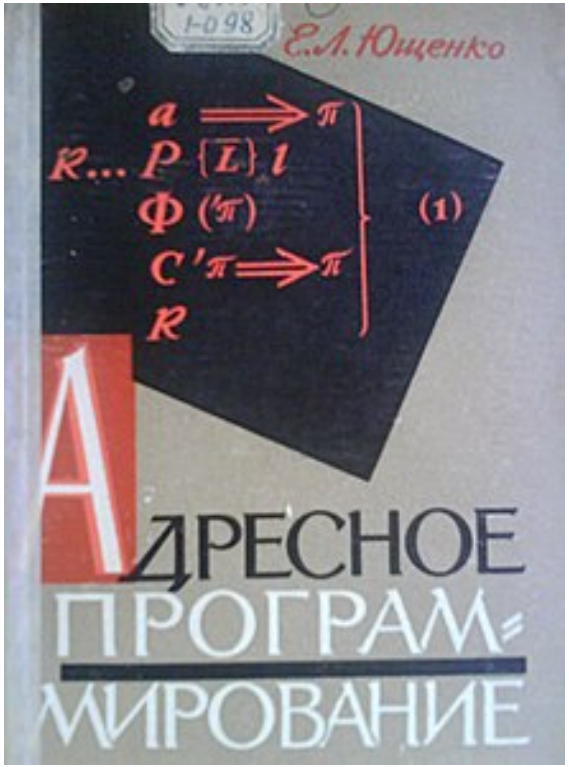
РЕАЛІЗАЦІЯ ІНТЕРПРЕТАТОРА БАЗОВОЇ ЧАСТИНИ АДРЕСНОЇ МОВИ ДЛЯ ОБРОБКИ ДЕРЕВОПОДІБНИХ ФОРМАТІВ

Виконав Санченко Георгій

ВСТУП

- 1950-60 роки - Адресна мова
- Штрих-операція
- Вказівники

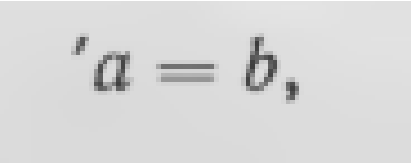


МЕТА ДОСЛІДЖЕННЯ

- Створення специфікації Адресної мови програмування
- Реалізація інтерпретатора за допомогою мови програмування Haskell
- Розробка на Адресній мові прикладів програм, що опрацьовують спискові ланцюжки та деревоподібні формати

ОГЛЯД АДРЕСНОЇ МОВИ

- Штрих-операція
- Мінус штрих-операція
- Відношення слідування



СИНТАКСИС

```
node_addr = 'node
P { node_addr == 0 } Ret |
node_val = '(node_addr + 1)

Pg bst_find_children { node_addr, left, right }

@case_left ...
    P { 'val < node_val } | case_right
    Pg bst_delete { 'val, 'left }
    end
@case_right ...
    P { 'val > node_val } | case_cur
    Pg bst_delete { 'val, 'right }
    end
```

ЛЕКСИЧНИЙ ТА СИНТАКСИЧНИЙ АНАЛІЗ

```
@eol_pattern      { tok' TNewLine }
"("               { tok' TLPar }
")"               { tok' TRPar }
"{"               { tok' TLCurly }
"}"               { tok' TRCurly }
"["               { tok' TLBrack }
"]"               { tok' TRBrack }
"|"               { tok' TBar }
";"               { tok' TSemi }
","               { tok' TComma }
"+"               { tok' TPlus }
"<+>"             { tok' TCirclePlus }
"-"               { tok' TMinus }
"*"               { tok' TTimes }
"/"               { tok' TDiv }
"%"               { tok' TMod }
"=="              { tok' TEq }
"/="              { tok' TNeq }
```

```
-- Expressions
Exp :  Exp "or" Exp          { BinOpApp Or $1 $3 }
    | Exp "and" Exp          { BinOpApp And $1 $3 }
    | Exp "==" Exp           { BinOpApp Equal $1 $3 }
    | Exp "/=" Exp           { BinOpApp NotEqual $1 $3 }
    | Exp "<=" Exp           { BinOpApp LessEqual $1 $3 }
    | Exp ">=" Exp           { BinOpApp GreaterEqual $1 $3 }
    | Exp "<" Exp            { BinOpApp Less $1 $3 }
    | Exp ">" Exp            { BinOpApp Greater $1 $3 }
    | Exp "+" Exp            { BinOpApp Add $1 $3 }
    | Exp "-" Exp            { BinOpApp Sub $1 $3 }
    | Exp "*" Exp            { BinOpApp Mul $1 $3 }
    | Exp "/" Exp            { BinOpApp Div $1 $3 }
    | Exp "%" Exp            { BinOpApp Mod $1 $3 }
    | Exp "<+>" Exp          { BinOpApp PtrAdd $1 $3 }
```

