CS 324 – Language Processors

MID-SEMESTER EXAMINATION

DATE: 20th September 2003 **Weightage: 20%**
Maximum Marks: 80 **Open Notes Exam Exam**

---

1. The intention behind this question is to test your overview of the area of language processing. Parts of this question relate to topics which have been touched upon, but not covered in details. Therefore, wherever appropriate, I am willing to tolerate certain amount of imprecision in your answers.

   Consider the Pascal program and the compilation of the function `average` shown below:

```
1.  type arr = array [1..10] of integer;
2.  var a  : arr;   sum : integer;

3.  function average(var a: arr; max : integer): integer;
4.  var i : integer;
5.  begin
6.     sum := 0;
7.     for i:= 1 to max do sum := sum + a[i];
8.     average := sum div  max;
9.  end; { average }

10. begin
11.    { the array a is read here }
12.    writeln('the average is', average(a, 10), 'the sum is', sum);
13. end.
```

```
average:            .L9:                          .L8:
pushl %ebp          movl 8(%ebp),%eax             movswl _SUM,%eax
movl %esp,%ebp      movswl -4(%ebp),%edx          movswl 12(%ebp),%edx
subl $8,%esp        movswl -2(%eax,%edx,2),%eax   movl %edx,%edi
movw $0,_SUM        movswl _SUM,%edx              cltd
movw 12(%ebp),%di   addl %eax,%edx                idivl %edi
movw %di,-8(%ebp)   movw %dx,_SUM                 movw %ax,-2(%ebp)
movw $1,-4(%ebp)    movw -4(%ebp),%ax             movw -2(%ebp),%ax
movw -4(%ebp),%ax   cmpw -8(%ebp),%ax             leave
cmpw -8(%ebp),%ax   jge .L8                       ret $8
jg .L8              incw -4(%ebp)
                    jmp .L9
```

   Now answer the following questions:

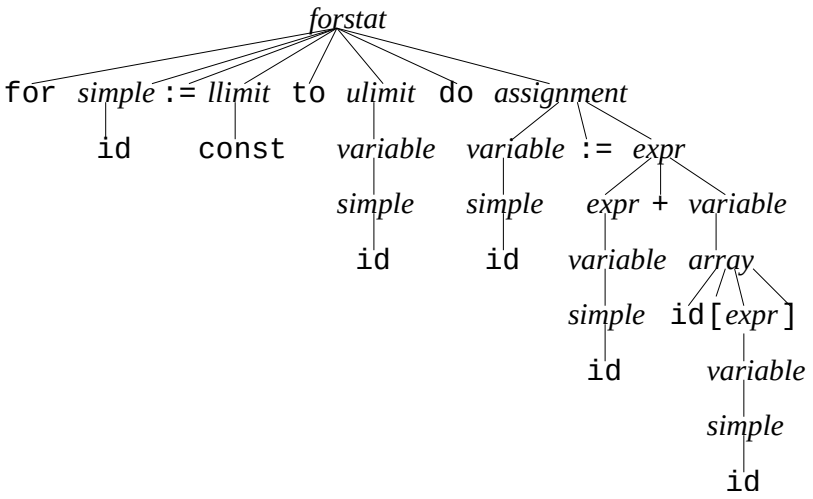
   (a) Find all lexemes in line 12 and group them according to tokens.
   Soln: These are:
   `writeln, average, a, sum` – identifier
   `(` – left parenthesis

`)` – right parenthesis
`'the average is', 'the sum is'` – strings
`,` – comma
`;` – semicolon
`10` – constant

(b) Write a grammar to express `for` statements whose body consists of a single assignment statement. The assignment statement should be made of simple and (one dimensional) array variables and the operators `:=` and `+`. The bounds of the for statement are variables or constants.

