2025_2 - COMPILADORES - METATURMA

Descrição Visualizar envios

AV10 - Verificação de Tipos

Data de entrega: quinta, 23 Out 2025, 23:59
Arquivos requeridos: driver.py, Lexer.py, Parser.py, Expression.py, Visitor.py (Baixar)
Tipo de trabalho: Trabalho individual

O objetivo deste trabalho é adicionar anotações de tipos e um sistema de verificação destas anotações à nossa linguagem de funções anônimas. As anotações de tipo aplicam-se às declarações de variáveis. Nossa linguagem declara variáveis de duas formas. Em blocos `let`, ou em definições `fn`. A figura abaixo ilustra alguns exemplos de anotações de tipo:

Figura 1: uma função é um valor de tipo "Arrow Type". Um "Arrow Type" tem a estrutura "type param -> type body":

```
(fn x:int => x + 1)
<class 'int'> -> <class 'int'>
```

Figure 2: nossa linguagem possui dois tipos primitivos: "int" e "bool":

```
(fn x:int => x * x) 2
<class 'int'>
```

```
(fn x:int => x > 0) 2
<class 'bool'>
```

Figure 3: a aplicação de uma função de tipo T0 -> T1 ao argumento de tipo T0 resulta no tipo T1:

```
(fn x:int => x > 1) 2
<class 'bool'>
```

Figure 4: a linguagem possui dois tipos de declarações: em blocos let ou em funções anônimas:

```
let
  cube: int->int <- fn x:int => x * x * x
in
  cube 4
end
<class 'int'>
```

Figure 5: anotações podem usar parênteses. Em alguns casos, parênteses são úteis para melhorar a legibilidade de programas:

```
let
  cube:(int->int) <- fn x:int => x * x * x
in
  cube 4 + 1
end
<class 'int'>
```

Figure 6: note que toda declaração de variáveis precisa ser anotada. Anotações não são opcionais:

```
(fn x:int->int => fn y:int => x (x y)) (fn a:int => a * a) 4
<class 'int'>
```

```
(fn x => fn y:int => x (x y)) (fn a:int => a * a) 4
Parse error
```

Figure 7: Arrow types se associam à direita. Então "int -> int -> int" é equivalente a int -> (int->int):

```
let
  add:int->int->int <- fn x:int => fn y:int => x + y
in
  add 1
end
<class 'int'> -> <class 'int'>
```

Neste trabalho você tem dois objetivos, a saber:

1. Modificar a sintáxe da linguagem para incorporar anotações de tipo
2. Implementar um Visitor que faça a verificação de tipos.

Abaixo salientamos as mudanças esperadas.

Expressões

Duas expressões serão modificadas: `Let` e `Fn`. Porém, essas modificações já estão implementadas para você. Em outras palavras, você não precisa alterar `Expression.py`. E quais são essas alterações? Agora, tanto `Let` quando `Fn` incorporam o tipo das variáveis que essas expressões declaram:

```
class Fn(Expression):
    def __init__(self, formal, tp_var, body):
        self.formal = formal
        self.tp_var = tp_var
        self.body = body
```

```
class Let(Expression):
    def __init__(self, identifier, tp_var, exp_def, exp_body):
        self.identifier = identifier
        self.tp_var = tp_var
        self.exp_def = exp_def
        self.exp_body = exp_body
```

Note que os métodos inicializadores agora recebem um parâmetro tp_var, que denota o tipo da variável declarada:

```
>>> # Declara f:int => y + x
>>> e0 = Fn('y', type(1), Add(Var('y'), Var('x')))

>>> # Declara let v:int <- 2 in v + 3 end
>>> e = Let('v', type(1), Num(2), Add(Var('v'), Num(3)))
```

Visitor

Você deverá implementar a classe `TypeCheckVisitor`, que faz a verificação de tipos de acordo com as regras da figura 8. Assim como no exercício de funções anônimas, cada método do novo visitor recebe um contexto (*environment*). Porém, este contexto não associa nomes de variáveis à valores. Ele associa nomes de variáveis à tipos:

```
>>> e = Var('t')
>>> ev = TypeCheckVisitor()
>>> e.accept(ev, {'t':type(1)})
<class 'int'>
```

