UNIVERSITY OF LIMERICK
OLLSCOIL LUIMNIGH

COLLEGE OF INFORMATICS & ELECTRONICS

DEPARTMENT OF ELECTRONIC & COMPUTER ENGINEERING

MODULE CODE:	**CE4717**
MODULE TITLE:	**Language Processors**
SEMESTER:	**Autumn 2004**
DURATION OF EXAM:	**2 Hours**
LECTURER:	**Dr. C. Flanagan**

INSTRUCTIONS TO CANDIDATES:

Please answer three questions.

All questions carry equal marks.

This exam represents 60% of the module assessment.

Please note that the instruction set of the target stack machine and the grammar for the compiler project language are included with this paper. If a question requires assembly code, it is expected that you will use stack machine assembly in your answer.

Q1. (a) What does it mean to say that a given grammar is LL(0)? Explain your answer, paying particular attention to how an LL(0) grammar may be distinguished from an LL(1) grammar. [4 Marks]

(b) Consider the following grammar:

$$\begin{array}{ll} \mathrm{S} & \rightarrow \mathrm{A} \\ \mathrm{A} & \rightarrow \mathrm{BBB} \\ \mathrm{B} & \rightarrow \mathrm{bb} \end{array}$$

i. Is the above grammar LL(0)? Justify your answer. [3 Marks]

ii. Is it LL(1)? Justify your answer. [2 Marks]

iii. How many sentences does this grammar generate? [1 Mark]

iv. It is possible for a valid LL(0) grammar to generate an infinite set of sentences? Justify your answer. [2 Marks]

v. Do you consider that LL(0) grammars are potentially useful in the design of practical programming languages? Justify your answer. [3 Marks]

(c) Give two possible parse trees for the expression "$n + n * n$", one in which "$+$" has higher precedence than "$*$", and the other where the converse applies. For each case express this precedence in terms of a generative grammar. [5 Marks]

Q2. Consider the following grammar:

$$\begin{array}{ll} \mathrm{S} & \rightarrow \mathrm{BA} \\ \mathrm{A} & \rightarrow \mathrm{aBA}|\epsilon \\ \mathrm{B} & \rightarrow \mathrm{DC} \\ \mathrm{C} & \rightarrow \mathrm{cDC}|\epsilon \\ \mathrm{D} & \rightarrow \mathrm{eSf}|\mathrm{g} \end{array}$$

i. Draw the syntax diagram for the grammar. [3 Marks]

ii. Calculate the FIRST sets of the grammar nonterminals. [5 Marks]

iii. Calculate the FOLLOW sets of the grammar nonterminals. [5 Marks]

iv. Calculate the PREDICT sets for *all* the grammar productions. [5 Marks]

v. Is the grammar LL(1)? Justify your answer. [2 Marks]

Q3. In LISP, an expression is formed as follows: a left parenthesis is followed by an operator, in its turn followed by one or more variables, numbers or expressions, the whole terminated by a right parenthesis. For example, the following are LISP expressions:

```
( + 2 a )
( * ( + 2 4 ) ( - b 7 ) )
( - 3 )
```

Assume that all variables are global and all numbers are unsigned integers. The operators are "$+$", "$-$", "$*$" and "$/$".

i. Write a grammar for LISP expressions. [4 Marks]

ii. Write a combined parser and code generator for such expressions. The code generated should use instructions from the stack machine. Assume an `Accept` routine is available which implements S-Algol error recovery, so you may ignore syntactic error recovery. Clearly indicate how both the parsing and code generation are achieved. Pay particular attention to handling unary operators and semantic errors. [16 Marks]

Q4. Consider the following regular expression:

$$(a|b)^*ba$$

i. Using Thompson's construction, generate an NFA for it. [8 Marks]

ii. Using the subset algorithm, convert the NFA to a DFA. [12 Marks]

Q5. Assume you wish to add a "**FOR**" loop capability to the compiler project language:

$$\begin{aligned} \langle \textit{ForStatement} \rangle \quad &:== \quad \text{“}\textbf{FOR}\text{”} \ \langle \textit{ForLimits} \rangle \ \text{“}\textbf{DO}\text{”} \ \langle \textit{Block} \rangle \\ \langle \textit{ForLimits} \rangle \quad &:== \quad \langle \textit{Identifier} \rangle \ \text{“}\textbf{:=}\text{”} \ \langle \textit{IntConst} \rangle \ \text{“}\textbf{TO}\text{”} \ \langle \textit{IntConst} \rangle \end{aligned}$$

You may assume that the procedure to parse a ⟨*Block*⟩ has already been written, and that tokens "**IDENTIFIER**" and "**INTCONST**" are returned by the scanner.

i. Sketch out how the "**FOR**" construct can be expressed in terms of conditional and unconditional jumps. [5 Marks]

ii. Define "*Backpatching*" and show where it is necessary in parsing the "**FOR**" construct. [3 Marks]

iii. Generate appropriate assembly code for the following program fragment. Assume all variables are global.

```
FOR i := 1 TO 10 DO
BEGIN
    WRITE( i );
END;
```

Explain how your assembly code implements the loop and why you chose your approach. [3 Marks]

iv. Write code for the appropriate routine(s) that are needed in a compiler to implement the "**FOR**" construct. [9 Marks]

The instruction set for the stack machine

`Add` Example: `Add`

$[\mathrm{SP}-1] \leftarrow [\mathrm{SP}-1] + [\mathrm{SP}]$; $\mathrm{SP} \leftarrow \mathrm{SP}-1$

"Addition". Pop the top two stack elements, add them and push the result back on the stack

`Sub` `Sub`

$[\mathrm{SP}-1] \leftarrow [\mathrm{SP}-1] - [\mathrm{SP}]$; $\mathrm{SP} \leftarrow \mathrm{SP}-1$

"Subtraction". Subtract the top of stack from the next element on the stack. Two stack elements are popped and the result is pushed back.

`Mult` `Mult`

$[\mathrm{SP}-1] \leftarrow [\mathrm{SP}-1] \times [\mathrm{SP}]$; $\mathrm{SP} \leftarrow \mathrm{SP}-1$

"Multiplication". Pop the top two stack elements, multiply them and push the result back on the stack.

`Div` `Div`

$[\mathrm{SP}-1] \leftarrow [\mathrm{SP}-1] \div [\mathrm{SP}]$; $\mathrm{SP} \leftarrow \mathrm{SP}-1$

"Division". Divide the top of stack by the next element on the stack. Two stack elements are popped and the result is pushed back.

`Neg` `Neg`

$[\mathrm{SP}] \leftarrow -[\mathrm{SP}]$

"Negation". Negate the top of stack, i.e., replace it with its two's complement.

`Br` $\langle addr \rangle$ `Br 200`

$\mathrm{PC} \leftarrow \langle addr \rangle$

"Branch to address". Branch to memory address $\langle addr \rangle$.

`Bgz` $\langle addr \rangle$ `Bgz 200`

if $[\mathrm{SP}] \geq 0$ then $\mathrm{PC} \leftarrow \langle addr \rangle$; $\mathrm{SP} \leftarrow \mathrm{SP}-1$

"Branch if Greater than or equal to Zero". Branch to memory address $\langle addr \rangle$ if the top of stack is greater than or equal to zero. Pop the stack.

`Bg` $\langle addr \rangle$ `Bg 200`

if $[\mathrm{SP}] > 0$ then $\mathrm{PC} \leftarrow \langle addr \rangle$; $\mathrm{SP} \leftarrow \mathrm{SP}-1$

"Branch if Greater than zero". Branch to memory address $\langle addr \rangle$ if the top of stack is greater than zero. Pop the stack.

