Examination in Compilers, EDAN65

Department of Computer Science, Lund University

2022–10–25, 14.00-19.00

SOLUTIONS

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

1 Lexical analysis

a) (4p)

Regular expressions

```
DO  = "do"
SID = [a-z]
ID  = [a-z][a-z]+
WS  = " " | \n
```

Note that `DO` needs to be defined before `ID` because of rule priority.

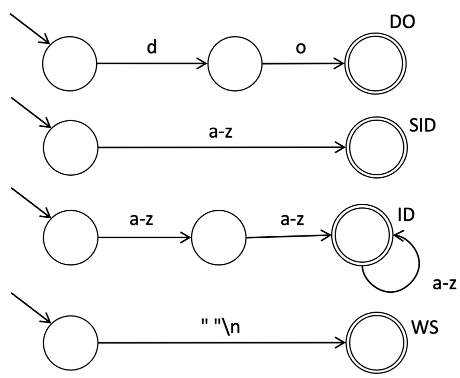
Note that this alternative definition of whitespace would also work fine:

```
WS = (" "|\n)+
```

Note also that using a character class would also work fine to define whitespace. E.g., `[\ \n]` or `[\ \n]+` .

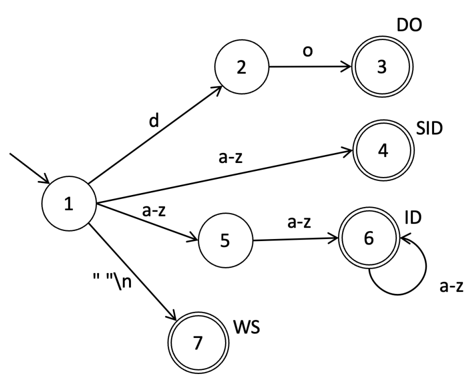
b) (4p)

Finite automata for the four regular expressions.



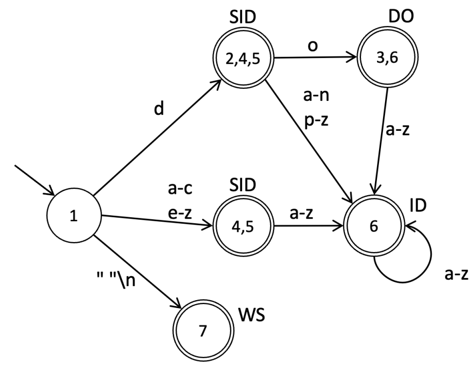
c) (2p)

Combined NFA.



d) (5p)

DFA for the NFA in 1c)



2 Grammars

a) (5p)

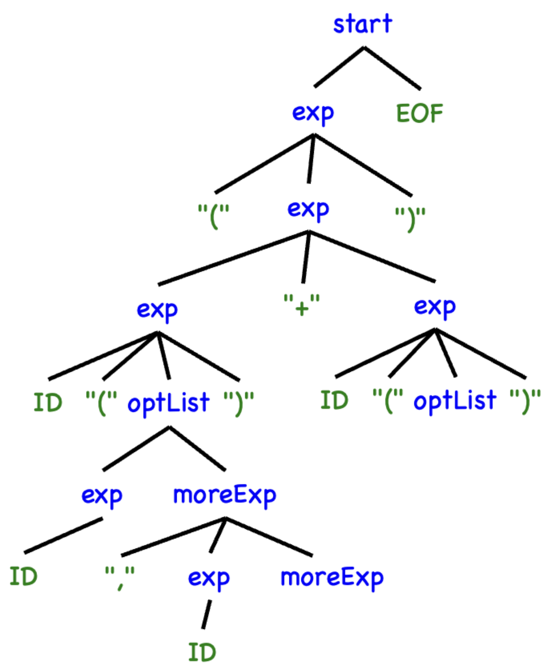
Equivalent grammar on canonical form:

p_0 : `start → exp EOF`
p_1 : `exp → ID`
p_2 : `exp → ID "(" optList ")"`
p_3 : `optList →` ϵ
p_4 : `optList → exp moreExp`
p_5 : `moreExp →` ϵ
p_6 : `moreExp → "," exp moreexp`
p_7 : `exp → exp "+" exp`
p_8 : `exp → "(" exp ")"`

Note that there are several other possible equivalent grammars.

b) (5p)

Parse tree according to the grammar in (a):



c) (5p)

The FOLLOW set for `exp` is

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

To prove that each of these terminals are in the FOLLOW set, we construct derivations from the start symbol that show that each of them can follow directly after an `exp` symbol. We can, for example, construct the following derivations:

```
start ⇒ exp EOF
start ⇒ exp EOF ⇒ "(" exp ")" EOF
start ⇒ exp EOF ⇒ exp "+" exp EOF
start ⇒ exp EOF
      ⇒ ID "(" optList ")" EOF
      ⇒ ID "(" exp moreExp ")" EOF
      ⇒ ID "(" exp "," exp moreExp ")" EOF
```

d) (5p)

Equivalent LL(1) grammar:

p_0 : `start → exp EOF`
p_1 : `exp → term exprest`
p_2 : `exprest → "-" exp`
p_3 : `exprest → "+" exp`
p_4 : `exprest →` ϵ
p_5 : `term → ID`
p_6 : `term → "(" exp ")"`

Note that there are several other equivalent LL(1) grammars. Details for how to arrive at this particular solution:

We start by rewriting the original grammar to canonical form:

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

We see now that there are ambiguities in the grammar, due to the productions

```
exp → exp "-" exp
exp → exp "+" exp
```

The ambiguities can be eliminated by replacing one of the `exp` operators in the binary expressions with a more restricted nonterminal, `term`. We choose to replace the left operand so that we introduce right recursion instead of left recursion. To restrict what a term can be, we change the two last productions to go from `term` instead of from `exp`. We also need to make it possible to derive a `term` from an `exp`, so we add that production. We now get:

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

We see now that there is a common prefix in the grammar. It needs to be eliminated to make the grammar LL(1). We do this by introducing a new nonterminal, `exprest`, for the remainder after the common prefix, resulting in the solution given earlier. We cannot see any obvious LL(1) problems in the solution grammar, but to be certain that it is LL(1), we would need to construct the LL(1) table, as will be done in (e).

e) (5p)

The LL(1) table:

	EOF	"-"	"+"	ID	"("	")"
start				p0	p0	
exp				p1	p1	
exprest	p4	p2	p3			p4
term				p5	p6	

Since there is no conflict, the grammar is LL(1).

3 Program analysis

a) (5p)

Attribute grammar:

```
inh Type ReturnStmt.funcType();
eq  Function.getChild().funcType() = getType();

syn boolean ReturnStmt.missingReturnValue() =
        !hasReturnValue() && !funcType().isVoid();
syn boolean ReturnStmt.uselessReturnValue() =
        hasReturnValue() && funcType().isVoid();

syn boolean Type.isVoid() = false;
eq  VoidType.isVoid() = true;
```

b) (5p)

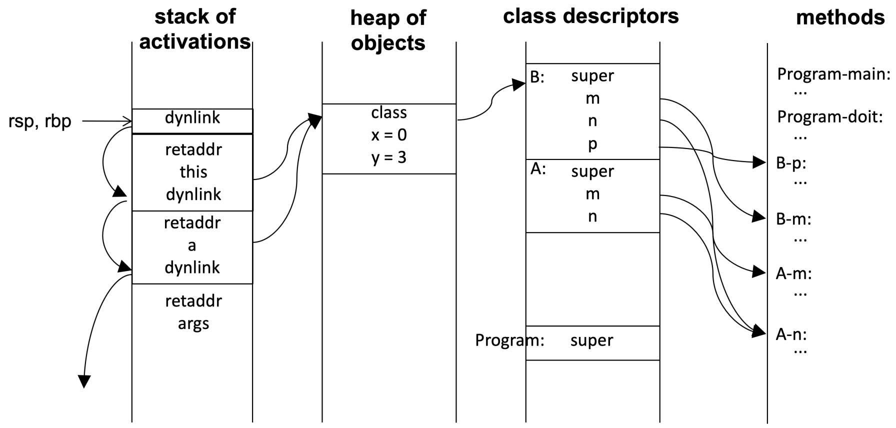
Attribute grammar:

```
syn boolean Function.sufficientReturns() = getBlock().sufficientReturns();
syn boolean Stmt.sufficientReturns() = false;
eq  Block.sufficientReturns() {
      for (Stmt s : getStmts()) {
          if (s.sufficientReturns()) return true;
      }
      return false;
    }
eq ReturnStmt.sufficientReturns() = true;
eq IfStmt.sufficientReturns() =
      hasElse() && getThen().sufficientReturns() && getElse().sufficientReturns();
```

4 Code generation and run-time systems

a) (5p)

The situation at runtime:



b) (5p)

Addresses used in `B.m`:

this object	16(%rbp)
y field	16(%rbp) + 16

x86 code for `B.m`:

```
B.m:
    pushq %rbp           # push old frame pointer (the new dynamic link)
    movq %rsp, %rbp      # set new frame pointer
    movq 16(%rbp), %rax  # address of this object -> rax
    movq $3, 16(%rax)    # 3 -> y
    movq %rbp, %rsp      # Move back stack pointer
    popq %rbp            # Restore the frame pointer
    ret                  # Return to calling method
```