T,F}, T = {a,b,+,*}, P, E} with the set of productions P as follows:

(1) E → E + T
(2) E → T
(3) T → T F
(4) T → F
(5) F → F *
(6) F → a
(7) F → b

For this CFG answer the following questions:

(a) [10 points] Compute the FIRST and FOLLOW for all nonterminal symbols in G.
(b) [10 points] Consider the augmented grammar G' = {NT, T, { (0) E' → E$ } + P, E' }. Compute the set of LR(0) items for G'.
(c) [05 points] Compute the LR(0) parsing table for G'. If there are shift-reduce conflicts use the SLR parse table construction algorithm and discuss if the conflicts are eliminated.
(d) [05 points] Show the movements of the parser for the input w = "a+ab*$".

Solution:

(a) We compute the FIRST and FOLLOW for the augmented grammar (0) E' → E$

FIRST(E) = {a,b}
FIRST(T) = {a,b}
FIRST(F) = {a,b}
FOLLOW(E) = {+,$}
FOLLOW(T) = FIRST(F) + FOLLOW(E) = {a,b,+,$}
FOLLOW(F) = {*,a,b,+,$}

(b) Consider the augmented grammars E' → E$ we compute the LR(0) set of items.

S0 = closure({[E' → •E$]})
= E' → •E$
E → •E + T
E → •T
T → •T F
T → •F
F → •F *
F → •a
F → •b

S1 = goto (I0,E)
= closure({[E' → E•$],[E → E• + T]})
= E' → E• $
E → E• + T

S2 = goto (I0,T)
= closure({[E → T•],[T → T• F]})
= E → E•
T → T•F
F → •F *
F → •a
F → •b

S3 = goto (S0, F)
= closure({[T → F•],[F →F• *]})
= T → F•
F →F• *

S4 = goto (S2,a)
= closure({[F → a•]})
= F → a•

S5 = goto (S2,b)
= closure({[F → b•]})
= F → b•

S6 = goto (S1, +)
= closure({[E → E+•T]})
= E → E+•T
T → •T F
T → •F
T → •F *
F → •a
F → •b

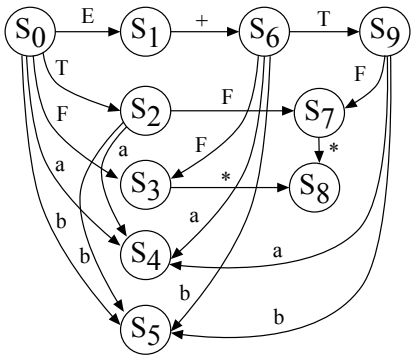
S3 = goto(S0 ,F)
= closure({[T → F•], [F →F•*]})

S7 = goto (S2,F)
= closure({[T → TF•],[F → F•*]})
= T → T F•
F → F•*

S8 = goto (S3,*)
= closure({[F → F*•]})
= F → F*•

S9 = goto (S6,T)
= closure({[E → E+T•], [E → T•F]})
= E → E + T•
E → T•F
F → •F *
F → •a
F → •b

goto (S9,a) = S4
goto (S9,b) = S5
goto (S9,F) = S7



State	Action					Goto		
	a	b	+	*	$	E	T	F
S0	shift S4	shift S5				goto S1	goto S2	goto S3
S1			shift S6		accept			
S2	shift S4	shift S5	reduce 2		reduce 2			goto S3
S3	reduce 4	reduce 4	reduce 4	shift S8	reduce 4			
S4	reduce 6	reduce 6	reduce 6	reduce 6	reduce 6			
S5	reduce 7	reduce 7	reduce 7	reduce 7	reduce 7			
S6	shift S4	shift S5					goto S9	goto S3
S7	reduce 3	reduce 3	reduce 3	shift S8	reduce 3			
S8	reduce 5	reduce 5	reduce 5	reduce 5	reduce 5			
S9	shift S4	shift S5	reduce 2		reduce 2			goto S7

(c) We cannot construct an LR(0) parsing table because states S1, S2, S3, S7 and S9 have "shift-reduce" conflicts. We use the FOLLOW sets to eliminate the conflicts and build the SLR parsing table above.

