Examination in Compilers, EDAN65

Department of Computer Science, Lund University

2024–10–29, 14.00-19.00

SOLUTIONS

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

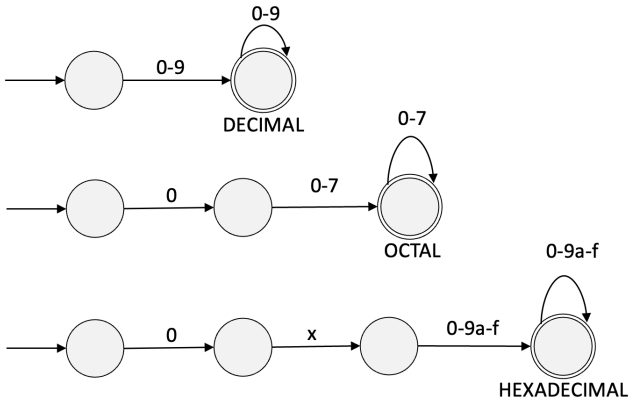
1 Lexical analysis

a) (3p)

The examples `00` and `007` fit the regular expressions for both `DECIMAL` and `OCTAL`. By placing the rule for `OCTAL` before the rule for `DECIMAL`, these two examples will be matched with `OCTAL`.

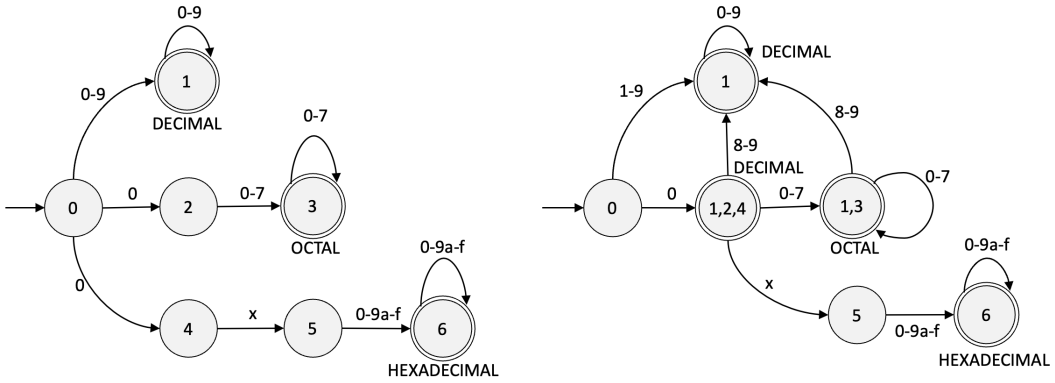
b) (3p)

The DFAs for the regular expressions:



c) (6p)

Combined NFA to the left. Equivalent DFA to the right.



d) (3p)

As alternatives to `HEXADECIMAL("0x19")`, the scanner could return either `DECIMAL("0")` or `HEXADECIMAL("0x1")` as the first token.

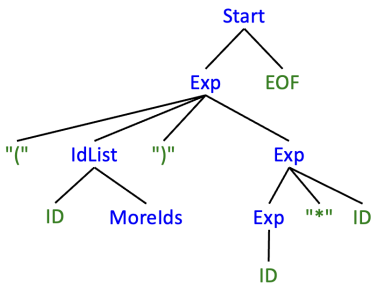
2 Context-Free Grammars

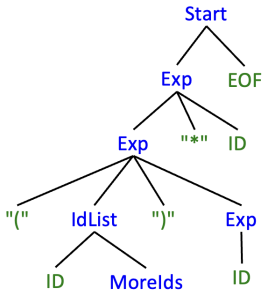
a) (5p)

An example sentence is

```
"(" ID ")" ID "*" ID EOF
```

For this sentence, the following two parse trees can be constructed:





b) (5p)

The following equivalent unambiguous grammar gives parse trees that are more similar to the left tree above (i.e., lambdas can have multiplications as subtrees, but not the other way around):

p_0: `Start -> Exp EOF`
p_1: `Exp -> "(" IdList ")" Exp`
p_2: `IdList -> ID MoreIds`
p_3: `MoreIds ->` ϵ
p_4: `MoreIds -> "," ID MoreIds`
p_5: `Exp -> Factor`
p_6: `Factor -> Factor "*" ID`
p_7: `Factor -> ID`

Another equivalent unambiguous grammar is the following, which gives parse trees more similar to the right-hand tree above (i.e., multiplications can have lambdas as subtrees, but not the other way around):

p_0: `Start -> Exp EOF`
p_1: `Exp -> Exp "*" ID`
p_2: `Exp -> Factor`
p_3: `Factor -> "(" IdList ")" Factor`
p_4: `IdList -> ID MoreIds`
p_5: `MoreIds ->` ϵ
p_6: `MoreIds -> "," ID MoreIds`
p_7: `Factor -> ID`

c) (5p)

First, we can realize that productions $p_2 - p_4$ can be simplified by using repetition instead of recursion as follows:

```
IdList -> ID ( "," ID)*
```

If we then inline `IdList` and write productions p_1, p_5, and p_6 as alternatives for `Exp`, we get the following equivalent EBNF grammar with only two nonterminals:

```
Start -> Exp EOF
Exp -> "(" ID ("," ID)* ")" Exp | Exp "*" ID | ID
```

