GRIFFITH UNIVERSITY

School of Information and Communication Technology

3136CIT Programming Language Implementation

Final Examination, Semester 1, 2006

Details

Total marks:	80 (40% of the total marks for this subject)
Perusal time:	15 minutes
Working time:	3 hours
Date:	Saturday 10 June 2006

Instructions

1. **Don't panic!**
2. Read these instructions carefully. Use the perusal time wisely.
3. **You may not write anything during perusal.**
4. **This is a closed book examination.** Written materials other than language translation dictionaries (on paper) are not permitted. Calculators, PDAs and other electronic devices are not permitted.
5. The examination has eight questions worth 10 marks each.
6. **Write answers to all questions in the answer booklets provided.** Start the answer to each question on a new page. Please write neatly. Use blue or black ink (and definitely not red ink). Dark pencil is acceptable.
7. In general, explicitly state any assumptions you are making and show all working. Credit may be given for incomplete answers provided the work shown is understandable (and correct).

Paper number: 365

3136CIT Programming Language Implementation (2006/1)

1. (a) Briefly describe the four main tasks of a lexical analyser.

   (b) Give a regular expression for the set of strings of 0's and 1's that start with 1 and contain an even number of 0's, *e.g.*, 1, 100, 1011101001.

   (c) Give a regular expression for the set of programming language identifiers, each of which starts with a letter and contains only letters and digits, *e.g.*, a, a123bc4,

   (d) Give a regular expression for the set of programming language floating point constants, where each such constant contains either a decimal point or an exponent part, *e.g.*, 123.45, 123e45, 1.23e45. The strings 123. and .45 are *not* floating point constants for the purposes of this question. Ignore the possibility of negative constants or negative exponents.

   In the last two parts, you may use the symbols LETTER and DIGIT to represent a letter or digit respectively.

2. (a) Briefly describe the two main tasks of a parser.

   (b) What language is described by the following context-free grammar?

   $S \rightarrow T \mid a\,S\,c$
   $T \rightarrow \epsilon \mid b\,T\,c$

   The terminal symbols in this grammar are a, b and c; ϵ denotes the empty sequence.

   (c) Give a context-free grammar for the set of nonempty balanced parenthesis strings, *e.g.*, (), ()(), ((())()).

   (d) Consider the following simple grammar for arithmetic expressions, *e.g.*, 2, 3*(4+5+6).

   $E \rightarrow \mathbf{num} \mid (\,E\,) \mid E * E \mid E + E$

   The terminal symbols in this grammar are numbers ($\mathbf{num}$), $($, $)$, $*$ and $+$.

   Use an example to show that the grammar is ambiguous. (Remember to state carefully why your example shows that the grammar is ambiguous.)

   (e) Give an unambiguous grammar for the same language.

3. Consider the following grammar for arithmetic expressions in Lisp notation, *e.g.*, 2, (+ 3 (* 4 5 6) (+ 7 8)).

   $E \rightarrow \mathbf{num} \mid (P\,L)$
   $P \rightarrow * \mid +$
   $L \rightarrow E \mid L\,E$

   The terminal symbols in this grammar are numbers ($\mathbf{num}$), $($, $+$, $*$ and $)$.

   (a) What is the start (or first) set of the nonterminal symbol L? What is the follow set of the nonterminal symbol L? Explain why this grammar is not LL(1).

   (b) Transform the grammar into an LL(1) grammar for the same language. **Hint** Use regular expressions on the right hand sides of rules in your transformed grammar.

   (c) Write a recursive descent parser in C (or Java, or clear pseudo-code) for arithmetic expressions in your transformed grammar.

Your parser only need to recognise whether or not an input string is a legal arithmetic expression defined by the initial grammar. You may use the global variable `token` and the (void) functions `getToken()` and `error()` in the normal way.

4. Consider the following grammar for identifiers or nested, parenthesized lists of identifiers, *e.g.*, a, (a,b), (a,(a,b,(c)), d).

   (1) $S \rightarrow$ **id**
   (2) $S \rightarrow (L)$
   (3) $L \rightarrow S$
   (4) $L \rightarrow L , S$

   The terminal symbols of this grammar are identifiers (**id**), $($, , and $)$.

   Now, consider the following SLR(1) parsing table for this language of nested parenthesized lists of identifiers.