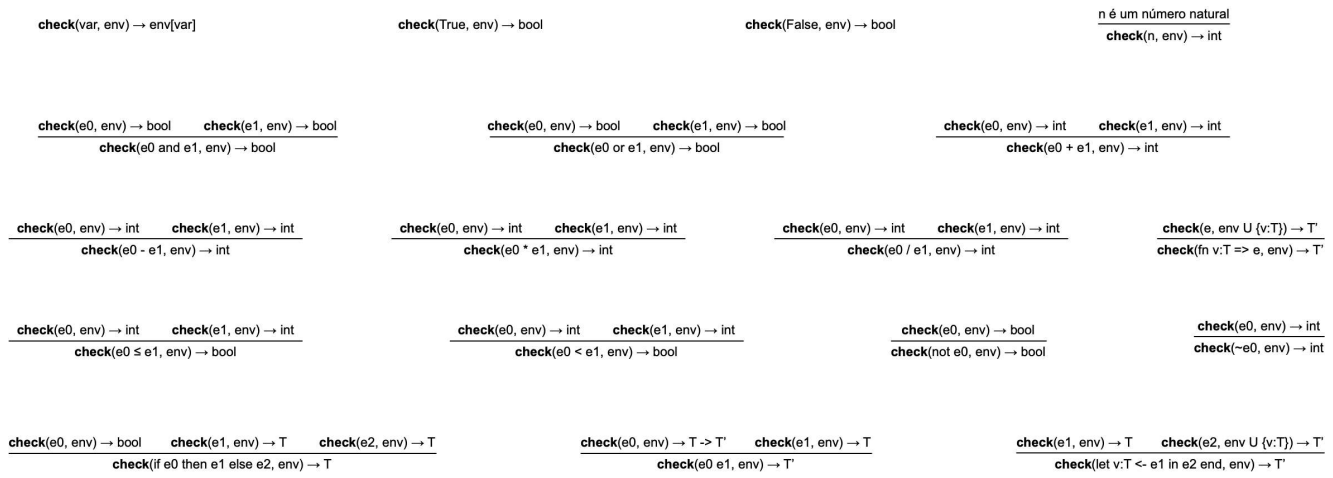

Figure 8: regras de verificação de tipagem. Essas regras devem ser usadas na implementação de `TypeCheckVisitor`.

Análise Léxica

O analisador léxico (`Lexer.py`) deverá ser alterado para incluir quatro novos tokens que participam das anotações de tipos:

1. TPF = 220: A seta que indica o tipo de função (`->`)
2. COL = 221: Os dois pontos que separam o nome da variável da anotação de tipo (`:`)
3. INT = 222: O tipo primitivo int ('`int`')
4. LGC = 223: O tipo primitivo bool ('`bool`')

Os novos tokens já estão declarados em `Lexer.py`. Porém, você deverá estender a implementação deste arquivo feita no exercício de funções anônimas para transformar strings nestes tokens.

Análise Sintática

O analisador sintático (`Parser.py`) deverá ser alterado para incluir anotações. A figura 9 contém uma sugestão de gramática.

```
fn_exp  ::= fn <var>: types => fn_exp
          | if_exp
if_exp  ::= <if> if_exp <then> fn_exp <else> fn_exp
          | or_exp
or_exp  ::= and_exp (or and_exp)*
and_exp ::= eq_exp (and eq_exp)*
eq_exp  ::= cmp_exp (= cmp_exp)*
cmp_exp ::= add_exp ([<=|<] add_exp)*
add_exp ::= mul_exp ([+|-] mul_exp)*
mul_exp ::= unary_exp ([*|/] unary_exp)*
unary_exp ::= <not> unary_exp
              | ~ unary_exp
              | let_exp
let_exp ::= <let> <var>: types <- fn_exp <in> fn_exp <end>
          | val_exp
val_exp ::= val_tk (val_tk)*
val_tk ::= <var> | ( fn_exp ) | <num> | <true> | <false>

types ::= type -> types | type

type ::= int | bool | ( types )
```

Figura 9: Sugestão de gramática com as corretas regras de precedência e associatividade

```
def types(self):
    tp = self.type()
    while self.match_and_consume(TokenType.TPF):
        tp = ArrowType(tp, self.type())
    return tp
```

Figura 10: A análise sintática de anotações de tipo, implementada com a associatividade à esquerda. Esta implementação simula a produção "types ::= type (-> type)*". De acordo com essa regra, o tipo int -> int -> int seria equivalente a ((int->int) -> int). Note que esta implementação está **incorreta**.

```
def types(self):
    tp = self.type()
    if self.match_and_consume(TokenType.TPF):
        tp = ArrowType(tp, self.types())
    return tp
```

Figura 10: A análise sintática de anotações de tipo, implementada com a associatividade à direita. Esta implementação simula a produção "types ::= type types | type". De acordo com essa regra, o tipo int -> int -> int seria equivalente a (int -> (int -> int)). Note que esta implementação está **correta**, representando assim o comportamento esperado de nossa linguagem

Submetendo e Testando

Para completar este VPL, você deverá entregar cinco arquivos: `Expression.py`, `Lexer.py`, `Parser.py`, `Visitor.py` e `driver.py`. Você não deverá alterar `driver.py` ou `Expression.py`. Para testar sua implementação localmente, você pode usar o comando abaixo:

```
python3 driver.py
(fn x: int => x * x) (4-1) # CTRL+D
<class 'int'>
```

A implementação dos diferentes arquivos possui vários comentários doctest, que testam sua implementação. Caso queira testar seu código, simplesmente faça:

```
python3 -m doctest xx.py
```

No exemplo acima, substituta `xx.py` por algum dos arquivos que você queira testar (experimente com `Visitor.py`, por exemplo). Caso você não gere mensagens de erro, então seu trabalho está (quase) completo!

Arquivos requeridos

driver.py

```python
import sys
from Expression import *
from Visitor import *
from Lexer import Lexer
from Parser import Parser

if __name__ == "__main__":
    """
    Este arquivo nao deve ser alterado, mas deve ser enviado para resolver o
    VPL. O arquivo contem o codigo que testa a implementacao do parser.
    """
    text = sys.stdin.read()
    lexer = Lexer(text)
    parser = Parser(lexer.tokens())
    exp = parser.parse()
    visitor = TypeCheckVisitor()
    print(f"{exp.accept(visitor, {})}")
```

