UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Faculty of Engineering, Built Environment and Information Technology
Department of Computer Science
COS341 *CompilerConstruction*
Exam Opportunity 2

17th of June 2022

Student: *Full Name:* Ryan Healy

Student: ***Number:*** 20667307

EXAMINERS

- Internal Examiner: Prof. *S. Gruner*,
- External Examiner: Prof.em. *D. Kourie*, University of Stellenbosch.

INSTRUCTIONS

- *Any electronic devices* (cell phones, laptop computers, tablet computers, pocket calculators, etc.) are *strictly forbidden.*
- Paper-based auxiliary materials (printed books and/or hand-scribbled crib notes) are *allowed up to a maximum weight* of 20 kilogram per student.
- The answers *must* be written in *indelible ink* into the **separate answer-booklet** which is provided together with this question-paper. Any answers *written with pencil (or other types of erasable ink) will not be marked* (=null points).
- You have 3 hours work-time to complete this exam: any extra time is *only* for those students who present a qualifying letter issued by the university's office for the disabled students.
- All in all there are *four Questions* with a total value of 40 Points.

- **Wait until the invigilator gives you permission** to start working!
- *Read* each question very carefully and thoroughly before you attempt to answer it.
- The invigilators in the exam room are *not allowed to provide any hints* which could possibly lead to the solution of a question that the student is supposed to answer.
- **Return this question-paper** (with the **marking-grid displayed below**) *together* with your answer-booklet.
- A perusal opportunity will be provided in due course, after all the papers have been marked.

MARKING

Question	Q1	Q2	Q3	Q4	Sum
Maximum	12	10	8	10	40
Result	10	10	7	8,5	35,5

Turn the page →

Question 1 .... **[12 Points]**

A type checker is based on a *recursive Boolean function* that "works" on the syntax tree of an input program that has already been parsed. After several recursions, the Boolean function eventually returns <u>true</u> if the input program, that corresponds to the syntax tree, is correctly typed – otherwise the recursive Boolean function returns <u>false</u>. To enable a compiler-engineer to implement such a tree-"crawling" recursive Boolean function already the syntactic rules of the underlying context-free grammar must be *annotated* with additional rules that carry semantic information by means of which the implementation of the type-checker can be guided.

Given is now the following small context-free grammar, however still without any annotations. It is your task to provide a Boolean-typed type-checking rule to *each* of the given grammar's syntactic rules. (<u>Hint</u>: *do not confuse* the Boolean return-type *of* the recursive type-checking function itself with the Boolean types *within* an input-program that is to be type-checked!)

1. PROG ::= SEQ
2. SEQ ::= INSTR ; SEQ
3. SEQ ::= ε *// epsilon, nothing*
4. INSTR::= ITE
5. INSTR::= ASGN
6. ITE ::= **if** (EXPR) **then** { SEQ$_1$ } **else** { SEQ$_2$ }
7. ASGN ::= VAR = EXPR
8. VAR ::= NVAR *// for numbers*
9. VAR ::= BVAR *// for Booleans*
10. EXPR ::= VAR$_1$ + VAR$_2$ *// numeric addition*
11. EXPR ::= VAR$_1$ ≤ VAR$_2$ *// smaller or equal*
12. EXPR ::= VAR$_1$ **v** VAR$_2$ *// logical disjunction*

Thereby, your annotations shall be structured as follows: use ***tc*** as the name of the **t**ype **c**hecker function, and let *NT* be any *Non-Terminal* symbol which ***tc*** takes as its input parameter, such that you can define ***tc***(*NT*) = ... for each *Non-Terminal* symbol which the given grammar contains. Since the grammar has twelve syntactic rules, you must provide twelve semantic annotations accordingly. To indicate the **t**ypes **o**f the syntactic entities themselves, use the notation ***to***(*NT*), such that -for example- ***to***(VAR) denotes the type of a variable, or ***to***(EXPR) denotes the type of an expression: The non-recursive ***to***(...) is thus used as an internal auxiliary means by the recursive ***tc***(...)

Question 2 .... **[10 Points]**

The well-known *Euclidean algorithm* for non-negative integer variables is given as follows:

```
if (a == 0) then { r = b ; }
             else {
                  while (b != 0) do
                        {
                          if (a > b) then { a = a-b ;}
                                     else { b = b-a ;}
                        }
                  r = a ;
                  }
```

Apply the *translation method of Chapter 6* (of our COS341 textbook) to accurately translate this algorithm into intermediate code, whereby the resulting intermediate code may *not* contain any "else"; (in other words: all "else"-branches *must* be represented by GOTO-jumps).

