2025_2 - COMPILADORES - METATURMA

Descrição Visualizar envios

AV7 - Inferência de Tipos

Data de entrega: sexta, 10 Out 2025, 23:59
Arquivos requeridos: driver.py, Lexer.py, Parser.py, Expression.py, Visitor.py, Unifier.py (Baixar)
Tipo de trabalho: Trabalho individual

O objetivo deste trabalho é implementar um inferidor de tipos para nossa linguagem de expressões aritméticas. Para tanto, você deverá implementar três algoritmos:

1. O primeiro algoritmo é um Visitor, que atravessa a árvore de sintaxe abstrata de programas gerando *constraints*.
2. O segundo algoritmo implementa a unificação das constraints, juntando variáveis de mesmo tipo em conjuntos.
3. O terceiro algoritmo encontra um nome para cada conjunto. O nome de um conjunto é seu tipo.

Geração de *Constraints*

Uma "*constraint*" é uma relação de equivalência, que determina que duas variáveis têm o mesmo tipo. Por exemplo, a relação ('TV_1', 'v') indica que a variável 'v' e a variável 'TV_1' possuem o mesmo tipo. De maneira similar, a relação ('v', <class 'int'>) indica que a variável 'v' possui tipo `int`. Para gerar equivalências, você deve implementar uma subclasse de `Visitor`, que será chamada `CtrGenVisitor`. Esse visitor atravesa a árvore de sintaxe abstrata produzindo relações de equivalência. Como exemplo, veja o resultado do teste abaixo:

```
>>> e = Let('v', Num(40), Let('w', Num(2), Add(Var('v'), Var('w'))))
>>> ev = CtrGenVisitor()
>>> sorted([str(ct) for ct in e.accept(ev, ev.fresh_type_var())])
        ["('TV_1', 'TV_2')", \
         "('TV_2', 'TV_3')", \
         "('v', <class 'int'>)", \
         "('w', <class 'int'>)", \
         "(<class 'int'>, 'TV_3')", \
         "(<class 'int'>, 'v')", \
         "(<class 'int'>, 'w')"]
```

Cada método de visita possui a seguinte assinatura: "`visit_var(self, exp, type_var)`". O argumento "`exp`" é a expressão que está sendo analisada. O argumento "`type_var`" é a variável de tipo que irá denotar o tipo de "`exp`". As Figuras 1-4 abaixo ilustram como se dá o processo de geração de relações de equivalência para uma expressão como `let v <- 3 * 7 in v + v end`:

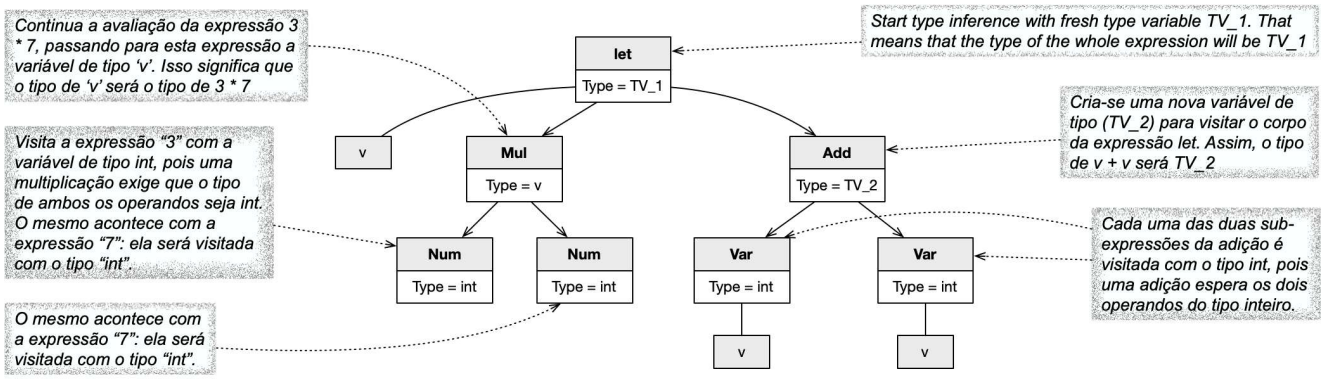

Figura 1: Como funciona a criação de variáveis de tipo para tipar uma expressão como "let v <- 3 * 7 in v + v end".