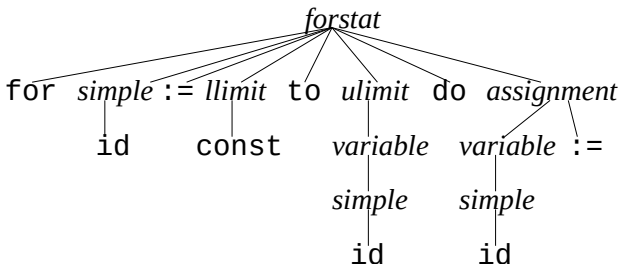
forstat → `for` *simple* `:=` *llimit* `to` *ulimit* `do` *assignment*
assignment → *variable* `:=` *expression*
variable → *simple* | *array*
simple → `identifier`
array → `identifier [` *expression* `]`
expression → *expression* `+` *variable* | *expression* `+ constant` | *variable* | `constant`
llimit → *variable* | `constant`
ulimit → *variable* | `constant`

(c) Construct the parse tree for the `for` statement in line 7 using the above grammar.



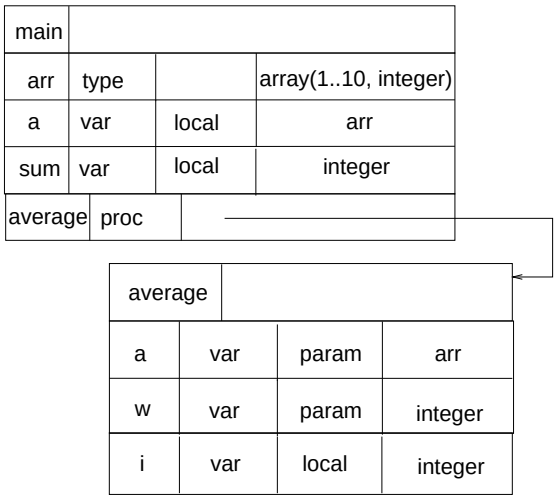
(d) What part of the parse tree would have been constructed after reading the second assignment operator at line 7, if the parser is a (i) top-down (ii) bottom-up parser?

Soln: In both cases, the parse tree constructed would be:



(e) Show the contents of the symbol table after processing line 4.

Soln:



(f) Describe how the symboltable would be used to locate the type of `sum` at line 7.

First the current symboltable, the symboltable of `average`, is searched for `sum`. Since the search will fail, the symboltable for `main` is searched and it is found there.

(g) Describe in not more than three lines, what is involved in type checking the statement `sum := sum + a[i]` at line 7.

The right hand side of the assignment is typechecked first. The type of `sum` is found to be an integer from the symboltable. Next, since `a` is found to be an array of integers and `i` is an integer, `a[i]` is deduced to be an integer. Thus the right hand side of the assignment is an integer. Since `sum`, the left hand side of the assignment, is also an integer, it is concluded that the assignment is type ok.

(h) What would the overloaded operator `+` in line 7 resolve to? How has this information been employed in the generated code?

Since both operands of `+` are integers, `+` represents integer addition.

Because of this the addition instruction in block `L9` is an integer addition. Otherwise a floating point addition would have been issued.

(i) What addresses have been given to the following items – (i) `max` (ii) `i`, and (iii) `sum`.

Soln: (i) `ebp + 12` (ii) `ebp - 4` (iii) `sum` has not been given an address on the stack. It has been given an absolute address denoted by `_SUM`.

(j) What items have been given the following addresses relative to the base pointer of `average` – (i) `ebp + 8` (ii) `ebp - 8`, and (iii) `ebp + 4`?

Soln: (i) Starting address of the array `a`, (ii) the upper limit of the for loop is stored here. The upper limit, in general, is an expression whose value needs to be stored somewhere, (iii) the return address.

$(2 + 5 + 2 + 3 + 4 + 3 + 3 + 2 + 6 + 6 = 36$ Marks)

2. Explain why the following Lex patterns are incorrect:

(a) `({DIGIT}*\.{DIGIT}*(e-?{DIGIT}+)?` as representation of real numbers.

(b) `"/*"[^*]*(\*([^/][^*]*)?)*"*/"` as representation of comments.

