Реалізація мови процедурного програмування типа Algol-60 засобами Haskell

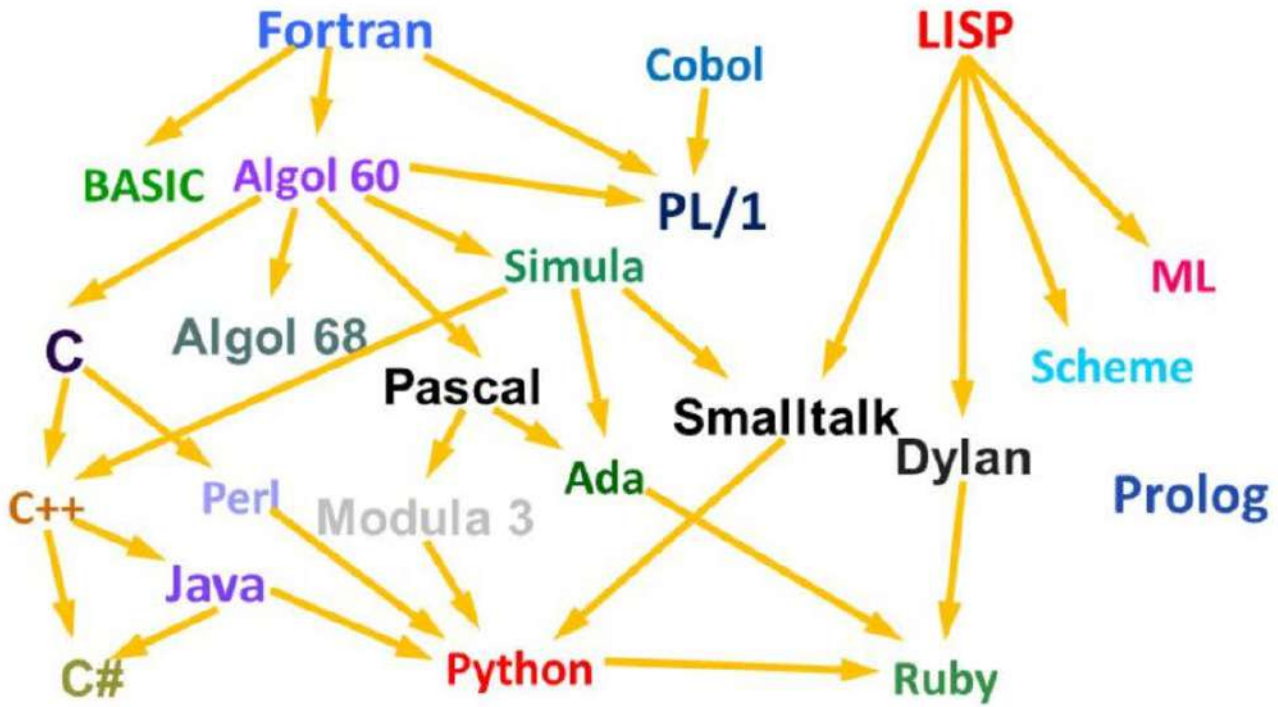
Виконав: Пінкевич В. М.

Науковий керівник: Проценко В. С.

