Name:

Midterm 1

CMSC 430
Introduction to Compilers
Fall 2014

October 13, 2014

Instructions

This exam contains 9 pages, including this one. Make sure you have all the pages. Write your name on the top of this page before starting the exam.

Write your answers on the exam sheets. If you finish at least 15 minutes early, bring your exam to the front when you are finished; otherwise, wait until the end of the exam to turn it in. Please be as quiet as possible.

If you have a question, raise your hand. If you feel an exam question assumes something that is not written, write it down on your exam sheet. Barring some unforeseen error on the exam, however, you shouldn't need to do this at all, so be careful when making assumptions.

Question	Score	Max
1		25
2		40
3		35
Total		100

Question 1. Short Answer (25 points).

a. (5 points) Briefly explain the difference between an *interpreter* and a *compiler*.

Answer: An interpreter is a program that executes another program. A compiler translates a program from a source language to a target language, and the target language program can then be executed either by an interpreter or by an actual CPU.

b. (5 points) In operational semantics, what does it mean for a term to be in *normal form*? Explain in at most 1-2 sentences.

Answer: A term is in *normal form* if it cannot be reduced any further.

c. (5 points) In at most a few sentences, explain how an LALR(1) parser DFA is derived from an LR(1) parser DFA. What is the advantage of an LALR(1) parser compared to an LR(1) parser?

> **Answer:** The LALR(1) parser DFA is the same as the LR(1) parser DFA, except states with the same *core* are merged. The core of a state is the set of LR(0) items derived by ignoring the lookaheads. An LALR(1) parser may therefore have fewer states, and hence require less space, than an LR(1) parser.

d. (5 points) Why do compilers sometimes need to introduce *temporary variables* when translating to an intermediate representation?

> **Answer:** If the IR is three-address code, then temporary variables are needed to hold intermediate computation results when executing complex expressions.

e. (5 points) Consider the type of lambda calculus terms from project 1 and one of the associated functions you wrote:

```
type expr =
    | Var of string
    | Lam of string * expr
    | App of expr * expr

val subst : string → expr → expr → expr
```

Also consider the big-step, call-by-value operational semantics for lambda calculus:

$$\begin{array}{rcl} v & ::= & \lambda x.e \\ e & ::= & x \mid v \mid e\ e \end{array}$$

$$\frac{}{(\lambda x.e) \rightarrow (\lambda x.e)} \qquad \frac{e_1 \rightarrow \lambda x.e \qquad e_2 \rightarrow v \qquad e[x \mapsto v] \rightarrow v'}{e_1\ e_2 \rightarrow v'}$$

Write a function `big_step : expr → expr` that implements the big-step semantics above. You may assume `subst` is correctly implemented as specified in the project:

```
(* subst x e1 e2 returns a copy of e2 in which free occurrences of x have been replaced by e1. Be sure to implement
   capture−avoiding substitution , i.e., bound variables should be renamed as needed to avoid capture free variables
   under a lambda. For example, subst "x" (Var "y") (Lam("y", Var "x")) should return something equivalent to
   Lam("z", Var "y") and not Lam("y", Var "y"). *)
```

Answer:

```
let rec big_step = function
| Lam _ as e → e
| App (e1, e2) →
  let Lam (x,e) = big_step e1 in
  let v = big_step e2 in
    big_step (subst x v e)
```

Question 2. Parsing (40 points).

a. (10 points) Below is a .output file from ocamlyacc. Fill in the table at the bottom of the page so it shows the sequence of steps taken by the parser described by this output. Note that the parse starts in state 1, and that the parse ends upon reduction r1. We've put the $end terminal on the end of the input for you. You may not use all rows, and you can add extra rows to the table below if you need.

```
   0  $accept : %entry% $end
   1  s : a b
   2  a : A c
   3    | A
   4  b : B
   5  c : C a
   6    | C
   7  %entry% : '\001' s


state 0
	$accept : . %entry% $end (0)

	'\001'  shift 1
	.  error
	%entry% goto 2

state 1
	%entry% : '\001' . s  (7)

	A  shift 3
	.  error
	s  goto 4
	a  goto 5
```

```
state 2
	$accept : %entry% . $end (0)

	$end  accept


state 3
	a : A . c  (2)
	a : A .  (3)

	C  shift 6
	B  reduce 3
	c  goto 7


state 4
	%entry% : '\001' s .  (7)

	.  reduce 7


state 5
	s : a . b  (1)

	B  shift 8
	.  error
	b  goto 9
```

