

CSE-504 Compiler Design

Spring '08

Final Exam

May 15, 2008

Duration: 2 hour, 30 minutes.

Name: _____

USB ID Number: _____

INSTRUCTIONS

Read the following carefully before answering any question.

- Make sure you have filled in your name and USB ID number in the space above.
- Write your answers in the space provided; Keep your answers brief and precise.
- The exam consists of **6** questions, in **11** pages (including this page) for a total of **30** points.

GOOD LUCK!

Question	Max.	Score
1.	6	
2.	4	
3.	4	
4.	8	
5.	3	
6.	5	
Total:	30	

1. **[Total: 6 points]** Consider a small markup language called TFL (which looks like XML) that is used to specify how segments of text must be formatted and rendered. The following is the grammar for TFL:

$$\begin{array}{ll}
 \textit{Para} & \longrightarrow \text{<P> } \textit{PhraseSeq} \text{ </>} \\
 \textit{PhraseSeq} & \longrightarrow \textit{PhraseSeq} \textit{Phrase} \\
 & \quad | \quad \epsilon \\
 \textit{Phrase} & \longrightarrow \text{ } \textit{Phrase} \text{ </>} \\
 & \quad | \quad \text{<N> } \textit{Phrase} \text{ </>} \\
 & \quad | \quad \text{<SMALL> } \textit{Phrase} \text{ </>} \\
 & \quad | \quad \text{<LARGE> } \textit{Phrase} \text{ </>} \\
 & \quad | \quad \textit{Phrase} \textit{Word} \\
 & \quad | \quad \epsilon \\
 \textit{Word} & \longrightarrow \textit{Word} \text{ Character} \\
 & \quad | \quad \text{Character}
 \end{array}$$

In the above grammar, <P>, , <N>, <SMALL>, <LARGE>, </>, and Character are terminal symbols.

The tag indicates that the phrase contained between and the corresponding </> must be rendered in **bold face**. The <N> tag indicates that the phrase contained between <N> and the corresponding </> must be rendered in normal face. The <SMALL> tag indicates that the phrase contained between <SMALL> and the corresponding </> must be rendered with fonts that are one size smaller than the current font. The <LARGE> tag indicates that the phrase contained between <LARGE> and the corresponding </> must be rendered with fonts that are one size larger than the current font. Font sizes are measured in “points” which are integer values.

Assume that the documents will always start out using normal face and a 12-point font.

For instance, the following TFL document:

This is <N>only</> an</> example.

will be rendered as follows:

This **is** only **an** example.

The following TFL document:

Do <LARGE> not <LARGE> use </> <SMALL>too</> many</> font <SMALL>sizes</>

will be rendered as follows:

Do not **USE** too many font sizes

[“Do” is in 12 point font; “not” is 13 points; “use” is 14 points; “too” is 12 points, “many” is 13 points; “font” is 12 points and “sizes” is 11 points].

[CONTINUED ON NEXT PAGE]

[CONTINUED FROM PREVIOUS PAGE]

- (a) [3 points] Give a syntax-directed definition (attribute grammar) that determines the “face” with which each word will be rendered. That is, define a “face” attribute for *Word*. The value of face attribute will be “normal” if the word will be rendered using a normal face, or “bold” if the word will be rendered in **bold face**.
- (b) [3 points] Give a syntax-directed definition that determines the font size used to render each word in a document. That is, define a “size” attribute for *Word* that represents the font size of each word.

2. **[Total: 4 points]** We wish to extend the Decaf language with “do-while” loops, defined by the following grammar rule:

$$Stmt \longrightarrow \text{do } Stmt \text{ while } Expr$$

where *Stmt* stands for statements and *Expr* stands for expressions. The statement following the “do” keyword is called the *body* of the loop, while the expression following the “while” keyword is called the *loop condition*. Operationally, a “do-while” loop is executed by first executing the loop body. The loop condition is then checked; if the condition is false, the loop is exited. If the condition is true, the loop is repeated (and the execution continues on until the loop condition becomes false).

- (a) [2 points] Write a syntax-directed definition to determine whether or not a “do-while” loop is type correct. Assume that *Expr* has an attribute called **type** which represents the type of that expression; and that *Stmt* has a boolean attribute **type_correct** that is true if and only if there are no type errors in that statement.