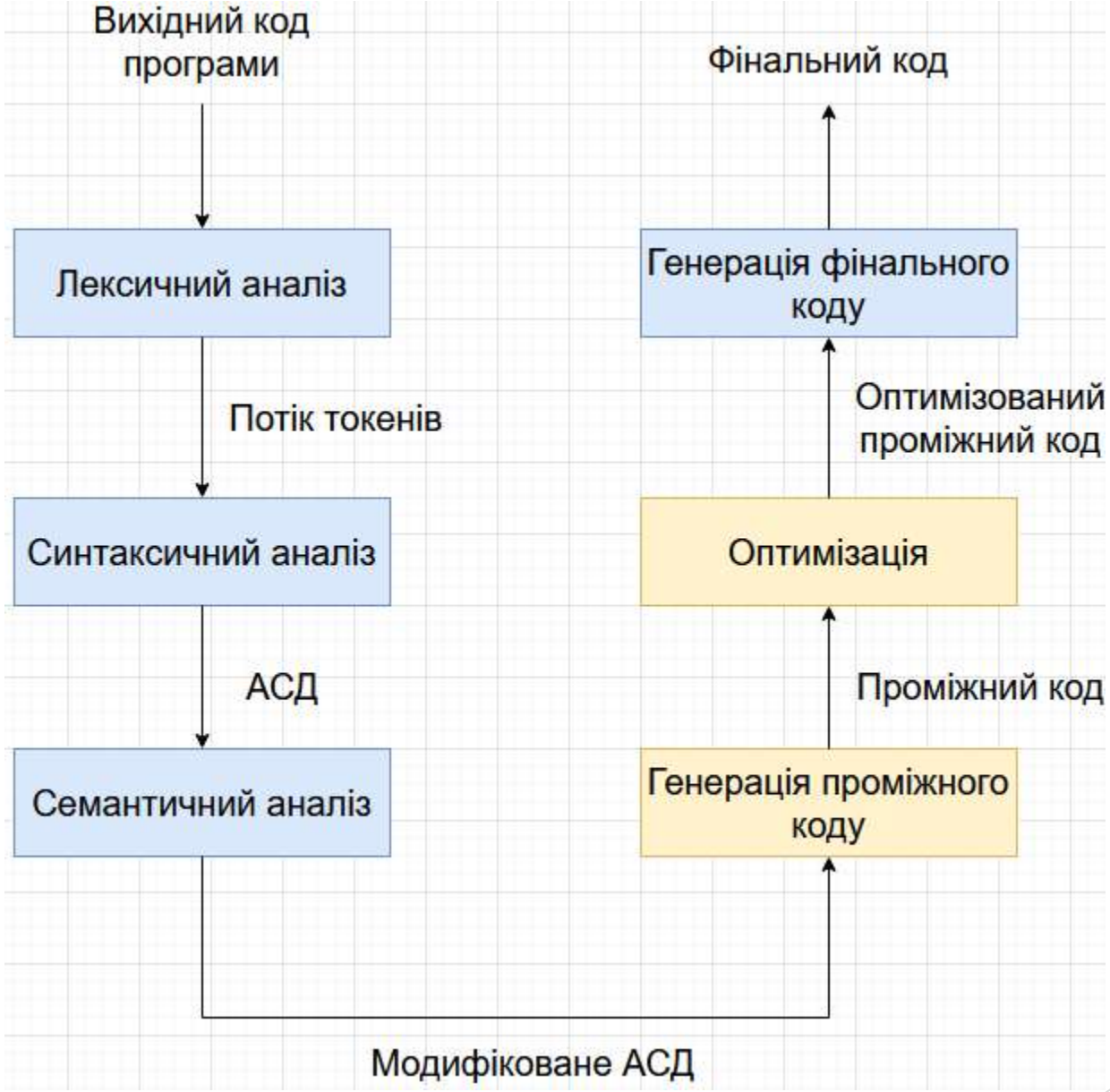
Мета роботи

- Розробка інтерпретатора підмножини елементів мови програмування Pascal;
- Дослідження особливостей імплементації компіляторів та інтерпретаторів залежно від специфічних аспектів цільової мови програмування;
- Дослідження доцільності використання функціональних мов програмування для розробки компіляторів та інтерпретаторів.

