CS3250

THE UNIVERSITY OF WARWICK

Third Year Examinations: Summer 2013

Compiler Design

Time allowed: 3 hours.

Read carefully the instructions on the answer book and make sure that the particulars required are entered on the cover of **each** answer book. Start the answer for each question on a new page. Calculators are neither allowed nor necessary.

Answer **FOUR** questions.

---

1. (a) Describe the functions performed by a lexer and describe the differences between a Finite State Machine and a lexer. [5]

   (b) Write grammars for the following languages:

   i. the set of non-negative odd integers; [4]

   ii. the set of non-negative even integers with no leading zeros permitted. [4]

   (c) Consider the following grammar:

   $$E \rightarrow E + T \mid T \qquad T \rightarrow T * id \mid id \mid (E)$$

   Is this an ambiguous grammar? Justify your answer. [5]

   (d) Explain the difference between context sensitive, context-free and D-context-free languages, and discuss the relevance of this distinction to compiler construction. [7]

---

2. (a) Explain the difference between top-down and bottom-up parsing. Explain the meaning of the abbreviations LL and LR which are often used to denote the two approaches. [8]

   (b) Consider the following grammar, where S is the start symbol:

   $$S \rightarrow A\,\$ \qquad A \rightarrow B\,A\,a \mid \epsilon \qquad B \rightarrow b\,B\,c \mid A\,A$$

   Calculate Nullable, First, and Follow for A and B. [9]

   (c) The First and Follow sets are used primarily to predict grammar rules in a top-down parser, but they can also be used for syntax error recovery. Show how syntax errors can be managed, using First and Follow sets for a grammar for arithmetic expressions as an example. [8]

Continued

---

3. (a) Explain the role of *item grammars* in building bottom-up parsers and the role of the look-ahead input token in resolving conflicts. [8]

   (b) Construct an LR(0) state machine for the following grammar, where S is the start symbol:

   $$S \rightarrow A\,\$ \mid x\,b\,\$ \qquad A \rightarrow a\,A\,b \mid B \qquad B \rightarrow x$$

   [8]

   (c) Consider the following grammar:

   $$\begin{array}{ll} (1) E \rightarrow E\ ;\ D & (4) T \rightarrow \texttt{int} \\ (2) E \rightarrow D & (5) T \rightarrow \texttt{real} \\ (3) D \rightarrow T\ L & (6) L \rightarrow L\ ,\ \texttt{id} \\ & (7) L \rightarrow \texttt{id} \end{array}$$

   Extend the grammar with attribute rules, associating each identifier with an attribute to represent its *type* and placing this information in a simple symbol table. Assume that there is an external function that associates each identifier with an attribute s such that `id.s` contains the string value of `id`.

   [9]

---

4. (a) Explain the usefulness of *virtual machines* as targets of compilation instead of creating object code directly. [8]

   (b) Describe *activation stacks*, and explain how they support the use of non-local names for languages with static binding. [8]

   (c) Describe the sequence of instructions that need to be performed when calling and returning from functions. In your explanation consider the following function, assuming that the values of the non-local integer variables `x` and `y` should not be modified and that the language uses dynamic binding:

```
f (x, y)
  { int z;
    z := 3 ;
    x := x * 3 ;
    y := g(x) + y ;
    return (x * y) }
```

   [9]

Continued

5. (a) Describe the main steps in the register allocation algorithm, including both coalescing and spilling. [8]

   (b) Compare and contrast the use of Maximal Munch with the use of Dynamic Programming for instruction selection. How does the target architecture affect the choice of instruction selection method? [8]

   (c) Explain the process of efficiently translating Boolean expressions into jump instructions to true and false branches. Illustrate the method using the following code fragment:

```
if x < y and z = a + b then x := y else x := -y ;
y := x ;
```

   [9]

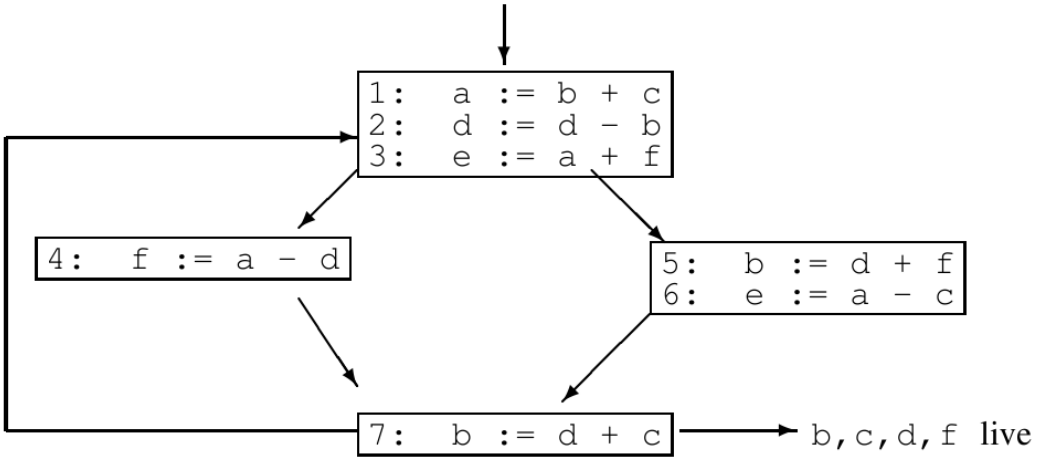
6. (a) For each of the optimisations below, explain what it is and what kind of information needs to be extracted from the program to be optimised:

   i. constant propagation; [5]

   ii. common sub-expression elimination; [5]

   iii. dead-code elimination. [5]