Alternative solutions

It is possible to eliminate recursion also for `Exp`, and replace recursion with repetition. To do this, we can realize that the token sequence resulting from an `Exp` will consist of zero or more

```
"(" IdList ")"
```

followed by a single

```
ID
```

and then followed by zero or more

```
"*" ID
```

Therefore, `Exp` can be rewritten as:

```
Exp -> ( "(" ID ("," ID)* ")" )*    ID    ("*" ID)*
```

Since `Exp` no longer is recursive, it can be inlined in `Start`, and we then get the following equivalent EBNF grammar with only one nonterminal:

```
Start -> ( "(" ID ("," ID)* ")" )*    ID    ( "*" ID)*    EOF
```

This is arguably not very easy to read, so fewer nonterminals is not necessarily preferable. If you managed to construct this solution, you will, however, get an extra bonus point!

d) (5p)

The LL(1) parser table

	EOF	"("	")"	ID	","	"*"
Start		p0		p0		
Exp		p1,p5		p5,p6		
IdList				p2		
MoreIds			p3		p4	

3 Program analysis

a) (5p)

Attribute grammar for `Action.numItemsBefore()` and `Action.numItemsAfter()`:

```
inh int Action.numItemsBefore();
syn int Action.numItemsAfter();

eq Program.getActionList().numItemsBefore() = 0;
eq ActionList1.getTail().numItemsBefore() = getHead().numItemsAfter();

eq Forward.numItemsAfter() = numItemsBefore();
eq Pick.numItemsAfter()    = min(numItemsBefore() + 1, 3);
eq Place.numItemsAfter()   = max(numItemsBefore() - 1, 0);
```

b) (5p)

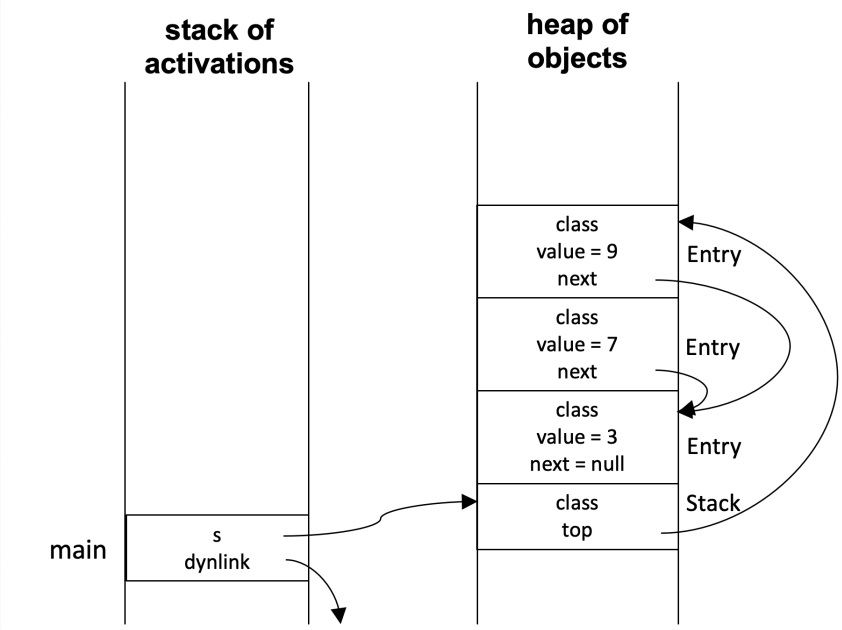
Attribute grammar for `Program.failedPicks()`:

```
coll Counter Program.failedPicksCount();

Pick contributes 1
when numItemsBefore() == 3
to Program.failedPicksCount();

syn int Program.failedPicks() = failedPicksCount().count();
```

4 Code generation and run-time systems

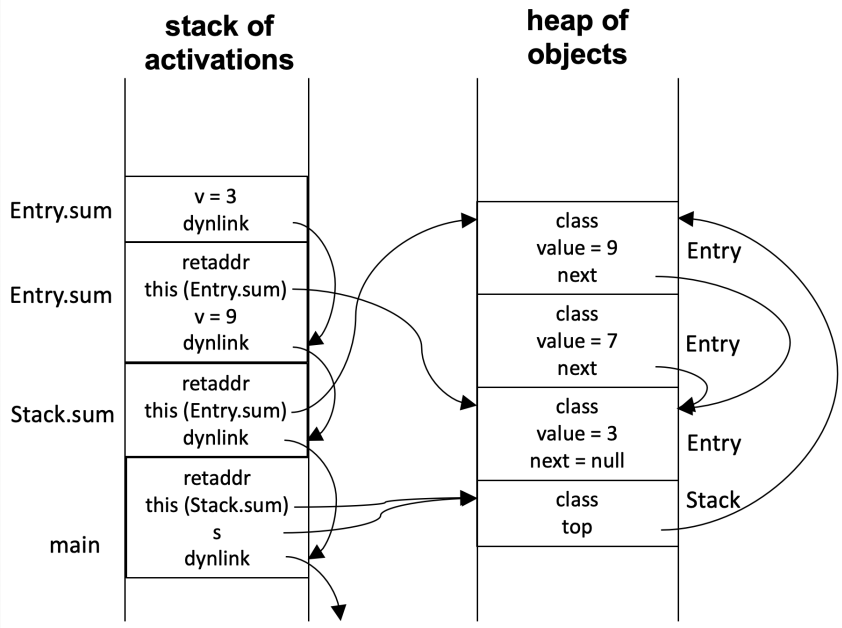
a) (5p)

The situation at *** PC1 ***



b) (5p)

The situation at *** PC2 ***



c) (5p)

The drawing enhanced with root pointers (R) as well as dead (D) and live (L) objects.