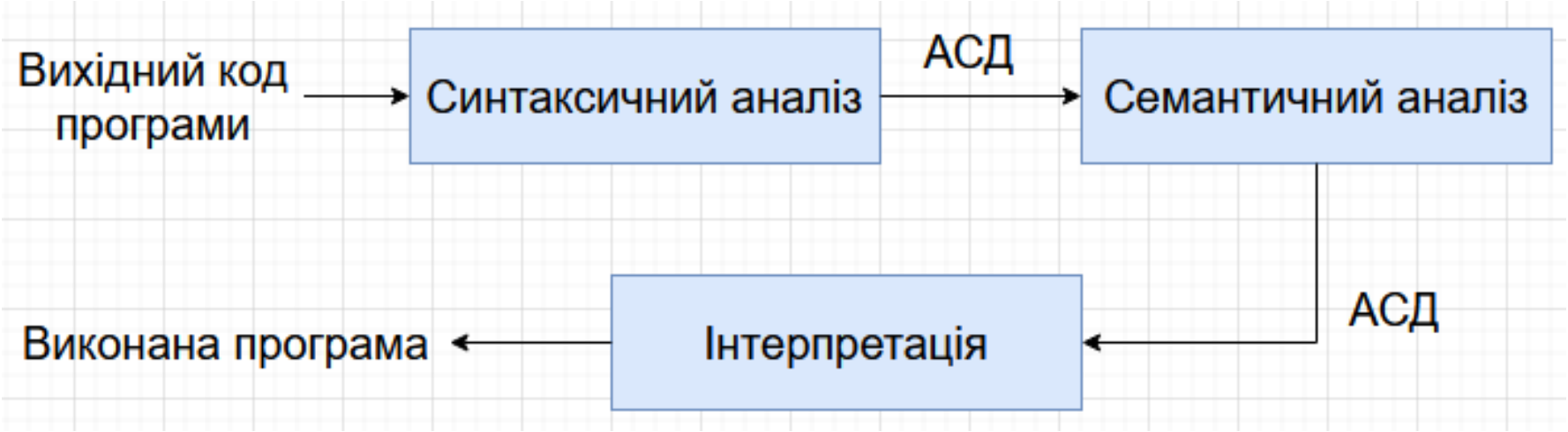
ALGOL-подібні мови програмування



Типові етапи компіляції



Етапи виконання розробленого інтерпретатора



Інструменти розробки інтерпретатора

PARSEC

Обрана підмножина елементів мови Pascal

Визначення: функції, змінні та процедури

Інструкції: оператор присвоєння, умовний оператор “if”, цикли “while” та “for”

Стандартна бібліотека: процедури “read”, “readln”, “write”, “writeln” для роботи з потоками введення-виведення

Типи даних: цілочисельний, тип дійсних чисел, булевий, символьний, стрічковий

Операції: всі унарні та бінарні операції, визначені у Pascal для обраних типів даних.

АСД (модуль Lexic)

```haskell
data Program = Program {pHeader :: Identifier, pBody :: Block} deriving (Show)

data Block = Block {bDeclarations :: [Declaration], bBody :: Statement} deriving (Show)

data Declaration = VarDecl [Variable] | FuncDecl Function | ProcDecl Procedure deriving (Show)

data Variable = Var {vName :: Identifier, vType :: DataType, vValue :: Maybe Value} deriving (Show)

data Function = Function {fName :: Identifier, fParams :: [FormalParam], fResType :: DataType, fBlock :: Block} deriving (Show)

data Procedure = Procedure {pName :: Identifier, pParams :: [FormalParam], pBlock :: Block} deriving (Show)

data FormalParam = FormalParam {fpName :: Identifier, fpType :: DataType} deriving (Show)
```

```haskell
data Statement
  = Assignment {aName :: Identifier, aValue :: Expression}
  | ProcCall {pcName :: Identifier, pcParams :: [Expression]}
  | Compound [Statement]
  | If {iCondition :: Expression, iIfRoute :: Statement, iElseRoute :: Maybe Statement}
  | While {wCondition :: Expression, wBody :: Statement}
  | Repeat {rCondition :: Expression, rBody :: [Statement]}
  deriving (Show)
```

```haskell
data DataType
  = DTBoolean
  | DTInteger
  | DTReal
  | DTChar
  | DTString
  deriving (Show, Eq)
```

