Department of Information and Computing Sciences
Utrecht University

INFOB3TC – Exam

Johan Jeuring

Monday, 31 January 2011, 09:00–11:30, EDUC-GAMMA

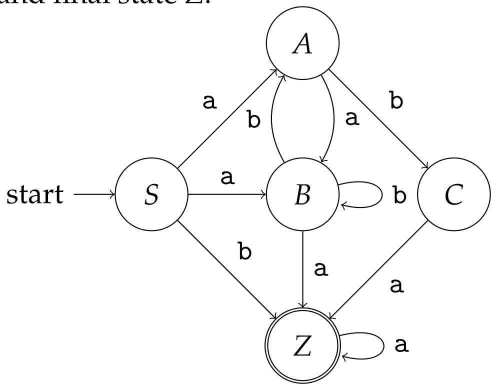
Preliminaries

- The exam consists of 13 pages (including this page). Please verify that you got all the pages.
- Write your **name** and **student number** on all submitted work. Also include the total number of separate sheets of paper.
- For each task, the maximum score is stated. The total amount of points you can get is 90.
- Try to give simple and concise answers. Write readable. Do not use pencils or pens with red ink.
- You may answer questions in Dutch, English, or Swedish.
- When writing Haskell code, you may use Prelude functions and functions from the *Data.List*, *Data.Maybe*, *Data.Map*, *Control.Monad* modules. Also, you may use all the parser combinators from the `uu-tc` package. If in doubt whether a certain function is allowed, please ask.

Good luck!

Regular grammars, NFAs, DFAs

Consider the following NFA (Nondeterministic Finite-state Automaton), with start state S, and final state Z.



1 (6 points)**.** Construct a regular grammar with the same language. •

2 (6 points)**.** Construct a DFA (Deterministic Finite-state Automaton) with the same language (you may draw a DFA). •

3 (6 points)**.** Suppose we have two context-free grammars $G_1 = (T_1, N_1, R_1, S_1)$ and $G_2 = (T_2, N_2, R_2, S_2)$, where the intersection of N_1 and N_2 is empty. Define $G = (T_1 \cup T_2, N_1 \cup N_2 \cup \{S\}, R_1 \cup R_2 \cup \{S \rightarrow S_1\ S_2\}, S)$, where S is the new startsymbol.

(a) What is the language of G?

(b) This construction does not work for regular grammars. Why not?

(c) Describe the construction of a grammar with the same language as G, which is regular if both G_1 and G_2 are regular.

•

Pumping lemmas

The language of sequences of nested pairs of brackets consists of sequences of open and close brackets that are well nested. Examples of sentences in this language are:

```
(((())))
(()())()
```

The empty sentence is also an element of this language.

4 (4 points)**.** Show that the language of nested pairs of brackets is context-free. •

5 (5 points)**.** The regular pumping lemma is useful in showing that a language does *not* belong to the family of regular languages. Its application is typical of pumping lemmas in general; it is used negatively to show that a given language does not belong to the family of regular languages. Give this negative version of the regular pumping lemma, which you can use to prove that a language is not regular. •

6 (9 points)**.** The language of sequences of nested pairs of brackets can be specified as follows: the string s belongs to the language if and only if:

- no prefix of s has fewer open brackets than close brackets,
- the numbers of open and close brackets in s are the same.

Prove that the language of sequences of nested pairs of brackets is not regular. •

LL and LR parsing

Consider the following context-free grammar with startsymbol S, terminals $\{\texttt{a}, \texttt{b}, \texttt{c}\}$, and productions:

$$\begin{aligned}
&S \rightarrow D\,\texttt{a}\,E\\
&D \rightarrow \texttt{b}\,SD\\
&D \rightarrow \varepsilon\\
&E \rightarrow D\\
&E \rightarrow \texttt{c}
\end{aligned}$$

7 (8 points)**.** Determine the empty property, and the first and follow sets for each of the nonterminals of the above grammar. •

8 (3 points)**.** Using empty, first, and follow, determine the lookahead set of each production in the above grammar. •

