CANDIDATE: please attach Student Support Unit sticker, if relevant

G5035

THE UNIVERSITY OF SUSSEX

INFORMATICS

BSc AND MComp SECOND YEAR EXAMINATION 2021

January 2021 (A1)

Compilers and Computer Architecture

Candidates should answer TWO questions out of THREE. If all three questions are attempted only the first two answers will be marked.

Each question is worth 50 marks.

Write your answers on A4 paper, scan and save as a single PDF file and upload to Canvas
PDF file name: candidate number_module title
Read Academic Integrity Statement

You are reminded that, unless you have been authorised to do so in School or specific assessment guidance, you should not access online materials, notes etc. during this examination or discuss this assessment with others before the end of its 24 hour window. By submitting this assessment you confirm that you have read the above Statement and are responsible for understanding and complying with our academic misconduct regulations (found on Student Hub and here: Academic Misconduct regulations).

1. This question is about lexing, parsing, and CFGs (= context free grammars).

   (a) For each of the following statements about ASTs (= abstract syntax trees), state whether they are true or false. Correct answers are awarded 2 marks; wrong answers are penalised by 1 mark (to a minimum of 0 marks for this question). Omitted answers do not contribute positively or negatively to your grade. [14 marks]

   i. The lexer takes as input a list of tokens, representing a program.

   ii. The lexer produces an AST from the list of tokens.

   iii. The token list is produced in the semantic analysis stage.

   iv. The purpose of the lexing phase is to make it easier to adapt a compiler to new machine architectures (such as moving from Intel to ARM processors).

   v. The purpose of constructing an AST is to represent the structure of the computer program, to simplify code generation.

   vi. The purpose of constructing an AST is to allow the compiler to generate faster executable machine code.

   vii. Dynamically typed languages like Python and Javascript don't need to do lexing and parsing.

   (b) In this question, we consider CFGs in which we take S to be the initial variable. For each of the following statements, state whether they are true or false. Correct answers are awarded 4 marks; wrong answers are penalised by 1 mark (to a minimum of 0 marks for this question). Omitted answers do not contribute positively or negatively to your grade. [16 marks]

   i. The CFG over the alphabet $\{a, b, =, +, 1, ;\}$ with reductions

   $$S \to A;S \;\Big|\; A \to a{=}E \;\Big|\; A \to b{=}E \;\Big|\; E \to a \;\Big|\; E \to b \;\Big|\; E \to 1 \;\Big|\; E \to V{+}V$$

   True or false: the language of this CFG is infinite (meaning: contains infinitely many strings).

   ii. The CFG over the alphabet $\{(,)\}$ with reductions

   $$S \to (S) \quad \Big| \quad S \to (SS) \quad \Big| \quad S \to ()$$

   True or false: the language of this CFG is precisely the set of non-empty strings over $\{(,)\}$ with balanced brackets, including e.g. $(), (()), (()(())), ...$

   iii. The CFG over the alphabet $\{(,)\}$ with reductions

   $$S \to (T) \quad \Big| \quad S \to T(T) \quad \Big| \quad S \to (T((T)T)) \quad \Big| \quad S \to \varepsilon \quad \Big| \quad T \to S$$

   True or false: the language of this CFG is precisely the set of (possibly empty) strings over $\{(,)\}$ with balanced brackets, including e.g. $(), (()), (()(())), ...$

iv. The CFG over the alphabet $\{a, !\}$ with reductions

$$S \to HSHS \quad \Big| \quad S \to aH!! \quad \Big| \quad S \to \varepsilon \quad \Big| \quad H \to SHS$$

True or false: the CFG is left-recursive.

(c) For this question, you are to provide a formal specification of the structure of the reduction relations of a CFG, given that a CFG can itself be described using a sequence of symbols.

Consider a context-free grammar $G = (A, V, S, X)$, with alphabet $A = \{a, b, c\}$, variables $V = \{P, Q, R, S, T\}$, and start variable S. Observe that one may specify the reduction relation X with a string of the following format:

- A single reduction rule has the format "$\langle \text{var} \rangle ::= \langle \text{string} \rangle$", where
  - $\langle \text{var} \rangle \in V$, and
  - $\langle \text{string} \rangle$ is either a single symbol 'epsilon', or a non-empty string over the set $A \cup V$.
- The set of all reduction rules consists either of just a single reduction rule, or a semicolon-separated list "$\langle \text{rule}_1 \rangle$; $\langle \text{rule}_2 \rangle$; $\cdots$" in which each $\langle \text{rule}_\text{j} \rangle$ is a single reduction rule.

