Examination in Compilers, EDAN65

Department of Computer Science, Lund University

2020–10–28, 09.00-12.00

SOLUTIONS

Max points: 36
For grade 3: Min 18
For grade 4: Min 24
For grade 5: Min 30

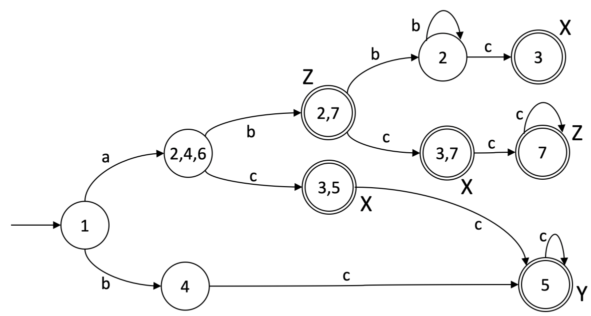
1 Lexical analysis

a) (3p)

```
X = ab*c
Y = [ab]c+
Z = abc*
```

b) (5p)

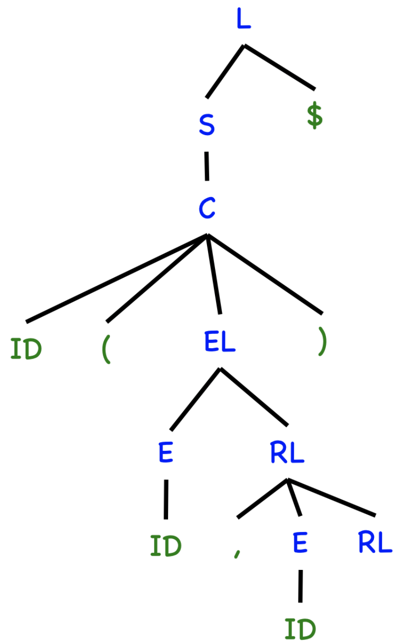
The DFA



2 Grammars

a) (3p)

Parse tree for `ID(ID,ID)$`



b) (4p)

EBNF:

```
L → S $
S → A | C
A → ID "=" E
C → ID "(" [ E ("," E)* ] ")"
E → ID | C
```

c) (5p)

We note that there are common prefixes in the grammar, one for S and one for E:

```
S => A => ID ...
S => C => ID ...

E => ID
E => C => ID ...
```

We can eliminate these common prefixes, resulting in the following grammar:

```
L → S $
S → ID SR
SR → "=" E
SR → "(" EL ")"
E → ID ER
ER → ϵ
ER → "(" EL ")"
EL → ϵ
EL → E RL
RL → ϵ
RL → "," E RL
```

We do not spot any more obvious common prefixes, nor any left recursion, nor any ambiguities. So we have reason to believe that the grammar is LL(1). To be really sure, however, we would need to construct the LL(1) table. In doing so, we see there are no conflicts, so the grammar is LL(1).

d) (5p)

LR parsing sequence with `stack • input ; action` in each step

```
• ID "(" ID "," ID ")" $       ; SHIFT
ID • "(" ID "," ID ")" $       ; SHIFT
ID "(" • ID "," ID ")" $       ; SHIFT
ID "(" ID • "," ID ")" $       ; REDUCE E → ID
ID "(" E • "," ID ")" $        ; SHIFT
ID "(" E "," • ID ")" $        ; SHIFT
ID "(" E "," ID • ")" $        ; REDUCE E → ID
ID "(" E "," E • ")" $         ; REDUCE RL → ϵ
ID "(" E "," E RL • ")" $      ; REDUCE RL → "," E RL
ID "(" E RL • ")" $            ; REDUCE EL → E RL
ID "(" EL • ")" $              ; SHIFT
ID "(" EL ")" • $              ; REDUCE C → ID "(" EL ")"
C • $                          ; REDUCE S → C
S • $                          ; ACCEPT
```

3 Program analysis

a) (4p)

Implementation of the attributes:

```
syn Block Program.localLookupBlock(String s) {
  for (Block b : getBlockList())
    if (b.getBId().equals(s)) return b;
  return null;
}
syn Port Block.localLookupPort(String s) {
  for (Port p : getPortList())
    if (p.getPId().equals(s)) return p;
  return null;
}
```

b) (5p)

Attribute grammar computing PortUse.port():

```
syn Port PortUse.port(){
  if (block() != null)
    return block().localLookupPort(getPId());
  else
    return null;
}
syn Block PortUse.block() = lookupBlock(getBId());
inh Block PortUse.lookupBlock(String s);
eq Program.getConnector().lookupBlock(String s) = localLookupBlock(s);
```

Here, the equation for `PortUse.port` can be rewritten to the following more elegant implementation, using a conditional expression:

```
syn Port PortUse.port() =
  (block() != null) ? block().localLookupPort(getPId()) : null;
```

There are other possible implementations that would work. For example, an inherited attribute for the `Program` root could be introduced, to let the `PortUse` directly call `localLookupBlock`. This solution, however, does not follow the lookup pattern, and makes the specification difficult to extend. For example, if we add nested blocks to the language, such a solution would need to be replaced rather than just extended.

4 Runtime systems

(2p)

A root pointer is a variable on the stack (or in global data) that points to an object on the heap. The garbage collector uses the root pointers to determine which objects are live, i.e., the ones reachable (directly or transitively) from a root pointer. The garbage collector can use this information to reclaim dead areas on the heap (to allocate new objects there), and for compacting the heap (move the live objects together to a contiguous area).