9 (3 points)**.** Is the above grammar LL(1)? Explain how you can determine this using the lookahead sets of the productions. •

10 (4 points)**.** The string `baca` is a sentence of the above grammar. Show how an LL(1) parser recognizes this string by using a stack. Show step by step the contents of the stack, the part of the input that has not been consumed yet, and which action you perform. If the above grammar is not LL(1), point at the step where different choices can be made. •

Consider the context-free grammar:

$$\begin{aligned}
&S \rightarrow AS\\
&S \rightarrow \texttt{b}\\
&A \rightarrow SA\\
&A \rightarrow \texttt{a}
\end{aligned}$$

We want to use an LR parsing algorithm to parse sentences from this grammar. We start with extending the grammar with a new start-symbol S', and a production

$$S' \rightarrow S\ \$$$

where \$ is a terminal symbol denoting the end of input.

11 (9 points)**.** Construct the LR(0) automaton for the extended grammar. •

12 (3 points)**.** This grammar is not LR(0). Explain why. •

13 (3 points)**.** The string `bab $` is a sentence of the above grammar. Show how an LR(0)-based parser recognizes this string by using a stack. Show step by step the contents of the stack mixed with the states in the LR(0) automaton you pass through, the part of the input that has not been consumed yet, and which action you perform. Explain at which step(s) different choices can be made. •

14 (3 points)**.** The extended grammar is SLR(1). Give the SLR(1) action table for this grammar. You do not have to give the complete table, but you do have to give the actions for the states in which conflicts appear. •

Code generation

15 (18 points)**.** The essential components of the third lab exercise are included below. Solve the 'additional task' 8 of the lab exercise, that is: include a **for** statement in the source language, and add functionality to compile a **for** statement. Here is an example of a **for** statement:

```
for (n=0; n<10; n++)
{ do something }
```

You can assume that the three components between parentheses are expressions, and that doing something is achieved by means of a block of statements. Or you make a different choice, but make sure you document your choice.

Annotate the text of the lab with positions, and give the code you have to add to these positions in order to also compile **for** statements. **Fill out your name on the exam/lab text as well!** •

