Examination in Compilers, EDAN65

Department of Computer Science, Lund University

2022–10–25, 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 has two kinds of identifiers: `SID`s and `ID`s. A `SID` consists of a single lowercase letter, like `a`, `b`, etc. An `ID` consists of lowercase letters, and must be at least two letters long, e.g., `ab`, `aabzcd`, etc. The language also has a number of keywords, e.g., `do`. Whitespace consists of blanks and newlines.

a) Use regular expressions to write down token definitions for `SID`, `ID`, `DO`, and `WHITESPACE`. The usual disambiguation rules of rule priority and longest match apply.

(4p)

b) Draw four small finite automata, one for each of `SID`, `ID`, `DO`, and `WHITESPACE`. Mark the final state of each automaton with the token in question. (4p)

c) Combine the four automata to one NFA by joining their start states, and mark each state with a unique number. (2p)

d) Construct a DFA that is equivalent to the NFA and that has as few states as possible. Mark each DFA state with the state numbers from the corresponding states in the NFA. Mark each final state by the appropriate token, taking disambiguation rules into account if necessary.

(5p)

2 Grammars

Consider the following EBNF grammar for expressions:

```
start → exp EOF
exp → ID | ID "(" [ exp ( "," exp )* ] ")" | exp "+" exp | "(" exp ")"
```

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

```
{ EOF, ID, "(", ")", ",", "+" }
```

a) Construct an equivalent grammar on canonical form. The grammar should include the same nonterminals `start` and `exp`, as well as additional nonterminals in order to eliminate the EBNF constructs. (5p)

b) Draw a parse tree for the following token sequence:

```
"(" ID "(" ID "," ID ")" "+" ID "(" ")" ")" EOF
```

The parse tree should follow your grammar from 2(a) *exactly*, including all terminals and nonterminals.

(5p)

c) Construct the FOLLOW set for the nonterminal `exp`. Prove that each element belongs to FOLLOW by showing derivations from the `exp` symbol, and marking out the element (e.g., by underscoring it). Use the grammar you constructed in 2(a). A derivation should be written in the form

```
start ⇒ ... ⇒ ...
```

where only one nonterminal is replaced in each derivation step.

(5p)

d) Consider the following grammar on EBNF form:

```
start → exp EOF
exp → ID | exp "-" exp | exp "+" exp | "(" exp ")"
```

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)

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

(5p)

3 Program analysis

MinC is a small C-like procedural language, where a program consists of a number of function definitions. Functions can be `void`, or return a value of type `int`. Below, parts of the abstract grammar for MinC is shown.

```
Program ::= Function*;
Function ::= Type IdDecl Param* Block;
Block : Stmt ::= Stmt*;
abstract Stmt;
IfStmt : Stmt ::= Cond:Expr Then:Stmt [Else:Stmt];
Assignment : Stmt ::= IdUse Expr;
ReturnStmt : Stmt ::= [ReturnValue:Expr];
CallStmt : Stmt ::= Func:IdUse Arg:Expr*;
abstract Type;
VoidType : Type;
IntType : Type;
 ...
```

If a function is `void`, the return statements inside it should not return any value. Functions of type `int`, on the other hand, must return a value on each possible execution path through the code.

For example, the following three functions are compile-time incorrect: `f1` will not return any value if the `p<0` condition is false; `f2` lacks a return value in the return statement; `f3` has a return statement with a useless return value.

```
int f1(int p) {
   if (p<0) {
      return 1;
   }
}
```

```
int f2(int p) {
   print(2);
   return;
}
```

```
void f3(int p) {
   print(3);
   return 3;
}
```

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

a) Construct an aspect that computes two boolean attributes:

`ReturnStmt.missingReturnValue()` that is `true` for return statements that lack a value and are located inside a non-void function.

`ReturnStmt.uselessReturnValue()` that is `true` for return statements that have a value and are located inside a void function.

(5p)

b) Construct an aspect that computes a boolean attribute `Function.sufficientReturns()` that is `true` if the function returns a value on each execution path through the function. Note that this attribute is defined for all functions, regardless of if they are void or not, but it will be interesting to access only for functions that are not void. (It's value for void functions does not matter.)

(5p)

4 Code generation and run-time systems

Consider the following program in a Java-like language:

```
class Program {
  public static void main(String[] args) {
    doit(new B());
  }
  static void doit(A a) {
    a.m();
  }
}

class A {
  int x = 0;
  void m() { x = 1; }
  void n() { x = 2; }
}
class B extends A {
  int y = 0;
  void p() { x = 7; }
  void m() { y = 3; }
}
```

The language is implemented using virtual tables, without any optimizations.

a) In the program above, the `main` method calls the `doit` method with a new `B` object. The method `doit` in turn calls the method `m` on that object. Draw the runtime situation right before the return of `m`. Your drawing should include:

- method activations with dynamic links, static links (`"this"` pointers), arguments, and local variables
- frame pointer and stack pointer
- objects with class descriptor links and fields
- class descriptors with virtual tables and pointers to the code

(5p)

b) Translate the method `B.m()` to x86 code. Use only the instructions on the x86 Cheat Sheet.

(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)