Name:

Midterm 1

CMSC 430
Introduction to Compilers
Fall 2013

October 16, 2013

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		40
3		15
4		25
Total		100

Question 1. Short Answer (20 points).

a. (5 points) Briefly explain the difference between *bottom-up* and *top-down* parsing.

b. (5 points) In at most a few sentences, explain what it means for a term to be *stuck*.

c. (5 points) Briefly explain what the *preservation theorem* is.

d. (5 points) Here is a simplified version of the `json` type from `Yojson.Safe`:

```
type json = [ ‘Assoc of (string * json) list
       | ‘Int of int
       | ‘List of json list ]
```

Write an OCaml function that returns the sum of all the `‘Int` elements of the type `json` above. For example, for `‘List [‘Int 2; ‘Assoc ["foo’’, ‘Int 3]]` it should return 5. You may use any standard OCaml library functions you like, including the following two:

```
# List.fold_left;;
- : (’a -> ’b -> ’a) -> ’a -> ’b list -> ’a = <fun>
# List.map;;
- : (’a -> ’b) -> ’a list -> ’b list = <fun>
```

Question 2. Parsing (40 points).

a. (25 points) Consider the following ocamlyacc output:

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

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

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

state 1
        %entry% : '\001' . s   (5)

        C   shift 3
        .   error
        s   goto 4

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

        $end  accept

state 3
        s : C .   (2)

        .   reduce 2

state 4
        s : s . a B  (1)
        %entry% : '\001' s .   (5)
        a : .   (3)

        A   shift 5
        $end   reduce 5
        B   reduce 3
        a   goto 6

state 5
        a : A . A  (4)

        A   shift 7
        .   error

vstate 6
        s : s a . B  (1)

        B   shift 8
        .   error

state 7
        a : A A .   (4)

        .   reduce 4

state 8
        s : s a B .   (1)

