Name:

Midterm 1

CMSC 430
Introduction to Compilers
Fall 2012

October 10, 2012

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		20
2		42
3		16
4		22
Total		100

Question 1. Short Answer (20 points).

a. (7 points) Briefly explain the difference between *big-step* and *small-step* operational semantics. What kinds of programs can we assign meaning to using a small-step semantics that are difficult to assign meaning to using a big-step semantics? (You might find it useful to refer to IMP in your discussion.)

Answer: A big-step semantics reduces a program "all at once" to a result. For example, in IMP, the big-step reduction for commands has the form $\langle c, \sigma \rangle \to \sigma'$, producing the new state σ' in one, big step. In contrast, a small-step semantics reduces one subexpression of the program, and may take many steps to reduce to the final result. In IMP, small-step reductions for commands have the form $\langle c, \sigma \rangle \to \langle c', \sigma' \rangle$. One advantage to small-step semantics is that it lets us assign meaning to non-termination programs which, since they never produce a final value, do not have meaning in a big-step setting.

b. (7 points) Explain briefly how LALR(1) parsing differs from LR(1) parsing, and give one advantage and one disadvantage of LALR(1) as compared to LR(1).

Answer: LALR(1) parsing compresses the action and goto tables of an LR(1) parser by combining DFA states whose LR(0) items are the same (i.e., taking the LR(1) items and ignoring the lookaheads). This can result in smaller parsing tables, which could shrink the size of the parser and potentially improve speed. But it also can introduce new reduce/reduce conflicts. (Note that it will never introduce shift/reduce conflicts.)

c. (6 points) Consider the data type for lambda expressions from Project 1:

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

Write a function `free_vars : expr -> string list` that returns the list of free variables in a lambda expression.

Answer: (No sample solution.)

Question 2. Parsing (42 points).

a. (10 points) Consider the following grammar and the corresponding action and goto tables.

	S'	$\rightarrow$	S
0.	S	$\rightarrow$	ABb
1.	A	$\rightarrow$	ε
2.	A	$\rightarrow$	aA
3.	B	$\rightarrow$	b
4.	B	$\rightarrow$	cB

	action				*goto*		
state	a	b	c	\$	A	B	S
0	$s1$	$r1$	$r1$		3		9
1	$s1$	$r1$	$r1$		2		
2			$r2$				
3		$s7$	$s8$			4	
4		$s5$					
5				$r0$			
6		$r4$					
7		$r3$					
8		$s7$	$s8$			6	
9				*accept*			

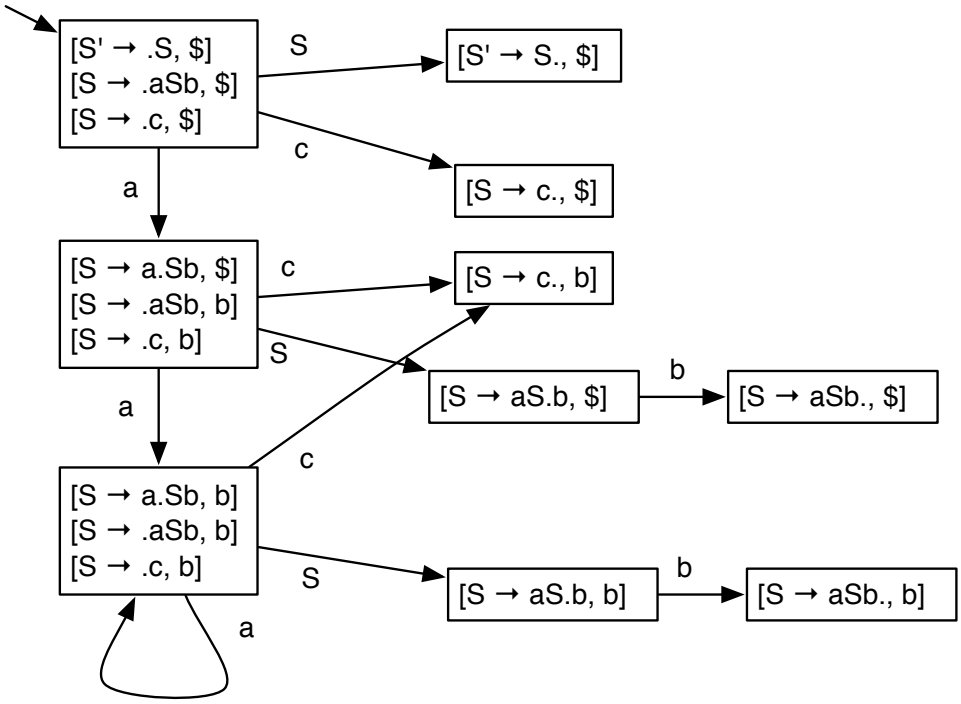
Fill in the following table with the steps taken by the parser on the given input string. Note that we have explicitly written in the end-of-file character, \$, at the end of the input, which we omitted on Homework 1. (You may not need to use all the lines below.)

