Examination in Compilers, EDAN65

Department of Computer Science, Lund University

2021–04–07, 09.00-12.00

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 name and signature on every sheet or 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: 36
For grade 3: Min 18
For grade 4: Min 24
For grade 5: Min 30

Good luck!

1 Lexical analysis

In the language FAVA, a floating point literal consists of an integer part, a decimal point, and a fractional part. The integer and fractional parts are optional, but may not be excluded at the same time. I.e., there must be at least one digit in the literal. The decimal point is not optional – it must always be there. Here are some examples of valid floating point literals in FAVA:

```
3.14
049.
.4
```

a) Construct a regular expression FLOAT for FAVA floating point literals. (3p)

b) Construct a DFA for recognizing FAVA FLOAT tokens. The DFA should be minimal (contain as few states as possible). Mark the final state(s) with FLOAT.

(4p)

c) In a later version of the language, FAVA2, a sequence of digits in a floating point literal may be separated by underscores, to enhance readability. For example, the following is a valid floating point literal in FAVA2:

```
1_903_344.123_45
```

However, underscores are not allowed in the beginning or in the end of the literal, or next to the decimal point. So the following are *not* valid floating point literals in FAVA2:

```
_1.2
3.4_
5_.6
7._8
```

Construct a DFA for recognizing FAVA2 FLOAT tokens. The DFA should be minimal (contain as few states as possible). Mark the final state(s) with FLOAT.

(4p)

2 Grammars

Consider the following context-free grammar with start symbol `E`, and where `ID` is a terminal symbol for identifiers, `ID=[a-z]`.

```
E → E "<" E
E → E "+" E
E → E "-" E
E → "(" E ")"
E → ID
```

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

b) Construct an equivalent unambiguous grammar that would be possible to parse using an LALR parser. The grammar should be on canonical form. The grammar should reflect the usual operator precedence rules for expressions:

- The operators "+" and "-" are left-associative, whereas "<" is nonassociative
- The operators "+" and "-" have higher priority than "<"

(5p)

c) Construct an equivalent grammar that is LL(1). The grammar should be on canonical form. (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.)

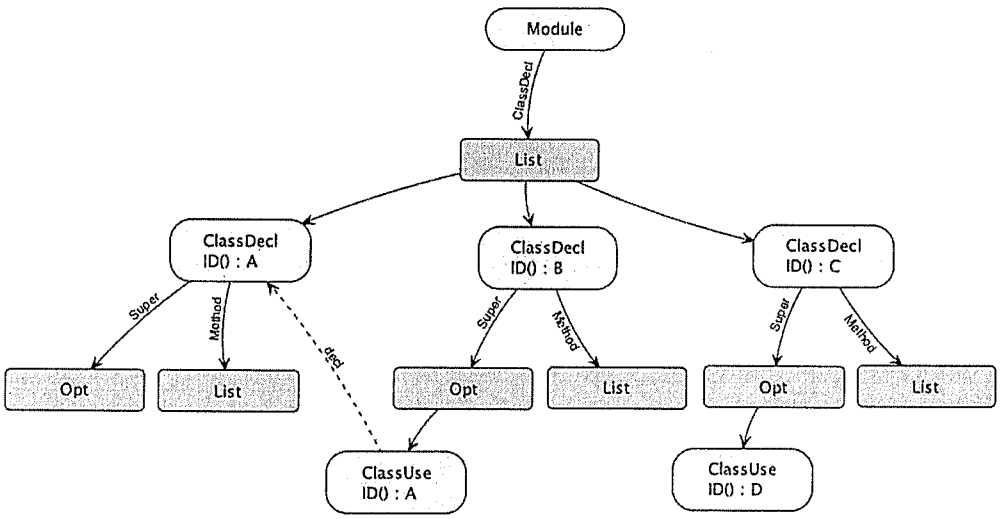
```
class A { ... }
class B extends A { ... }
class C extends D { ... }
```

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`.



(4p)

b) 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.

(2p)

4 Runtime systems

Consider the following program in a C-like language:

```
int b(int x, int y) {
  int z = x+y;
  // ** PC **
  return z;
}
int t(int x) {
  return x*3;
}
int f(int x, int y) {
  int r = 3 + b(t(x+1)+2, y+3);
  return r + 2
}
static void main() {
  int a = f(3, 7);
  print(a);
}
```

A compiler for the language generates unoptimized code that pushes temporaries on the stack. Arguments are passed on the stack. The return value is passed in the `rax` register.

Draw the situation on stack when the execution has reached the location indicated by `** PC **`.

Your drawing should show stack frames, stack pointer, frame pointer, dynamic links, local variables, temporaries, and arguments. Include the actual values known at that point in execution, including dynamic links, and mark which frame is which.

(4p)