Examination in Compilers, EDAN65

Department of Computer Science, Lund University

2018–04–10, 08.00-13.00

Note! **Your exam will be marked only if you have completed all six programming lab assignments in advance.**

Start each solution (1, 2, 3, 4) on a separate sheet of paper. Write your *personal identifier*[1] on every sheet of paper. Write clearly and legibly. Try to find clear, readable solutions with meaningful names. Unnecessary complexity will result in point reduction.

The following documents may be used during the exam:

- *Reference manual for JastAdd2*
- *x86 Cheat Sheet*

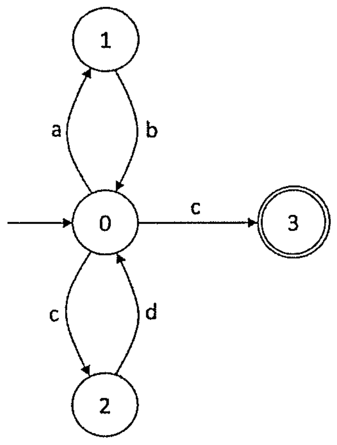
You may also use a dictionary from English to your native language.

Max points: 60
For grade 3: Min 30
For grade 4: Min 40
For grade 5: Min 50

[1]The *personal identifier* is a short phrase, a code or a brief sentence of your choice. It can be anything, but not something that can reveal your identity. The purpose of this identifier is to make it possible for you to identify your exam in case something goes wrong with the anonymous code on the exam cover (such as if it is confused with another code due to sloppy writing).

1 Lexical analysis

Consider the following NFA that defines a language over the alphabet $\{a, b, c, d\}$.



a) Construct a regular expression that is equivalent to the NFA. (5p)

b) Construct a DFA that is equivalent to the NFA. Each state should be labelled with the set of corresponding states in the original NFA. The DFA should have as few states as possible. (5p)

c) List the 7 shortest strings that belong to the language. (5p)

2 Context-Free Grammars

Consider the following context-free grammar for email addresses:

p_0 : `start → address EOF`
p_1 : `address → name "@" name "." ID`
p_2 : `name → ID`
p_3 : `name → ID "." name`

where `EOF` is the terminal symbol for end-of-file, and `ID` is a terminal symbol corresponding to a token defined as `ID = [A-Za-z]+`

a) Construct a parse tree following the above grammar and that uses each production at least once. (5p)

b) The grammar is not LL(1). Construct an equivalent grammar that is LL(1). (5p)

c) Construct the FIRST and FOLLOW sets for the nonterminals of your transformed grammar. (5p)

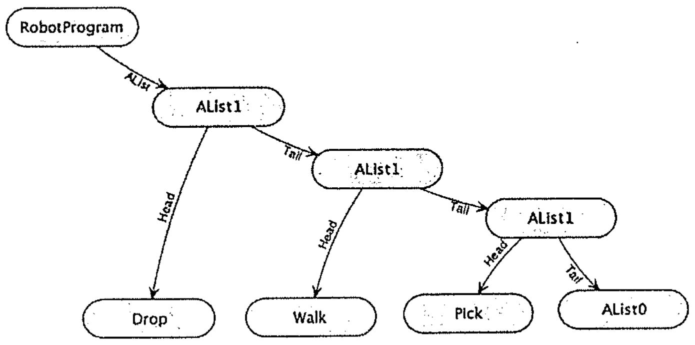
d) Construct an LL(1) table for your transformed grammar, with one column for each of the terminals `"@"`, `"."`, `ID`, and `EOF`. (5p)

3 Program analysis

The following abstract grammar defines a robot language with instructions Walk, Pick, and Drop. At a Pick action, the robot picks up an item, but it can hold at most three items, so if it already has three, the Pick action has no effect. The Drop action drops one of the items, but has no effect if the robot has no items. At the Walk action, the robot takes a step forward. The actions are arranged in a sequence, using AList nodes, where AList0 is an empty list, and AList1 is a list with at least one action.

```
RobotProgram ::= AList;
abstract AList;
AList0 : AList;
AList1 : AList ::= Head:Action Tail:AList;
abstract Action;
Drop : Action;
Pick : Action;
Walk : Action;
```

The figure below shows an example robot program where the robot does a Drop, a Walk, and a Pick action. Initially, the robot has no items, so after executing this sequence it has one item left.



a) Construct an attribute grammar that defines an integer attribute `Action.items()` whose value is the number of items that the robot holds after executing that action, i.e., a value between 0 and 3. Furthermore, define an integer attribute `RobotProgram.items()` whose value is the number of items the robot holds after executing all the actions. For the above program, the value would be 1. Your solution should use inherited and synthesized attributes, and may not use `instanceof`.

You may assume the existence of two methods

```
int min(int v1, int v2) { ... }
int max(int v2, int v2) { ... }
```

Hint! Define an additional attribute `prevItems()` for the number of items the robot has prior to each action. Which attributes should be synthesized and which should be inherited? (10p)

b) Add a collection attribute `RobotProgram.failedPicks()` that counts the number of failed Pick actions in the program, i.e., Pick actions for which the robot already holds three items. The type of the collection attribute should be `Counter`, defined as follows:

```
public class Counter {
  private int count = 0;
  public void add(int increment) {
    count = count + increment;
  }
  public int count() {
    return count;
  }
}
```

(5p)

Turn page for problem 4

4 Code generation and run-time systems

Consider the following recursive program computing fibonacci numbers in a C-like language:

```
int fib(int n) {
  if ( n <= 1 ) // ** PC **
    return n;
  return fib(n-1) + fib(n-2);
}
void main() {
  int s = fib(3);
  ...
}
```

a) A compiler for the language generates unoptimized code that pushes temporaries on the stack. Arguments are passed on the stack (not in registers). The return value is passed in the rax register.

Write down the x86 code generated by the compiler for the fib method.

Use only the instructions on the x86 Cheat Sheet. Use rbp as frame pointer and rsp as stack pointer. You are encouraged to comment your code to help us understand your intention. For simplicity and readability, you may leave out the characters q, %, and , in the code. For example, you may write `add $8 rax` instead of `addq $8, %rax`. (5p)

b) Draw the situation on the stack when the execution has reached the location indicated by `** PC **` for the first call to `fib(0)`.

Your drawing should include stack frames for main and fib activations, stack pointer, frame pointer, dynamic links, return addresses, arguments and temporary variables. Include the actual values for dynamic links, arguments and temporary variables according to what is known at that point in the execution.

Mark each frame with what activation it is, e.g., `fib(3)`. Mark arguments with *arg* and temporaries with *temp*. The drawing should be consistent with your code from problem 4 a). (5p)