Stack	Input	Action
0	*accbb*\$	$s1$
$0, a, 1$	*ccbb*\$	$r1$
$0, a, 1, A, 2$	*ccbb*\$	$r2$
$0, A, 3$	*ccbb*\$	$s8$
$0, A, 3, c, 8$	*cbb*\$	$s8$
$0, A, 3, c, 8, c, 8$	*bb*\$	$s7$
$0, A, 3, c, 8, c, 8, b, 7$	*b*\$	$r3$
$0, A, 3, c, 8, c, 8, B, 6$	*b*\$	$r4$
$0, A, 3, c, 8, B, 6$	*b*\$	$r4$
$0, A, 3, B, 4$	*b*\$	$s5$
$0, A, 3, B, 4, b, 5$	\$	$r0$
$0, S, 9$	\$	*accept*

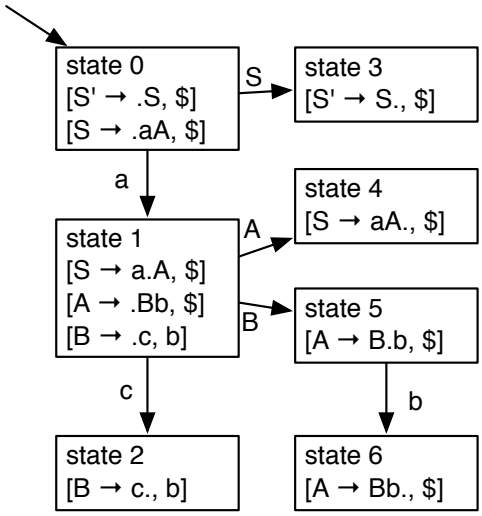
b. (14 points) Draw the LR(1) parsing DFA for the following grammar.

$$\begin{array}{lcl} S & \rightarrow & aSb \\ S & \rightarrow & c \end{array}$$

Answer: First we need to augment the grammar with an additional production $S' \rightarrow S$.



c. (12 points) Write down the grammar and the action and goto tables corresponding to the following LR(1) parsing DFA.



Answer:

	S'	$\rightarrow$	S
0.	S	$\rightarrow$	aA
1.	A	$\rightarrow$	Bb
2.	B	$\rightarrow$	c

	action				*goto*		
state	a	b	c	\$	A	B	S
0	$s1$						3
1			$s2$		4	5	
2		$r2$					
3				*accept*			
4				$r0$			
5		$s6$					
6				$r1$			

d. (6 points) Write down one production such that, if it were added to the grammar to part c just above, it would introduce a shift/reduce conflict. Justify your answer by describing which state would change and how it would exhibit the conflict.

Answer: There are many possible answers. If we add a production $S \rightarrow acb$ to the grammar, then we'll add item $[S \rightarrow .acb, \$]$ to state 0, item $[S \rightarrow a.cb, \$]$ to state 1, and $[S \rightarrow ac.b, \$]$ to state 2. But then in state 2 upon seeing a b we don't know whether to shift or reduce.

Question 3. Operational semantics (16 points). Consider extending the call-by-value lambda calculus with exceptions. To keep things simple, we will have one exceptional value, *error*, that will propagate through evaluation until caught by a *try e with e* handler (or until it reaches the top level of the expression).

$$\begin{array}{lcl} e & ::= & v \mid x \mid e\ e \mid \mathit{error} \mid \mathit{try}\ e\ \mathit{with}\ e \\ v & ::= & n \mid \lambda x.e \end{array}$$

$$\textsc{Beta}\ \frac{}{(\lambda x.e_1)\ v_2 \to e1[x \mapsto v_2]} \qquad \textsc{Left}\ \frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \textsc{Right}\ \frac{e_2 \to e_2'}{v\ e_2 \to v\ e_2'} \qquad \textsc{Err-left}\ \frac{e_1 \to \mathit{error}}{e_1\ e_2 \to \mathit{error}} \qquad \textsc{Err-right}\ \frac{e_2 \to \mathit{error}}{v\ e_2 \to \mathit{error}}$$

$$\textsc{Try-v}\ \frac{}{\mathit{try}\ v\ \mathit{with}\ e \to v} \qquad \textsc{Try-err}\ \frac{}{\mathit{try}\ \mathit{error}\ \mathit{with}\ e \to e} \qquad \textsc{Try-ctxt}\ \frac{e_1 \to e_1'}{\mathit{try}\ e_1\ \mathit{with}\ e_2 \to \mathit{try}\ e_1'\ \mathit{with}\ e_2}$$

a. (10 points) What is the normal form of *try* $(\lambda x.\mathit{try}\ x\ 3\ 4\ \mathit{with}\ 5)\ (\lambda y.\mathit{error})\ \mathit{with}\ 6$ under these semantics? Show each step of evaluation to justify your answer. (You need not show the derivations underlying each step, but do show the final $e \to e'$ steps.)