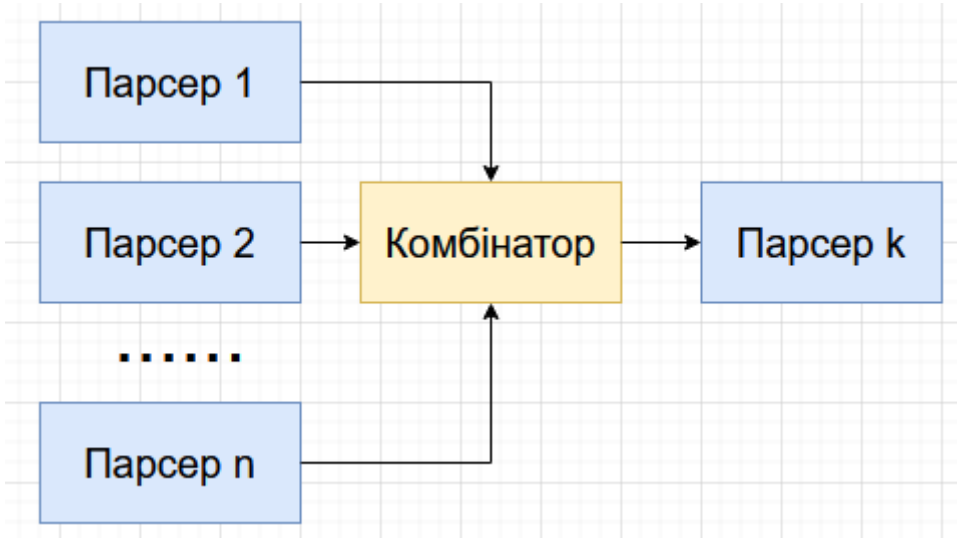
```haskell
data Expression
  = Val Value
  | VarRef Identifier
  | FuncCall {fcName :: Identifier, fcParams :: [Expression]}
  | UnOp UnaryOp Expression
  | BinOp {boOp :: BinaryOp, boLeft :: Expression, boRight :: Expression}
  | Paren Expression
  deriving (Show)
```

```haskell
data Value
  = Boolean Bool
  | IntNum Int
  | RealNum Double
  | Character Char
  | Str String
  deriving (Show)
```

Синтаксичний аналіз (модуль Parser)

- Основна мета – побудова абстрактного синтаксичного дерева на основі вихідного коду програми;

- Використання бібліотеки Parsec та підходу “комбінаторів парсерів”.



Семантичний аналіз (модуль Analyzer)

```haskell
data Scope = Scope
  { variables :: Map.Map String VarInfo,
    functions :: Map.Map String FuncInfo,
    procedures :: Map.Map String ProcInfo,
    scopeLevel :: Int,
    parentScope :: Maybe Scope
  }
  deriving (Show)

data VarInfo = VI {viName :: String, viType :: DataType} deriving (Show)
data FuncInfo = FI {fiName :: String, fiParams :: [ParamInfo], fiResType :: DataType} deriving (Show)
data ProcInfo = PRI {priName :: String, priParams :: [ParamInfo]} deriving (Show)
data ParamInfo = PAI {paiName :: String, paiType :: DataType} deriving (Show)
```

```haskell
data AnalysisError = AnalysisError AnalysisErrorType String (Maybe AnalysisError) deriving (Show)
data AnalysisErrorType
  = IdentifierAlreadyDefinedError
  | VariableDoesNotExistError
  | FunctionDoesNotExistError
  | ProcedureDoesNotExistError
  | ActualParameterError
  | TypeMismatchError
  | FormalParameterDeclarationError
  | VariableDeclarationError
  | FunctionDeclarationError
  | ProcedureDeclarationError
  | AssignmentError
  | ProcedureCallError
  | IfStatementError
  | WhileStatementError
  | RepeatStatementError
  | VariableReferenceError
  | FunctionCallError
  | UnaryOperatorError
  | BinaryOperatorError
  deriving (Show)
```

- Основна мета – аналіз абстрактного синтаксичного дерева для виявлення семантичних помилок.