State	**Action**					**Goto**	
	id	(	,	)	$	S	L
0	Shift 1	Shift 2				3	
1			Reduce(1)	Reduce(1)	Reduce(1)		
2	Shift 1	Shift 2				4	5
3					Accept		
4			Reduce (3)	Reduce (3)			
5			Shift 7	Shift 6			
6			Reduce(2)	Reduce(2)	Reduce(2)		
7	Shift 1	Shift 2				8	
8			Reduce(4)	Reduce(4)			

   (a) Each state consists of a set of "items" of the form $A \rightarrow X_1 \ldots X_m \bullet X_{m+1} \ldots X_n$. Carefully describe what each such item means.

   (b) Given the input $(a, (b))\$$, trace the behaviour of this parser. Initially, the parsing stack contains state 0. At each step, show the parsing stack, the remaining input, and the action taken.

   (c) In this parser, state 0 consists of the following items.
   $S' \rightarrow . S$
   $S \rightarrow .$ **id**
   $S \rightarrow . (L)$
   (Here, S' is a new nonterminal symbol.) Which items does state 2 consist of?

5. (a) Briefly describe the two main tasks of a semantic analyser.

   (b) What are the two main data structures required by a semantic analyser to perform these tasks?

   (c) Briefly describe a typical data structure used to implement a symbol table, and justify two main features of the data structure.

(d) Consider the following (ambiguous) grammar for arithmetic expressions with embedded assignments, *e.g.*, 3 + (x = 5.0+1) + 7.0.

$E \to \mathbf{id}$
$E \to \mathbf{int}$
$E \to \mathbf{float}$
$E \to (\, E \,)$
$E \to \mathbf{id} = E$
$E \to E + E$

The terminal symbols in this grammar are $\mathbf{id}$, $\mathbf{int}$, $\mathbf{float}$, $($, $)$, $=$ and $+$.

Informally, each expression has a type *Int* or *Float*, each $\mathbf{int}$ (resp., $\mathbf{float}$) expression has type *Int* (resp., *Float*), each $\mathbf{id}$ expression has type given by applying the function *typeOf* to the *name* of the $\mathbf{id}$, the sum of two expressions of type *Int* has type *Int*, otherwise a sum has type *Float*, the type of an assignment ($\mathbf{id} = E$) is the type of the expression E, and an assignment ($\mathbf{id} = E$) is valid if **id** and E have the same type or $\mathbf{id}$ has type *Float* and E has type *Int*.

Give an attribute grammar that assigns a type attribute to each subexpression in the syntax tree for an expression and checks that all assignments are valid.

6. (a) What is the value of the following Scheme expression?

```
(let ((m 1))
  (define (f) (+ m 2))
  (define (g m) (f))
  (g 4))
```

(b) Transform the Scheme let*-expression

```
(let* ((a 1) (b (+ a 2)))
  (+ a b 3))
```

into an equivalent Scheme expression using lambda-expressions without let*-expressions. Recall that let*-expressions bind their local variables sequentially.

(c) Describe the Scheme evaluation process when a procedure (or closure) *proc* is applied to a list *args* of (evaluated) arguments, You may use either a Scheme function definition or clear pseudocode. You may assume the procedure is a compound procedure, not a primitive procedure.

(d) Consider the following Scheme function for efficiently raising a value `x` to a non-negative integer power `n`.

```
(define (power x n)
  (cond ((zero? n) 1)
        ((even? n) (power (* x x) (/ n 2)))
        (else (* x (power x (- n 1))))))
```

Transform this function definition into continuation-passing style.

Explain why such transformations into continuation-passing style are important.

7. Answer **either** part (a) **or** part (b) below (but not both).

   (a) Carefully describe a mechanism for dynamic storage management, on a heap, capable of implementing C `malloc()` and `free()` functions. In your description, indicate how these two functions should be implemented, discuss the extent to which internal or external fragmentation may occur, and consider the efficiency of your implementation.

   (b) Carefully describe the structure and contents of an activation record (or stack frame) for a procedural language with nested scopes, *i.e.*, with nonlocal variables. Assuming the existence of registers *sp* (stack pointer) and *fp* (frame pointer), carefully describe what operations must be performed on procedure call and on procedure return. Also indicate how both local and nonlocal variables are accessed.

8. (a) Briefly describe three different, possible types of target languages for code generation.

   (b) Give a sequence of three-address code instructions that is equivalent to the following source code fragment. Avoid any unnecessary unconditional jumps in your solution.

```
read a, b;
while b > 0 do
    c = b;
    b = a % b
    a = c;
end;
write a;
```

   (c) Give a sequence of three-address code instructions that is equivalent to the following conditional statement. Avoid any Boolean operations or unnecessary unconditional jumps in your solution.

```
if (i < 0 || i > j) && (done || n > 500) S1 else S2 end
```

   Here, `S1` and `S2` are abitrary statement sequences.

   (d) Name two important optimisation techniques and give a clear example of each one.