[CONTINUED ON NEXT PAGE]

[CONTINUED FROM PREVIOUS PAGE]

- (b) [2 points] Write a syntax-directed definition to generate code for “do-while” loops. You may generate code for Cream (as in HW6). Recall that Cream has three flavors of jump instructions: `jmp` for unconditional branch, `jz` for jump on zero (i.e. if the top-of stack is zero), and `jnz` for jump on non-zero. `jz` and `jnz` remove the top-most element on the stack regardless of whether the jump was taken or not. *You may also generate three address code (following the book).*

Clearly state and define any additional attributes you may use.

3. **[Total: 4 points]** Consider introducing a “conditional” expression of the form:

$e1 ? e2 : e3$

The first sub-expression $e1$ should be a Boolean expression. If that evaluates to true, then the value of the whole expression is same as the value of $e2$; otherwise the value of the whole expression is same as the value of $e3$. The following grammar rule derives a conditional expression:

$$Expr \longrightarrow Expr ? Expr : Expr$$

- (a) [2 points] Write a syntax-directed definition to infer the type of a conditional expression. The inferred type should be *error* if there is some type error in the expression.

- (b) [2 points] Write a syntax-directed definition to generate code for a conditional expression. You may generate code for Cream or a Cream-like stack machine, or generate three-address code.

4. **[Total: 8 points]** Consider introducing a statement in Decaf that does *simultaneous assignment* (as in Python):

`(x, y, z) := (x+1,x+2,x+3);`

i.e., an assignment with multiple sources (right-hand sides) and multiple destinations (left-hand sides). The number of left- and right-hand sides will be equal in type-correct statements. The semantics of a simultaneous assignment statement is to evaluate the right hand side values, and move all of those, at the same time, to the left hand side destinations.

If the value of `x` is initially 1, the execution of the above example statement will leave `x` with value 2, `y` with 3 and `z` with 4.

Note that unlike the assignment expression in Decaf, the above uses a `:=` symbol, and is considered a *statement*.

- (a) [2 points] Give a grammar rule that represents such a simultaneous assignment statement. Your grammar rule may permit assignments where the left- and right-hand sides have different number of expressions. We can impose the constraints that the two sides have the same number of expressions when building the AST.

- (b) [2 points] How will you represent such an assignment statement in abstract syntax? You may write your answer in terms of a C++ class definition, or using tree datatypes (as used in class and in HW4 description).

[CONTINUED ON NEXT PAGE]

[CONTINUED FROM PREVIOUS PAGE]

- (c) [2 points] Write a syntax-directed definition to type check the simultaneous assignment statement. (*Hint*: first determine how you will represent the type of a comma-separated list of expressions.)

- (d) [2 points] Write a syntax-directed definition to generate code for a simultaneous assignment statement. For this purpose, you may assume that the left hand side of a simultaneous assignment statement consists only of local variables (not arbitrary lhs expressions). You may also assume that each local variable has an “offset” attribute that specifies the offset of the variable in the current activation record. You may generate code for Cream or a Cream-like stack machine, or generate three-address code.

5. **[3 points]** Below, the figure on the left represents the heap of a program. Each heap cell is a single word, which can hold address or non-address data. Addresses in the heap start from 1. Objects used by the program are allocated a set of consecutive cells, and the address of an object is the address of the first cell allocated to it. In the figure, each object is separated from the next by a double line. The figure shows the address of each object and the contents of each cell in the heap. Each number in the figure represents an address. Each non-number (e.g. “A”) represents a non-address word. For example, there is an object with three words (A, and address 4 and 15) at address 1 in the heap.

The figure on the left is the heap **before** an invocation of a copying garbage collector. (That is, what is shown is the “from” space of the copying collector). Write, in the figure on the right, the effect of copying collection: i.e. the “to” space of the copying collector **after** the collection is done.

Assume that the root of the collection is address 1 (only one object in the root set).

Heap Before Copying Collection

1	A
	4
	15
4	B
	15
6	C
	4
8	D
	10
10	E
	8
12	F
	8
	6
15	G
	10

Heap After Copying Collection

1	
---	--

6. **[Total: 5 points]** This question is on data-flow analysis. For the purpose of this question, we'll consider flow graphs where each node in the flow graph is either the program entry or exit node, or is a single assignment statement of the form $x = Expr$.

A variable x is said to be “*well defined*” at some program point P *only if*:

- Every path leading to the program point P has an assignment to x , and
- For *every* path leading to P , let the last assignment to x (of the form $x = Expr$) be at program point P' . Then every variable in $Expr$ is well-defined at P' .

The set of well-defined variables is empty at program entry.

