Examination in Compilers, EDAN65

Department of Computer Science, Lund University

2023–10–27, 14.00-19.00

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 only on one side of each sheet. Write your *anonymous code* and *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*

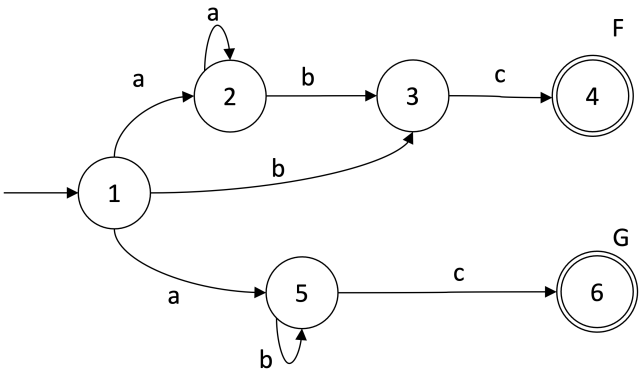
Max points: 60
For grade 3: Min 30
For grade 4: Min 40
For grade 5: Min 50

Good luck!

[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

A language defines the tokens `F` and `G` over the alphabet {`a, b, c`}. The tokens are defined by the following NFA, where `F` has rule priority over `G` in case of ambiguities.



a) For each token, list all strings of length 5 or shorter, taking rule priority into account. (5p)

b) Write down the regular expressions for the two tokens. Each expression should be as simple as possible. (5p)

c) Construct a DFA that is equivalent to the NFA and where each state is labelled with the set of corresponding NFA states. The DFA should have as few states as possible. Each final state should be labelled with the appropriate token, taking rule priority into account. (5p)

2 Grammars

Consider the following grammar for a small language:

p_0 : `start` $\rightarrow$ `stm EOF`
p_1 : `stm` $\rightarrow$ `"if" exp "then" stmlist "fi"`
p_2 : `stm` $\rightarrow$ `"if" exp "then" stmlist "else" stmlist "fi"`
p_3 : `stm` $\rightarrow$ `ID "=" exp ";"`
p_4 : `stmlist` $\rightarrow$ `stm stmlist`
p_5 : `stmlist` $\rightarrow \epsilon$
p_6 : `exp` $\rightarrow$ `exp "+" factor`
p_7 : `exp` $\rightarrow$ `factor`
p_8 : `factor` $\rightarrow$ `ID`
p_9 : `factor` $\rightarrow$ `"(" exp ")"`

where `start` is the start symbol and the alphabet used is

`EOF, "if", "then", "fi", "else", ID, "=", ";", "+", "(", ")"`

a) Draw a derivation tree for the following token sequence:

`"if" ID "then" ID "=" ID ";" "fi" EOF`

The derivation tree should follow the grammar *exactly*, including all terminals and nonterminals.

(5p)

b) Consider again the input sequence `"if" ID "then" ID "=" ID ";" "fi" EOF`. Write down the sequence of steps that an LR parser would take for parsing this program according to the grammar above. For each step, show the stack contents, the remaining input, and the shift/reduce/accept action taken as in the following table:

stack	•	*remaining input*	*action*
	•	`"if" ID "then" ID "=" ID ";" "fi" EOF`	...
...	•	...	...

(5p)

c) The grammar above is not LL(1). Construct an equivalent grammar that is on canonical form and that is LL(1). Enumerate the productions of the LL(1) grammar (i.e., $p_1, p_2, ...$ and so on).

(5p)

d) Construct an LL(1) table for the grammar you constructed in 2(c).

(5p)

3 Program analysis

Consider the following example in a small language where a program is simply a list of variable declarations, assignments, and function calls. (The functions are defined elsewhere.)

```
int a;
int b;
a = 1 + f1(2,8) + f2(4,3,4);
b = a + 2;
f3(a + 3 + f4(5+a));
```

In this example, the value of `b` is never accessed in any computation. The variable occurs only on the left-hand side of an assignment, and not in any position where its value is accessed: `b` is not used in any right-hand side of an assignment, or as an argument in a function call. We say that the variable `b` is *dead*: it could be removed without changing the meaning of the program.

Solve the problems below using reference attribute grammars. Note that you may neither use `instanceof` nor the `getParent()` method.

a) Construct a JastAdd abstract grammar for this language. The grammar should make use of the following two existing node classes for variable declarations and uses, respectively:

```
IdDecl ::= <ID>;
IdUse ::= <ID>;
```

(3p)

b) Assume that a name analysis aspect already has been implemented, defining the following attributes

```
syn IdDecl IdUse.decl() = ...
syn nta Undeclared Program.undeclared() = ...
```

Here, `decl` refers to the appropriate `IdDecl`, or to `undeclared` if the declaration is missing. `Undeclared` is an abstract grammar subclass of `IdDecl`:

```
Undeclared : IdDecl;
```

Define a collection attribute `IdDecl.uses()` that contains the set of `IdUse`s bound to that `IdDecl`. (4p)

c) Define a boolean inherited attribute `IdUse.valueAccessed()` that is true if the `IdUse` is in a position where its value is accessed (like in the right-hand side of an assignment, or in the argument to a function call). (4p)

d) Define a boolean synthesized attribute `IdDecl.dead()` that is true if none of its `IdUse`s are in a position where the variable value is accessed. To help defining this attribute, use the attributes defined in 3b and 3c.

In the example above, `IdDecl.dead()` should be false for `a` and true for `b`. (4p)

Note

It can be noted that the analysis in 3d is very simple, and will not find all variables that can be removed. For example, if a statement `c = b` is added, then the analysis would no longer consider `b` to be dead, even if the value of `c` is not used (and therefore dead). A more sophisticated analysis could handle such transitivity, and it could also take the order of assignments into account.

Turn page for problem 4

4 Code generation and run-time systems

Consider the following program in a C-like language:

```
int a(int x, int y) {
  int z = x+y;
  // ** PC **
  return z;
}
int b(int x, int y) {
  return x*y;
}
int c(int x) {
  return x*2;
}
int d(int x, int y, int z) {
  int r = z;
  r = b(a(c(y), r), x+1);
  return r + 1;
}
void main() {
  int s = d(5, 2, 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. Arguments should be pushed right-to-left so that the first argument is closest to the frame of the called method. It is ok to also evaluate the arguments from right to left.

Write down the x86 code generated by the compiler for the `d` 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) The execution starts by a call to `main`. Draw the situation on the stack when the execution has reached the location indicated by `** PC **`.

Your drawing should include stack frames, stack pointer, frame pointer, dynamic links, parameters and local variables. Temporary values do not have to be drawn. Include the actual values known at that point in execution, including dynamic links, and mark which frame is which. The drawing should be consistent with the code in problem 4 a). (5p)