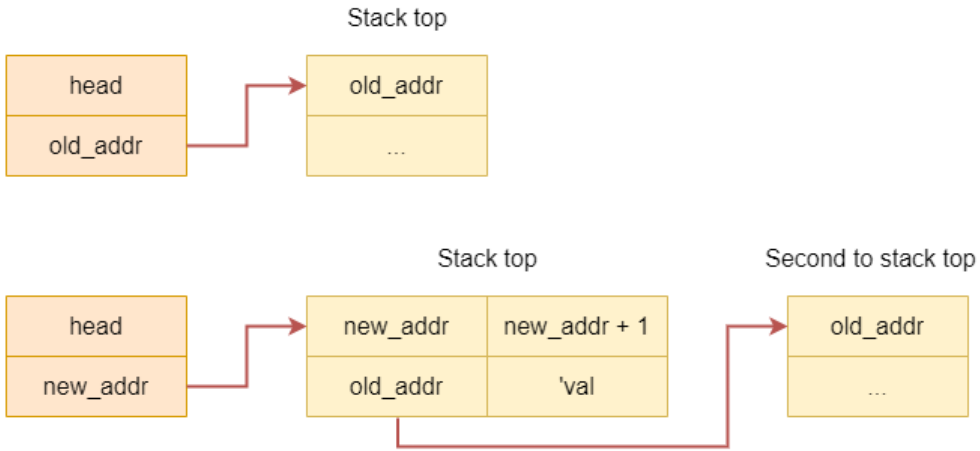
КОМПІЛЯТОР

```
data CompState = CompState
  { curLine :: IORef Int,
    curChunk :: IORef Chunk,
    labelOffsetMap :: IORef LabelOffsetMap,
    jumpPatches :: IORef [(Int, String)],
    loopPatches :: IORef [LoopPatch],
    labelRefPatches :: IORef [(Int, String)],
    csFnVars :: FnVarMap,
    csFnMap :: LineFnMap,
    csProgLines :: [ProgLine],
    csReps :: IORef [Int]
  }
```

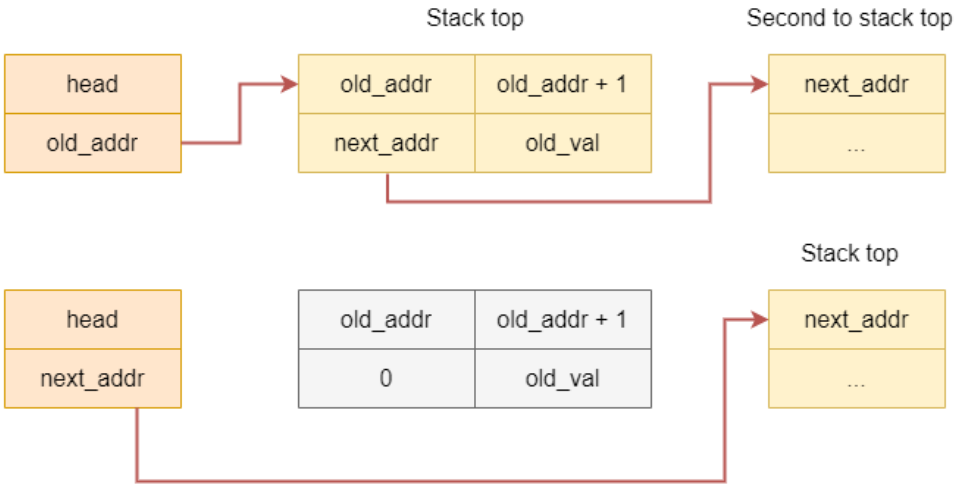
ВІРТУАЛЬНА МАШИНА

```haskell
runVm :: VM -> IO InterpretResult
runVm vm = do
  (Just intRes) <- untilM isJust runStep Nothing
  return intRes
  where
    runStep _ = do
      instr <- toEnum <$> readByte vm
      execInstruction instr vm `catch` handler
      where
        handler (ErrorCallWithLocation msg _) = do
          lineNum <- getCurrentLine vm
          putStrLn $ "Runtime error at line " ++ show lineNum
          putStrLn msg
          return $ Just RUNTIME_ERR
```

СТЕК


Stack top
head
old_addr
old_addr
...
Stack top
Second to stack top
head
new_addr
new_addr
new_addr + 1
old_addr
'val
old_addr
...

СТЕК


Stack top
Second to stack top
head
old_addr
old_addr
old_addr + 1
next_addr
old_val
next_addr
...
Stack top
head
next_addr
old_addr
old_addr + 1
0
old_val
next_addr
...

СПИСОК

```
@map ... Nil => f, Nil => list_addr, Nil => r
    head = 'list_addr
    Pg list_empty { 'r }

    P { 'head == 0 } Ret |

    L { 'head, 'Nil, P { 'pi /= 0 } => pi } l1
        val = '('pi + 1)
        Pg ['f] { val, new_val }
        Pg list_add { 'new_val, 'r }
    @l1 ...
    Ret
```

БІНАРНЕ ДЕРЕВО

```
@bst_insert ... Nil => val, Nil => node
    node_addr = 'node
    node_val = '(node_addr + 1)

    P { 'val == node_val } Ret |

    Pg bst_find_children { node_addr, left, right }

    is_left = 'val < node_val
    P { is_left } cur_child = 'left | cur_child = 'right

    P { cur_child == 0 } case_create | case_insert
    @case_create ...
        Pg bst_create_child { 'val, node_addr, cur_child_new }
        end
    @case_insert ...
        Pg bst_insert { 'val, cur_child }
        end

    @end ...
    Ret
```

ВИСНОВКИ

- Розробка видозміни синтаксису Адресної мови
- Реалізація мовою функційного програмування Haskell інтерпретатора ADPL
- Реалізовано програми обробки спискових ланцюжків та деревоподібних форматів

ДЯКУЮ ЗА УВАГУ!