`Blz` $\langle addr \rangle$ `Blz 200`

if $[\mathrm{SP}] \leq 0$ then $\mathrm{PC} \leftarrow \langle addr \rangle$; $\mathrm{SP} \leftarrow \mathrm{SP}-1$

"Branch if Less than or equal to Zero". Branch to memory address $\langle addr \rangle$ if the top of stack is less than or equal to zero. Pop the stack.

`Bl` $\langle addr \rangle$ `Bl 200`

if $[\mathrm{SP}] < 0$ then $\mathrm{PC} \leftarrow \langle addr \rangle$; $\mathrm{SP} \leftarrow \mathrm{SP}-1$

"Branch if Less than zero". Branch to memory address $\langle addr \rangle$ if the top of stack is less than zero. Pop the stack.

`Bz` $\langle addr \rangle$ `Bz 200`

if $[\mathrm{SP}] = 0$ then $\mathrm{PC} \leftarrow \langle addr \rangle$; $\mathrm{SP} \leftarrow \mathrm{SP}-1$

"Branch if equal to Zero". Branch to memory address $\langle addr \rangle$ if the top of stack is equal to zero. Pop the stack.

`Bnz` $\langle addr \rangle$ `Bnz 100`

if $[\mathrm{SP}] \neq 0$ then $\mathrm{PC} \leftarrow \langle addr \rangle$; $\mathrm{SP} \leftarrow \mathrm{SP}-1$

"Branch if not equal to Zero". Branch to memory address $\langle addr \rangle$ if the top of stack is not equal to zero. Pop the stack.

`Call` $\langle addr \rangle$ `Call 234`

$\mathrm{SP} \leftarrow \mathrm{SP}+1$; $[\mathrm{SP}] \leftarrow \mathrm{PC}+1$; $\mathrm{PC} \leftarrow \langle addr \rangle$;

"Call a subroutine". First push the address of the instruction following the `Call` onto the stack, then jump to $\langle addr \rangle$.

`Ret` `Ret`

$\mathrm{PC} \leftarrow [\mathrm{SP}]$; $\mathrm{SP} \leftarrow \mathrm{SP}-1$;

"Return from a subroutine". Pop the top of stack and place its value in the program counter.

`Bsf` `Bsf`

$\langle temp \rangle \leftarrow \mathrm{FP}$; $\mathrm{FP} \leftarrow \mathrm{SP}$; $\mathrm{SP} \leftarrow \mathrm{SP}+1$; $[\mathrm{SP}] \leftarrow \langle temp \rangle$;

"Build Stack Frame". Prepare to enter a subroutine by building the dynamic link portion of a stack frame.

`Rsf` `Rsf`

$\mathrm{SP} \leftarrow \mathrm{FP}-1$; $\mathrm{FP} \leftarrow [\mathrm{FP}+1]$

"Remove Stack Frame". Remove a stack frame on exiting a subroutine. Restore the Frame Pointer to point to the base of the previous stack frame.

`Ldp` $\langle addr \rangle$ `Ldp 103`

$[\mathrm{FP}] \leftarrow [\langle addr \rangle]$; $[\langle addr \rangle] \leftarrow \mathrm{FP}$

"Load Display Pointer". Copy the old display pointer in $\langle addr \rangle$ to the location pointed to by the Frame Pointer, then replace it with the current value of the Frame Pointer.

`Rdp` $\langle addr \rangle$ `Rdp 103`

$[\langle addr \rangle] \leftarrow [\mathrm{FP}]$

"Restore old Display Pointer". Replace the display pointer currently at $\langle addr \rangle$ with the previously active display pointer, currently preserved in the location pointed at by the Frame Pointer.

`Inc` $\langle words \rangle$ `Inc 4`

$\mathrm{SP} \leftarrow \mathrm{SP} + \langle words \rangle$

"Increment the stack pointer". Create $\langle words \rangle$ words of local variable space on the stack.