        .   reduce 1
```

i. (5 points) What are the terminals of this grammar, and what are the nonterminals? Exclude the extra symbols added by ocamlyacc.

ii. (15 points) Consider the following table showing the sequence of steps taken by the above parser. Fill in the missing entries in the input and stack columns. 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.

Stack	Input	Action
1 [nothing else to fill in]	[fill in] $end	$s3$
	$end	$r2$
	$end	$s5$
	$end	$s7$
	$end	$r4$
	$end	$s8$
	$end	$r1$

iii. (5 points) Write down a production such that, if it were added to the grammar above, ocamlyacc would report a reduce/reduce conflict. Justify your answer by explaining what state would have a conflict and why.

b. (15 points) Draw the LR(1) parsing DFA for the following grammar:

$$\begin{array}{rcl} S & ::= & Sab \mid D \\ D & ::= & dD \mid \varepsilon \end{array}$$

Question 3. Operational semantics (15 points). Consider extending the lambda calculus to include *options*:

$$\begin{array}{rcl} e & ::= & v \mid x \mid e\ e \mid \text{None} \mid \text{Some}\ e \mid \text{case}\ e\ \text{of None} \to e, \text{Some}\ x \to e \\ v & ::= & n \mid \lambda x.e \mid \text{None} \mid \text{Some}\ v \end{array}$$

Here are the operational semantics rules:

$$\textsc{Beta}\ \frac{}{(\lambda x.e_1)\ v_2 \to e_1[x \mapsto v_2]} \qquad \textsc{Left}\ \frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \textsc{Right}\ \frac{e_2 \to e_2'}{v\ e_2 \to v\ e_2'}$$

$$\textsc{Case}\ \frac{e_0 \to e_0'}{\begin{array}{r}(\text{case}\ e_0\ \text{of None} \to e_1, \text{Some}\ x \to e_2) \to \\ (\text{case}\ e_0'\ \text{of None} \to e_1, \text{Some}\ x \to e_2)\end{array}}$$

$$\textsc{Case-None}\ \frac{}{(\text{case None of None} \to e_1, \text{Some}\ x \to e_2) \to e_1} \qquad \textsc{Case-Some}\ \frac{}{(\text{case Some}\ v\ \text{of None} \to e_1, \text{Some}\ x \to e_2) \to e_2[x \mapsto v]}$$

a. (5 points) Following the above small-step rules, reduce the following term to a normal form. Show each step $e \to e'$ of reduction, but don't show the derivations underlying each step. For example, if we asked you to reduce $((\lambda x.x)\ (\lambda y.y))\ (\lambda z.z)$, your answer would be:

$$\begin{array}{rl} ((\lambda x.x)\ (\lambda y.y))\ (\lambda z.z) & \to \\ (\lambda y.y)\ (\lambda z.z) & \to \\ \lambda z.z & \end{array}$$

Reduce this term:

$$((\lambda x.\text{case}\ x\ \text{of None} \to (\lambda z.z), \text{Some}\ y \to (\lambda w.y))\ (\text{Some}\ \ 42))\ 5$$

b. (5 points) We left out one operational semantics rule above. Write down the missing rule, along with a program that cannot be evaluated without it.

c. (5 points) Write down two *big-step* operational semantics rules for "case" that are equivalent to the small-step rules Case, Case-None, and Case-Some. As a reminder, here is a big-step rule for evaluating function application:

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

Question 4. Type checking (25 points). Now suppose we extend the simply typed lambda calculus to support options; thus we need to add *option types*:

$$
\begin{array}{rcl}
e & ::= & v \mid x \mid e\ e \mid \text{None} \mid \text{Some } e \mid \text{case } e \text{ of None} \to e, \text{Some } x{:}\,t \to e \\
v & ::= & n \mid \lambda x{:}\,t.e \mid \text{None} \mid \text{Some } v \\
t & ::= & \mathit{int} \mid t \to t \mid t\ \mathit{option} \\
A & ::= & \cdot \mid x{:}\,t, A
\end{array}
$$

Here are some of the type rules for this language:

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

a. (10 points) Fill in the types in the typing derivation below. You will need to determine what types t_0, t_1, and t_2 are. To save writing, write i for *int* and *io* for *int option*.

$t_0 = \boxed{}$ $\qquad t_1 = \boxed{}$ $\qquad t_2 = \boxed{}$

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{y{:}\boxed{}, x{:}\boxed{} \vdash x : \boxed{}}
\quad
\dfrac{\dfrac{}{y{:}\boxed{}, x{:}\boxed{} \vdash y : \boxed{}}}{y{:}\boxed{}, x{:}\boxed{} \vdash \text{Some } y : \boxed{}}
}{y{:}\boxed{}, x{:}\boxed{} \vdash x\ (\text{Some } y) : \boxed{}}
}{x{:}\boxed{} \vdash \lambda y{:}\,t_1.x\ (\text{Some } y)) : \boxed{}}
}{\cdot \vdash (\lambda x{:}\,t_0.\lambda y{:}\,t_1.x\ (\text{Some } y)) : \boxed{}}
\quad
\dfrac{\dfrac{}{z : \boxed{} \vdash z : \boxed{}}}{\cdot \vdash (\lambda z{:}\,t_2.z) : \boxed{}}
}{\cdot \vdash (\lambda x{:}\,t_0.\lambda y{:}\,t_1.x\ (\text{Some } y))\ (\lambda z{:}\,t_2.z) : \boxed{}}
\quad
\dfrac{}{\cdot \vdash 42 : \boxed{}}
}{\cdot \vdash (\lambda x{:}\,t_0.\lambda y{:}\,t_1.x\ (\text{Some } y))\ (\lambda z{:}\,t_2.z)\ 42 : \boxed{}}
$$

b. (5 points) Write down the most general possible type rule for "None."

c. (5 points) Write down the most general possible type rule for "case."

d. (5 points) Using the following subset of the grammar that is missing lambda:

$$\begin{array}{rcl} e & ::= & v \mid x \mid e\ e \mid \text{None} \mid \text{Some } e \mid \text{case } e \text{ of None} \rightarrow e, \text{Some } x{:}\, t \rightarrow e \\ v & ::= & n \mid \text{None} \mid \text{Some } v \\ t & ::= & \mathit{int} \mid t \rightarrow t \mid t\ \mathit{option} \\ A & ::= & \cdot \mid x{:}\, t, A \end{array}$$

write down a term that is ill-typed according to the type rule from part c and yet does not get stuck at run time.