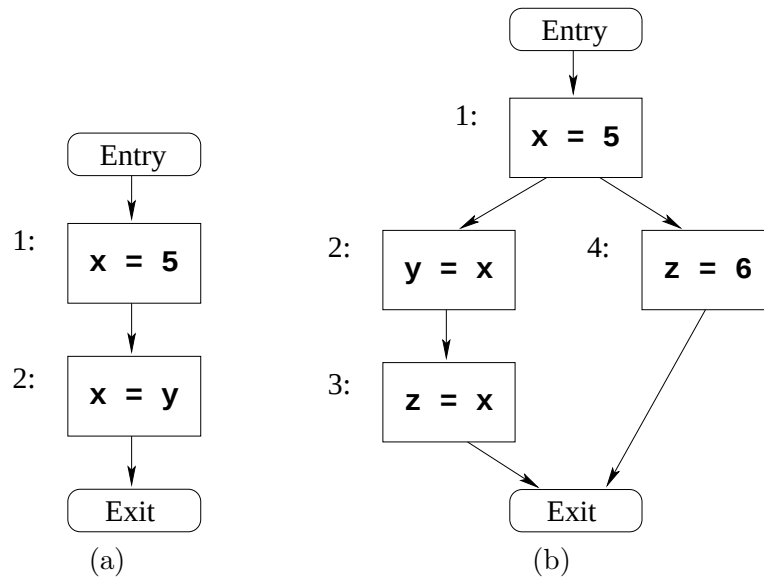


Figure 1: Example flow graphs

For instance, consider the flow graphs in Figure 1.

In Fig. 1(a), after statement 1, the set of well-defined variables is $\{x\}$. After statement 2, the set of well-defined variables is again empty (since y is not well defined after statement 1, x is no longer well-defined).

For the flow graph in Fig. 1(b), the set of well-defined variables after statement 1 is $\{x\}$. After statement 2, $\{x,y\}$ are well defined, and after statement 3 $\{x,y,z\}$ are well defined. After statement 4, $\{x,z\}$ are well defined. Hence at Exit, only $\{x,z\}$ are well defined. (y is not well defined since it is not assigned any value in the path Entry,1,4,Exit.)

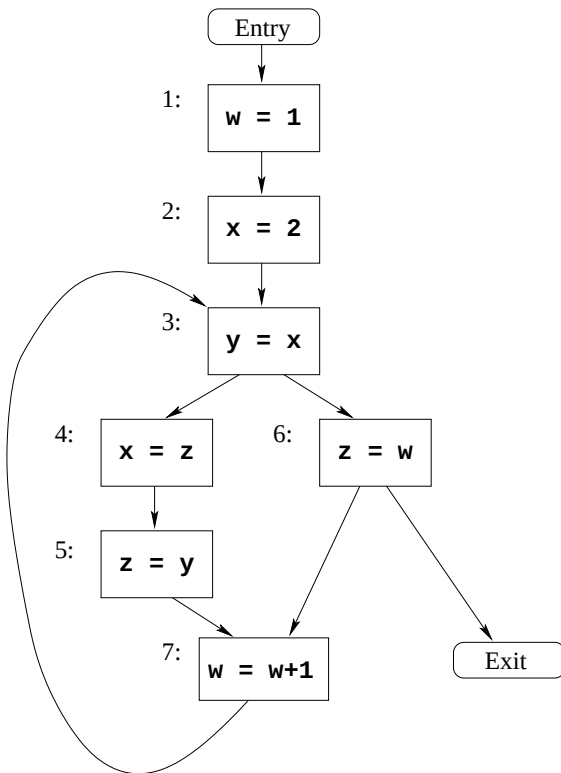
- (a) [1 point] Is this a forward analysis or backward analysis? Justify.

[CONTINUED ON NEXT PAGE]

[CONTINUED FROM PREVIOUS PAGE]

- (b) [3 points] Consider a data flow analysis to compute the set of well-defined variables. Define the sets $IN(B)$ and $OUT(B)$ where $IN(B)$ is the set of well-defined variables at entry to block B , and $OUT(B)$ is the set of well-defined variables at exit from block B . Assume that $vars(Expr)$ gives the set of variables used in expression $Expr$.

- (c) [2 points] Consider the flow graph below. At each node, write the set of variables that are well-defined at entry and exit from that node (i.e. the IN and OUT sets).



Block B	IN(B)	OUT(B)
Entry		
1		
2		
3		
4		
5		
6		
7		
Exit		

END OF EXAM