JavaLex.hs

```
module JavaLex where

import Data.Char
import Control.Monad
import ParseLib.Abstract

data Token = POpen    | PClose        -- parentheses     ()
           | SOpen    | SClose        -- square brackets []
           | COpen    | CClose        -- curly braces    {}
           | Comma    | Semicolon
           | KeyIf    | KeyElse
           | KeyWhile | KeyReturn
           | KeyTry   | KeyCatch
           | KeyClass | KeyVoid
           | StdType   String         -- the 8 standard types
           | Operator  String         -- the 15 operators
           | UpperId   String         -- uppercase identifiers
           | LowerId   String         -- lowercase identifiers
           | ConstInt  Int
           | ConstBool Bool
           deriving (Eq, Show)

keyword :: String -> Parser Char String
keyword []                    = succeed ""
keyword xs@(x:_) | isLetter x = do
                                  ys <- greedy (satisfy isAlphaNum)
                                  guard (xs == ys)
                                  return ys
                 | otherwise  = token xs

greedyChoice :: [Parser s a] -> Parser s a
greedyChoice = foldr (<<|>) empty

terminals :: [(Token, String)]
terminals =
    [( POpen     , "("      )
    ,( PClose    , ")"      )
    ,( SOpen     , "["      )
    ,( SClose    , "]"      )
    ,( COpen     , "{"      )
    ,( CClose    , "}"      )
    ,( Comma     , ","      )
    ,( Semicolon , ";"      )
    ,( KeyIf     , "if"     )
    ,( KeyElse   , "else"   )
    ,( KeyWhile  , "while"  )
    ,( KeyReturn , "return" )
    ,( KeyTry    , "try"    )
    ,( KeyCatch  , "catch"  )
    ,( KeyClass  , "class"  )
    ,( KeyVoid   , "void"   )
    ]


lexWhiteSpace :: Parser Char String
lexWhiteSpace = greedy (satisfy isSpace)

lexLowerId :: Parser Char Token
lexLowerId =  (\x xs -> LowerId (x:xs))
          <$> satisfy isLower
          <*> greedy (satisfy isAlphaNum)

lexUpperId :: Parser Char Token
lexUpperId =  (\x xs -> UpperId (x:xs))
          <$> satisfy isUpper
          <*> greedy (satisfy isAlphaNum)

lexConstInt :: Parser Char Token
lexConstInt = (ConstInt . read) <$> greedy1 (satisfy isDigit)

lexEnum :: (String -> Token) -> [String] -> Parser Char Token
lexEnum f xs = f <$> choice (map keyword xs)

lexTerminal :: Parser Char Token
lexTerminal = choice (map (\ (t,s) -> t <$ keyword s) terminals)

stdTypes :: [String]
stdTypes = ["int", "long", "double", "float",
            "byte", "short", "boolean", "char"]

operators :: [String]
operators = ["+", "-", "*", "/", "%", "&&", "||",
             "^", "<=", "<", ">=", ">", "==",
             "!=", "="]


lexToken :: Parser Char Token
lexToken = greedyChoice
             [ lexTerminal
             , lexEnum StdType stdTypes
             , lexEnum Operator operators
             , lexConstInt
             , lexLowerId
             , lexUpperId
             ]

lexicalScanner :: Parser Char [Token]
lexicalScanner = lexWhiteSpace *> greedy (lexToken <* lexWhiteSpace) <* eof


sStdType :: Parser Token Token
sStdType = satisfy isStdType
       where isStdType (StdType _) = True
             isStdType _           = False

sUpperId :: Parser Token Token
sUpperId = satisfy isUpperId
       where isUpperId (UpperId _) = True
             isUpperId _           = False
```

```
sLowerId :: Parser Token Token
sLowerId = satisfy isLowerId
       where isLowerId (LowerId _) = True
             isLowerId _           = False

sConst :: Parser Token Token
sConst  = satisfy isConst
       where isConst (ConstInt  _) = True
             isConst (ConstBool _) = True
             isConst _             = False

sOperator :: Parser Token Token
sOperator = satisfy isOperator
       where isOperator (Operator _) = True
             isOperator _            = False


sSemi :: Parser Token Token
sSemi =  symbol Semicolon
```

JavaGram.hs

```
module JavaGram where

import ParseLib.Abstract hiding (braced, bracketed, parenthesised)
import JavaLex


data Class = Class     Token [Member]
          deriving Show

data Member = MemberD  Decl
            | MemberM  Type Token [Decl] Stat
          deriving Show

data Stat = StatDecl   Decl
          | StatExpr   Expr
          | StatIf     Expr Stat Stat
          | StatWhile  Expr Stat
          | StatReturn Expr
          | StatBlock  [Stat]
          deriving Show

data Expr = ExprConst  Token
          | ExprVar    Token
          | ExprOper   Token Expr Expr
          deriving Show

data Decl = Decl       Type Token
          deriving Show

data Type = TypeVoid
          | TypePrim   Token
          | TypeObj    Token
          | TypeArray  Type
          deriving (Eq,Show)

parenthesised p = pack (symbol POpen) p (symbol PClose)
bracketed     p = pack (symbol SOpen) p (symbol SClose)
braced        p = pack (symbol COpen) p (symbol CClose)

pExprSimple :: Parser Token Expr
pExprSimple =  ExprConst <$> sConst
           <|> ExprVar   <$> sLowerId
           <|> parenthesised pExpr

pExpr :: Parser Token Expr
pExpr = chainr pExprSimple (ExprOper <$> sOperator)



pMember :: Parser Token Member
pMember =   MemberD <$> pDeclSemi
        <|> pMeth

pStatDecl :: Parser Token Stat
pStatDecl =   pStat
          <|> StatDecl <$> pDeclSemi

pStat :: Parser Token Stat
pStat =  StatExpr
         <$> pExpr
         <*  sSemi
     <|> StatIf
         <$  symbol KeyIf
         <*> parenthesised pExpr
         <*> pStat
         <*> option ((\_ x -> x) <$> symbol KeyElse <*> pStat) (StatBlock [])
     <|> StatWhile
         <$  symbol KeyWhile
         <*> parenthesised pExpr
         <*> pStat
     <|> StatReturn
         <$  symbol KeyReturn
         <*> pExpr
         <*  sSemi
     <|> pBlock


pBlock :: Parser Token Stat
pBlock  =  StatBlock
           <$> braced( many pStatDecl )



pMeth :: Parser Token Member
pMeth =  MemberM
         <$> (   pType
             <|> const TypeVoid <$> symbol KeyVoid
             )
         <*> sLowerId
         <*> parenthesised (option (listOf pDecl
                                           (symbol Comma)
                                   )
                                   []
                           )
         <*> pBlock

pType0 :: Parser Token Type
pType0 =  TypePrim <$> sStdType
      <|> TypeObj  <$> sUpperId

pType :: Parser Token Type
pType = foldr (const TypeArray)
        <$> pType0
        <*> many (bracketed (succeed ()))


pDecl :: Parser Token Decl
pDecl = Decl
        <$> pType
        <*> sLowerId

pDeclSemi :: Parser Token Decl
pDeclSemi = const <$> pDecl <*> sSemi

pClass :: Parser Token Class
pClass = Class
        <$  symbol KeyClass
        <*> sUpperId
        <*> braced ( many pMember )
```