Інтерпретація (модуль Interpreter)

```haskell
data ScopeRecord = SR
  { srName :: String,
    srLevel :: Int,
    srType :: RecordType,
    parameters :: Maybe [ParamInfo],
    srReturnType :: Maybe DataType,
    variables :: Map String VarInfo,
    functions :: Map String ScopeRecord,
    procedures :: Map String ScopeRecord,
    srBody :: Statement,
    parentSR :: Maybe ScopeRecord
  }

data RecordType = RTProgram | RTFunction | RTProcedure deriving (Show)

data VarInfo = VI {viName :: String, viType :: DataType, viValue :: Maybe Value} deriving (Show)
data ParamInfo = PAI {paiName :: String, paiType :: DataType} deriving (Show)
```

```haskell
data InterpretationError = InterpretationError InterpretationErrorType String (Maybe InterpretationError) deriving (Show)
data InterpretationErrorType
  = WrongTypeError
  | DivisionByZeroError
  | WrongReadProcedureArgumentError
  deriving (Show)
```

```haskell
data ActivationRecord = AR
  { arName :: String,
    arLevel :: Int,
    arType :: RecordType,
    vars :: Map String VarInfo
  }
  deriving (Show)
```

```haskell
data Interpreter = I
  { callStack :: [ActivationRecord],
    currentSR :: ScopeRecord
  }
  deriving (Show)
```

- Основна мета – інтерпретація програми використовуючи абстракте синтаксичне дерево.

Використання інтерпретатора

Команда для виклику інтерпретатора:
<Назва_виконуваного_файлу> run <Шлях_до_файлу> --debug

```
PS D:\Projects\haskell\pascal-interpreter\.stack-work\install\fb26c89a\bin> .\pascal-interpreter-exe.exe
ERROR: Command was not provided
PS D:\Projects\haskell\pascal-interpreter\.stack-work\install\fb26c89a\bin>
```

```
PS D:\Projects\haskell\pascal-interpreter\.stack-work\install\fb26c89a\bin> .\pascal-interpreter-exe.exe random-command
ERROR: Unknown command 'random-command'
PS D:\Projects\haskell\pascal-interpreter\.stack-work\install\fb26c89a\bin>
```

```
PS D:\Projects\haskell\pascal-interpreter\.stack-work\install\fb26c89a\bin> .\pascal-interpreter-exe.exe run
ERROR: expecting file path to run the program
PS D:\Projects\haskell\pascal-interpreter\.stack-work\install\fb26c89a\bin>
```

```
PS D:\Projects\haskell\pascal-interpreter\.stack-work\install\fb26c89a\bin> .\pascal-interpreter-exe.exe run ./random-fi
le-path.pas
pascal-interpreter-exe.exe: ./random-file-path.pas: openFile: does not exist (No such file or directory)
PS D:\Projects\haskell\pascal-interpreter\.stack-work\install\fb26c89a\bin>
```

Програма для обрахування заданої кількості чисел Фібоначчі

```
program Fibonacci;

function calculateFibonacci(position: Integer): Integer;
begin
if position < 0 then
  calculateFibonacci := -1
else if position = 0 then
  calculateFibonacci := 0
else if position < 3 then
  calculateFibonacci := 1
else
  calculateFibonacci := calculateFibonacci(position - 1) +
                        calculateFibonacci(position - 2);
end;

var index: Integer;
var amount: Integer;

begin
  write('Please, enter the required number of Fibonacci numbers to calculate: ');
  read(amount);

  index := 0;
  while (index <= amount) do
    begin
      writeln('Fibonacci number at position ', index, ' - ', calculateFibonacci(index));
      index := index + 1;
    end;
end.
```