(d) For the input = a+ab*$ the parser actions would be:

$0	shift S4	
$0a4	reduce 6	F → a
$0F3	reduce 4	T → F
$0T2	reduce 2	E → T
$0E1	shift S6	
$0E1+6	shift S4	
$0E1+6a4	reduce 6	F → a
$0E1+6F3	reduce 4	T → F
$0E1+6T9	shift S5	
$0E1+6T9b5	reduce 7	F → b
$0E1+6T9F7	shift S8	
$0E1+6T9F7*8	reduce 5	F → F*
$0E1+6T9F7	reduce 3	T → TF
$0E1+6T9	reduce 4	E → E+T
$0E1	***accept***	

Problem 3: Syntax-Directed Translation and Intermediate Representation [50 points]

In this problem you are asked to develop a syntax-directed translation (SDT) scheme to support an analysis over the statement of the input program. In particular it is important to understand which variables are first read and then written in a sequence of statements in what it is called "*upwards exposed*". In the example "x = y + 1; y = x;" the variable y is upwards-exposed but the variable x is not, as an read operation for y occurs before the write operation for y, whereas for the variable x a write occurs before the read. In order to uncover which variables are upwards exposed you need to keep track of the assignment statements in the grammar and the accesses on both the right- and left-hand side of these assignment statements.

For the purposes of this exercise consider the fragment of the CFG below where **id** and **const** denote terminal symbols associated with variables and integer constants in the input program as recognized by a scanner.

```
Block    → StatList
StatList → Stat ';' StatList
         → ε
Stat     → Assign
         →  ...
Assign   → id '=' Expr
Expr     → Expr  '+' Term
Expr     → Expr  '-' Term
Expr     → Term
Term     → Term '*' Factor
Term     → Term  '/' Factor
Term     → Factor
Factor   → '(' Expr ')'
Factor   →  id
Factor   →  const
```

In developing your SDT scheme answer the following questions:

a) [20 points] For the CFG fragment above define the attributes for the non-terminal symbols and terminal symbols for the grammar and develop semantic rules that can be used to compute as the value of an attribute at the Block non-terminal symbols (unique for each sequence of code), the set of upwards exposed variables, i.e. variables that are first read and then possibly written. Clearly, the values at the Block symbol will have to be derived from the values at the intermediate non-terminal symbols such as StatList and Assign, so you will need to make sure the values of the attributes are propagated properly up and down the tree. In your description make sure you identify the attributes as inherited and synthesized.

b) [10 points] Using the attribute grammar developed in a) show the annotated parse tree for the sequence of statements "x = y + 1; y = z;" illustrating the dependences between the evaluation of the attributes.

c) [10 points] Discuss how you would combine the information of multiple statements in a conditional construct, i.e., if you had a statement of the form "if-then-else" and had computed the attribute values for the sequence of statements in both branches of the " if " construct, how would you derive the combined values of the attributes for the entire " if-then-else" construct?

d) [10 points] For the specific code example in b) derive a 3-address representation. Describe a simple algorithm based on the 3-address representation to derive the same upwards-exposed information as your SDT is designed to do. Discuss possible advantages of this 3-address representation versus the abstract-syntax tree representation for this particular problem of upwards-exposed variables.

Solution:

a) This problem requires the use of both inherited and synthesized attributes so that we can capture the flow of information along the various statements in the parse tree and thus emulate the evaluation order of the corresponding instructions or operations. The solution describe here is not necessarily the most compact in terms of the minimization of the number of attributes associated with each symbol. Instead it is the most straightforward to understand.

In particular, we need to track, which variables are read and which is written to observe if a given variable is read before it is possibly written. As such we define for the Block, StatList, Stat and Assign 6 attributes, 3 inherited and 3 synthesized. Two of the inherited attributes capture the variables that are read and written respectively before the statement. Similarly, we define two synthesized attributes to indicate which variables are read and written before the statement. As such we name these attributes $read_{in}$, $write_{in}$ and $read_{out}$ and $write_{out}$, respectively. In addition for the Block, StatList, Stat and Assign we define another set of two attributes, one inherited and another synthesized, that hold the variables that are upwards exposed, named up_{in} and up_{out}. Lastly, for the id and const terminal symbols we define a synthesized access attribute which in the case of the const terminal always denotes the empty set.

The table below summarizes these attributes along with their type.

Symbol	Attribute	Type	Values
Block, StatList, Stat, Assign	$read_{in}$, $read_{out}$, $write_{in}$, $write_{out}$, up_{in}, up_{out}	inherited, synthesized, inherited, synthesized, inherited synthesized	set of variables
Expr, Term, Factor	$read_{in}$, $read_{out}$, $write_{in}$, $write_{out}$,	inherited, synthesized, inherited, synthesized	set of variables
id	access	synthesized	set of variables
const	access	synthesized	set of variables

As to the sematic rules there are ample copy operations just to convey the information up and down the parse tree and alongside the statements in the list of statements as depicted below for each production.

Block → StatList ||
$StatList.read_{in} = Block.read_{in}$;
$StatList.write_{in} = Block.write_{in}$;
$StatList.up_{in} = Block.up_{in}$;
$Block.read_{out} = StatList.read_{out}$;
$Block.write_{out} = StatList.write_{out}$;
$Block.up_{out} = StatList.up_{out}$;

$StatList_1$ → Stat ';' $StatList_2$ ||
$Stat.read_{in} = StatList_1.read_{in}$;
$Stat.write_{in} = StatList_1.write_{in}$;
$Stat.up_{in} = StatList.up_{in}$;
$StatList_2.read_{in} = Stat.read_{out}$;
$StatList_2.write_{in} = Stat.write_{out}$;
$StatList_2.up_{in} = Stat.up_{out}$;
$StatList_1.read_{out} = StatList_2.read_{out}$;
$StatList_2.write_{out} = StatList_2.write_{out}$;
$StatList_1.up_{out} = Stat\ List_2.up_{out}$;

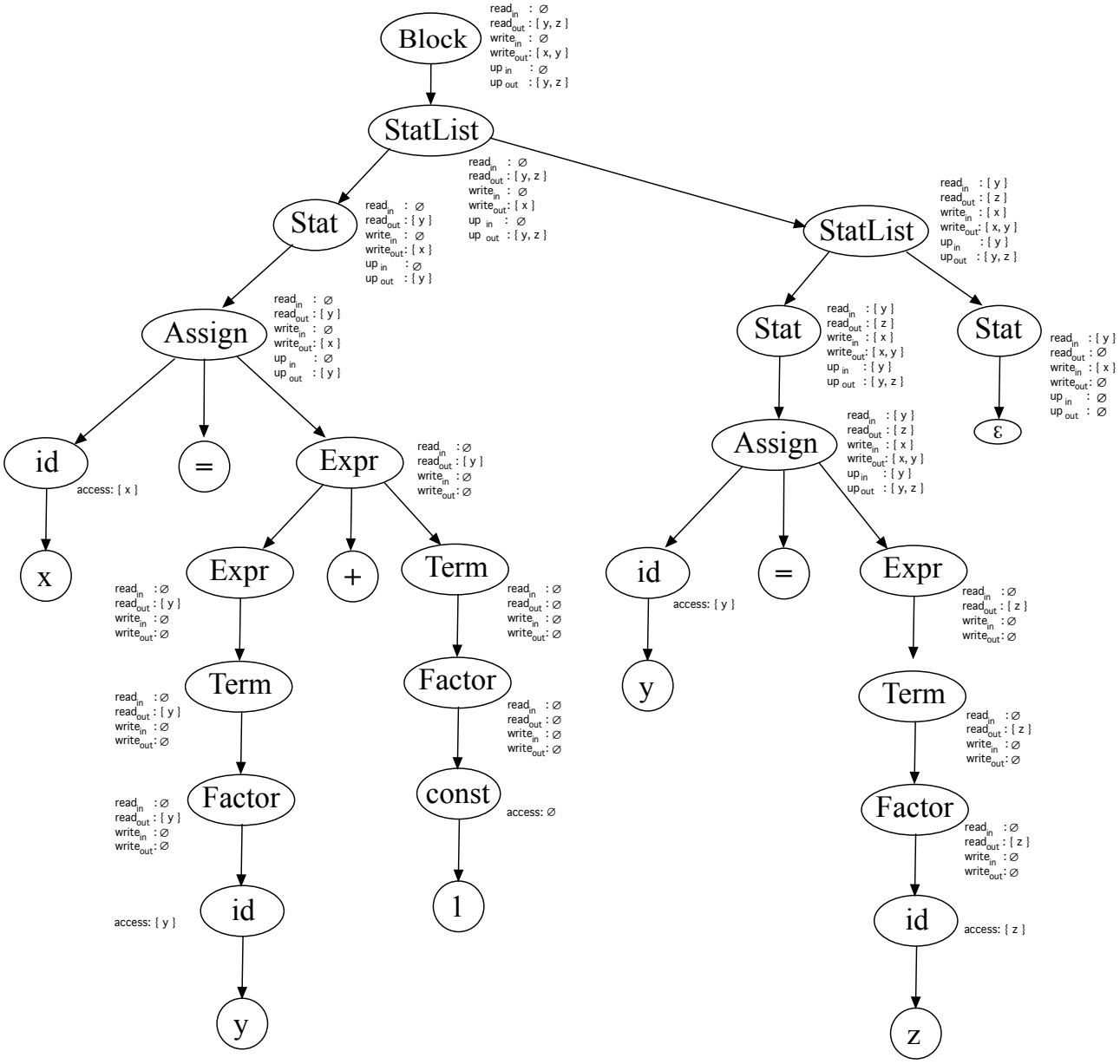
→ ε ||
$Stat.read_{out} = Stat.read_{in}$;
$Stat.write_{out} = Stat.write_{in}$;
$Stat.up_{out} = Stat.up_{in}$;