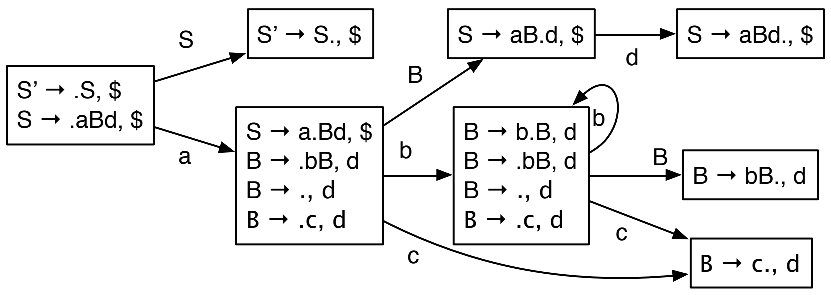
```
state 6
	c : C . a  (5)
	c : C .  (6)

	A  shift 3
	B  reduce 6
	a  goto 10


state 7
	a : A c .  (2)
	.  reduce 2


state 8
	b : B .  (4)
	.  reduce 4


state 9
	s : a b .  (1)
	.  reduce 1


state 10
	c : C a .  (5)
	.  reduce 5
```

Stack	Input	Action
1	$ACAB\$$	$s3$
$1, A, 3$	$CAB\$end$	$s6$
$1, A, 3, C, 6$	$AB\$end$	$s3$
$1, A, 3, C, 6, A, 3$	$B\$end$	$r3$
$1, A, 3, C, 6, a, 10$	$B\$end$	$r5$
$1, A, 3, c, 7$	$B\$end$	$r2$
$1, a, 5$	$B\$end$	$s8$
$1, a, 5, B, 8$	$\$end$	$r4$
$1, a, 5, b, 9$	$\$end$	$r1$
	$\$end$	

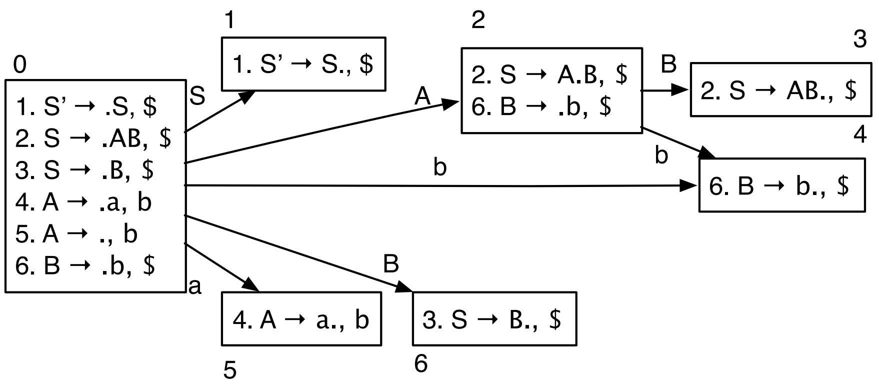
b. (20 points) Draw the LR(1) parsing DFA for the following grammar:

$$\begin{array}{lcl} S & \rightarrow & aBd \\ B & \rightarrow & bB \\ B & \rightarrow & \varepsilon \\ B & \rightarrow & c \end{array}$$

Answer:



c. (10 points) Consider the following LR(1) parsing DFA:



Fill in the action/goto table below so it matches the above DFA. One of the states has a conflict. What is the state, and what is the conflict?

state	action			goto		
	a	b	$	A	B	S
0	s5	r5, s4		2	6	1
1			accept			
2		s4			3	
3			r2			
4			r6			
5		r4				
6			r3			

Describe the conflict in the parsing DFA:
There is a shift/reduce conflict in state 0 on seeing b in the lookahead.

Question 3. Operational semantics (35 points). Consider lambda calculus extended with integers and updatable references. (Here ℓ ("location") is a pointer, and a store S maps locations to values.)

$$\begin{array}{rcl} v & ::= & n \mid \lambda x.e \mid \ell \\ e & ::= & v \mid x \mid e\ e \mid \text{ref}\ e \mid !e \mid e_1 := e_2 \\ S & ::= & \emptyset \mid S[\ell \mapsto v] \end{array}$$

