Examination in Compilers, EDAN65

Department of Computer Science, Lund University

2018–10–30, 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) (5p)

Strings of length 5 or shorter that belong to the language.

```
ac
abc
abbc
abbbc
bc
aabc
aaabc
```

b) (5p)

Regular expression:

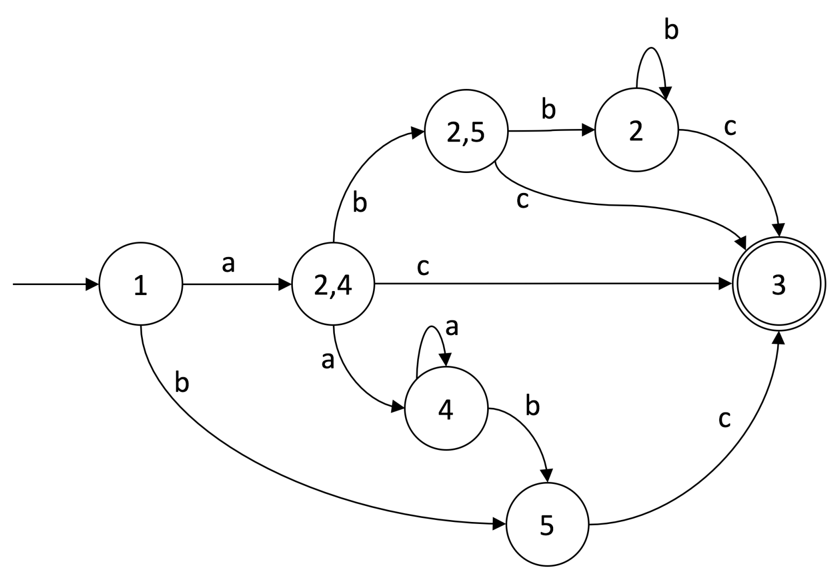
```
ab*c | a*bc
```

or

```
(ab* | a*b) c
```

c) (5p)

DFA:

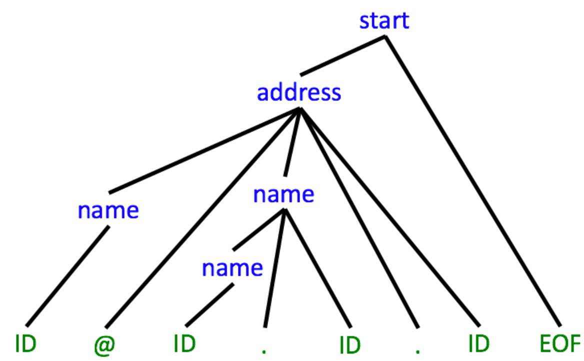


Note! It would be possible to minimze this DFA by joining the states {2,5} and {2}.

2 Context-Free Grammars

a) (5p)

Example parse tree:



b) (5p)

An equivalent grammar on EBNF form, with as few nonterminals as possible:

p_0 : `start → ID ("." ID)* "@" ID ("." ID)+ EOF`

Note! It does not matter if the iterations are done using left or right associativity. Another possible solution would be, for example:

p_0 : `start → (ID ".")* ID "@" (ID ".")+ ID EOF`

c) (5p)

The LL(1) table:

	"@"	"."	ID	EOF
start			p0	
address			p1	
name			p2,p3	

Since there is a conflict, the grammar is not LL(1).

d) (5p)

We see immediately that the grammar is not LL(1) since it has a left recursion in p_3. To eliminate it, we can note that `name "." ID` generates the same language as `ID "." name`, so we can rewrite p_3, resulting in the following grammar:

p_0 : `start → address EOF`
p_1 : `address → name "@" name "." ID`
p_2 : `name → ID`
p_3 : `name → ID "." name`

There is now a common prefix for the nonterminal `name` that needs to be eliminated. We can do this by introducing a nonterminal `rest` for the remainder:

p_0 : `start → address EOF`
p_1 : `address → name "@" name "." ID`
p_2 : `name → ID rest`
p_3 : `rest → "." name`
p_4 : `rest →` ϵ

To check if this grammar is LL(1), we can construct an LL(1) table for it. We can do this by computing FIRST for all symbols, then observing that `rest` is the only nullable symbol, so we also compute FOLLOW for `rest`:

```
FIRST(start) = {ID}
FIRST(address) = {ID}
FIRST(name) = {ID}
FIRST(rest) = {"."}
FOLLOW(rest) = {"@", "."}
```

We can now construct the LL(1) table:

```
          "@"     "."     ID     EOF
          ---------------------------------
start                     p0
address                   p1
name                      p2
rest      p4      p3,p4
```

As we see, there is a conflict in the table, so the grammar is still not LL(1). The problem is that the second `name` in p_1 will expand to `ID rest`, and an LL(1) parser cannot determine if that `rest` nonterminal should be expanded using p_3 or p_4 when the lookahead is `"."`.