JavaAlgebra.hs

```
module JavaAlgebra where

import JavaLex
import JavaGram

type JavaAlgebra clas memb stat expr
  = (  ( Token -> [memb]                 -> clas
       )
    ,  ( Decl                            -> memb
       , Type -> Token -> [Decl] -> stat -> memb
       )
    ,  ( Decl                            -> stat
       , expr                            -> stat
       , expr -> stat -> stat            -> stat
       , expr -> stat                    -> stat
       , expr                            -> stat
       , [stat]                          -> stat
       )
    ,  ( Token                           -> expr
       , Token                           -> expr
       , Token -> expr -> expr           -> expr
       )
    )


foldJava :: JavaAlgebra clas memb stat expr -> Class -> clas
foldJava ((c1),(m1,m2),(s1,s2,s3,s4,s5,s6),(e1,e2,e3)) = fClas
 where fClas (Class      c ms)     = c1 c (map fMemb ms)
       fMemb (MemberD    d)        = m1 d
       fMemb (MemberM    t m ps s) = m2 t m ps (fStat s)
       fStat (StatDecl   d)        = s1 d
       fStat (StatExpr   e)        = s2 (fExpr e)
       fStat (StatIf     e s1 s2)  = s3 (fExpr e) (fStat s1) (fStat s2)
       fStat (StatWhile  e s1)     = s4 (fExpr e) (fStat s1)
       fStat (StatReturn e)        = s5 (fExpr e)
       fStat (StatBlock  ss)       = s6 (map fStat ss)
       fExpr (ExprConst  con)      = e1 con
       fExpr (ExprVar    var)      = e2 var
       fExpr (ExprOper   op e1 e2) = e3 op (fExpr e1) (fExpr e2)
```