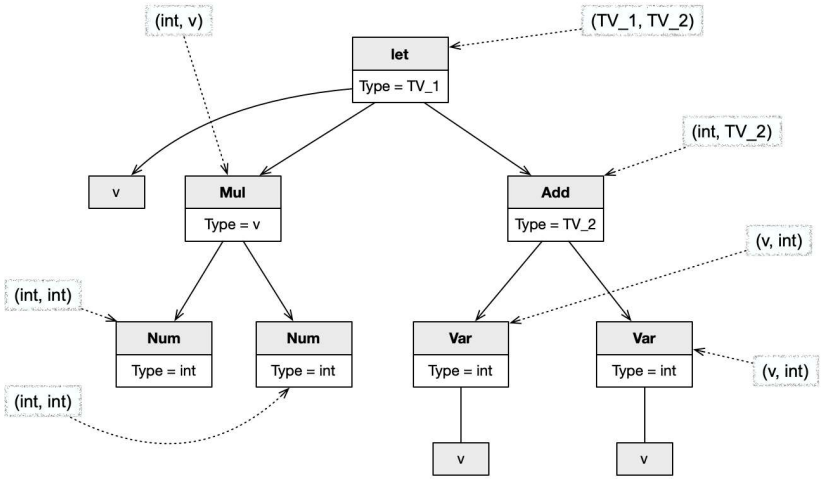

Figura 2: Lista de relações produzidas durante a visitação da expressão "let v <- 3 * 7 in v + v end"

$$\frac{\text{type}(E0, v) = K0 \quad TV\_2 \text{ is fresh} \quad \text{type}(E1, TV\_2) = K1}{\text{type}(\text{let } v <- E0 \text{ in } E1 \text{ end}, TV\_1) = K0 \cup K1 \cup \{(TV\_1, TV\_2)\}}$$

Figura 3: Regras para geração de relações de uma expressão let

```
def visit_let(self, exp, TV_1):
  K0 = exp.exp_def.accept(self, exp.identifier)
  TV_2 = self.fresh_type_var()
  K1 = exp.exp_body.accept(self, TV_2)
  return K0 | K1 | {(TV_1, TV_2)}
```

Figura 4: Implementação da geração de constraints em Python.

O visitor que gera relações de equivalência irá gerar o seguinte conjunto de equivalências para a expressão acima: `{(int, v), (TV_1, TV_2), (int, int), (int, int), (int, TV_2), (v, int), (v, int)}`. Os nomes TV_1 e TV_2 são variáveis de tipos, criadas para unificar os tipos das diferentes sub-expressões que formam `let v <- 3 * 7 in v + v` end. Para sabermos o tipo da variável `v`, é necessário resolver essas relações.

Unificação de Equivalências

A fim de descobrir o tipo das variáveis de um programa faz-se necessário "unificar" essas variáveis. Em nosso caso, unificação é bastante simples, e deve ser implementada na função `unify`, no arquivo `Unifier.py`. A unificação constrói um mapa de variáveis para conjuntos. Cada conjunto contém todas as variáveis que possuem tipos equivalentes no programa. O teste abaixo mostra o que deve ser feito:

```
>>> type_sets = unify([('a', 'TV_1'), ('b', 'TV_2'), ('TV_2', type(1)), ('TV_1',
type(True))], {})
>>> for var, type_set in type_sets.items():
...   print(var, type_set)
...
a {<class 'bool'>, 'a', 'TV_1'}
TV_1 {<class 'bool'>, 'a', 'TV_1'}
b {'b', <class 'int'>, 'TV_2'}
TV_2 {'b', <class 'int'>, 'TV_2'}
<class 'int'> {'b', <class 'int'>, 'TV_2'}
<class 'bool'> {<class 'bool'>, 'a', 'TV_1'}
```

Na prática, um truque simples para resolver essa parte da inferência é fazer com que todas as referências na tabela "type_sets" apontem para os mesmos conjuntos. Você pode processar cada equivalência em separado, unificando conjuntos conforme mostra a Figura 5 abaixo.