Lexer.py

```
import sys
import enum


class Token:
    """
    This class contains the definition of Tokens. A token has two fields: its
    text and its kind. The "kind" of a token is a constant that identifies it
    uniquely. See the TokenType to know the possible identifiers (if you want).
    You don't need to change this class.
    """
    def __init__(self, tokenText, tokenKind):
        # The token's actual text. Used for identifiers, strings, and numbers.
        self.text = tokenText
        # The TokenType that this token is classified as.
        self.kind = tokenKind


class TokenType(enum.Enum):
    """
    These are the possible tokens. You don't need to change this class at all.
    """

    EOF = -1  # End of file
    NLN = 0  # New line
    WSP = 1  # White Space
    COM = 2  # Comment
    NUM = 3  # Number (integers)
    STR = 4  # Strings
    TRU = 5  # The constant true
    FLS = 6  # The constant false
    VAR = 7  # An identifier
    LET = 8  # The 'let' of the let expression
    INX = 9  # The 'in' of the let expression
    END = 10  # The 'end' of the let expression
    EQL = 201  # x = y
    ADD = 202  # x + y
    SUB = 203  # x - y
    MUL = 204  # x * y
    DIV = 205  # x / y
    LEQ = 206  # x <= y
    LTH = 207  # x < y
    NEG = 208  # ~x
    NOT = 209  # not x
    LPR = 210  # (
    RPR = 211  # )
    ASN = 212  # The assignment '<-' operator
    ORX = 213  # x or y
    AND = 214  # x and y
    IFX = 215  # The 'if' of a conditional expression
    THN = 216  # The 'then' of a conditional expression
    ELS = 217  # The 'else' of a conditional expression
    FNX = 218  # The 'fn' that declares an anonymous function
    ARW = 219  # The '=>' that separates the parameter from the body of function
    TPF = 220  # The arrow that indicates a function type
    COL = 221  # The colon that separates type annotations
    INT = 222  # The int type ('int')
    LGC = 223  # The boolean (logic) type ('bool')


class Lexer:

    def __init__(self, source):
        """
        The constructor of the lexer. It receives the string that shall be
        scanned.
        TODO: You will need to implement this method.
        """
        pass

    def tokens(self):
        """
        This method is a token generator: it converts the string encapsulated
        into this object into a sequence of Tokens. Notice that this method
        filters out three kinds of tokens: white-spaces, comments and new lines.

        Examples:

        >>> l = Lexer("1 + 3")
        >>> [tk.kind for tk in l.tokens()]
        [<TokenType.NUM: 3>, <TokenType.ADD: 202>, <TokenType.NUM: 3>]

        >>> l = Lexer('1 * 2\\n')
        >>> [tk.kind for tk in l.tokens()]
        [<TokenType.NUM: 3>, <TokenType.MUL: 204>, <TokenType.NUM: 3>]

        >>> l = Lexer('1 * 2 -- 3\\n')
        >>> [tk.kind for tk in l.tokens()]
        [<TokenType.NUM: 3>, <TokenType.MUL: 204>, <TokenType.NUM: 3>]

        >>> l = Lexer("1 + var")
        >>> [tk.kind for tk in l.tokens()]
        [<TokenType.NUM: 3>, <TokenType.ADD: 202>, <TokenType.VAR: 7>]

        >>> l = Lexer("let v <- 2 in v end")
        >>> [tk.kind.name for tk in l.tokens()]
        ['LET', 'VAR', 'ASN', 'NUM', 'INX', 'VAR', 'END']

        >>> l = Lexer("v: int -> int")
        >>> [tk.kind.name for tk in l.tokens()]
        ['VAR', 'COL', 'INT', 'TPF', 'INT']

        >>> l = Lexer("v: int -> bool")
```

```
104          >>> [tk.kind.name for tk in l.tokens()]
105          ['VAR', 'COL', 'INT', 'TPF', 'LGC']
106          """
107          token = self.getToken()
108          while token.kind != TokenType.EOF:
109              if token.kind != TokenType.WSP and token.kind != TokenType.COM \
110                      and token.kind != TokenType.NLN:
111                  yield token
112              token = self.getToken()
113
114      def getToken(self):
115          """
116          Return the next token.
117          TODO: Implement this method!
118          """
119          token = None
120          return token
```