Stat → Assign	‖	$Assign.read_{in} = Stat.read_{in}$; $Stat.read_{out} = Assign.read_{out}$; $Assign.write_{in} = Stat.write_{in}$; $Stat.write_{out} = Stat.write_{out}$; $Assign.up_{in} = Stat.up_{in}$; $Stat.up_{out} = Assign.up_{out}$;

Assign → id '=' Expr	‖	$Expr.read_{in} = Assign.read_{in}$;
		$Expr.write_{in} = Assign.write_{in}$;
		$Assign.up_{out} = Assign.up_{in}$;
		for all $v \in Expr.read_{out}$ do
		if ($v \notin Assign.up_{in}$) then
		$Assign.up_{out} = Assign.up_{out} \cup \{ v \}$
		end if
		end for
		$Assign.read_{out} = Assign.read_{in} \cup Expr.read_{out}$
		$Assign.write_{out} = Assign.write_{in} \cup \{$ id.access $\}$

$Expr_1$ → $Expr_2$ '+' Term	‖	$Expr_2.read_{in} = Expr_1.read_{in}$; $Term.read_{in} = Expr_1.read_{in}$; $Expr_2.write_{in} = Expr_1.write_{in}$; $Term.write_{in} = Expr_1.write_{in}$; $Expr_1.read_{out} = Expr_2.read_{out} \cup Term.read_{out}$; $Expr_1.write_{out} = Expr_2.write_{out} \cup Term.write_{out}$;
$Expr_1$ → $Expr_2$ '-' Term	‖	<same as for the production $Expr_1$ → $Expr_2$ '+' Term >
Expr → Term	‖	$Term.read_{in} = Expr.read_{in}$; $Expr.read_{out} = Term.read_{out}$; $Term.write_{in} = Expr.write_{in}$; $Expr.write_{out} = Term.write_{out}$;
$Term_1$ → $Term_2$ '*' Factor	‖	$Term_2.read_{in} = Term_1.read_{in}$; $Factor.read_{in} = Term_1.read_{in}$; $Term_2.write_{in} = Term_1.write_{in}$; $Factor.write_{in} = Term_1.write_{in}$; $Term_1.read_{out} = Term_2.read_{out} \cup Factor.read_{out}$; $Term_1.write_{out} = Term_2.write_{out} \cup Factor.write_{out}$;
$Term_1$ → $Term_2$ '/' Factor	‖	<same as for the production $Term_1$ → $Term_2$ '*' Factor >
Term → Factor	‖	$Factor.read_{in} = Term.read_{in}$; $Term.read_{out} = Factor.read_{out}$; $Factor.write_{in} = Term.write_{in}$; $Term.write_{out} = Factor.write_{out}$;
Factor → '(' Expr ')'	‖	$Expr.read_{in} = Factor.read_{in}$; $Factor.read_{out} = Expr.read_{out}$; $Expr.write_{in} = Factor.write_{in}$; $Factor.write_{out} = Expr.write_{out}$;
Factor → **id**	‖	$Factor.read_{out} = Factor.read_{in} \cup \{$ id.name $\}$; $Factor.write_{out} = Factor.write_{in}$;
Factor → **const**	‖	$Factor.read_{out} = Factor.read_{in}$; $Factor.write_{out} = Factor.write_{in}$;

b) The figure below depicts the attributes for the parse tree resulting from parsing the input code example with the grammar and rules described above.



c) In the presence of an if-then-else statement, the semantic rule would copy the inherited attributes of the statement to the statement corresponding to the evaluation of the predicate followed by copying the resulting output values to the statement list symbols or block symbols for the two branches of the if-then-else construct. To compute the synthesized attributes, the upwards-exposed attribute up, would be the union of the up attributes of the symbols corresponding to the two branches of the if-then-else construct. Similarly, for the attributes $read_{out}$ and $write_{out}$.

d) A 3-address representation corresponding to the two instructions provided could be as shown below where we make use of a single temporary variable t1. Note that this variable t1 need not be included in the analysis of upward-exposed variables.

```
t1 = y + 1
x = t1
y = z
```

A simple algorithm would scan the instructions from top to bottom (in effect emulating its execution) and determine if a given variable is read before it is written. Associated with each instruction we could define 4 attributes as before and simply track each of the variables that are read that have not yet been written. The algorithm would be as follows:

```
for all instruction a = b op c do
  ReadSet ← { b, c }
  if (b ∉ WriteSet) then
     UpExp ← { b }
  end if
  if (c ∉ WriteSet) then
     UpExp ← { c }
  end if
   WriteSet ← { a }
end for
```

At the of the last instruction the UpExp set would contains the set of variables that are upwards exposed in the corresponding sequence of instructions.