Compiler Theory CS447b: Final exam. $\boxed{\textbf{3 hours.}}$ **UWO, April 13, 2004.**

Student ID number:

Guidelines. The exam is closed book and all notes are forbidden. However some formulas from the course are given in annexes. There are 10 exercises which have the following allocation of marks 4, 6, 2, 6, 2, 4, 4, 2, 4, 6, respectively. The total is 40. All answers should be written in the *answer boxes*. No justifications for the answers are needed **unless** explicitly required. You are expected to do this exam on your own without assistance from anyone else in the class. If possible, please avoid pencils and use pens with dark ink. Thank you.

Exercise 1. We consider the alphabet $\Sigma = \{a, b\}$. For each of the following languages over Σ give an automaton (deterministic or not) that recognizes this language.

1. L_1 is the language consisting of all words w over Σ such that w is not the empty word and none of the words aa, bb is a factor of w. **Hint.** It would be useful for the remaining questions to build a deterministic automaton of L_1.
2. $\overline{L_1}$ is the language consisting of all words w over Σ such that w is the empty word or one of the words aa, bb is a factor of w.
3. L_{12} is the language consisting of all words w over Σ such that w is not the empty word and no words of the form a^n, b^n for $n \geq 3$ is a factor of w.
4. L_2 is the language consisting of all words w over Σ that belong to L_{12} but not to L_1.

Answer 1

Exercise 2. Consider the following grammar G with terminals a, b, c, d and non-terminals S, A, B, C.

$$\begin{array}{lcl} S & \longmapsto & C\,A \\ A & \longmapsto & a\,B\,A \;\mid\; \varepsilon \\ B & \longmapsto & b\,B \;\mid\; C\,d \\ C & \longmapsto & c\,C \;\mid\; \varepsilon \end{array}$$

We recall the rules for the computation of the FOLLOW set of each nonterminal.

$$\begin{array}{lcll} \text{if} & A \longmapsto \alpha B \beta & \text{then} & \text{FOLLOW}(B) := \text{FIRST}(\beta) \setminus \{\varepsilon\} \cup \text{FOLLOW}(B) \\ \text{if} & A \longmapsto \alpha B & \text{then} & \text{FOLLOW}(B) := \text{FOLLOW}(B) \cup \text{FOLLOW}(A) \\ \text{if} & \left\{ \begin{array}{l} A \longmapsto \alpha B \beta \\ \beta \stackrel{*}{\Longrightarrow} \varepsilon \end{array} \right. & \text{then} & \text{FOLLOW}(B) := \text{FOLLOW}(B) \cup \text{FOLLOW}(A) \end{array}$$

Recall also that if \$ denotes the end-of-line symbol and if S is the start symbol of the grammar (as it is the case above) then we have $\$ \in \text{FOLLOW}(S)$. Give the parsing table of G (for the non-recursive implementation of predictive parsing).

Answer 2

Exercise 3. Consider the following syntax-directed definition.

Production	Semantic Rule
$S \longmapsto E$	$E.env$:= $\mathsf{initialEnv}()$
$E_1 \longmapsto$ **let id** $= E_2$ **in** E_3 **end**	$E_2.env$:= $E_1.env$
	$E_3.env$:= $\mathsf{Update}(E_1.env, \mathbf{id}, E_2.val)$
	$E_1.val$:= $E_3.val$
$E_1 \longmapsto (E_2 + E_3)$	$E_1.val$:= $E_2.val + E_3.val$
	$E_2.env$:= $E_1.env$
	$E_3.env$:= $E_1.env$
$E \longmapsto \mathbf{id}$	$E.val$:= $\mathsf{LookUp}(\mathbf{id}, E.env)$
$E \longmapsto \mathbf{1}$	$E.val$:= $\mathbf{1}$
$E \longmapsto \mathbf{0}$	$E.val$:= $\mathbf{0}$

Write a translation scheme for this syntax-directed definition.