Parser.py

```
import sys

from Expression import *
from Lexer import Token, TokenType

"""
This file implements a parser for SML with anonymous functions and type
annotations. The grammar is as follows:

fn_exp  ::= fn <var>: types => fn_exp
          | if_exp
if_exp  ::= <if> if_exp <then> fn_exp <else> fn_exp
          | or_exp
or_exp  ::= and_exp (or and_exp)*
and_exp ::= eq_exp (and eq_exp)*
eq_exp  ::= cmp_exp (= cmp_exp)*
cmp_exp ::= add_exp ([<=|<] add_exp)*
add_exp ::= mul_exp ([+|-] mul_exp)*
mul_exp ::= unary_exp ([*|/] unary_exp)*
unary_exp ::= <not> unary_exp
             | ~ unary_exp
             | let_exp
let_exp ::= <let> <var>: types <- fn_exp <in> fn_exp <end>
          | val_exp
val_exp ::= val_tk (val_tk)*
val_tk ::= <var> | ( fn_exp ) | <num> | <true> | <false>

types ::= type -> types | type

type ::= int | bool | ( types )

References:
    see https://www.engr.mun.ca/~theo/Misc/exp_parsing.htm#classic
"""

class Parser:
    def __init__(self, tokens):
        """
        Initializes the parser. The parser keeps track of the list of tokens
        and the current token. For instance:
        """
        self.tokens = list(tokens)
        self.cur_token_idx = 0 # This is just a suggestion!
        # You can (and probably should!) modify this method.

    def parse(self):
        """
        Returns the expression associated with the stream of tokens.

        Examples:
        >>> parser = Parser([Token('123', TokenType.NUM)])
        >>> exp = parser.parse()
        >>> ev = TypeCheckVisitor()
        >>> exp.accept(ev, None)
        <class 'int'>

        >>> parser = Parser([Token('True', TokenType.TRU)])
        >>> exp = parser.parse()
        >>> ev = TypeCheckVisitor()
        >>> exp.accept(ev, None)
        <class 'bool'>

        >>> parser = Parser([Token('False', TokenType.FLS)])
        >>> exp = parser.parse()
        >>> ev = TypeCheckVisitor()
        >>> exp.accept(ev, None)
        <class 'bool'>

        >>> tk0 = Token('~', TokenType.NEG)
        >>> tk1 = Token('123', TokenType.NUM)
        >>> parser = Parser([tk0, tk1])
        >>> exp = parser.parse()
        >>> ev = TypeCheckVisitor()
        >>> exp.accept(ev, None)
        <class 'int'>

        >>> tk0 = Token('3', TokenType.NUM)
        >>> tk1 = Token('*', TokenType.MUL)
        >>> tk2 = Token('4', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2])
        >>> exp = parser.parse()
        >>> ev = TypeCheckVisitor()
        >>> exp.accept(ev, None)
        <class 'int'>

        >>> tk0 = Token('3', TokenType.NUM)
        >>> tk1 = Token('*', TokenType.MUL)
        >>> tk2 = Token('~', TokenType.NEG)
        >>> tk3 = Token('4', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2, tk3])
        >>> exp = parser.parse()
        >>> ev = TypeCheckVisitor()
        >>> exp.accept(ev, None)
        <class 'int'>

        >>> tk0 = Token('30', TokenType.NUM)
        >>> tk1 = Token('/', TokenType.DIV)
        >>> tk2 = Token('4', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2])
        >>> exp = parser.parse()
        >>> ev = TypeCheckVisitor()
        >>> exp.accept(ev, None)
        <class 'int'>
```

```
>>> tk0 = Token('3', TokenType.NUM)
>>> tk1 = Token('+', TokenType.ADD)
>>> tk2 = Token('4', TokenType.NUM)
>>> parser = Parser([tk0, tk1, tk2])
>>> exp = parser.parse()
>>> ev = TypeCheckVisitor()
>>> exp.accept(ev, None)
<class 'int'>

>>> tk0 = Token('30', TokenType.NUM)
>>> tk1 = Token('-', TokenType.SUB)
>>> tk2 = Token('4', TokenType.NUM)
>>> parser = Parser([tk0, tk1, tk2])
>>> exp = parser.parse()
>>> ev = TypeCheckVisitor()
>>> exp.accept(ev, None)
<class 'int'>

>>> tk0 = Token('2', TokenType.NUM)
>>> tk1 = Token('*', TokenType.MUL)
>>> tk2 = Token('(', TokenType.LPR)
>>> tk3 = Token('3', TokenType.NUM)
>>> tk4 = Token('+', TokenType.ADD)
>>> tk5 = Token('4', TokenType.NUM)
>>> tk6 = Token(')', TokenType.RPR)
>>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6])
>>> exp = parser.parse()
>>> ev = TypeCheckVisitor()
>>> exp.accept(ev, None)
<class 'int'>

>>> tk0 = Token('4', TokenType.NUM)
>>> tk1 = Token('==', TokenType.EQL)
>>> tk2 = Token('4', TokenType.NUM)
>>> parser = Parser([tk0, tk1, tk2])
>>> exp = parser.parse()
>>> ev = TypeCheckVisitor()
>>> exp.accept(ev, None)
<class 'bool'>

>>> tk0 = Token('4', TokenType.NUM)
>>> tk1 = Token('<=', TokenType.LEQ)
>>> tk2 = Token('4', TokenType.NUM)
>>> parser = Parser([tk0, tk1, tk2])
>>> exp = parser.parse()
>>> ev = TypeCheckVisitor()
>>> exp.accept(ev, None)
<class 'bool'>

>>> tk0 = Token('4', TokenType.NUM)
>>> tk1 = Token('<', TokenType.LTH)
>>> tk2 = Token('4', TokenType.NUM)
>>> parser = Parser([tk0, tk1, tk2])
>>> exp = parser.parse()
>>> ev = TypeCheckVisitor()
