CSCE 531, Spring 2015
Final Exam Answer Key

1. (40 points total) Consider the following grammar with start symbol S':

$$\begin{aligned} S' &\to S \\ S &\to aSb \\ S &\to T \\ T &\to Ta \\ T &\to cS \\ T &\to \epsilon \end{aligned}$$

   (a) (10 points) Find $\mathit{FIRST}(S)$, $\mathit{FIRST}(T)$, $\mathit{FOLLOW}(S)$, and $\mathit{FOLLOW}(T)$. You need not justify your answer.

   (b) (15 points) Using the method described in class or the text, start to construct the set

$$\{I_0, I_1, I_2, \ldots\}$$

   of states for a canonical LR(1) parser for this grammar, defining the transition function at the same time. Compute the start state I_0 first, then *only* the possible transitions from I_0. This should produce four additional states I_1, I_2, I_3, I_4, and you do not compute any further transitions. You should display states in condensed form, that is, if there are multiple lookaheads attached to the same LR(0) item, then list the item once with the lookaheads separated by slashes, for example,

$$T \to .Ta,\ \$/a$$

   (Omit the brackets for all items.) Note that in class I denoted the transition function as *trans*. The textbook denotes the same function as *goto*. You may use the letter t to denote this function.

   (c) (10 points) Give as much of the action table for this parser as you can, based on your partial construction above. That is, define action$[i, a]$ for each state I_i and terminal a (including \$) that you can establish from your construction. Omit "error" entries.

   (d) (5 points) Describe any conflicts you find, if any, giving the conflicting actions.

Answer:

(a)

$$\mathit{FIRST}(S) = \mathit{FIRST}(T) = \{a, c, \epsilon\}\ , \qquad \mathit{FOLLOW}(S) = \mathit{FOLLOW}(T) = \{a, b, \$\}\ .$$

(b)

$I_0 = \text{start state}:$
$S' \to .S,\ \$$
$S \to .aSb,\ \$$
$S \to .T,\ \$$
$T \to .Ta,\ \$/a$
$T \to .cS,\ \$/a$
$T \to .,\ \$/a$

$I_2 = t(I_0, a):$
$S \to a.Sb,\ \$$
$S \to .aSb,\ b$
$S \to .T,\ b$
$T \to .Ta,\ b/a$
$T \to .cS,\ b/a$
$T \to .,\ b/a$

$I_4 = t(I_0, c)$
$T \to c.S,\ \$/a$
$S \to .aSb,\ \$/a$
$S \to .T,\ \$/a$
$T \to .Ta,\ \$/a$
$T \to .cS,\ \$/a$
$T \to .,\ \$/a$

$I_1 = t(I_0, S):$
$S' \to S.,\ \$$

$I_3 = t(I_0, T)$
$S \to T.,\ \$$
$T \to T.a,\ \$/a$

(c) We omit any action items that mention states we have not already constructed.

	a	b	c	\$
0	shift 2, reduce $T \to \epsilon$		shift 4	reduce $T \to \epsilon$
1				accept
2	reduce $T \to \epsilon$	reduce $T \to \epsilon$		
3				reduce $S \to T$
4	reduce $T \to \epsilon$		shift 4	reduce $T \to \epsilon$

(d) Our table in (c) above indicates a shift/reduce conflict in $action[0, a]$ and no other conflicts. However (and this is optional), we know that there will be shift/reduce conflicts in $action[2, a]$ and $action[4, a]$.

2. (20 points) Consider a bottom-up grammar in BNF for control flow constructs in a typical Pascal-like programming language

$\langle start \rangle$ ::= $\langle stmt \rangle$
$\langle stmt \rangle$ ::= **if** e **then** $\langle stmt \rangle_1$
$\langle stmt \rangle$ ::= **if** e **then** $\langle stmt \rangle_1$ **else** $\langle stmt \rangle_2$
$\langle stmt \rangle$ ::= **repeat** $\langle stmt_list \rangle$ **until** e
$\langle stmt \rangle$ ::= **begin** $\langle stmt_list \rangle$ **end**
$\langle stmt \rangle$ ::= **break**
$\langle stmt \rangle$ ::= **other**
$\langle stmt_list \rangle$ ::= `/* empty */`
$\langle stmt_list \rangle$ ::= $\langle stmt_list \rangle_1$ `';'` $\langle stmt \rangle$

A **break** statement is meant to unconditionally jump to the end of the closest surrounding **repeat**-loop, if there is one. (Note that a **break** statement might jump out of an arbitrary number of nested **if** and **begin–end** statements.)