Приклади виконання програми

```
PS D:\Projects\haskell\pascal-interpreter\.stack-work\install\fb26c89a\bin> .\pascal-interpreter-exe.exe run D:\Projects
\haskell\pascal-interpreter\sources\fibonacci.pas
Interpreting source file: 'D:\Projects\haskell\pascal-interpreter\sources\fibonacci.pas'...

Please, enter the required number of Fibonacci numbers to calculate: 10
Fibonacci number at position 0 - 0
Fibonacci number at position 1 - 1
Fibonacci number at position 2 - 1
Fibonacci number at position 3 - 2
Fibonacci number at position 4 - 3
Fibonacci number at position 5 - 5
Fibonacci number at position 6 - 8
Fibonacci number at position 7 - 13
Fibonacci number at position 8 - 21
Fibonacci number at position 9 - 34
Fibonacci number at position 10 - 55
PS D:\Projects\haskell\pascal-interpreter\.stack-work\install\fb26c89a\bin>
```

```
PS D:\Projects\haskell\pascal-interpreter\.stack-work\install\fb26c89a\bin> .\pascal-interpreter-exe.exe run D:\Projects
\haskell\pascal-interpreter\sources\fibonacci.pas
Interpreting source file: 'D:\Projects\haskell\pascal-interpreter\sources\fibonacci.pas'...

Please, enter the required number of Fibonacci numbers to calculate: 13abc
InterpretationError-[type = WrongReadProcedureArgumentError, message = Wrong input! Can't read value of type 'Integer' f
rom the input: "13abc"]
PS D:\Projects\haskell\pascal-interpreter\.stack-work\install\fb26c89a\bin>
```