>>> exp.accept(ev, None)
<class 'bool'>

>>> tk0 = Token('not', TokenType.NOT)
>>> tk1 = Token('(', TokenType.LPR)
>>> tk2 = Token('4', TokenType.NUM)
>>> tk3 = Token('<', TokenType.LTH)
>>> tk4 = Token('4', TokenType.NUM)
>>> tk5 = Token(')', TokenType.RPR)
>>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5])
>>> exp = parser.parse()
>>> ev = TypeCheckVisitor()
>>> exp.accept(ev, None)
<class 'bool'>

>>> tk0 = Token('true', TokenType.TRU)
>>> tk1 = Token('or', TokenType.ORX)
>>> tk2 = Token('false', TokenType.FLS)
>>> parser = Parser([tk0, tk1, tk2])
>>> exp = parser.parse()
>>> ev = TypeCheckVisitor()
>>> exp.accept(ev, None)
<class 'bool'>

>>> tk0 = Token('true', TokenType.TRU)
>>> tk1 = Token('and', TokenType.AND)
>>> tk2 = Token('false', TokenType.FLS)
>>> parser = Parser([tk0, tk1, tk2])
>>> exp = parser.parse()
>>> ev = TypeCheckVisitor()
>>> exp.accept(ev, None)
<class 'bool'>

>>> t0 = Token('let', TokenType.LET)
>>> t1 = Token('v', TokenType.VAR)
>>> t2 = Token(':', TokenType.COL)
>>> t3 = Token('int', TokenType.INT)
>>> t4 = Token('<-', TokenType.ASN)
>>> t5 = Token('42', TokenType.NUM)
>>> t6 = Token('in', TokenType.INX)
>>> t7 = Token('v', TokenType.VAR)
>>> t8 = Token('end', TokenType.END)
>>> parser = Parser([t0, t1, t2, t3, t4, t5, t6, t7, t8])
>>> exp = parser.parse()
>>> ev = TypeCheckVisitor()
>>> exp.accept(ev, {})
<class 'int'>
```

```
207
208         >>> t0 = Token('let', TokenType.LET)
209         >>> t1 = Token('v', TokenType.VAR)
210         >>> t2 = Token(':', TokenType.COL)
211         >>> t3 = Token('int', TokenType.INT)
212         >>> t4 = Token('<-', TokenType.ASN)
213         >>> t5 = Token('21', TokenType.NUM)
214         >>> t6 = Token('in', TokenType.INX)
215         >>> t7 = Token('v', TokenType.VAR)
216         >>> t8 = Token('+', TokenType.ADD)
217         >>> t9 = Token('v', TokenType.VAR)
218         >>> tA = Token('end', TokenType.END)
219         >>> parser = Parser([t0, t1, t2, t3, t4, t5, t6, t7, t8, t9, tA])
220         >>> exp = parser.parse()
221         >>> ev = TypeCheckVisitor()
222         >>> exp.accept(ev, {})
223         <class 'int'>
224
225         >>> tk0 = Token('if', TokenType.IFX)
226         >>> tk1 = Token('2', TokenType.NUM)
227         >>> tk2 = Token('<', TokenType.LTH)
228         >>> tk3 = Token('3', TokenType.NUM)
229         >>> tk4 = Token('then', TokenType.THN)
230         >>> tk5 = Token('1', TokenType.NUM)
231         >>> tk6 = Token('else', TokenType.ELS)
232         >>> tk7 = Token('2', TokenType.NUM)
233         >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6, tk7])
234         >>> exp = parser.parse()
235         >>> ev = TypeCheckVisitor()
236         >>> exp.accept(ev, None)
237         <class 'int'>
238
239         >>> tk0 = Token('if', TokenType.IFX)
240         >>> tk1 = Token('false', TokenType.FLS)
241         >>> tk2 = Token('then', TokenType.THN)
242         >>> tk3 = Token('1', TokenType.NUM)
243         >>> tk4 = Token('else', TokenType.ELS)
244         >>> tk5 = Token('2', TokenType.NUM)
245         >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5])
246         >>> exp = parser.parse()
247         >>> ev = TypeCheckVisitor()
248         >>> exp.accept(ev, None)
249         <class 'int'>
250
251         >>> tk0 = Token('fn', TokenType.FNX)
252         >>> tk1 = Token('v', TokenType.VAR)
253         >>> tk2 = Token(':', TokenType.COL)
254         >>> tk3 = Token('int', TokenType.INT)
255         >>> tk4 = Token('=>', TokenType.ARW)
256         >>> tk5 = Token('v', TokenType.VAR)
257         >>> tk6 = Token('+', TokenType.ADD)
258         >>> tk7 = Token('1', TokenType.NUM)
259         >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6, tk7])
260         >>> exp = parser.parse()
261         >>> ev = TypeCheckVisitor()
262         >>> print(exp.accept(ev, {}))
263         <class 'int'> -> <class 'int'>
264
265         >>> t0 = Token('(', TokenType.LPR)
266         >>> t1 = Token('fn', TokenType.FNX)
267         >>> t2 = Token('v', TokenType.VAR)
268         >>> t3 = Token(':', TokenType.COL)
269         >>> t4 = Token('int', TokenType.INT)
270         >>> t5 = Token('=>', TokenType.ARW)
271         >>> t6 = Token('v', TokenType.VAR)
272         >>> t7 = Token('+', TokenType.ADD)
273         >>> t8 = Token('1', TokenType.NUM)
274         >>> t9 = Token(')', TokenType.RPR)
275         >>> tA = Token('2', TokenType.NUM)
276         >>> parser = Parser([t0, t1, t2, t3, t4, t5, t6, t7, t8, t9, tA])
277         >>> exp = parser.parse()
278         >>> ev = TypeCheckVisitor()
279         >>> exp.accept(ev, {})
280         <class 'int'>
281         """
282         # TODO: implement this method.
283         return None
```