Add semantic rules to compute the following inherited attribute: $\langle stmt \rangle.break$ is a string that is the destination label to which to jump if a break statement is encountered within $\langle stmt \rangle$. If the statement is not inside any **repeat** statement, then use the special value NIL for this

attribute. You may assume a function `newSymbol()` is available that returns a fresh label every time it is called.

Also add semantic actions that

- emit an unconditional jump to the proper destination whenever a **break** statement is encountered (if there is no proper destination, issue an error message), and
- emit the corresponding break destination as a label (i.e., followed by a colon) after each **repeat** statement.

You may assume a function `emit(...)` is available that emits whatever string is passed to it.

You may define any additional attributes that you find helpful, but you must pass all data around on the semantic stack, i.e., you may not declare or use any external (global) variables or data structures. Also, your syntax-directed translation scheme must be L-attributed, and no intermediate actions are allowed.

Answer:

Production	Semantic action
$\langle start \rangle ::= \langle stmt \rangle$	$\langle stmt \rangle.break := \text{NIL}$
$\langle stmt \rangle ::= \textbf{if } e \textbf{ then } \langle stmt \rangle_1$	$\langle stmt \rangle_1.break := \langle stmt \rangle.break$
$\langle stmt \rangle ::= \textbf{if } e \textbf{ then } \langle stmt \rangle_1 \textbf{ else } \langle stmt \rangle_2$	$\langle stmt \rangle_1.break := \langle stmt \rangle.break;$ $\langle stmt \rangle_2.break := \langle stmt \rangle.break$
$\langle stmt \rangle ::= \textbf{repeat } \langle stmt_list \rangle \textbf{ until } e$	$\langle stmt_list \rangle.break := \text{newSymbol}();$ $\text{emit}(\langle stmt_list \rangle.break); \text{emit}(\text{“:”})$
$\langle stmt \rangle ::= \textbf{begin } \langle stmt_list \rangle \textbf{ end}$	$\langle stmt_list \rangle.break := \langle stmt \rangle.break$

$\langle stmt \rangle ::= \textbf{break}$	**if** $\langle stmt \rangle.break <> \text{NIL}$ **then**
	emit(“goto”); emit($\langle stmt \rangle.break$)
	else
	semantic_error()

$\langle stmt \rangle ::= \textbf{other}$	
$\langle stmt_list \rangle ::=$ `/* empty */`	
$\langle stmt_list \rangle ::= \langle stmt_list \rangle_1$ `';'` $\langle stmt \rangle$	$\langle stmt_list \rangle_1.break := \langle stmt_list \rangle.break;$ $\langle stmt \rangle.break := \langle stmt_list \rangle.break$

3. (20 points) Consider the following bison grammar for a C-like programming language fragment (with start symbol `assignment`):

```
assignment :
      expr '=' expr ';'
    ;
expr :
      expr '+' unary
    | unary
    ;
unary :
      '*' unary
    | '&' unary
    | factor
    ;
factor :
      CONST
    | VAR
    | '(' expr ')'
    ;
```

Note:

- The assignment operator `'='` requires an L-value for the left-hand expr, and an R-value for the right-hand expr. (The assignment itself returns no value.)
- The addition operator `'+'` requires two R-values and returns an R-value.
- The pointer indirection operator `'*'` requires an R-value and returns an L-value.
- The address-of operator `'&'` requires an L-value and returns an R-value.
- Constants (`CONST`) are R-values, and variables (`VAR`) are L-values.
- Parentheses have no effect on the kind of value (R- versus L-).

Add bison-appropriate semantic actions that do three things:

- Compute, as synthesized attributes of `expr`, `unary`, and `factor`, whether or not the (sub)expression is an R-value or an L-value. Use the integer value 0 to mean R-value and 1 to mean L-value. Compute these using the usual `$`-references.
- Issue instructions (for a human being) to dereference an operand whenever it is an L-value but an R-value is expected for the operation. For example, when the left operand of the '+' operator is an L-value, then include the action

  ```
  msg("deref left operand");
  ```

  in that production (and similarly for the right operand).
- Whenever an R-value is present where an L-value is expected, include the action

  ```
  error("L-val expected");
  ```

Do not use any intermediate actions, only reduce-actions. You do not need to provide the human with any other information, e.g., which production generates a particular message. You also need not worry about (data) types of expressions.