Here is most of an operational semantics for this language:

$$\textsc{Beta}\ \frac{}{\langle S, (\lambda x.e_1)v_2\rangle \to \langle S, e_1[x \mapsto v_2]\rangle} \qquad \textsc{AppL}\ \frac{\langle S, e_1\rangle \to \langle S', e_1'\rangle}{\langle S, e_1\ e_2\rangle \to \langle S', e_1'\ e_2\rangle} \qquad \textsc{AppR}\ \frac{\langle S, e_2\rangle \to \langle S', e_2'\rangle}{\langle S, v\ e_2\rangle \to \langle S', v\ e_2'\rangle}$$

$$\textsc{Ref}\ \frac{\ell \notin dom(S) \qquad S' = S[\ell \mapsto v]}{\langle S, \text{ref}\ v\rangle \to \langle S', \ell\rangle} \qquad \textsc{RefIn}\ \frac{\langle S, e\rangle \to \langle S', e'\rangle}{\langle S, \text{ref}\ e\rangle \to \langle S', \text{ref}\ e'\rangle}$$

$$\textsc{Assign}\ \frac{S' = S[\ell \mapsto v]}{\langle S, \ell := v\rangle \to \langle S', v\rangle} \qquad \textsc{AssignL}\ \frac{\langle S, e_1\rangle \to \langle S', e_1'\rangle}{\langle S, e_1 := v\rangle \to \langle S', e_1' := v\rangle} \qquad \textsc{AssignR}\ \frac{\langle S, e_2\rangle \to \langle S', e_2'\rangle}{\langle S, e_1 := e_2\rangle \to \langle S', e_1 := e_2'\rangle}$$

a. (5 points) In this semantics, is the left-hand side of an assignment evaluated first; is the right-hand side evaluated first; or is the choice non-deterministic? Explain your answer briefly.

Answer: The right-hand side is evaluated first, as can be seen in Rule AssignL, which requires the right hand side be fully evaluated.

b. (10 points) Let $\emptyset$ be the empty store. Show a derivation that in this operational semantics, the reduction at the bottom holds. (You can draw your derivation up, above the reduction.)

Answer:

$$\frac{\dfrac{\langle \emptyset, \text{ref}\ 42\rangle \to \langle [\ell \mapsto 42], \ell\rangle}{\langle \emptyset, (\lambda x.x)\ (\text{ref}\ 42)\rangle \to \langle [\ell \mapsto 42], (\lambda x.x)\ \ell\rangle}}{\langle \emptyset, ((\lambda x.x)\ (\text{ref}\ 42))\ (\lambda z.z)\rangle \to \langle [\ell \mapsto 42], ((\lambda x.x)\ \ell)\ (\lambda z.z)\rangle}$$

$$\langle \emptyset, ((\lambda x.x)\ (\text{ref}\ 42))\ (\lambda z.z)\rangle \quad \to \quad \langle [\ell \mapsto 42], ((\lambda x.x)\ \ell)\ (\lambda z.z)\rangle$$

c. (10 points) Evaluate the following configuration until no more reductions are possible. For this part, just show the reduction steps and not the derivations of the steps. Use ℓ_1, ℓ_2, ℓ_3, etc if you need more locations. (Note this example will use a pointer to a pointer, which is allowed.)

$$\langle \emptyset, ((\lambda x.x)(\text{ref } 42)) := ((\lambda y.\lambda z.y)\ (\text{ref } 43)\ 44)\rangle \qquad \rightarrow$$

Answer:

$$\begin{aligned}
\langle [\ell_1 \mapsto 43], ((\lambda x.x)(\text{ref } 42)) := ((\lambda y.\lambda z.y)\ \ell_1\ 44)\rangle &\quad \rightarrow \\
\langle [\ell_1 \mapsto 43], ((\lambda x.x)(\text{ref } 42)) := ((\lambda z.\ell_1)\ 44)\rangle &\quad \rightarrow \\
\langle [\ell_1 \mapsto 43], ((\lambda x.x)(\text{ref } 42)) := \ell_1\rangle &\quad \rightarrow \\
\langle [\ell_1 \mapsto 43, \ell_2 \mapsto 42], ((\lambda x.x)\ell_2) := \ell_1\rangle &\quad \rightarrow \\
\langle [\ell_1 \mapsto 43, \ell_2 \mapsto 42], \ell_2 := \ell_1\rangle &\quad \rightarrow \\
\langle [\ell_1 \mapsto 43, \ell_2 \mapsto \ell_1], \ell_1\rangle &
\end{aligned}$$

d. (10 points) Write the missing operational semantics rule(s) for dereference (written $!e$).

Answer:

$$\textsc{Deref}\ \frac{}{\langle S, \text{ref } \ell\rangle \rightarrow \langle S, S(\ell)\rangle} \qquad \textsc{DerefIn}\ \frac{\langle S, e\rangle \rightarrow \langle S', e'\rangle}{\langle S, !e\rangle \rightarrow \langle S', !e'\rangle}$$