Expression.py

```python
from abc import ABC, abstractmethod
from Visitor import *


class Expression(ABC):
    @abstractmethod
    def accept(self, visitor, arg):
        raise NotImplementedError


class Var(Expression):
    """
    This class represents expressions that are identifiers. The value of an
    indentifier is the value associated with it in the environment table.
    """

    def __init__(self, identifier):
        self.identifier = identifier

    def accept(self, visitor, arg):
        return visitor.visit_var(self, arg)


class Bln(Expression):
    """
    This class represents expressions that are boolean values. There are only
    two boolean values: true and false. The semantics of such an expression is
    the boolean itself.
    """

    def __init__(self, bln):
        self.bln = bln

    def accept(self, visitor, arg):
        return visitor.visit_bln(self, arg)


class Num(Expression):
    """
    This class represents expressions that are numbers. The semantics of such
    an expression is the number itself.
    """

    def __init__(self, num):
        self.num = num

    def accept(self, visitor, arg):
        return visitor.visit_num(self, arg)


class BinaryExpression(Expression):
    """
    This class represents binary expressions. A binary expression has two
    sub-expressions: the left operand and the right operand.
    """

    def __init__(self, left, right):
        self.left = left
        self.right = right

    @abstractmethod
    def accept(self, visitor, arg):
        raise NotImplementedError


class Eql(BinaryExpression):
    """
    This class represents the equality between two expressions. The semantics
    of such an expression is True if the subexpressions are the same, or false
    otherwise.
    """

    def accept(self, visitor, arg):
        return visitor.visit_eql(self, arg)


class Add(BinaryExpression):
    """
    This class represents addition of two expressions. The semantics of such
    an expression is the addition of the two subexpression's values.
    """

    def accept(self, visitor, arg):
        return visitor.visit_add(self, arg)


class And(BinaryExpression):
    """
    This class represents the logical disjunction of two boolean expressions.
    The evaluation of an expression of this kind is the logical AND of the two
    subexpression's values.
    """

    def accept(self, visitor, arg):
        return visitor.visit_and(self, arg)


class Or(BinaryExpression):
    """
    This class represents the logical conjunction of two boolean expressions.
    The evaluation of an expression of this kind is the logical OR of the two
    subexpression's values.
    """
```

```
104
105     def accept(self, visitor, arg):
106         return visitor.visit_or(self, arg)
107
108
109 class Sub(BinaryExpression):
110     """
111     This class represents subtraction of two expressions. The semantics of such
112     an expression is the subtraction of the two subexpression's values.
113     """
114
115     def accept(self, visitor, arg):
116         return visitor.visit_sub(self, arg)
117
118
119 class Mul(BinaryExpression):
120     """
121     This class represents multiplication of two expressions. The semantics of
122     such an expression is the product of the two subexpression's values.
123     """
124
125     def accept(self, visitor, arg):
126         return visitor.visit_mul(self, arg)
127
128
129 class Div(BinaryExpression):
130     """
131     This class represents the integer division of two expressions. The
132     semantics of such an expression is the integer quocient of the two
133     subexpression's values.
134     """
135
136     def accept(self, visitor, arg):
137         return visitor.visit_div(self, arg)
138
139
140 class Leq(BinaryExpression):
141     """
142     This class represents comparison of two expressions using the
143     less-than-or-equal comparator. The semantics of such an expression is a
144     boolean value that is true if the left operand is less than or equal the
145     right operand. It is false otherwise.
146     """
147
148     def accept(self, visitor, arg):
149         return visitor.visit_leq(self, arg)
150
151
152 class Lth(BinaryExpression):
153     """
154     This class represents comparison of two expressions using the
155     less-than comparison operator. The semantics of such an expression is a
156     boolean value that is true if the left operand is less than the right
157     operand. It is false otherwise.
158     """
159
160     def accept(self, visitor, arg):
161         return visitor.visit_lth(self, arg)
162
163
164 class UnaryExpression(Expression):
165     """
166     This class represents unary expressions. A unary expression has only one
167     sub-expression.
168     """
169
170     def __init__(self, exp):
171         self.exp = exp
172
173     @abstractmethod
174     def accept(self, visitor, arg):
175         raise NotImplementedError
176
177
178 class Neg(UnaryExpression):
179     """
180     This expression represents the additive inverse of a number. The additive
181     inverse of a number n is the number -n, so that the sum of both is zero.
182     """
183
184     def accept(self, visitor, arg):
185         return visitor.visit_neg(self, arg)
186
187
188 class Not(UnaryExpression):
189     """
190     This expression represents the negation of a boolean. The negation of a
191     boolean expression is the logical complement of that expression.
192     """
193
194     def accept(self, visitor, arg):
195         return visitor.visit_not(self, arg)
196
197
198 class Let(Expression):
199     """
200     This class represents a let expression.
201     """
202
203     def __init__(self, identifier, tp_var, exp_def, exp_body):
204         self.identifier = identifier
205         self.tp_var = tp_var
206         self.exp_def = exp_def
```

```
207         self.exp_body = exp_body
208
209     def accept(self, visitor, arg):
210         return visitor.visit_let(self, arg)
211
212
213 class Fn(Expression):
214     """
215     This class represents an anonymous function.
216     """
217
218     def __init__(self, formal, tp_var, body):
219         self.formal = formal
220         self.tp_var = tp_var
221         self.body = body
222
223     def accept(self, visitor, arg):
224         return visitor.visit_fn(self, arg)
225
226
227 class App(Expression):
228     """
229     This class represents a function application, such as 'e0 e1'.
230     """
231
232     def __init__(self, function, actual):
233         self.function = function
234         self.actual = actual
235
236     def accept(self, visitor, arg):
237         return visitor.visit_app(self, arg)
238
239
240 class IfThenElse(Expression):
241     """
242     This class represents a conditional expression.
243     """
244
245     def __init__(self, cond, e0, e1):
246         self.cond = cond
247         self.e0 = e0
248         self.e1 = e1
249
250     def accept(self, visitor, arg):
251         return visitor.visit_ifThenElse(self, arg)
```