Answer:

```
assignment :
      expr '=' expr ';'  { if (!$1) error("L-val expected");
                           if ($3) msg("deref right-hand side");
                         }
    ;
expr :
      expr '+' unary     { if ($1) msg("deref left operand");
                           if ($3) msg("deref right operand");
                           $$ = 0;
                         }
    | unary              /* default action: $$ = $1; */
    ;
unary :
      '*' unary          { if ($2) msg("deref operand");
                           $$ = 1;
                         }
    | '&' unary          { if (!$2) error("L-val expected");
                           $$ = 0;
                         }
    | factor             /* default action */
    ;
factor :
      CONST              { $$ = 0; }
    | VAR                { $$ = 1; }
    | '(' expr ')'       { $$ = $2; }
    ;
```

4. (15 points) Consider the following Pascal declaration:

```
var
   a : array[3..10] of array[2..9] of array[4..8] of Real;
```

Assuming that a Pascal Real is eight bytes and that the base address of a is 1500, find the base address of the Real variable `a[9][6][5]`.

Answer: [Note: the original question mistakenly suggested that `a[9][6][5]` is an integer variable. It is a Real variable.] Here are the sizes of the relevant types:

type	size
`Real`	8
`array[4..8] of Real`	$(8-4+1)\cdot 8 = 40$
`array[2..9] of array[4..8] of Real`	$(9-2+1)\cdot 40 = 320$

Thus the base address of `a[9][6][5]` is

$$1500 + (9-3)320 + (6-2)40 + (5-4)8 = 1500 + 2088 = 3588 .$$

5. (25 points total) Consider the following three-address code (line numbers added):

```
1     L1:     i := a
2             t1 := i + 5
3     L2:     if t1 <= 10 then goto L4
4             j := i + 1
5             if j >= 0 then goto L2
6             a := a + j
7     L3:     if j <= 10 then goto L5
8     L4:     i := j + 1
9             t1 := 2 * j
10            goto L3
11    L5:     a := t1 - 1
12            if a > 0 then goto L1
13
```

Assume that control enters at line 1 and that there are no other entry points.

(a) (5 points) Describe the basic blocks $B_1, B_2, \ldots$ by giving an inclusive range of line numbers for each block.

(b) (10 points) Draw the flow diagram as a directed graph, labeling the nodes $B_1, B_2, \ldots$. Give dangling arrows both for the entry point and for the exit point of the code as a whole.

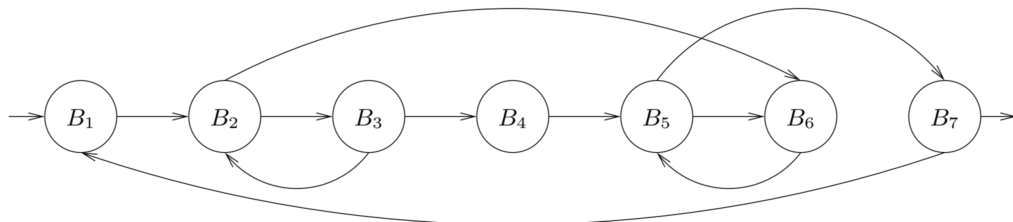
(c) (5 points) Using the strict definition of a loop as defined in class and in the text, list any sets of vertices that constitute loops. Label any inner loop(s) as such.

(d) (5 points) Is the variable `j` alive between lines 2 and 3? Explain.

Answer:

(a)

Block	**Line(s)**
B_1	1–2
B_2	3
B_3	4–5
B_4	6
B_5	7
B_6	8–10
B_7	11–12



(b)

(c) There are two loops:

- $\{B_1, B_2, B_3, B_4, B_5, B_6, B_7\}$.
- $\{B_2, B_3\}$ (inner loop).

(d) Yes, `j` is alive. Line 3 could go to line 8, where `j` is used but not set in between.