Soln: (i) accepts `.` (ii) accepts `/*abc**/*/`. $(4 + 5 = 9$ Marks)

3. The grammar $S \rightarrow a\ S\ a \mid a\ a$ generates all even length strings of a's except for the empty string. Consider a recursive descent parser with backtracking that tries the $a\ S\ a$ alternative before $a\ a$.

   (a) Does the parser succeed on (i) 4 a's (ii) 6 a's?
   Soln: (i) Yes (ii) No.

   (b) Let us denote by s_n the n^{th} shortest string accepted by the parser. Now complete the following.
   $s_1 = aa$
   $s_n = s_{n-1}s_{n-1}$

   I shall explain this later when I have time.

4. A *no* answer to any part of the following question has to be justified by writing a parsing table entry with a conflict. A *yes* answer requires no explanation.

   (a) Is the following grammar (i) SLR(1) (ii) LALR(1) (iii) LR(1)?
   $S \rightarrow d\ M\ N\ c \mid N \mid L$
   $N \rightarrow d\ b \mid c\ d$
   $L \rightarrow d\ M\ d\ M$
   $M \rightarrow b$
   Soln: (i) The grammar is not SLR(1) because it is not LALR(1) (ii) It is not LALR(1) because the state corresponding to the viable prefix db contains the items $N \rightarrow d\ b\ \circ, \{\$\}$ and $M \rightarrow b\ \circ, \{d,\ c\}$. The state corresponding to the viable prefix $dMdb$, on the other hand contains the items $N \rightarrow d\ b\ \circ, \{c\}$ and $M \rightarrow b\ \circ, \{\$\}$. When these two state are merged, we get a state containing the items $N \rightarrow d\ b\ \circ, \{\$,\ c\}$ and $M \rightarrow b\ \circ, \{\$,\ d,\ c\}$. (iii) The grammar is LR(1).

   (b) Is the following grammar (i) LL(1) (ii) SLR(1) (iii) LALR(1)?
   $S \rightarrow d\ B \mid A\ d \mid B\ c$
   $A \rightarrow C\ c$
   $B \rightarrow a$
   $C \rightarrow d\ a$ $(7 + 7 = 14$ Marks)
   Soln: (i) This grammar is not LL(1) because $FIRST(d\ B)$ and $FIRST(A\ d)$ have d in common.
   (ii) The grammar is not SLR(1) because, in the state corresponding to viable prefix da, there are two items $C \rightarrow d\ a\ \circ$ and $B \rightarrow a\ \circ$. Since c is in $FOLLOW$ of both C and B, there is a reduce-reduce conflict.
   (iii) The grammar is LALR(1). The item $B \rightarrow a\ \circ,\ \{c\}$ will not be there in the LALR(1) parsing table.

5. Construct a grammar which is LL(1) but not LALR(1). Explain your answer. (6 Marks)
   Soln: First make two distinct viable prefixes starting with a and b. The idea is that a LR(1) parser would have kept the states at these viable prefixes apart. A LALR(1) parser will merge them instead.
   $S \rightarrow a\ A \mid b\ B$
   Now create symmetrical situations for A and B. First start with A

$A \rightarrow C\ a \mid D\ b$
$C \rightarrow E$
$D \rightarrow E$

So, on the viable prefix aE, there will be a state with the items $C \rightarrow E \circ$, $\{a\}$ and $D \rightarrow E \circ$, $\{b\}$.

Now do a similar thing for B.

$B \rightarrow C\ b \mid D\ a$

So that for the viable prefix bE there will be a state with items $C \rightarrow E \circ$, $\{b\}$ and $D \rightarrow E \circ$, $\{a\}$. LALR(1) would then combine these two states and get into a conflict.

We have made the grammar non LALR(1) without having defined E yet. We have to do this in a way that makes the grammar LL(1). How does one do it? Simple. Define E as ϵ so that LL(1) uses the power of FIRST to bore[1] through E (and therefore through C and D) to see what lies beyond.

See what I mean?

[1] *bore*, as in *the engineers bored a tunnel*, and not as in *AS bored us to death in the class today.*