We can make the grammar LL(1) by again noting that `name "." ID` generates the same language as `ID "." name`, and transform the production for `address`, resulting in the following grammar:

p_0 : `start → address EOF`
p_1 : `address → name "@" ID "." name`
p_2 : `name → ID rest`
p_3 : `rest → "." name`
p_4 : `rest →` ϵ

We then construct the LL(1) table for this grammar:

```
          "@"     "."     ID     EOF
          ---------------------------------
start                     p0
address                   p1
name                      p2
rest      p4      p3             p4
```

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

3 Program analysis

a) (5p)

Attribute grammar:

```
inh int Action.prevItems();
eq RobotProgram.getAList().prevItems() = 0;
eq AList1.getTail().prevItems() = getHead().items();

syn int Action.items();
eq Walk.items() = prevItems();
eq Pick.items() = min(prevItems() + 1, 3);
eq Drop.items() = max(prevItems() - 1, 0);
```

b) (5p)

Three are several solution approaches to this problem. One possibility is to introduce an attribute `items()` on AList. To implement this attribute, one solution is to introduce `prevItems()` also on AList:

```
syn int RobotProgram.items() = getAList().items();

syn int AList.items();
eq AList0.items() = prevItems();
eq AList1.items() = getTail().items();

inh int AList.prevItems();
eq AList1.getHead().prevItems() = prevItems();
```

(Note that the last equation is actually not needed, since the equation one node up for `AList1.getTail().prevItems()` in problem (a) provides the same value.)

An alternative solution is to implement `items()` on AList by asking the tail if it is the last action:

```
syn int RobotProgram.items() = getAList().items();

syn int AList.items();
eq AList0.items() = 0;
eq AList1.items() =
    getTail().isLast()?
        getHead().items():
        getTail().items();

syn boolean AList.isLast() = false;
eq AList0.isLast() = true;
```

A third alternative solution is to implement a reference attribute `last` on AList that returns the last element of the list, and introduce the `prevItems()` attribute on AList:

```
syn int RobotProgram.items() = getAList().last().prevItems();
syn AList AList.last() = this;
eq AList1.last() = getTail().last();
inh int AList.prevItems();
```

c) (5p)

Attribute grammar:

```
inh RobotProgram Action.root();
eq RobotProgram.getChild().root() = this;

coll Counter RobotProgram.failedPicksCounter()
    [new Counter()]
    with add
    root RobotProgram;

Pick contributes 1
when prevItems() == 3
to RobotProgram.failedPicksCounter() for root();

syn int RobotProgram.failedPicks() = failedPicksCounter().count();
```

4 Code generation and run-time systems

a) (5p)

Assembly code:

(Note that the result from the first call needs to be saved on the stack. It cannot be saved in a register, since the next call would in that case destroy the value in that register.)

```
fib:              # fib method
  push rbp          # push dynamic link
  mov rsp rbp       # set new base pointer
  mov 16(rbp) rax   # n -> rax
  mov $1 rbx        # 1 -> rbx
  cmp rbx rax       # compare n and 1
  jg fib_if_end     # jump if greater, to end of if-statement
  mov 16(rbp) rax     # n -> rax
  jmp fib_return      # jump to return-part
fib_if_end:         # end-if:
  mov 16(rbp) rax     # n -> rax
  mov $1 rbx          # 1 -> rbx
  sub rbx rax         # rax - rbx -> rax
  push rax            # push arg
  call fib            # first recursive call
  pop rbx             # pop arg
  push rax            # push first result on temp stack
  mov 16(rbp) rax     # n -> rax
  mov $2 rbx          # 2 -> rax
  sub rbx rax         # n - 2 -> rax
  push rax            # push arg
  call fib            # second recursive call
  pop rbx             # pop arg
  mov rax rbx         # second result -> rbx
  pop rax             # pop first result -> rax
  add rbx rax         # add results -> rax
fib_return:        # return-part:
  pop rbp             # reset base pointer to caller frame
  ret                 # return


Address table:
--------------
n   16(rbp)
```

b) (5p)

Stack at first call to `fib(0)`: