Compiler Theory CS447b: Final exam. **UWO, April 15, 2003.**

Student ID number:

Guidelines. The exam is closed book and all notes are forbidden. The duration of the exam is 3 hours. Each of the 10 exercices will marked be over 4 points except for exercices 9 and 10 which will be marked 2 and 6 respectively. 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 $\Sigma = \{a, b, c\}$. For each of the following languages over Σ give a deterministic automaton that recognizes this language

1. L_1 is the language consisting of all words over Σ except the word $w = ba$.
2. L_2 is the language consisting of all words over Σ for which the word $w = ba$ is **not** a factor. This means that a word over Σ belongs to L_2 if and only if it does **not** match the LEX regular expression `[a-c]*ba[a-c]*`.

Answer 1

Exercise 2. We consider the alphabet $\Sigma = \{(,)\}$ consisting of the opening parenthesis and the closing parenthesis. For each of the following grammars, if its generated language is regular then give a finite automaton recognizing this language otherwise justify briefly why it is not regular.

$$G_1 \begin{cases} S & \longmapsto & A \\ A & \longmapsto & (\,B \\ A & \longmapsto & (\,A \\ B & \longmapsto &)\,B \\ B & \longmapsto &) \end{cases}$$

$$G_2 \begin{cases} S & \longmapsto & E \\ E & \longmapsto & (\,E\,)\,E \\ E & \longmapsto & \varepsilon \end{cases}$$

Answer 2

Exercise 3. We consider the alphabet $\Sigma = \{a, b\}$ and the grammar G below. Let L be the language generated by G over Σ.

$$G \begin{cases} S & \longmapsto & AB \\ A & \longmapsto & aAA \\ A & \longmapsto & \varepsilon \\ B & \longmapsto & bBB \\ B & \longmapsto & \varepsilon \end{cases}$$

1. Give a grammar G' generating $L \setminus \{\varepsilon\}$ and with **no** ε-productions, that is **no** rules of the form $X \longmapsto \varepsilon$, where ε denotes the empty word.
2. If L is regular then give a regular expression for L otherwise tell whether G' is anbiguous or not.

Answer 3

Exercise 4. Consider the following grammar G with terminals a, b, e, i, t and nonterminals S, S', E.

$$\begin{array}{lcl} S & \longmapsto & i\,E\,t\,S\,S' \mid a \\ S' & \longmapsto & e\,S \mid \varepsilon \\ E & \longmapsto & b \end{array}$$

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

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

Give the parsing table (for the non-recursive implementation of predictive parsing)

Answer 4

Exercise 5. Consider the following syntax-directed definition.

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

Write a translation scheme for this syntax-directed definition.

Answer 5

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

Exercise 6. 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 tranalation 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 $E.false := S.next$ $S_1.next := S.next$ $S.code := E.code \mid\mid$ generate($E.true$ '**:**') $\mid\mid S_1.code$
$S \longmapsto$ **if** E **then** S_1 **else** S_2	$E.true$:= newlabel $E.false$:= newlabel $S_1.next := S.next$ $S_2.next := S.next$ $code_1 := E.code \mid\mid$ generate($E.true$ '**:**') $\mid\mid S_1.code$ $code_2$:= generate('**goto**' $S.next$) $\mid\mid$ $code_3$:= generate($E.false$ '**:**') $\mid\mid S_2.code$ $S.code := code_1 \mid\mid code_2 \mid\mid code_3$
$S \longmapsto$ **while** E **repeat** S_1	$S.begin$:= newlabel $E.true$:= newlabel $E.false := S.next$ $S_1.next := S.begin$ $code_1$:= generate($S.begin$ '**:**') $\mid\mid E.code$ $code_2$:= generate($E.true$ '**:**') $\mid\mid S_1.code$ $code_3$:= generate('**goto**' $S.begin$) $S.code := code_1 \mid\mid code_2 \mid\mid code_3$

In C, the **for**-statement has the following form.

```
for ( s1 ; e ; s2 ) s3
```

Taking its meaning to be

```
s1;
while ( e ) {
     s3 ;
     s2 ;
}
```

construct a syntax-directed definition to translate C-style **for**-statements into three-address code.

Answer 6

Exercise 7. 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).

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

There's one sligth 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 $S_1;\ S_2$ 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 7

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

Exercise 8. Some languages such as PL1 permit a list of names to be given a list of attributes. The grammar G below abstracts the problem.

$$\begin{array}{rcl} D & \longmapsto & namelist\ \ attriblist; \\ namelist & \longmapsto & \mathbf{id}, namelist \\ namelist & \longmapsto & \mathbf{id} \\ attriblist & \longmapsto & A\ \ attriblist \\ attriblist & \longmapsto & A \\ A & \longmapsto & \mathbf{long} \\ A & \longmapsto & \mathbf{double} \\ A & \longmapsto & \mathbf{single} \\ A & \longmapsto & \mathbf{integer} \\ A & \longmapsto & \mathbf{float} \end{array}$$

where D, $namelist$, $attriblist$ and A are nonterminals and where **id**, **long**, **double**, **single**, **integer**, **float** and **;** are terminals. As usual **id** is the *identifier* token. For instance each of the following lines is a declaration generated by G.

```
i, j single integer;
u, v, w double float;
```

Give a syntax-directed definition for declarations generated by G. Here are some hints.

- You may simply write N and T instead of $namelist$ and $attriblist$ respectively.
- The code in the actions does **not** need to be C code but may be any pseudo-code that is convenient to you.
- You may consider that **id** has an attribute $\mathbf{id}.name$ (as usual) giving the lexical value of **id**.
- Moreover you may assume that every variable in the symbol table may be associated with a list of attributes by an instruction like enter($\mathbf{id}.name$, $T.list$)
- The instruction $T.list$:= makelist($A.attrib$) will create the list consisting of the single item $A.attrib$.
- The instruction $T.list$:= cons($A.attrib$, $T.list$) will add the element $A.attrib$ at the beginning of the list $T.list$.

Answer 8

Exercise 9. The goal of this exercice is to translate into three-address code a C-like piece of code for the multiplications of two square matrices. To do so we assume that our three-address code implemnts arrays like in C. A part from that, the three-address code instructions you may need are of one of the usual forms:

- `goto L` where `L` is a symbolic label (like `L1, L2, ...`)
- `if x relop y goto L` where `relop` can be = or <
- `x:= y op z` where `op` can be − or + or `*`

Moreover your solution does **not** need to be obtained by one of the algorithms of the course. Just consider that you are programming a very basic calculator. Here's the C-like piece of code we want to translate.

```
for (i=0; i<n; i++) {
     for (j=0; j<n; j++) {
          c[i,j] = 0;
          for (k=0; k<n; k++) {
               c[i,j] = c[i,j] + a[i,k] * b[k,j];
          }
     }
}
```

Answer 9

Exercise 10. Consider the following three-address code program.

```
L0  n := 10
    s := 0
    i := 0
L1  if i > n goto L6
    j := i
    m := n
    p := m
    f := 1
L2  if p = 1 goto L3
    f := f * p
    p := p - 1
    goto L2
L3  a := f
    p := m - j
    f := 1
L4  if p = 1 goto L5
    f := f * p
    p := p - 1
    goto L4
L5  b := f
    c := a quo b
    s := s + c
    i := i - 1
    goto L1
L6
```

1. Give the flow graph of this program.
2. Identify the back edges and the loops.
3. For each basic block B compute the data-flow sets $gen(B)$ and $kill(B)$ of the generated and killed variable definitions.
4. Deduce from this study a significant optimization for the above code.

Answer 10