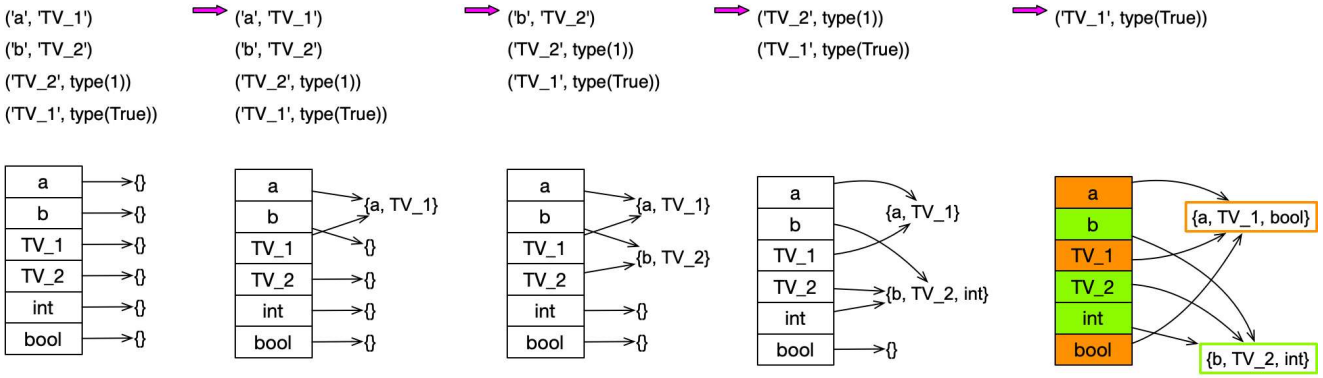

Figura 5: Unificação de regras de equivalência.

De Conjuntos de Equivalências para Nomes de Tipos

A parte final do trabalho consiste em substituir conjuntos de tipos pelo nome desses conjuntos. O nome de um conjunto de tipos é o nome do tipo concreto que existe dentro dele. Por exemplo, na Figura 5 temos dois conjuntos de tipos: `S1 = {a, TV_1, bool}` e `S2 = {b, TV_2, int}`. O tipo concreto que existe em S1 é `bool`, e o tipo concreto que existe em S2 é `int`. Assim, o nome de S1 é `bool`, e o nome de S2 é `int`.

A fim de resolver essa parte do exercício, você terá de implementar uma função `infer_types`, que recebe como entrada uma expressão, e produz uma tabela, que associa cada variável naquela expressão a um nome de tipos. Note que esta tabela pode contar mais informações do que as variáveis do programa. Por exemplo, ela pode conter as variáveis de tipos e os tipos concretos presentes na expressão. O teste abaixo mostra como `infer_types` pode ser usada:

```
>>> e0 = Let('v', Num(1), Let('y', Var('v'), Var('y')))
>>> e1 = IfThenElse(Lth(e0, Num(2)), Bln(True), Bln(False))
>>> e2 = Let('w', e1, And(Var('w'), Var('w')))
>>> type_names = infer_types(e2)
>>> type_names['v']
<class 'int'>
>>> type_names['w']
<class 'bool'>
>>> type_names['y']]
<class 'int'>
```

Note que a tabela `type_names`, no exemplo acima, associa cada variável a uma instância de "`Type`" em Python. Você pode usar `type(1)` para produzir uma instância de `<class 'int'>` e `type<True>` para produzir uma instância de `<class 'bool'>`.

Além de encontrar nomes para os conjuntos de tipos, é a função `name_sets` que determina se o programa contém erros de tipo. Quando um erro de tipos ocorre, o programa deve abortar com a mensagem "`Type error`" (sem ponto final!). Neste trabalho, reconhecemos dois tipos de erros:

1. **Tipo polimórfico**: um conjunto de tipos não contém nenhum tipo primitivo (nem int nem bool).
2. **Tipo ambíguo**: um conjunto de tipos contém os dois tipos primitivos (int e bool).

Note que independente do erro de tipos, a mesma mensagem deve ser impressa: "`Type error`".