Answer 3

$S \quad \longmapsto$
$E_1 \quad \longmapsto$
$E_1 \quad \longmapsto$
$E \quad \longmapsto$
$E \quad \longmapsto$
$E \quad \longmapsto$

Exercise 4. The grammar below generates sequences of declarations of the form $\mathbf{id} : T$.

$$\begin{array}{lcl} P & \longmapsto & D \\ D & \longmapsto & D \, ; D \\ D & \longmapsto & \mathbf{id} : T \\ T & \longmapsto & \mathbf{integer} \\ T & \longmapsto & \mathbf{float} \\ T & \longmapsto & \mathbf{array} \, [\, \mathbf{num} \,] \, \mathbf{of} \, T_1 \\ T & \longmapsto & \uparrow T_1 \end{array}$$

As usual **id** is the token for identifiers, **integer** denotes a type of integers, **float** denotes a type of floating point numbers and **num** is the token for integer values. Note that **array**, $\uparrow$, **;** , **:**, **[**, **]** and **of** are terminals too. Moreover,

- $\mathbf{array} \, [\, \mathbf{num} \,] \, \mathbf{of} \, T$ denotes the type of arrays of length $\mathbf{num}.val$ with elements of type $T.type$.
- $\uparrow T$ denotes the type of pointers to elements of type $T.type$.

You are asked to write a **translation scheme** which computes the memory address of every declared name and stores this information in the symbol table. To reach that goal you may associate the nonterminal T with a synthesized attribute $T.width$ that indicates the number of memory units taken by objects of type T.

- For $T \longmapsto \mathbf{integer}$ and $T \longmapsto \mathbf{float}$ it is usual to have $T.width = 4$ and $T.width = 8$ respectively.
- For $T \longmapsto \mathbf{array} \, [\, \mathbf{num} \,] \, \mathbf{of} \, T$ the formula for $T.width$ is easy to guess.
- For $T \longmapsto \uparrow T_1$ remember that a pointer is an integer.

As usual, the nonterminal T has a synthesized attribute $T.type$. You are **not** asked to give the semantic rules computing $T.type$ (although this is easy). You may need a global variable $offset$ which gives the current available memory address:

- Before the first declaration is made, $offset$ should be initialized to 0.
- As each name $\mathbf{id}.name$ is seen, that name is entered in the symbol table with $offset$ equal to the current value and $offset$ is incremented by the width of the type of $\mathbf{id}.name$.
- The procedure call enter($\mathbf{id}.name, T.type, offset$) creates a symbol-table entry for $\mathbf{id}.name$, with type $T.type$ and memory address $offset$.

Answer 4

Exercise 5. We recall the syntax-directed definition for the translation of flow of control statements into three-address code, using the jump method with symbolic labels. In each of the above productions, E is a boolean expression to be translated. This boolean expression E is associated with two labels (which are inherited attributes)

- $E.true$ the label to which control flows if E is **true**,
- $E.false$ the label to which control flows if E is **false**.

Similarly, S is a flow of control statement associated with 2 or 3 attributes among

- $S.begin$ which is a label attached to the first 3-address code in the translation of S, $S.begin$ is a synthesized attribute,
- $S.next$ which is a label attached to the first 3-address code statement to be executed after the code for S , $S.next$ is an **inherited** attribute,
- $S.code$ is the translation code for S, as usual it is a **synthesized** attribute.

Production	Semantic Rule
$S \longmapsto$ **if** E **then** S_1	$E.true$:= newlabel $E.false$:= $S.next$ $S_1.next$:= $S.next$ $S.code$:= $E.code$ || generate($E.true$ '**:**') || $S_1.code$
$S \longmapsto$ **if** E **then** S_1 **else** S_2	$E.true$:= newlabel $E.false$:= newlabel $S_1.next$:= $S.next$ $S_2.next$:= $S.next$ $code_1$:= $E.code$ || generate($E.true$ '**:**') || $S_1.code$ $code_2$:= generate('**goto**' $S.next$) || $code_3$:= generate($E.false$ '**:**') || $S_2.code$ $S.code$:= $code_1$ || $code_2$ || $code_3$
$S \longmapsto$ **while** E **repeat** S_1	$S.begin$:= newlabel $E.true$:= newlabel $E.false$:= $S.next$ $S_1.next$:= $S.begin$ $code_1$:= generate($S.begin$ '**:**') || $E.code$ $code_2$:= generate($E.true$ '**:**') || $S_1.code$ $code_3$:= generate('**goto**' $S.begin$) $S.code$:= $code_1$ || $code_2$ || $code_3$

Some languages allow **repeat-until** loop statements of the form

```
repeat {
      s1;
}
until (e);
```

where `s1` is a statement and `e` is a boolean expression. (Iterations of the loop body are executed until `e` becomes *true*. Construct a syntax-directed definition for the translation of these **repeat-until** loop statements into three-address code.