Write a formal grammar (involving alphabet symbols '$::=$', 'epsilon', '$;$', 'a', 'b', 'c', 'P', 'Q', *etc.*) which accepts precisely the strings with the above format, for a CFG $G = (A, V, S, X)$ as set out above. (**Hint:** use notation which distinguishes the rules of the CFG G, from the rules of the formal grammar which you use to *describe* G.) [20 marks]

2. This question is about code generation. The first two parts make reference to the following program fragment in a Java-like language:

```
class MyInt {
  private int n;
  public MyInt (int j)   { n=j; }
  public int f (int x)   { n=n+1; return n*x; }
  public MyInt g (int x) {
    int j = x+n;
    MyInt r = new MyInt(j);
    return r;
  }
}
```

   (a) Consider an invocation of the form `a.g(z)` for variables `a` (of type `MyInt`) and `z` (of type `int`). Describe the locations in memory of the variables `x`, `j`, `n`, and `r` (making reference to the stack, activation frame, symbol table, or heap as appropriate), and explain the reason for your answer.

      i. Briefly describe the location of the variable `x`, and the reason for this. [4 marks]

      ii. Describe the location of the variable `j`. Does it have any simple relationship to the location of `x`? Briefly explain your answer. [8 marks]

      iii. Describe the location of the variable `n`. Does it have any simple relationship to the location of `x` or `j`? Briefly explain your answer. [8 marks]

      iv. Describe the location of the object `r`. Does it have any simple relationship to the location of `x`, `j`, or `n`? Briefly explain your answer. [8 marks]

   (b) Explain how the compilation of methods and method invocation in object-oriented languages, can be reduced to the compilation of procedures and procedure calls. Your answer should suppose an approach involving method tables.

      i. Write pseudocode representing how `MyInt::f` would be realised as a procedure, and how an invocation `a.f(x)` would be realised as a procedure invocation for an object `a` of class `MyInt`. [6 marks]

      ii. Explain briefly how method bodies are stored in memory, and how these memory locations may be found at run-time in a method invocation. [10 marks]

      iii. Explain briefly what advantage there is to perform such a conversion, e.g., in a compiler accepting multiple source languages. [6 marks]

3. This question is about computer architecture.

   (a) In most common architectures, a memory access to an odd-valued address may either fail, or be slower than a memory access to an address which is a multiple of 4 or 8. Briefly explain why. [10 marks]

   (b) Describe the two main issues which apply to the use and availability of the following for kinds of memory. Describe how these issues apply to each sort of memory (a very brief description for each). [10 marks]

      i. Registers;
      ii. Flash memory;
      iii. Dynamic RAM (main memory);
      iv. Static RAM (cache);

   (c) In this question, you will write (short) RISC-V assembly programs for two different tasks. Your programs may only use the registers `x0`, `a0`, `t1`, `t2`, `sp`, and `fp`, and the following RISC-V assembly commands:

`lw`	*register*$_1$	*offset* `(` *register*$_2$ `)`		load a word of data at *offset* from an address stored in *register*$_2$, into *register*$_1$.
`sw`	*register*$_1$	*offset* `(` *register*$_2$ `)`		store the value of *register*$_1$ into memory at *offset* from an address stored in *register*$_2$.
`add`	*register*$_1$	*register*$_2$	*register*$_3$	add the values stored in *register*$_3$ and *register*$_2$, and store the result in *register*$_1$.
`addi`	*register*$_1$	*register*$_2$	*constant*	add the signed 16-bit value *constant* to the value stored in *register*$_2$ and and store the result in *register*$_1$.
`b`	*address*			jump to the instruction at *address* .
`beq`	*register*$_1$	*register*$_2$	*address*	if the contents of *register*$_1$ and *register*$_2$ are equal, jump to the instruction at *address* .
`bne`	*register*$_1$	*register*$_2$	*address*	if the contents of *register*$_1$ and *register*$_2$ are different, jump to the instruction at *address* .

   Each instruction is to be written on a separate line. Use the convention that the stack grows "downwards" (*i.e.*, that the front of the stack is at a lower address than the rest of the stack) and that `sp` points at an unused word in memory below the front of the stack.

i. Write a program that pushes the value 0 onto the front of the stack. [10 marks]

ii. Write a RISC-V assembly program to add a sequence of numbers which are stored on the stack. Assume the following:

- The last item on the stack to be added, is at the address which is pointed to by `fp`; and that the initial value stored by `fp` is an address in memory which is higher than that stored in `sp`.
- In particular: there is at least one item on the stack to be added.

The resulting sum is to be stored at the front of the stack. The first instruction should have a symbolic address "`start`", and the last instruction should jump to a symbolic address "`done`". [20 marks]

End of Paper