Submetendo e Testando

Para completar este VPL, você deverá entregar seis arquivos: `Expression.py`, `Lexer.py`, `Parser.py`, `Unifier.py`, `Visitor.py` e `driver.py`. Você não deverá alterar `driver.py`. Na verdade, os únicos arquivos que você terá de modificar são `Visitor.py` e `Unifier.py`. Você pode usar, como ponto de partida, os arquivos disponíveis no exercício anterior (Verificação dinâmica de tipos). Para testar sua implementação localmente, você pode usar o comando abaixo:

```
python3 driver.py
2 + let v <- 3 in v * v end # CTRL+D
Type(v): int
Type(w): NoneType
Type(x): NoneType
Type(y): NoneType
Type(z): NoneType
```

Note que o teste busca encontrar o tipo de cinco variáveis: v, w, x, y e z. Caso essas variáveis não existam no programa de teste, não se preocupe: será impresso o tipo "`NoneType`". Você pode fazer algumas suposições neste trabalho:

1. Programas de teste não redefinem variáveis. Em outras palavras, cada expressão let cria sempre uma variável com um nome diferente.
2. Caso haja variáveis com nomes diferentes de v, w, x, y e z, isso não importa para o teste. Somente essas cinco variáveis são pesquisadas.
3. Caso algumas dessas variáveis não seja definida, isso também não importa: o tipo NoneType será impresso ao final.

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 Unifier import *
from Lexer import Lexer
from Parser import Parser

def print_types(type_names):
    """
    This method runs the type inference engine, and prints the type of
    variables 'x', 'y' and 'z', if they exist in the test case.
    """
    print(f"Type(v): {type_names.setdefault('v', type(None)).__name__}")
    print(f"Type(w): {type_names.setdefault('w', type(None)).__name__}")
    print(f"Type(x): {type_names.setdefault('x', type(None)).__name__}")
    print(f"Type(y): {type_names.setdefault('y', type(None)).__name__}")
    print(f"Type(z): {type_names.setdefault('z', type(None)).__name__}")

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())
    expression = parser.parse()
    print_types(infer_types(expression))
```

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


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']
        """
        token = self.getToken()
        while token.kind != TokenType.EOF:
            if token.kind != TokenType.WSP and token.kind != TokenType.COM \
                    and token.kind != TokenType.NLN:
                yield token
            token = self.getToken()

    def getToken(self):
        """
        Return the next token.
        TODO: Implement this method!
        """
```

```
104         token = None
105         return token
```