`Dec` $\langle \textit{words} \rangle$ `Dec 4`

SP $\leftarrow$ SP $-$ $\langle \textit{words} \rangle$

"Decrement the stack pointer". Remove $\langle \textit{words} \rangle$ words of local variable space from the stack.

`Push FP` `Push FP`

SP $\leftarrow$ SP $+ 1$; [SP] $\leftarrow$ FP

"Push the Frame Pointer". Push the current value of the Frame Pointer register onto the top of the stack.

`Load #`$\langle \textit{datum} \rangle$ `Load #-200`

SP $\leftarrow$ SP $+ 1$; [SP] $\leftarrow$ $\langle \textit{datum} \rangle$

"Load immediate datum". Increment the Stack Pointer. Load the immediate value $\langle \textit{datum} \rangle$ to the Top of Stack.

`Load` $\langle \textit{addr} \rangle$ `Load 200`

SP $\leftarrow$ SP $+ 1$; [SP] $\leftarrow$ [$\langle \textit{addr} \rangle$]

"Load from absolute address". Increment the Stack Pointer. Load the contents of memory word $\langle \textit{addr} \rangle$ to the Top of Stack.

`Load FP`$+\langle \textit{offset} \rangle$ `Load FP-4`

SP $\leftarrow$ SP $+ 1$; [SP] $\leftarrow$ [FP $+$ $\langle \textit{offset} \rangle$]

"Load FP relative." Increment the Stack Pointer. Form a memory address by adding $\langle \textit{offset} \rangle$ to the value of the Frame Pointer. Load the contents of memory at this address to the Top of Stack.

`Load [SP]`$+\langle \textit{offset} \rangle$ `Load [SP]+2`

[SP] $\leftarrow$ [[SP] $+$ $\langle \textit{offset} \rangle$]

"Load SP indirect relative". Form a memory address by adding $\langle \textit{offset} \rangle$ to the value on the Top of Stack. Replace the Top of Stack contents with the contents of memory at this address. Do not change the Stack Pointer.

`Store` $\langle \textit{addr} \rangle$ `Store 200`

[$\langle \textit{addr} \rangle$] $\leftarrow$ [SP]; SP $\leftarrow$ SP $- 1$

"Store to absolute address". Store the Top of Stack value at $\langle \textit{addr} \rangle$. Decrement the Stack Pointer.

`Store FP`$+\langle \textit{offset} \rangle$ `Store FP+3`

[FP $+$ $\langle \textit{offset} \rangle$] $\leftarrow$ [SP]; SP $\leftarrow$ SP $- 1$

"Store FP relative". Form a memory address by adding $\langle \textit{offset} \rangle$ to the value of the Frame Pointer. Store the value at the Top of Stack to this address. Decrement the Stack Pointer.

`Store [SP]`$+\langle \textit{offset} \rangle$ `Store [SP]+2`

[[SP] $+$ $\langle \textit{offset} \rangle$] $\leftarrow$ [SP $- 1$]; SP $\leftarrow$ SP $- 2$

"Store SP indirect relative". Form a memory address by adding $\langle \textit{offset} \rangle$ to the value on the Top of Stack. Store the value at SP $- 1$ at this address, then adjust the Stack Pointer to pop the top two stack elements.

`Read` `Read`

SP $\leftarrow$ SP $+ 1$; [SP] $\leftarrow$ $\langle \textit{integer read from input} \rangle$

"Read from Keyboard". Read an integer from the keyboard and push it onto the stack.

`Write` `Write`

$\langle \textit{screen} \rangle$ $\leftarrow$ [SP]; SP $\leftarrow$ SP $- 1$;

"Write to Screen". Display the integer on the top of the stack to the screen and pop the stack,

`Halt` `Halt`

"Halt Execution". Stop executing instructions.