Приклад виконання програми в розширеному режимі

```
PS D:\Projects\haskell\pascal-interpreter\.stack-work\install\fb26c89a\bin> .\pascal-interpreter-exe.exe run D:\Projects\haskell\pascal-interpreter\sources\fibonacci.pas --debug
Interpreting source file: 'D:\Projects\haskell\pascal-interpreter\sources\fibonacci.pas'...

Parsing program...
Program {pHeader = Identifier {idValue = "Fibonacci"}, pBody = Block {bDeclarations = [FuncDecl (Function {fName = Identifier {idValue = "calculateFibonacci"}, fParams = [FormalParam {fpName = Identifier {idVa
lue = "position"}, fpType = DTInteger}], fResType = DTInteger, fBlock = Block {bDeclarations = [], bBody = Compound [If {iCondition = BinOp {boOp = Lt, boLeft = VarRef (Identifier {idValue = "position"}), boRi
ght = Val (IntNum 0)}, iIfRoute = Assignment {aName = Identifier {idValue = "calculateFibonacci"}, aValue = UnOp UnaryMinus (Val (IntNum 1))}, iElseRoute = Just (If {iCondition = BinOp {boOp = Eql, boLeft = Va
rRef (Identifier {idValue = "position"}), boRight = Val (IntNum 0)}, iIfRoute = Assignment {aName = Identifier {idValue = "calculateFibonacci"}, aValue = Val (IntNum 0)}, iElseRoute = Just (If {iCondition = Bi
nOp {boOp = Lt, boLeft = VarRef (Identifier {idValue = "position"}), boRight = Val (IntNum 3)}, iIfRoute = Assignment {aName = Identifier {idValue = "calculateFibonacci"}, aValue = Val (IntNum 1)}, iElseRoute
= Just (Assignment {aName = Identifier {idValue = "calculateFibonacci"}, aValue = BinOp {boOp = Plus, boLeft = FuncCall {fcName = Identifier {idValue = "calculateFibonacci"}, fcParams = [BinOp {boOp = Minus, b
oLeft = VarRef (Identifier {idValue = "position"}), boRight = Val (IntNum 1)}]}, boRight = FuncCall {fcName = Identifier {idValue = "calculateFibonacci"}, fcParams = [BinOp {boOp = Minus, boLeft = VarRef (Iden
tifier {idValue = "position"}), boRight = Val (IntNum 2)}]}}})})})}]}}),VarDecl [Var {vName = Identifier {idValue = "index"}, vType = DTInteger, vValue = Nothing}],VarDecl [Var {vName = Identifier {idValue = "
amount"}, vType = DTInteger, vValue = Nothing}]], bBody = Compound [ProcCall {pcName = Identifier {idValue = "write"}, pcParams = [Val (Str "Please, enter the required number of Fibonacci numbers to calculate:
 ")]},ProcCall {pcName = Identifier {idValue = "read"}, pcParams = [VarRef (Identifier {idValue = "amount"})]},Assignment {aName = Identifier {idValue = "index"}, aValue = Val (IntNum 0)},While {wCondition = P
aren (BinOp {boOp = Lte, boLeft = VarRef (Identifier {idValue = "index"}), boRight = VarRef (Identifier {idValue = "amount"})}), wBody = Compound [ProcCall {pcName = Identifier {idValue = "writeln"}, pcParams
= [Val (Str "Fibonacci number at position "),VarRef (Identifier {idValue = "index"}),Val (Str " - "),FuncCall {fcName = Identifier {idValue = "calculateFibonacci"}, fcParams = [VarRef (Identifier {idValue = "i
ndex"})]}]},Assignment {aName = Identifier {idValue = "index"}, aValue = BinOp {boOp = Plus, boLeft = VarRef (Identifier {idValue = "index"}), boRight = Val (IntNum 1)}]}]}]}
----------------------------------------
Analyzing parsed program...
A {currentScope = Scope {variables = fromList [("amount",VI {viName = "amount", viType = DTInteger}),("index",VI {viName = "index", viType = DTInteger})], functions = fromList [("calculateFibonacci",FI {fiName
 = "calculateFibonacci", fiParams = [PAI {paiName = "position", paiType = DTInteger}], fiResType = DTInteger})], procedures = fromList [("read",PRI {priName = "read", priParams = []}),("readln",PRI {priName =
"readln", priParams = []}),("write",PRI {priName = "write", priParams = []}),("writeln",PRI {priName = "writeln", priParams = []})], scopeLevel = 1, parentScope = Nothing}, globalScope = Scope {variables = fro
mList [("amount",VI {viName = "amount", viType = DTInteger}),("index",VI {viName = "index", viType = DTInteger})], functions = fromList [("calculateFibonacci",FI {fiName = "calculateFibonacci", fiParams = [PAI
 {paiName = "position", paiType = DTInteger}], fiResType = DTInteger})], procedures = fromList [("read",PRI {priName = "read", priParams = []}),("readln",PRI {priName = "readln", priParams = []}),("write",PRI
{priName = "write", priParams = []}),("writeln",PRI {priName = "writeln", priParams = []})], scopeLevel = 1, parentScope = Nothing}}
----------------------------------------
Interpreting program...
Program output:
Please, enter the required number of Fibonacci numbers to calculate: 10
Fibonacci number at position 0 - 0
Fibonacci number at position 1 - 1
Fibonacci number at position 2 - 1
Fibonacci number at position 3 - 2
Fibonacci number at position 4 - 3
Fibonacci number at position 5 - 5
Fibonacci number at position 6 - 8
Fibonacci number at position 7 - 13
Fibonacci number at position 8 - 21
Fibonacci number at position 9 - 34
Fibonacci number at position 10 - 55

----------------------------------------
Final interpreter state:
I {callStack = [AR {arName = "Fibonacci", arLevel = 1, arType = RTProgram, vars = fromList [("amount",VI {viName = "amount", viType = DTInteger, viValue = Just (IntNum 10)}),("index",VI {viName = "index", viTy
pe = DTInteger, viValue = Just (IntNum 11)})]}], currentSR = SR { srName = Fibonacci, srLevel = 1, srType = RTProgram, parameters = Nothing, srReturnType = Nothing, variables = fromList [("amount",VI {viName =
 "amount", viType = DTInteger, viValue = Nothing}),("index",VI {viName = "index", viType = DTInteger, viValue = Nothing})], functions = fromList [("calculateFibonacci",SR { srName = calculateFibonacci, srLevel
 = 2, srType = RTFunction, parameters = Just [PAI {paiName = "position", paiType = DTInteger}], srReturnType = Just DTInteger, variables = fromList [], functions = fromList [], procedures = fromList [], srBody
 = Compound [If {iCondition = BinOp {boOp = Lt, boLeft = VarRef (Identifier {idValue = "position"}), boRight = Val (IntNum 0)}, iIfRoute = Assignment {aName = Identifier {idValue = "calculateFibonacci"}, aValu
e = UnOp UnaryMinus (Val (IntNum 1))}, iElseRoute = Just (If {iCondition = BinOp {boOp = Eql, boLeft = VarRef (Identifier {idValue = "position"}), boRight = Val (IntNum 0)}, iIfRoute = Assignment {aName = Iden
tifier {idValue = "calculateFibonacci"}, aValue = Val (IntNum 0)}, iElseRoute = Just (If {iCondition = BinOp {boOp = Lt, boLeft = VarRef (Identifier {idValue = "position"}), boRight = Val (IntNum 3)}, iIfRoute
 = Assignment {aName = Identifier {idValue = "calculateFibonacci"}, aValue = Val (IntNum 1)}, iElseRoute = Just (Assignment {aName = Identifier {idValue = "calculateFibonacci"}, aValue = BinOp {boOp = Plus, bo
```

