**UNIVERSITY OF LIMERICK**
**OLLSCOIL LUIMNIGH**

**COLLEGE OF INFORMATICS & ELECTRONICS**

**DEPARTMENT OF ELECTRONIC & COMPUTER ENGINEERING**

| | |
|---|---|
| **MODULE CODE:** | **CE4717** |
| **MODULE TITLE:** | **Language Processors** |
| **SEMESTER:** | **Autumn 2003** |
| **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.**

**The following information is supplied with the paper:**

- **A grammar for the language of the term project;**
- **The instruction set of the stack machine used as a target in the term project, and;**
- **A description and partial instruction set for a RISC-style machine architecture.**

**In answering Question 3, please refer to the RISC machine description. For any other question requiring assembly code it is expected that you will employ the stack machine instruction set.**

---

Q1. (a) This is code for an `Accept` routine without any error recovery.

```
PRIVATE void Accept( int ExpectedToken )
{
    if ( CurrentToken.code == ExpectedToken )
        CurrentToken = GetToken();
    else  {
        SyntaxError( CurrentToken, ExpectedToken );
        exit( EXIT_FAILURE );
    }
}
```

Modify it so that it incorporates S-Algol error recovery. [5 Marks]

(b) The following program, based on the grammar for the compiler project, contains a syntax error.

```
PROGRAM test;
VAR i;
BEGIN DO
    i := 2;
    WRITE( i );
END.
```

i. Identify the error. [2 Marks]

ii. Show clearly what would be "inserted" in the source by the S-Algol error recovery scheme, what part (if any) of the input skipped, and where re-synchronisation would take place. [5 Marks]

iii. Write code for a routine to parse a $\langle \textit{Block} \rangle$ using augmented S-Algol error recovery. Include all code unique to the augmented S-Algol scheme and any synchronising sets required in this case. [7 Marks]

iv. At what point in the input does the augmented scheme re-synchronise? [1 Mark]

Q2. It is desired to add *functions* to the Compiler Project Language, so that code like the following may be written.

```
PROGRAM combinations;
    int nCr;

    FUNCTION factorial(n):
        VAR i;
    BEGIN
        factorial := 1;
        i := 2;
        WHILE ( i <= n ) DO BEGIN
            factorial := factorial * i;
            i := i + 1;
        END;
    END;

BEGIN
    nCr := factorial(6) / (factorial(2) * factorial(4));
    WRITE(nCr);
END.
```

i. Design a run-time storage layout which allows functions to be included in the language. Show this layout for the case of a call `factorial(5)`. Be sure to explain how the result of the function call is returned to the calling environment.

[10 Marks]

ii. Modify the Compiler Project Language grammar to allow functions to be compiled. Is any semantic processing needed to disambiguate the new grammar? Discuss.

[10 Marks]

Q3. (a) The following are a portion of C source code and the assembly language corresponding to it generated by a highly optimising compiler.

```
#define PI 3.1415926
#define SCALE 100.0

double  alpha[100][4];
        ⋮
for (i = 0; i < 100; i++)
    for (j = 0; j < 4; j++)
        a[i][j] = 2.0 * PI * SCALE * (i + 1);
```

```
1000:    xor.w   r4 ← r4, r4
1004:    ldi.d   f0 ← 628.31853
1016:    shl.w   r5 ← r4, 3
1020:    inc.w   r4
1024:    cvtid   f2 ← r4
1028:    mul.d   f4 ← f0, f2
1032:    st.d    8000(r5) ← f4
1036:    st.d    8008(r5) ← f4
1040:    st.d    8016(r5) ← f4
1044:    st.d    8024(r5) ← f4
1048:    cmpi.w  r4, 100
1052:    bne     1016
```

Identify any instances of optimisations in the assembly language, name them, and discuss why they are appropriate. [15 Marks]

(b) i. The compiler project grammar is not LL(1). One EBNF production violates the LL(1) criterion. Which one? Explain why this parsing conflict arises. [2 Marks]

ii. How does the compiler resolve the conflict? [3 Marks]

Q4. Consider the following grammar.