The grammar for the compiler project language

⟨*Program*⟩	:==	“**PROGRAM**” ⟨*Identifier*⟩ “**;**” [⟨*Declarations*⟩] { ⟨*ProcDeclaration*⟩ } ⟨*Block*⟩ “**.**”
⟨*Declarations*⟩	:==	“**VAR**” ⟨*Variable*⟩ { “**,**” ⟨*Variable*⟩ } “**;**”
⟨*ProcDeclaration*⟩	:==	“**PROCEDURE**” ⟨*Identifier*⟩ [⟨*ParameterList*⟩] “**;**” [⟨*Declarations*⟩] { ⟨*ProcDeclaration*⟩ } ⟨*Block*⟩ “**;**”
⟨*ParameterList*⟩	:==	“**(**” ⟨*FormalParameter*⟩ { “**,**” ⟨*FormalParameter*⟩ } “**)**”
⟨*FormalParameter*⟩	:==	[“**REF**”] ⟨*Variable*⟩
⟨*Block*⟩	:==	“**BEGIN**” { ⟨*Statement*⟩ “**;**” } “**END**”
⟨*Statement*⟩	:==	⟨*SimpleStatement*⟩ | ⟨*WhileStatement*⟩ | ⟨*IfStatement*⟩ | ⟨*ReadStatement*⟩ | ⟨*WriteStatement*⟩
⟨*SimpleStatement*⟩	:==	⟨*Variable*⟩ ⟨*RestOfStatement*⟩
⟨*RestOfStatement*⟩	:==	⟨*ProcCallList*⟩ | ⟨*Assignment*⟩ | ϵ
⟨*ProcCallList*⟩	:==	“**(**” ⟨*ActualParameter*⟩ { “**,**” ⟨*ActualParameter*⟩ } “**)**”
⟨*Assignment*⟩	:==	“:=” ⟨*Expression*⟩
⟨*ActualParameter*⟩	:==	⟨*Variable*⟩ | ⟨*Expression*⟩
⟨*WhileStatement*⟩	:==	“**WHILE**” ⟨*BooleanExpression*⟩ “**DO**” ⟨*Block*⟩
⟨*IfStatement*⟩	:==	“**IF**” ⟨*BooleanExpression*⟩ “**THEN**” ⟨*Block*⟩ [“**ELSE**” ⟨*Block*⟩]
⟨*ReadStatement*⟩	:==	“**READ**” ⟨*ProcCallList*⟩
⟨*WriteStatement*⟩	:==	“**WRITE**” ⟨*ProcCallList*⟩
⟨*Expression*⟩	:==	⟨*CompoundTerm*⟩ { ⟨*AddOp*⟩ ⟨*CompoundTerm*⟩ }
⟨*CompoundTerm*⟩	:==	⟨*Term*⟩ { ⟨*MultOp*⟩ ⟨*Term*⟩ }
⟨*Term*⟩	:==	[“−”] ⟨*SubTerm*⟩
⟨*SubTerm*⟩	:==	⟨*Variable*⟩ | ⟨*IntConst*⟩ | “**(**” ⟨*Expression*⟩ “**)**”
⟨*BooleanExpression*⟩	:==	⟨*Expression*⟩ ⟨*RelOp*⟩ ⟨*Expression*⟩
⟨*AddOp*⟩	:==	“+” | “−”
⟨*MultOp*⟩	:==	“∗” | “/”
⟨*RelOp*⟩	:==	“=” | “<=” | “>=” | “<” | “>”
⟨*Variable*⟩	:==	⟨*Identifier*⟩
⟨*IntConst*⟩	:==	⟨*Digit*⟩ { ⟨*Digit*⟩ }
⟨*Identifier*⟩	:==	⟨*Alpha*⟩ { ⟨*AlphaNum*⟩ }
⟨*AlphaNum*⟩	:==	⟨*Alpha*⟩ | ⟨*Digit*⟩
⟨*Alpha*⟩	:==	“**A**” . . . “**Z**” | “**a**” . . . “**z**”
⟨*Digit*⟩	:==	“**0**” . . . “**9**”