Question 3 .... [8 Points]

The Euclidean algorithm of the previous questions can also be embedded into a *function* which has the following form:

```
int Euc(int a, int b) // name and parameters
     {
      localV r;    // declaration
      algorithm_from_Question_2 // euc_code_2
      return r;
     }
```

Assume now that enough CPU registers are available (no "spilling" into main memory), and also assume a "*Callee-Save*" *strategy* according to Section 9.4 of our COS341 textbook. On this basis, *apply the methods of Chapter 9* to generate code (call-sequences *with prologue and epilogue*) for the given **Euc** function, whereby you need ***not*** again provide the previously generated algorithm code from Question 2: For the sake of brevity, you simply *represent* all code from Question 2 by the single-word pseudo-command "**euc_code_2**" here in Question 3. In other words: the focus of this question is the generation of the "boiler plate code" that is needed to enable correct function calls at run-time. The function's inner local variable declaration, which "does" nothing, you will translate as described in Chapter 6 (Fig. 6.12: "**id**").

Question 4 ....

In geometry, for any proper triangle with edges a,b,c, the well-known *triangle inequalities* state that $a<b+c$ and $b<a+c$ and $c<a+b$. The following program, which is already in the form of *intermediate code*, tests whether the values of three given variables (which are already presumed to be positive) a,b,c fulfil the above-mentioned triangle inequality; the program also uses an auxiliary 4th variable d.

```
1.  d := b+c
2.  IF a<d GOTO 4      // comment: then-case
3.  GOTO 9             // comment: else-case
4.  d := a+c
5.  IF b<d GOTO 7
6.  GOTO 9
7.  d := a+b
8.  IF c<d GOTO 11
9.  d := 0             // comment: false, no triangle
10. GOTO 12
11. d := 1             // comment: true, it is a triangle
12. RETURN d           // comment: output the result: end of program.
```

In the following sub-questions of this Question 4, you will be asked to *find out how many registers will be needed* if we hope to keep all of the given program's variables in the registers (without the need to "spill" any of them into the main memory).

4a .... [3 Points]

Apply the methods from Textbook Chapter 8 to provide the |succ[*i*]|gen[*i*]|kill[*i*]|-Table of the given program.

4b .... [3 Points]

Apply the methods from Textbook Chapter 8 to provide the |**out**[*i*]|**in**[*i*]|-Table of the given program.

Turn the page →

4c .... **[2 Points]**
Apply the methods from Textbook Chapter 8 to provide the Interference-Table of the given program.

4d .... **[2 Points]**
Draw the interference graph, provide its nodes with the *best possible* "colouring" (as *few* colours as possible), and *indicate how many registers will be needed* for the given program if *no* "spilling" shall occur. (Since you only have your ink pen in this exam, *write* the "colours" as WORDS with capital letters, for example: RED, BLUE, etc.)

There are no further questions.

This blank space is for rough-scribbles that will NOT be marked.

This blank space is for rough-scribbles that will NOT be marked.

10 1

~~tc(NT) = case PROG of~~

~~SEQ return tc~~

~~tc(SEQ) = case SEQ of~~

~~tc(NT) = case NT of~~

~~PROG~~

tc(PROG) = case PROG of

SEQ	return tc(SEQ)

tc(SEQ) = case SEQ of

ε	return true
INSTR; SEQ	t_1 = tc(INSTR) t_2 = tc(SEQ) if t_1 and t_2 then return true else return false

tc(INSTR) = case INSTR of

ITE	return tc(ITE)
ASGN	return tc(ASGN)

tc(ITE) = case ITE of

if(EXPR) then{SEQ_1} else{SEQ_2}	t_1 = to(EXPR) t_2 = tc(SEQ_1) t_3 = tc(SEQ_2) if t_1 = Boolean and t_2 and t_3 then return true else return false

tc(ASGN) = case ASGN of

VAR = EXPR	t_1 = to(VAR) t_2 = to(EXPR) if $t_1 = t_2$ then return true else return false

to(VAR) = case VAR of

NVAR	return number
BVAR	return Boolean

For EXPR you now need to, not tc. That is because you had correctly called to(EXPR) in the ITE-rule and in the ASGN-rule of above

tc(EXPR) = case EXPR of

$VAR_1 + VAR_2$	t_1 = to(VAR_1) t_2 = to(VAR_2) if t_1 = number and t_2 = number then return true else return false
$VAR_1 \leq VAR_2$	t_1 = to(VAR_1) t_2 = to(VAR_2) if t_1 = number and t_2 = number then return true else return false
$VAR_1 \vee VAR_2$	t_1 = to(VAR_1) t_2 = to(VAR_2) if $t_1 = t_2$ must be booleans then return true else return false

to(EXPR) = case EXPR of

$VAR_1 + VAR_2$	return number
$VAR_1 \leq VAR_2$	return Boolean
$VAR_1 \vee VAR_2$	return Boolean.

seen

(2) Labels will be given as $L_0, L_1, \ldots, L_m$

Variable names from the algorithm will remain the same in the following intermediate code. Temporary variables will be given as $t_0, t_1, \ldots, t_n$

Translating line 1: [Cond → a==0 $Stat_1$ → r=b, $Stat_2$ → lines 3-7]

$label_1 = l_0$ true

$label_2 = l_1$ false

$label_3 = l_2$ end.

if - then t
label f
—
goto end
label t
end

$code_1 = Trans_{Cond}(Cond, label_1, label_2, vtable, ftable)$

$code_2 = trans_{Stat}(Stat_1, vtable, ftable)$ true

$code_3 = trans_{Stat}(Stat_2, vtable, ftable)$ false

→ $code_1$ ++ [LABEL l_1] ++ $code_3$ ++ [GOTO l_2] ++ [LABEL l_0] ++ $code_2$ ++ [LABEL l_2]

~~Transcond Cond;~~

Generating $Code_1$: [Exp_1 → a Exp_2 → 0]

$Code_4 = TransExp(Exp_1, vtable, ftable, t_1)$

$Code_5 = TransExp(Exp_2, vtable, ftable, t_2)$

→ $code_4$ ++ $code_5$ ++ [IF $t_1 == t_2$ THEN l_0]

Generating $Code_4$: [place → t_1]

x = lookup(vtable, getname(a)) = a

→ [t_1 := a]

Generating $Code_5$: [place → t_2]

v = getvalue(0) = 0

→ [t_2 := 0]

Generating $code_5$:

$place = newvar = t_3$

$x = lookup(vtable, getname(r)) = r$

$code_6 = Trans_{Exp}(b, vtable, ftable, place)$

$\rightarrow code_6 ++ [r := t_3]$

Generating $code_6$: [$place \rightarrow t_3$] m

$x = lookup(vtable, getname(b)) = b$

$\rightarrow [t_3 = b]$

Currently, the intermediate code is as follows:

$t_1 := a$

$t_2 := 0$

IF $t_1 == t_2$ THEN l_0

LABEL l_1

[$code_3$]

GOTO l_2

LABEL l_0

$t_3 = b$

$r := t_3$

LABEL l_2

Generating $code_3$: [$cond \rightarrow b != 0$ $stat_1 \rightarrow$ lines 5-6 $cond \rightarrow b != 0$]

$label_1 = l_3$

$label_2 = l_4$

$label_3 = l_5$

$code_7 = transcond(cond, label_2, label_3, vtable, ftable)$

$code_8 = Trans_{Stat}(stat_1, vtable, ftable)$

$\rightarrow$ [LABEL l_3] ++ $code_7$ ++ [LABEL l_4] ++ $code_8$ ++ [GOTO ~~label~~ l_3] ++ [LABEL l_5]

NOTE: Branch switching has been performed

Generating code$_7$: [Exp1 => b Exp2 -> 0]

code$_9$ = transExp(b, vtab, ftab, t_4)

code$_{10}$ = transExp(0, vtab, ftab, t_5)

~~[IF t_4 == t_5~~

-> code$_9$ + code$_{10}$ + [IF t_4 == t_5 THEN L_5] ~~[LABEL~~

Gen code$_9$:

x = lookup(vtable, getname(b)) = b

-> [t_4 := b]

Gen code$_{10}$:

v = getvalue(0) = 0

-> [t_5 := 0]

Gen code 8:

label$_1$ = L_6 true

label$_2$ = L_7 false

label$_3$ = L_8 end

code$_{11}$ = transCond(a>b)

code$_{12}$ = transStat (a = a-b) true

code$_{13}$ = transStat (b = b-a) false

-> code 11 ++ [LABEL L_7] ++ code$_{13}$ ++ [GOTO L_8] ++ [LABEL L_6]

++ code 12 ++ [LABEL L_8]

Gen code 11:

code$_{14}$ = transExp(a) = [t_6 := a]

code$_{15}$ = transExp(b) = [t_7 := b]

-> code 14 ++ code 15 ++ [IF t_6 < t_7 THEN L_6]

Gen $code_{12}$: $[a = a - b]$

place $= t_8$

$x = lookup_p(vtab, getname(a)) = a$

$code16 = transexp_p(a-b, t_8)$

$\rightarrow code_{16}$ ++ $[a = t_8]$

gen code 16:

$code_{17} = [t_9 := a]$

$code_{18} = [t_{10} := b]$

$\rightarrow code\ 17$ ++ $code\ 18$ ++ $[t_8 := t_9 - t_{10}]$

Gen $code_{13}$: $[b = b - a]$

$x = lookup_p(vtab, getname(b)) = b$

$code_{19} = transexp(b-a, t_{11})$

$\rightarrow code_{19}$ ++ $[b = t_{11}]$

Gen code 19

$code_{20} = [t_{12} := b]$

$code_{21} = [t_{13} := a]$

$\rightarrow code_{20}$ ++ $code_{21}$ ++ $[t_{11} = t_{12} - t_{13}]$

Gen code 22: $[r = a]$

$x = lookup(vtab, getname(r)) = r$

$code_{23} = [t_{14} = a]$

$\rightarrow code_{23}$ ++ $[r = t_{14}]$

The final intermediate code is as follows:

$t_1 := a$

$t_2 := 0$

IF $t_1 == t_2$ THEN L_0

LABEL L_1

LABEL L_2

$t_4 := b$

$t_5 := 0$

IF $t_4 == t_5$ THEN L_5

LABEL L_4

$t_6 := a$

$t_7 := b$

IF $t_6 < t_7$ THEN L_6

LABEL L_7

$t_{12} := b$

$t_{13} := a$

$t_{11} = t_{12} - t_{13}$

$b = t_{11}$

GOTO L_8

LABEL L_6

$t_9 := a$

$t_{10} := b$

$t_8 = t_9 - t_{10}$

$a = t_8$

LABEL L_8

GOTO L_3

LABEL L_5

$t_{14} = a$

$r = t_{14}$

GOTO L_2

LABEL L_0

$t_3 := b$

$r := t_3$

LABEL L_2

10

8,5

(4) 3 a)

i	succ[i]	gen[i]	kill[i]
1	2	b, c	d
2	3, 4	a, d	
3	9		
4	5	a, c	d
5	6, 7	b, d	
6	9	.	
7	8	a, b	d
8	9, 11	c, d	
9	10		d
10	12		
11	12		d
12		d	

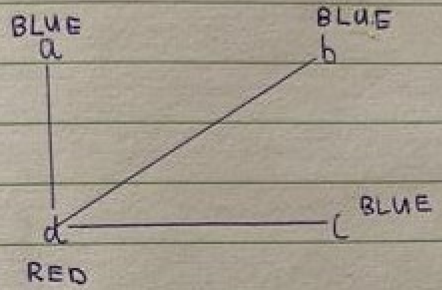
3 b)

i	iteration1 out[i]	iteration1 in[i]	iteration2 out[i]	iteration2 in[i]
1	a, d, c, b	b, c, a,	a, d, c, b	b, c, a
2	a, c, b	a, d, c, b	a, c, b	a, d, c, b
3	.	.	.	.
4	b, d, a, c	a, c, b	b, d, a, c	a, c, b
5	a, b, c	b, d, a, c	a, b, c	b, d, a, c
6	. .	.	.	.
7	c, d	a, b, c	c, d	a, b, c
8	.	c, d	.	c, d
9	d	.	d	.
10	d	d	d	d
11	d	.	d	.
12	.	d	.	d

0.5c) Instruction | LHS | Interferes with

Instruction	LHS	Interferes with
1	d	b, c ✗
4	d	a, c ✗
7	d	a, b ✗

2 d)



③ LABEL Euc

$SP := SP - framesize_{Euc} - 4*(k+1)$

$M[SP + framesize_{Euc}] := R0$

...

$M[SP + framesize_{Euc} + 4*k] := R\boxed{k}$

How large is K in our ~~example~~ example?

Where is the beginning and the ending of Prologue, Body, Epilogue?

$a := M[SP + framesize_{Euc} + 4*(k+1)]$

$b := M[SP + framesize_{Euc} + 4*(k+2)]$

$\left.\begin{array}{l} t_1 = newvar() \\ vtable_1 = bind(vtable, getname(r)) \end{array}\right\}$ this is invisible in the finally generated code!

euc_code_2

$M[SP + framesize_{Euc} + 4*(k+1)] := t_1$

$R0 := M[SP + framesize_{Euc}]$

...

$R\boxed{k} := M[SP + framesize_{Euc} + 4*k]$

How large is K in our example?

$SP := SP + framesize_{Euc} + 4*(k+1)$

GOTO M[SP]