Visitor.py

```
import sys
from abc import ABC, abstractmethod
from Expression import *


class Visitor(ABC):
    """
    The visitor pattern consists of two abstract classes: the Expression and the
    Visitor. The Expression class defines on method: 'accept(visitor, args)'.
    This method takes in an implementation of a visitor, and the arguments that
    are passed from expression to expression. The Visitor class defines one
    specific method for each subclass of Expression. Each instance of such a
    subclasse will invoke the right visiting method.
    """

    @abstractmethod
    def visit_var(self, exp, arg):
        pass

    @abstractmethod
    def visit_bln(self, exp, arg):
        pass

    @abstractmethod
    def visit_num(self, exp, arg):
        pass

    @abstractmethod
    def visit_eql(self, exp, arg):
        pass

    @abstractmethod
    def visit_and(self, exp, arg):
        pass

    @abstractmethod
    def visit_or(self, exp, arg):
        pass

    @abstractmethod
    def visit_add(self, exp, arg):
        pass

    @abstractmethod
    def visit_sub(self, exp, arg):
        pass

    @abstractmethod
    def visit_mul(self, exp, arg):
        pass

    @abstractmethod
    def visit_div(self, exp, arg):
        pass

    @abstractmethod
    def visit_leq(self, exp, arg):
        pass

    @abstractmethod
    def visit_lth(self, exp, arg):
        pass

    @abstractmethod
    def visit_neg(self, exp, arg):
        pass

    @abstractmethod
    def visit_not(self, exp, arg):
        pass

    @abstractmethod
    def visit_let(self, exp, arg):
        pass

    @abstractmethod
    def visit_ifThenElse(self, exp, arg):
        pass

    @abstractmethod
    def visit_fn(self, exp, arg):
        pass

    @abstractmethod
    def visit_app(self, exp, arg):
        pass


class ArrowType:
    """
    This is the class that represents function types. A function type is just
    that: the type of a function. Function types are also called "arrow types",
    because they tend to be represented as 't0 -> t1'. Thus, an arrow type
    has two parts: the head type (t0) and the tail type (t1).

    Usage:
        >>> t = ArrowType(type(1), type(True))
        >>> t.hd == type(1) and t.tl == type(True)
        True

        >>> t = ArrowType(type(1), type(True))
        >>> str(t)
        "<class 'int'> -> <class 'bool'>"
```

```
    """

    def __init__(self, tp_formal, tp_body):
        self.hd = tp_formal
        self.tl = tp_body

    def __eq__(self, other):
        """
        Two arrow types are equal if their head and tail are equal.
        Example:
            >>> t0 = ArrowType(type(1), ArrowType(type(1), type(1)))
            >>> t1 = ArrowType(type(1), ArrowType(type(1), type(1)))
            >>> t0 == t1
            True
        """
        if isinstance(other, ArrowType):
            return self.hd == other.hd and self.tl == other.tl
        else:
            return False

    def __repr__(self):
        if isinstance(self.hd, ArrowType):
            hd_str = f"( {str(self.hd)} )"
        else:
            hd_str = str(self.hd)
        if isinstance(self.tl, ArrowType):
            tl_str = f"( {str(self.tl)} )"
        else:
            tl_str = str(self.tl)
        return f"{hd_str} -> {tl_str}"


class TypeCheckVisitor(Visitor):
    """
    The TypeCheckVisitor class evaluates logical and arithmetic expressions. The
    result of evaluating an expression is the value of that expression. The
    inherited attribute propagated throughout visits is the environment that
    associates the names of variables with values.
    """

    def visit_var(self, exp, env):
        """
        Usage:
            >>> e = Var('t')
            >>> ev = TypeCheckVisitor()
            >>> e.accept(ev, {'t':type(1)})
            <class 'int'>

            >>> e = Var('t')
            >>> ev = TypeCheckVisitor()
            >>> e.accept(ev, {'t':ArrowType(type(1), type(True))})
            <class 'int'> -> <class 'bool'>
        """
        if exp.identifier in env:
            return env[exp.identifier]
        else:
            sys.exit("Def error")

    def visit_bln(self, exp, env):
        """
        Usage:
            >>> e = Bln(True)
            >>> ev = TypeCheckVisitor()
            >>> e.accept(ev, None)
            <class 'bool'>
        """
        return type(exp.bln)

    def visit_num(self, exp, env):
        """
        Usage:
            >>> e = Num(1)
            >>> ev = TypeCheckVisitor()
            >>> e.accept(ev, None)
            <class 'int'>
        """
        return type(exp.num)

    def visit_eql(self, exp, env): # Implemented for you :)
        """
        Usage:
            >>> e = Eql(Num(1), Num(2))
            >>> ev = TypeCheckVisitor()
            >>> e.accept(ev, None)
            <class 'bool'>
        """
        if exp.left.accept(self, env) == exp.right.accept(self, env):
            return type(True)
        else:
            sys.exit("Type error")

    def visit_and(self, exp, env):
        """
        Usage:
            >>> e = And(Bln(True), Bln(False))
            >>> ev = TypeCheckVisitor()
            >>> e.accept(ev, None)
            <class 'bool'>
        """
        # TODO: Implement this method.
        raise NotImplementedError

    def visit_or(self, exp, env):
        """
```

```
207         """
208         Usage:
209             >>> e = Or(Bln(True), Bln(False))
