Examination in Compilers, EDAN65

Department of Computer Science, Lund University

2020–04–15, 08.00-13.00

This is part 3 of the retake exam. This is an open book exam. Read through the instructions carefully (at `http://fileadmin.cs.lth.se/cs/Education/EDAN65/2019/web/retake.html`).

Remember to:

- Write your name and signature on each solution.
- Name your files with your initials and the problem you are solving.
- Make sure each submitted file has good contrast.

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

1 Lexical analysis

a) An identifier, ID, starts with a lowercase letter, followed by any number of lowercase letters or digits. Define the token ID using a regular expression. (2p)

b) In Java, numeric literals can include underscores to enhance readability. For example, we can write 832_759_234 instead of 832759234. Several underscores in a row are permitted, like 12___34. However, underscores in the beginning and end are not permitted, so, for example, _42 and 42_ are illegal. Use a regular expression to define a token INT for integer literals that may include underscores in this manner. (3p)

c) Construct three minimal finite automata: one for ID, one for INT, and one for the keyword end. Mark all final states with the appropriate token, and number all the states with unique numbers (don't use the same numbers in the different automata). (5p)

d) Construct a minimal DFA that combines the three automata for ID, INT and end, by joining their start states and eliminating nondeterminism. Label each final state by the appropriate token. Assume the usual disambiguation rule of keywords taking priority over identifiers. Number each state in the DFA with the set of corresponding states in the automata from the previous problem. (5p)

2 Context-Free Grammars

Consider the following context-free grammar for statements:

```
p0 : start → stmt EOF
p1 : stmt → "{" stmt "}"
p2 : stmt → stmt stmt
p3 : stmt → ID "=" NUM ";"
p4 : stmt → ε
```

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

```
{ "{", "}", "=", ";", ID, NUM, EOF }
```

a) The grammar is ambiguous. Prove this by constructing two different derivation trees for the same sentence. (5p)

b) Construct an equivalent grammar on EBNF form with as few nonterminals and productions as possible.

(5p)

c) Transform the grammar to an equivalent grammar that is LL(1) and on canonical form.

(5p)

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

(If you didn't manage to solve 2 c), you can instead construct the LL(1) table for the original grammar.)

(5p)

3 Program analysis

Consider the following class declarations in a Java-like object-oriented language. (The example has a static-semantic error, because the D class is missing.)

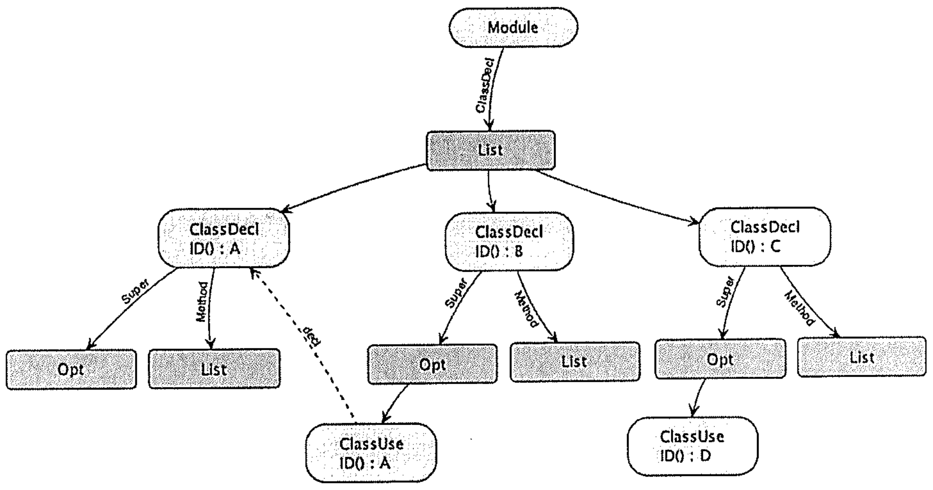
```
class A { ... }
class B extends A { ... }
class C extends D { ... }
```

The following abstract grammar defines part of this language:

```
Module ::= ClassDecl*;
ClassDecl ::= <ID> [Super:ClassUse] MethodDecl*;
ClassUse ::= <ID>;
MethodDecl ::= ...
```

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

a) Define an attribute decl() for ClassUse. The attribute should refer to the referenced ClassDecl if there is one, and should otherwise be null. This is illustrated in the figure below. Note that the value of decl in the use of D is not shown, because it is null.



(5p)

b) Define an attribute `isCyclic()` for `ClassDecl`, that is true if the class has itself on its superclass chain, and false otherwise.

For example, if we have two classes

```
A class B { ... }
B class A { ... }
```

then `isCyclic` would be `true` for both classes.

Hint! Use a circular attribute.

(5p)

c) Define a collection attribute `subclasses()` that contains a set of references to the immediate subclasses of a class. You may use the standard Java class `HashSet` to represent the sets.

(5p)

Turn page for problem 4

4 Code generation and run-time systems

Consider the following C function:

```
int f(int n, int a, int b) {
    int y;
    if (n>0) y = 3;
    else y = b;
    return a*y;
}
```

A C compiler can use an optimization called *constant propagation* to replace the call `f(1, 4, x)` by the value 12.

a) Suppose that `f` is not a C function, but a method in a Java class `A`:

```
class A {
  int f(int n, int a, int b) {
     ...
  }
}
```

Explain why it is difficult for a Java compiler to apply constant propagation for the call `a.f(1, 4, x)` in the method `m` below.

```
static int m(A a, int x) {
    int result;
    result = a.f(1, 4, x); // Can the call be optimized?
    return result;
}
```

(5p)

b) Consider the following main program that calls `m`.

```
static void main() {
    A a = new A();
    int r = m(a, 7);
    print(r);
}
```

Draw the situation on stack and heap right before the method `f` returns (assuming no optimizations are applied). Local variables and arguments should be stored on the stack, but the return value is assumed to be returned in a register. Your drawing should show the frame pointer, dynamic links, local variables, arguments, static link ("this pointer"), and return addresses (where this follows from the code above).
(5p)