Answer:

$$\begin{array}{r} \mathit{try}\ (\lambda x.\mathit{try}\ x\ 3\ 4\ \mathit{with}\ 5)\ (\lambda y.\mathit{error})\ \mathit{with}\ 6 \to \\ \mathit{try}\ (\mathit{try}\ (\lambda y.\mathit{error})\ 3\ 4\ \mathit{with}\ 5)\ \mathit{with}\ 6 \to \\ \mathit{try}\ (\mathit{try}\ \mathit{error}\ 4\ \mathit{with}\ 5)\ \mathit{with}\ 6 \to \\ \mathit{try}\ (\mathit{try}\ \mathit{error}\ \mathit{with}\ 5)\ \mathit{with}\ 6 \to \\ \mathit{try}\ 5\ \mathit{with}\ 6 \to \\ 5 \end{array}$$

b. (6 points) Suppose we consider *error* to be a value, i.e., we modify the production for v to be $v ::= n \mid \lambda x.e \mid \mathit{error}$. Give a program that could behave differently, and unexpectedly (compared to typical implementations of exceptions), with this change, and explain your answer.

Answer: If we allow *error* to be a value, we could pass it as a parameter or return it as a result, rather than propagating up the call stack. Thus it could potentially be discarded. For example, this would allow $(\lambda x.42)$ *error* to be reduced to either *error* or (unexpectedly) 42, whereas the latter is not allowed in the original semantics.

(The modified semantics has another oddity—It would also allow *try error with e* to evaluate to *error*.)

Question 4. Type checking (22 points). Now consider type checking the language from Question 3. Here is the language again, with type annotations this time:

$$\begin{array}{rcl} e & ::= & v \mid x \mid e\ e \mid \mathit{error} \mid \mathit{try}\ e\ \mathit{with}\ e \\ v & ::= & n \mid \lambda x : t.e \\ t & ::= & \mathit{int} \mid t \to t \\ A & ::= & \cdot \mid x : t, A \end{array}$$

Here are the four standard type rules for simply typed lambda calculus.

$$\textsc{Int}\ \frac{}{A \vdash n : \mathit{int}} \qquad \textsc{Var}\ \frac{x \in \mathit{dom}(A)}{A \vdash x : A(x)} \qquad \textsc{Lam}\ \frac{x : t, A \vdash e : t'}{A \vdash \lambda x : t.e : t \to t'} \qquad \textsc{App}\ \frac{A \vdash e_1 : t \to t' \qquad A \vdash e_2 : t}{A \vdash e_1\ e_2 : t'}$$

a. (10 points) Let $B = x : \mathit{int}, y : \mathit{int} \to \mathit{int}$. What type does $\lambda x : \mathit{int} \to \mathit{int}.\lambda z : \mathit{int}.x\ (y\ z)$ have in environment B? Justify your answer by drawing a derivation showing the typing holds. (To save some writing, you can use i instead of int. Also feel free to introduce new abbreviations, e.g., C, D, E, for environments, as needed.)

Answer: Let $C = x : \mathit{int} \to \mathit{int}, B$ and let $D = z : \mathit{int}, C$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{D \vdash x : \mathit{int} \to \mathit{int}} \qquad \dfrac{\dfrac{}{D \vdash y : \mathit{int} \to \mathit{int}} \qquad \dfrac{}{D \vdash z : \mathit{int}}}{D \vdash y\ z : \mathit{int}}}{D \vdash x\ (y\ z) : \mathit{int}}}{C \vdash \lambda z : \mathit{int}.x\ (y\ z) : \mathit{int} \to \mathit{int}}}{B \vdash \lambda x : \mathit{int} \to \mathit{int}.\lambda z : \mathit{int}.x\ (y\ z) : (\mathit{int} \to \mathit{int}) \to \mathit{int} \to \mathit{int}}$$

b. (6 points) Write down the most general possible type rule for *try* e_1 *with* e_2, and explain briefly why your rule is correct.

Answer:

$$\frac{A \vdash e_1 : t \qquad A \vdash e_2 : t}{A \vdash \mathit{try}\, e_1\ \mathit{with}\, e_2 : t}$$

Since we can't statically predict whether a *try* expression will evaluation to e_1 or e_2, we need to require both to have the same type.

c. (6 points) Write down the most general possible type rule for *error*, and explain why your rule is correct. (Hint: think about what type `raise exn` has in OCaml, for some exception `exn`.)

Answer:

$$\frac{}{A \vdash \mathit{error} : t}$$

We can try *error* as having any type because it never returns normally—its value always propagates by Err-Left and Err-Right until it reaches the top level, or until it's caught by *try*, in which case the value of the *with* handler is returned.