210             >>> ev = TypeCheckVisitor()
211             >>> e.accept(ev, None)
212             <class 'bool'>
213         """
214         # TODO: Implement this method.
215         raise NotImplementedError
216
217     def visit_add(self, exp, env):
218         """
219         Usage:
220             >>> e = Add(Num(1), Num(2))
221             >>> ev = TypeCheckVisitor()
222             >>> e.accept(ev, None)
223             <class 'int'>
224         """
225         # TODO: Implement this method.
226         raise NotImplementedError
227
228     def visit_sub(self, exp, env):
229         """
230         Usage:
231             >>> e = Sub(Num(1), Num(2))
232             >>> ev = TypeCheckVisitor()
233             >>> e.accept(ev, None)
234             <class 'int'>
235         """
236         # TODO: Implement this method.
237         raise NotImplementedError
238
239     def visit_mul(self, exp, env):
240         """
241         Usage:
242             >>> e = Mul(Num(1), Num(2))
243             >>> ev = TypeCheckVisitor()
244             >>> e.accept(ev, None)
245             <class 'int'>
246         """
247         # TODO: Implement this method.
248         raise NotImplementedError
249
250     def visit_div(self, exp, env):
251         """
252         Usage:
253             >>> e = Div(Num(1), Num(0))
254             >>> ev = TypeCheckVisitor()
255             >>> e.accept(ev, None)
256             <class 'int'>
257         """
258         # TODO: Implement this method.
259         raise NotImplementedError
260
261     def visit_leq(self, exp, env):
262         """
263         Usage:
264             >>> e = Leq(Num(1), Num(0))
265             >>> ev = TypeCheckVisitor()
266             >>> e.accept(ev, None)
267             <class 'bool'>
268         """
269         # TODO: Implement this method.
270         raise NotImplementedError
271
272     def visit_lth(self, exp, env):
273         """
274         Usage:
275             >>> e = Lth(Num(1), Num(0))
276             >>> ev = TypeCheckVisitor()
277             >>> e.accept(ev, None)
278             <class 'bool'>
279         """
280         # TODO: Implement this method.
281         raise NotImplementedError
282
283     def visit_neg(self, exp, env):
284         """
285         Usage:
286             >>> e = Neg(Num(1))
287             >>> ev = TypeCheckVisitor()
288             >>> e.accept(ev, None)
289             <class 'int'>
290         """
291         # TODO: Implement this method.
292         raise NotImplementedError
293
294     def visit_not(self, exp, env):
295         """
296         Usage:
297             >>> e = Not(Bln(False))
298             >>> ev = TypeCheckVisitor()
299             >>> e.accept(ev, None)
300             <class 'bool'>
301         """
302         # TODO: Implement this method.
303         raise NotImplementedError
304
305     def visit_ifThenElse(self, exp, env):
306         """
307         Usage:
308             >>> e0 = Lth(Num(1), Num(0))
309             >>> e = IfThenElse(e0, Bln(True), e0)
310             >>> ev = TypeCheckVisitor()
```

```
            >>> ev = TypeCheckVisitor()
            >>> e.accept(ev, {})
            <class 'bool'>

            >>> e0 = Lth(Num(1), Num(0))
            >>> e = IfThenElse(e0, Var('v'), Var('w'))
            >>> ev = TypeCheckVisitor()
            >>> tp0 = ArrowType(type(1), type(2))
            >>> tp1 = ArrowType(type(3), type(4))
            >>> e.accept(ev, {'v':tp0, 'w':tp1})
            <class 'int'> -> <class 'int'>
        """
        # TODO: Implement this method.
        raise NotImplementedError

    def visit_let(self, exp, env):
        """
        Usage:
            >>> e = Let('v', type(True), Not(Bln(False)), Var('v'))
            >>> ev = TypeCheckVisitor()
            >>> e.accept(ev, {})
            <class 'bool'>

            >>> e = Let('v', type(1), Num(2), Add(Var('v'), Num(3)))
            >>> ev = TypeCheckVisitor()
            >>> e.accept(ev, {})
            <class 'int'>
        """
        # TODO: Implement this method.
        raise NotImplementedError

    def visit_fn(self, exp, env):
        """
        Usage:
            >>> e = Fn('v', type(True), Var('v'))
            >>> ev = TypeCheckVisitor()
            >>> e.accept(ev, {})
            <class 'bool'> -> <class 'bool'>

            >>> e = Fn('v', type(1), Add(Var('v'), Num(3)))
            >>> ev = TypeCheckVisitor()
            >>> e.accept(ev, {})
            <class 'int'> -> <class 'int'>

            >>> e0 = Fn('y', type(1), Add(Var('y'), Var('x')))
            >>> e1 = Fn('x', type(1), e0)
            >>> ev = TypeCheckVisitor()
            >>> e1.accept(ev, {})
            <class 'int'> -> ( <class 'int'> -> <class 'int'> )
        """
        # TODO: Implement this method.
        raise NotImplementedError

    def visit_app(self, exp, env):
        """
        Usage:
            >>> e0 = Fn('v', type(1), Add(Var('v'), Num(3)))
            >>> e1 = App(e0, Num(1))
            >>> ev = TypeCheckVisitor()
            >>> e1.accept(ev, {})
            <class 'int'>

            >>> e0 = Fn('y', type(1), Add(Var('y'), Var('x')))
            >>> e1 = Fn('x', type(1), e0)
            >>> e2 = App(e1, Num(1))
            >>> ev = TypeCheckVisitor()
            >>> e2.accept(ev, {})
            <class 'int'> -> <class 'int'>
        """
        # TODO: Implement this method.
        raise NotImplementedError
```

VPL