   (b) Given the following flow graph:



   Calculate live variables at each statement. Explain how you have obtained this information. [10]

End

Marking Information - DO NOT COPY

1. (a) Mostly definition of lexing. In lectures we compared FSMs with lexing informally. Marks for mentioning that in lexing we combine several languages into one recogniser, try to recognise the longer word, return the actual word itself, also deal with errors and comments and white space.

   (b) These examples haven't been looked at in lectures.

   i. $S \to NO, \quad N \to DN|\epsilon, \quad D \to 0 \ldots 9, \quad O \to 1|3|5|7|9$

   ii. $S \to NE, \quad N \to PN|\epsilon,$

   (c) This grammar is not ambiguous, in fact it is the commonly known disambiguation for the ambiguous naive grammar for arithmetic expressions. The justification is that for every arithmetic expression there is only one possible leftmost and rightmost derivation.

   (d) The Chomsky hierarchy was studied in lectures. Context sensitive grammars are ones characterised by grammars with productions of the form $\alpha \to \beta$ where $|\alpha| < |\beta|$ while both context-free and D-context-free are given by grammars with productions of form $A \to \beta$. The difference between these two is that D-context-free languages have *deterministic* recognisers while generally context-free languages do not. The relevance to compiler design is that we can construct more efficient parsers when we do not need to model non-determinism.

2. (a) Bookwork, but involves explaining clearly explaining concepts that were explained separately, in different lectures. Top-down parsing involves starting with the start symbol and predicting productions for non-terminals, until being able to match against input, while bottom-up parsing involves scanning the input and creating partial trees by reducing sequences of symbols that match rhs of productions. In both methods the input is matched Left to right, but while top-down parsing produces a Leftmost derivation, bottom-up parsing produces a Rightmost derivation.

   (b) Partial marks for showing calculations. Direct application of algorithm, however the presence of an ϵ rule increases the difficulty.

	Nullable	First	Follow
S	no	{ a, b, $ }	
A	yes	{a,b }	{ a, b, c, $ }
B	yes	{ a,b }	{ a, b, c }

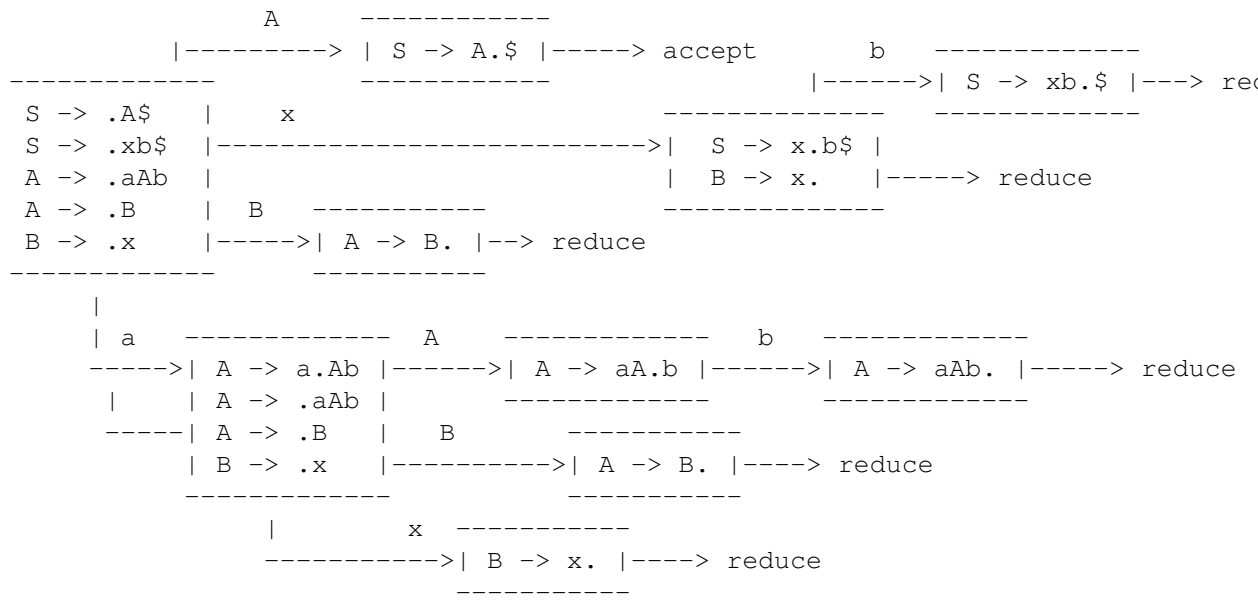
   (c) Discussed informally in lectures in the context of LL parse tables, and how the blank spaces in tables can be used. Follow sets are most useful for error recovery. When an error is found, one can scan and discard all lexemes until reaching a lexeme which is in the Follow of non-terminal being matched. Other techniques would involve replacing a wrong lexeme with one in the First set and continue parsing.