Answer 5

Exercise 6. We consider the following grammar. As in the course, the nonterminals P, D, S, T, E stand for *Program*, *Declaration*, *Statement*, *Type*, *Expression*. The terminals **id**, **boolean**, **integer**, **literal**, **num** stand for *identifier*, *boolean* (as a type), *integer* (as a type), *boolean literal* (as a value), *integer number* (as a value).

$$
\boxed{\begin{array}{l}
P \longmapsto D;\ S \\
D \longmapsto D;\ D \\
D \longmapsto \mathbf{id} bf\ : T \\
T \longmapsto \mathbf{boolean} \\
T \longmapsto \mathbf{integer} \\
E \longmapsto \mathbf{literal} \\
E \longmapsto \mathbf{num} \\
E \longmapsto \mathbf{id} \\
E \longmapsto E_1 + E_2 \\
E \longmapsto E_1 = E_2 \\
S \longmapsto S_1;\ S_2 \\
S \longmapsto E \\
S \longmapsto \mathbf{id{:=}}\, S_1 \\
S \longmapsto \mathbf{if}\ E\ \mathbf{then}\ \{S_1\} \\
S \longmapsto \mathbf{while}\ E\ \mathbf{repeat}\ \{S_1\}
\end{array}}
$$

There's one slight difference w.r.t. the grammar of the course in the *Type checking* chapter: each valid statement has a value and thus a type (rather than no value and type `void`). The following rules compute the value of a valid statement.

- if S_1 and S_2 are valid statements then the value of $\boxed{S_1;\ S_2}$ is that of S_2.
- if $\boxed{\mathbf{id{:=}}\, S_1}$ is a valid statement then its value is that of S_1.
- if $\boxed{\mathbf{if}\ E\ \mathbf{then}\ \{S_1\}}$ is a valid statement then its value is that of S_1.
- if $\boxed{\mathbf{while}\ E\ \mathbf{repeat}\ \{S_1\}}$ is a valid statement then its value is that of S_1.

We associate attributes $T.type$, $E.type$, $S.type$, to the grammar symbols T, E, S. Each of these attributes may be **boolean**, **integer** or **type_error**. Then, for instance, the statement $\boxed{\mathbf{while}\ E\ \mathbf{repeat}\ \{S_1\}}$ is valid as soon as $E.type = \mathbf{boolean}$ and $S_1.type \neq \mathbf{type\_error}$. The attribute $\mathbf{id}.entry$ refers to the entry of **id** in the symbol table.

Question. Complete the following type checker for the above grammar.

Answer 6

$D \longmapsto \mathbf{id}\!:T$	{ addtype(**id**.$entry$, $T.type$) }
$T \longmapsto \mathbf{boolean}$	{ $T.type$:= **boolean** }
$T \longmapsto \mathbf{integer}$	{ $T.type$:= **integer** }
$E \longmapsto \mathbf{literal}$	{ $E.type$:= **boolean** }
$E \longmapsto \mathbf{num}$	{ $E.type$:= **integer** }
$E \longmapsto \mathbf{id}$	{ $E.type$:= lookup(**id**.$entry$) }
$E \longmapsto E_1 + E_2$	{ $E.type$:=
$E \longmapsto E_1 = E_2$	{ $E.type$:=
$S \longmapsto S_1;\ S_2$	{ $S.type$:=
$S \longmapsto E$	{ $S.type$:=
$S \longmapsto \mathbf{id}\mathbf{:=}S_1$	{ $S.type$:=
$S \longmapsto \mathbf{if}\ E\ \mathbf{then}\ \{S_1\}$	{ $S.type$:=
$S \longmapsto \mathbf{while}\ E\ \mathbf{repeat}\ \{S_1\}$	{ $S.type$:=

Note that **only** the type of the non-terminals E, T is required, **not their value**.

Exercise 7. We extend the grammar of Exercise 2 in order to handle unary functions with a single returned value. Here's a program showing these new features.

```
f: integer -> integer;
f(n: integer): integer == {
  if n = 0 then {1} else {n * f(n-1)};
}
```

To do so, we add productions for generating the type of a function, the definition of a function, and a call to a function. Here's the enhanced grammar.

$$
\begin{array}{|l|}
\hline
P \longmapsto D;\ S \\
D \longmapsto D;\ D \\
D \longmapsto \mathbf{id} : T \\
T \longmapsto A \\
A \longmapsto \mathbf{boolean} \\
A \longmapsto \mathbf{integer} \\
T \longmapsto A_1 {\to} A_2 \\
E \longmapsto \mathbf{literal} \\
E \longmapsto \mathbf{num} \\
E \longmapsto \mathbf{id} \\
E \longmapsto E_1 + E_2 \\
E \longmapsto E_1 = E_2 \\
E \longmapsto \mathbf{id}(E_1) \\
S \longmapsto S_1;\ S_2 \\
S \longmapsto E \\
S \longmapsto \mathbf{id}{:=}\, S_1 \\
S \longmapsto \mathbf{if}\ E\ \mathbf{then}\ \{S_1\}\mathbf{else}\ \{S_2\} \\
S \longmapsto \mathbf{while}\ E\ \mathbf{repeat}\ \{S_1\} \\
S \longmapsto \mathbf{id}\,(\mathbf{id} :\ A_1){:}A_2 == \{S_1\} \\
\hline
\end{array}
$$

To keep things simple, observe that we impose the following constraints.

(C_1) The input type and the output type of a function can only be *primitive types*, that is **boolean** or **integer**. This appears in the above grammar.

(C_2) In our language, there is **only one** scope for variables and functions. In other words, every variable or function is global.

(C_3) In addition, each formal parameter of a function is viewed as a variable.

These constraints imply that **only one** symbol table is needed, as in the previous exercise. To denote the type of a function you can use a record of the form $[in : T_1, out : T_2]$. So, coming back to the above example

- parsing the first line would create an entry like $\boxed{f \mid [in : integer, out : integer]}$ in the symbol table,
- parsing the function definition would create an entry like $\boxed{n \mid integer}$.

Finally, since in this language every statement has a value (see the previous exercise), a function definition must have a value too. This value is simply the defined function. So, coming back again to the above example, the type of the statement corresponding to the function definition is just $[in : integer, out : integer]$.

Question. Complete the type checker for the following rules. You can use any pseudo-code style that you like or even write sentences like *if there exists an entry* e *with type* T *then* T.

Answer 7

$T \longmapsto A_1 \rightarrow A_2$	$\{\ T.type :=$
$E \longmapsto \mathbf{id}(E_1)$	$\{\ E.type :=$
$S \longmapsto \mathbf{id}\,(\mathbf{id} :\ A_1){:}A_2 == \{S_1\}$	$\{\ S.type :=$

Exercise 8. We consider the following basic block

```
a := b + c
z := a * a
x := 0
y := b + c
w := y * y
u := x + 3
v := u + v
```

Assume that only v and z are live at the exit of this block. Applying the following optimizations

- common sub-expressions elimination
- copy propagation
- dead code elimination

explain how the above basic block can be transformed into another one which has only 3 TAC statements. Note that each optimization can be applied several times.

Answer 8

Exercise 9. Consider the following three-address code program (where `a quo 2` denotes the quotient in the Euclidean division of `a` by `2` and `a mod b` denotes the remainder in the Euclidean division of `a` by `b`)

```
L0: a := 10
    if a = 2 goto L6
L1: b := 2
L2: d := a / 2
    c := a mod b
    if c = 0 goto L5
L3: if b >= d goto L6
L4: b := b + 1
    goto L2
L5: a := a + 1
    goto L1
L6: p := a
```

1. Draw a control-flow graph for this program.
2. Identify the loops of this control-flow graph.
3. Indicate the set of live variables at the exit of each basic block. You can give these data-flow sets **without** any justification.

Note that no optimizations is required in this exercise.

Answer 9

Exercise 10. Consider the following three-address code program fragment where `t3 <> 0` is true if and only if `t3` is **not** zero.

```
L0:  s:=0
     i:=0
     n:=10
L1:  t1 := a-b
     if t1 = 0 goto L3
L2:  t2 := i*4
     s := s+t2
     goto L4
L3:  s := s+i
L4:  i := i+1
     t3 := n-i
     if t3 <> 0 goto L1
L5:  t4 := a-b
```

1. Give the control-flow graph of this program.
2. Identify the back edges and the loops.
3. Which optimization (among *copy propagation*, *dead-code elimination*, *common sub-expression elimination*, *invariant code motion*) can be applied to this program? Show the intermediate transformations of the program. Computing the data-flow sets is **not** required. A brief justification of each transformation will be sufficient.