$$\begin{aligned}
\langle \textit{Expression} \rangle &::== \langle \textit{Factor} \rangle \ \langle \textit{RestOfExpr} \rangle \\
\langle \textit{RestOfExpr} \rangle &::== \langle \textit{AddOp} \rangle \ \langle \textit{Factor} \rangle \ \langle \textit{RestOfExpr} \rangle \ | \ \epsilon \\
\langle \textit{Factor} \rangle &::== \langle \textit{Term} \rangle \ \langle \textit{RestOfFact} \rangle \\
\langle \textit{RestOfFact} \rangle &::== \langle \textit{MultOp} \rangle \ \langle \textit{Term} \rangle \ \langle \textit{RestOfFact} \rangle \ | \ \epsilon \\
\langle \textit{Term} \rangle &::== \langle \textit{UnaryMinus} \rangle \ \langle \textit{SubTerm} \rangle \\
\langle \textit{SubTerm} \rangle &::== \text{“(”} \langle \textit{Expression} \rangle \text{“)”} \ | \ \text{“n”} \\
\langle \textit{AddOp} \rangle &::== \text{“+”} \ | \ \text{“-”} \\
\langle \textit{MultOp} \rangle &::== \text{“*”} \ | \ \text{“/”} \\
\langle \textit{UnaryMinus} \rangle &::== \text{“-”} \ | \ \epsilon
\end{aligned}$$

i. Prove that this grammar is LL(1). [10 Marks]

ii. Derive the LL(1) parse table for this grammar. [6 Marks]

iii. Show the actions of an LL(1) table-driven parser for the grammar on the single token string “n”. [4 Marks]

Q5. i. Use Thompson’s construction to convert the regular expression

$$(a|b)^*a(a|b|\epsilon)$$

into an NFA. Be sure to show the steps involved in the construction of the final, composite NFA from the initial, primitive NFA’s. [7 Marks]

ii. Convert the NFA of part (i) into a DFA using the subset construction. [10 Marks]

iii. Show the action of the DFA on the 8 character input string “abbaaaab”. [2 Marks]

iv. Show the action of the DFA on the 3 character input string “abb”. [1 Mark]

# 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` $\langle words \rangle$ `Dec 4`

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

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

`Push FP` `Push FP`

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

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

`Load #`$\langle datum \rangle$ `Load #-200`

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

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

`Load` $\langle addr \rangle$ `Load 200`

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

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

`Load FP`$+\langle offset \rangle$ `Load FP-4`

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

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

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

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

"Load SP indirect relative". Form a memory address by adding $\langle offset \rangle$ 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` $\langle addr \rangle$ `Store 200`

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

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

`Store FP`$+\langle offset \rangle$ `Store FP+3`

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

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

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

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

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

`Read` `Read`

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

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

`Write` `Write`

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

"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*⟩ | $\epsilon$

⟨*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**”

**RISC Machine Opcodes and Architecure**

The machine referred to in Question 3 is a high-performance Reduced Instruction Set Computer (RISC) with the following important characteristics:

- It is a load-store architecture which performs all arithmetic and logical operations in its CPU registers.
- It has two sets of CPU registers, 32 integer registers, with mnemonic names `r0` ... `r31`, and 16 double-precision floating-point registers, with mnemonic names `f0` ... `f15`.
- It is byte-addressed.
- C `int`s occupy 4 bytes (32 bits).
- C `double`s occupy 8 bytes (64 bits).

The following table gives a subset of the opcodes defined for this machine:

| | |
|---|---|
| `xor.w` | Perform exclusive-or operation on two integers. |
| `ldi.w` | Load integer constant into integer register. |
| `ldi.d` | Load double constant into double-precision floating-point register. |
| `shl.w` | Left-shift integer. |
| `inc.w` | Increment contents of integer register. |
| `cvtid` | Convert an integer value to a double. |
| `mul.d` | Double-precision multiplication. |
| `st.d` | Store the contents of a double-precision register to memory. |
| `cmpi.w` | Compare the contents of an integer register to an integer constant and set the CPU flags. |
| `bne` | Branch if the $Z$ flag is clear. |

In the assembly language code the *target* register of an operation appears on the left hand side of the "assignment arrow" (←). The source operand(s) appear on its right hand side. For example:

```
2804:    xor.w   r30 ← r7, r9
```

Here the contents of register `r7` are xor'ed with those of `r9`, and the result is written to `r30`. `2804` is the address in memory of the instruction.
Some instructions have a single register which is both source and target, e.g.:

```
2900:    inc.w   r30
```

This instruction adds 1 to the current contents of `r30` and writes the new value back to that register.