3. (a) This is the basis of LR parsing. Dots in the items represent the possible state in the matching of the rhs of each production. Transitions from one state to another involve

End

moving the dot one position to the right, over a terminal or non-terminal. We looked at several states which include items such as $X \rightarrow L.$ and $X \rightarrow L., S$, and decision is taken based on whether next token is ',' or not.

(b) The main difficulties in this are to ensure the closures are done properly and the one loop on the transition on `a`.



(c) The example involves first building a list of identifiers and then at the next stage processing each element of list and creating a new list which associates each identifier with its type.

```
(1) E0.d = D.d :: E1.d
(2) E.d  = D.d
(3) D.d  = map (\x.(T.t, x)) L.l
(4) T.t  = int
(5) T.t  = real
(6) L0.l = id.s :: L1.l
(7) L.l  = id.s
```

4. (a) Bookwork; in introducing intermediate languages we went over several of the benefits: portability, optimisation, correctness, adaptation, etc.

(b) Notes and textbook include detailed schema for activation records, showing all the information which is stored for each activation of a procedure: space for local variables, parameters, registers, and return address. In addition, for static binding activation records need to store pointer to most recent activation of enclosing scope, optionally through a display table.

End

	...
	Next activation records
	Space for storing local variables for spill and for storing live variables allocated to caller-saves registers across function calls
	Space for storing callee-saves registers that are used in the body
	Incoming parameters in excess of four
	Return address
FP ⟶	Static link (SL)
	Previous activation records
	...

Figure 10.13: Activation record with static link

	...			...	
	y			q	
f:	x		g:	p	
	Return address			Return address	
FP →	SL (null)		FP →	SL (to f)	
	...			...	

Figure 10.14: Activation records for `f` and `g` from figure 10.11

(c) Calling sequences depend on whether we wish to use caller-saving or callee-saving, students can use either of these approaches but should be aware off the difference. In each case, the stack pointer needs to be updates, parameters placed, area for local variables allocated, and contents of registers copied..

5. (a) We spent several lectures going over graph colouring and register allocation. Algorithm 9.3 in textbook is a high-level summary of the use of graph colouring to initialise, simplify, and select registers. The addition of coalescing and spilling involves a higher level loop:

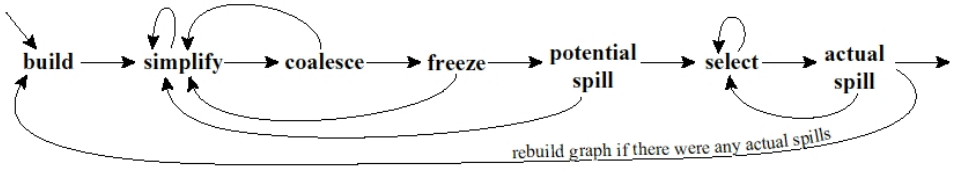


FIGURE 11.4. Graph coloring with coalescing.

(b) Maximal Munch involves instruction selection in one step, where instructions are matched to root node of a subtree and the match that covers most nodes is chosen, and the remaining subtrees are processed recursively. This suits RISC architectures where the different instructions cost roughly the same. Dynamic programming is useful in CISC architectures as there can be a significant different in costs and therefore the best choice for each possible subtree is first calculated before making the choice for instruction of the root node.

End

(c) In lectures we discussed extensively the issue of needing to fill in addresses of instructions still to be generated, and the use of *backpatching* as a way to fill in addresses later, when they are known. The example code also shows the efficiency gain obtained by *boolean short-circuiting*. While the actual attribute grammar is quite complex, the example can be used in answer, as long as students obtain code similar to:

```
     if x /< y goto ---          fill with l3
     t1 := a + b
l1   if z =/= t1 goto ---        fill with l3
l2   x = y
     goto ---                    fill with l4
l3   x := - y
l4   y := x
```

6. (a) Bookwork. Each of these optimisations covered in lectures and textbook. The information that needs to be extracted from program:

constant propagation	reaching definitions
common sub-exp elimination	available expressions
dead-code elimination	live variables

(b) Calculations of live variables can be done informally, by annotating graph, as long as there is some explanation. Ideally students should show the table where the information is incremented until there is no change in information added:

	use	kill	in	out	in	out	in	out
1	bc	a	bc	db	bcd	af	bcdf	abdcf
2	db	d	db	af	abdf	adf	abcdf	adcf
3	af	e	af	adf	adf	acd	acdf	adcf
4	ad	f	ad	dc	adc	dcf	adc	cdf
5	df	b	df	ac	acdf	dc	adcf	adcf
6	ac	e	ac	dc	acd	dcf	adcf	cdf
7	dc	b	dc	bcdf	dcf	bcdf	dcf	bcdf

End