JavaCode.hs

```haskell
module JavaCode where

import Prelude hiding (LT, GT, EQ)
import Data.Map as M
import JavaLex
import JavaGram
import JavaAlgebra
import SSM

data ValueOrAddress = Value | Address
  deriving Show

codeAlgebra :: JavaAlgebra Code
                           Code
                           Code
                           (ValueOrAddress -> Code)

codeAlgebra = ( (fClas)
              , (fMembDecl,fMembMeth)
              , (fStatDecl,fStatExpr,fStatIf,fStatWhile,fStatReturn,fStatBlock)
              , (fExprCon,fExprVar,fExprOp)
              )
 where
 fClas       c ms     = [Bsr "main", HALT] ++ concat ms

 fMembDecl   d        = []
 fMembMeth   t m ps s = case m of
                          LowerId x -> [LABEL x] ++ s ++ [RET]

 fStatDecl   d        = []
 fStatExpr   e        = e Value ++ [pop]
 fStatIf     e s1 s2  = let c  = e Value
                            n1 = codeSize s1
                            n2 = codeSize s2
                        in  c ++ [BRF (n1 + 2)] ++ s1 ++ [BRA n2] ++ s2
 fStatWhile  e s1     = let c = e Value
                            n = codeSize s1
                            k = codeSize c
                        in  [BRA n] ++ s1 ++ c ++ [BRT (-(n + k + 2))]
 fStatReturn e        = e Value ++ [pop] ++ [RET]
 fStatBlock  ss       = concat ss

 fExprCon    c        va = case c of
                              ConstInt n -> [LDC n]
 fExprVar    v        va = case v of
                             LowerId x -> let loc = 37
                                          in  case va of
                                                Value    ->  [LDL  loc]
                                                Address  ->  [LDLA loc]
 fExprOp     o e1 e2  va =
   case o of
     Operator "=" -> e2 Value ++ [LDS 0] ++ e1 Address  ++ [STA 0]
     Operator op  -> e1 Value ++ e2 Value ++ [opCodes ! op]

opCodes :: Map String Instr
opCodes
 = fromList
     [ ( "+" , ADD )
     , ( "-" , SUB )
     , ( "*" , MUL )
     , ( "/" , DIV )
     , ( "%" , MOD )
     , ( "<=", LE  )
     , ( ">=", GE  )
     , ( "<" , LT  )
     , ( ">" , GT  )
     , ( "==", EQ  )
     , ( "!=", NE  )
     , ( "&&", AND )
     , ( "||", OR  )
     , ( "^" , XOR )
     ]
```

SSM.hs

```
module SSM where

data Reg = PC | SP | MP | R3 | R4 | R5 | R6 | R7
   deriving Show

r0, r1, r2, r3, r4, r5, r6, r7 :: Reg
r0 = PC
r1 = SP
r2 = MP
r3 = R3
r4 = R4
r5 = R5
r6 = R6
r7 = R7
```

```
data Instr
 = STR Reg | STL Int  | STS Int  | STA Int  -- Store from stack
 | LDR Reg | LDL Int  | LDS Int  | LDA Int  -- Load on stack
 | LDC Int | LDLA Int | LDSA Int | LDAA Int -- Load on stack
 | BRA Int | Bra String                     -- Branch always (relative/to label)
 | BRF Int | Brf String                     -- Branch on false
 | BRT Int | Brt String                     -- Branch on true
 | BSR Int | Bsr String                     -- Branch to subroutine
 | ADD | SUB | MUL | DIV | MOD              -- Arithmetical operations on 2 stack operand
 | EQ  | NE  | LT  | LE  | GT  | GE         -- Relational   operations on 2 stack operand
 | AND | OR  | XOR                          -- Bitwise      operations on 2 stack operand
 | NEG | NOT                                --              operations on 1 stack operand
 | RET | UNLINK | LINK Int | AJS Int        -- Procedure utilities
 | SWP | SWPR Reg | SWPRR Reg Reg | LDRR Reg Reg  -- Various swaps
 | JSR | TRAP Int | NOP | HALT              -- Other instructions
 | LABEL String                             -- Pseudo-instruction for generating a label
   deriving Show

pop :: Instr
pop = AJS (-1)

type Code = [Instr]

formatInstr :: Instr -> String
formatInstr (LABEL s)   = s ++ ":"
formatInstr x           = '\t' : show x

formatCode :: Code -> String
formatCode = filter clean . concatMap ((++"\n") . formatInstr)
  where
    clean :: Char -> Bool
    clean x = notElem x "()\""

codeSize :: Code -> Int
codeSize = sum . map instrSize

instrSize :: Instr -> Int
instrSize (LDRR  _ _) =  ...
```