Answer 10

Annexe 1. Recall that for any string α of symbols the set FIRST(α) satisfies the following conditions for every terminal a and every string of symbols β

- FIRST(α) $\subseteq V_T \cup \{\varepsilon\}$
- $\alpha \stackrel{*}{\Longrightarrow} a\beta$ iff $a \in$ FIRST(α)
- $\alpha \stackrel{*}{\Longrightarrow} \varepsilon$ iff $\varepsilon \in$ FIRST(α).

For a symbol $X \in V_T \cup V_N$ the set FIRST(X) can be computed as follows

Algorithm 1

Input: $X \in V_T \cup V_N$ where V_T and V_N are the sets of terminals and nonterminals, respectively.

Output: FIRST(X).

```
if X ∈ V_T then
      FIRST(X) := {X}
else
      if X ⟼ ε then
         FIRST(X) := {ε}
      else
         FIRST(X) := ∅
for every X ⟶ α with α ≠ ε repeat
      let α = X_1 X_2 ··· X_ℓ where each X_i is a grammar symbol
      for i = 1 ··· ℓ repeat
            FIRST(X) := (FIRST(X_i) \ {ε}) ∪ FIRST(X)
      until ε ∉ FIRST(X_i)
      if ε ∈ ⋂_{1≤i≤n} FIRST(X_i) then
            FIRST(X) := FIRST(X) ∪ {ε}
```

Algorithm 2

Input: $X = X_1X_2\cdots X_n$ where every $X_i \in V_T \cup V_N$.

Output: FIRST($X_1X_2\cdots X_n$).

FIRST(X) := $\emptyset$
for $i = 1\cdots n$ **repeat**
  FIRST(X) := (FIRST(X_i) $\setminus\{\varepsilon\}$) $\cup$ FIRST(X)
until $\varepsilon \notin$ FIRST(X_i)
if $\varepsilon \in \bigcap_{1\leq i\leq n}$FIRST($X_i$) **then**
  FIRST(X) := FIRST(X) $\cup \{\varepsilon\}$

Recall that FOLLOW(A) is the set of the terminals that can appear immediately to the right of the nonterminal A in some sentential form. Moreover $\$$ belongs to FOLLOW(A) if A is the rightmost symbol in some sentential form.

Algorithm 3

Input: $G = (V_T, V_N, S, P)$.

Output: Parsing table M.

for every $A \in V_N$ **repeat**
  for every $a \in V_T \cup \{\$\}$ **repeat**
    $M[A, a]$:= $\emptyset$
for every $(A \longmapsto \alpha) \in P$ **repeat**
  for every $a \in$ FIRST(α) **repeat**
    if $\alpha \neq \varepsilon$ **then**
      $M[A, a]$:= $M[A, a] \cup \{A \longmapsto \alpha\}$
  if $\varepsilon \in$ FIRST(α) **then**
    for every $b \in$ FOLLOW(A) **repeat**
      $M[A, b]$:= $M[A, b] \cup \{A \longmapsto \alpha\}$
for every $A \in V_N$ **repeat**
  for every $a \in V_T \cup \{\$\}$ **repeat**
    if $M[A, a] = \emptyset$ **then**
      $M[A, a]$:= *error*

Annexe 2. Recall that a variable is *live* at a point p of a program $\mathcal{P}$ if it is referenced further in $\mathcal{P}$ (which can decided by considering the control flow graph) otherwise it is said to be dead. For a basic block B

- $use_L(B)$ is the set of the variables which are used in B before they are defined or re-defined in B. Thus, such variables rely on some earlier definitions (in the control flow graph).
- $def_L(B)$ is the set of the variables which are defined in B before they are used anywhere in the program. This can be computed by using a topological sort of the control flow graph (after its back edges have been removed).
- $bottom_L(B)$ is the set of variables that are live at $exit(B)$.
- $top_L(B)$ is the set of variables that are live at $entry(B)$.

The following iterative algorithm propagates liveness information backward around the control flow graph.

- For each basic block B we set
$$top_L(B) = use_L(B) \text{ and } bottom_L(B) = \emptyset \tag{1}$$
- For each basic block B
$$\begin{array}{rcl} bottom_L(B) & = & \bigcup_{B' \in \mathsf{pred}(B)} top_L(B') \\ top_L(B) & = & (bottom_L(B) \setminus def_L(B)) \bigcup use_L(B) \end{array} \tag{2}$$
- Repeat previous step until none of the sets change.