Програма для обрахування заданої кількості чисел Фібоначчі з помилкою

```
program Fibonacci;

function calculateFibonacci(position: Integer): Integer;
begin
if position < 0 then
  calculateFibonacci := -1
else if position = 0 then
  calculateFibonacci := 0
else if position < 3 then
  calculateFibonacci := 1
else
  calculateFibonacci := calculateFibonacci(position - 1) +
                        calculateFibonacci(position - 2);
end;

var index: Boolean;
var amount: Integer;

begin
  write('Please, enter the required number of Fibonacci numbers to calculate: ');
  read(amount);

  index := 0;
  while (index <= amount) do
    begin
      writeln('Fibonacci number at position ', index, ' - ', calculateFibonacci(index));
      index := index + 1;
    end;
end.
```

Результат виконання програми

```
PS D:\Projects\haskell\pascal-interpreter\.stack-work\install\fb26c89a\bin> .\pascal-interpreter-exe.exe run D:\Projects
\haskell\pascal-interpreter\sources\fibonacci.pas
Interpreting source file: 'D:\Projects\haskell\pascal-interpreter\sources\fibonacci.pas'...

AnalysisError-[type = AssignmentError, message = Wrong expression type in assignment statement!, source = AnalysisError-
[type = TypeMismatchError, message = Expression: Val (IntNum 0) has wrong type! Expected one of these types: ["DTBoolean
"]. Actual type: DTInteger]]
PS D:\Projects\haskell\pascal-interpreter\.stack-work\install\fb26c89a\bin>
```

Можливості розширення інтерпретатора

- Підтримка складених типів даних: масивів, множин, записів;
- Додавання операцій над складеними типами даних: перевірка на належність елемента до множини, читання та оновлення елементів масиву, операції для роботи з полями записів;
- Можливість передачі параметрів функцій та процедур за посиланням;
- Розширення стандартної бібліотеки: додавання математичних функцій та процедур для роботи з потоками введення-виведення.

Результати роботи

- Розроблено інтерпретатор підмножини елементів мови Pascal достатньої для виконання значної кількості програм;
- Описано переваги та недоліки розробленого інтерпретатора, а також можливості для його розширення та покращення;
- Проаналізовано принципи розробки компіляторів та інтерпретаторів в залежності від специфічних особливостей цільової мови програмування (у випадку поточної роботи – мови Pascal).

Висновки (1)

- Особливості мови програмування, для якої розробляється компілятор чи інтерпретатор, впливають на кількість етапів компіляції чи інтерпретації, а також на їх імплементацію;
- Використання проміжних структур даних для виконання кожного з етапів компіляції чи інтерпретації дозволяє зробити кожен етап незалежним від інших;
- Використання техніки комбінації парсерів (яка реалізована у бібліотеці Parsec) спрощує розробку синтаксичних аналізаторів та дозволяє описувати їх у вигляді, дуже схожому на опис синтаксису у формі Бекуса-Наура;

Висновки (2)

- Використання функціональних, статично і строго типізованих мов програмування (таких як Haskell) для розробки компіляторів та інтерпретаторів є доцільним, оскільки дозволяє легко представляти складні структури даних (такі як АСД) та синтаксичні структури мови (наприклад, вирази та інструкції), декларативно описувати операції над ними, а також виконувати аналіз та перевірку типів, що спрощує відстеження та усунення помилок при розробці.

Дякую за увагу!