Parser.py

```
import sys

from Expression import *
from Lexer import Token, TokenType

"""
This file implements the parser of logic and arithmetic expressions.

Precedence table:
    1: not ~ ()
    2: *   /
    3: +   -
    4: <   <=   >=   >
    5: =
    6: and
    7: or
    8: if-then-else

Notice that not 2 < 3 must be a type error, as we are trying to apply a boolean
operation (not) onto a number.

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!

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

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

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

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

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

        >>> tk0 = Token('3', TokenType.NUM)
        >>> tk1 = Token('*', TokenType.MUL)
        >>> tk2 = Token('4', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2])
        >>> exp = parser.parse()
        >>> ev = EvalVisitor()
        >>> exp.accept(ev, None)
        12

        >>> 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 = EvalVisitor()
        >>> exp.accept(ev, None)
        -12

        >>> tk0 = Token('30', TokenType.NUM)
        >>> tk1 = Token('/', TokenType.DIV)
        >>> tk2 = Token('4', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2])
        >>> exp = parser.parse()
        >>> ev = EvalVisitor()
        >>> exp.accept(ev, None)
        7

        >>> tk0 = Token('3', TokenType.NUM)
        >>> tk1 = Token('+', TokenType.ADD)
        >>> tk2 = Token('4', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2])
        >>> exp = parser.parse()
        >>> ev = EvalVisitor()
        >>> exp.accept(ev, None)
        7

        >>> tk0 = Token('30', TokenType.NUM)
```

```
>>> tk1 = Token('-', TokenType.SUB)
>>> tk2 = Token('4', TokenType.NUM)
>>> parser = Parser([tk0, tk1, tk2])
>>> exp = parser.parse()
>>> ev = EvalVisitor()
>>> exp.accept(ev, None)
26

>>> 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 = EvalVisitor()
>>> exp.accept(ev, None)
14

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

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

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

>>> 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 = EvalVisitor()
>>> exp.accept(ev, None)
True

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

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

>>> tk0 = Token('let', TokenType.LET)
>>> tk1 = Token('v', TokenType.VAR)
>>> tk2 = Token('<-', TokenType.ASN)
>>> tk3 = Token('42', TokenType.NUM)
>>> tk4 = Token('in', TokenType.INX)
>>> tk5 = Token('v', TokenType.VAR)
>>> tk6 = Token('end', TokenType.END)
>>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6])
>>> exp = parser.parse()
>>> ev = EvalVisitor()
>>> exp.accept(ev, {})
42

>>> tk0 = Token('let', TokenType.LET)
>>> tk1 = Token('v', TokenType.VAR)
>>> tk2 = Token('<-', TokenType.ASN)
>>> tk3 = Token('21', TokenType.NUM)
>>> tk4 = Token('in', TokenType.INX)
>>> tk5 = Token('v', TokenType.VAR)
>>> tk6 = Token('+', TokenType.ADD)
>>> tk7 = Token('v', TokenType.VAR)
>>> tk8 = Token('end', TokenType.END)
>>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6, tk7, tk8])
>>> exp = parser.parse()
>>> ev = EvalVisitor()
```

```
        >>> exp.accept(ev, {})
        42

        >>> tk0 = Token('if', TokenType.IFX)
        >>> tk1 = Token('2', TokenType.NUM)
        >>> tk2 = Token('<', TokenType.LTH)
        >>> tk3 = Token('3', TokenType.NUM)
        >>> tk4 = Token('then', TokenType.THN)
        >>> tk5 = Token('1', TokenType.NUM)
        >>> tk6 = Token('else', TokenType.ELS)
        >>> tk7 = Token('2', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6, tk7])
        >>> exp = parser.parse()
        >>> ev = EvalVisitor()
        >>> exp.accept(ev, None)
        1

        >>> tk0 = Token('if', TokenType.IFX)
        >>> tk1 = Token('false', TokenType.FLS)
        >>> tk2 = Token('then', TokenType.THN)
        >>> tk3 = Token('1', TokenType.NUM)
        >>> tk4 = Token('else', TokenType.ELS)
        >>> tk5 = Token('2', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5])
        >>> exp = parser.parse()
        >>> ev = EvalVisitor()
        >>> exp.accept(ev, None)
        2
        """
        # TODO: implement this method.
        return None
```

Expression.py

```
"""
This file implements the data structure that represents the logic and
arithmetic expressions in our language.

IMPORTANT: There is no need to change this file to solve this VPL!
"""
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):
        """
        Example:
        >>> e = Var('var')
        >>> ev = EvalVisitor()
        >>> e.accept(ev, {'var': 42})
        42

        >>> e = Var('v42')
        >>> ev = EvalVisitor()
        >>> e.accept(ev, {'v42': True, 'v31': 5})
        True
        """
        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 acceptuation of such an expression is
    the boolean itself.
    """
    def __init__(self, bln):
        self.bln = bln
    def accept(self, visitor, arg):
        """
        Example:
        >>> e = Bln(True)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        True
        """
        return visitor.visit_bln(self, arg)

class Num(Expression):
    """
    This class represents expressions that are numbers. The acceptuation of such
    an expression is the number itself.
    """
    def __init__(self, num):
        self.num = num
    def accept(self, visitor, arg):
        """
        Example:
        >>> e = Num(3)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        3
        """
        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 acceptuation
    of such an expression is True if the subexpressions are the same, or false
    otherwise.
    """
    def accept(self, visitor, arg):
        """
        Example:
        >>> n1 = Num(3)
        >>> n2 = Num(4)
        >>> e = Eql(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        False

        >>> n1 = Num(3)
        >>> n2 = Num(3)
        >>> e = Eql(n1, n2)
```

```
104         >>> ev = EvalVisitor()
105         >>> e.accept(ev, None)
106         True
107         """
108         return visitor.visit_eql(self, arg)
109
110 class Add(BinaryExpression):
111     """
112     This class represents addition of two expressions. The acceptuation of such
113     an expression is the addition of the two subexpression's values.
114     """
115     def accept(self, visitor, arg):
116         """
117         Example:
118         >>> n1 = Num(3)
119         >>> n2 = Num(4)
120         >>> e = Add(n1, n2)
121         >>> ev = EvalVisitor()
122         >>> e.accept(ev, None)
123         7
124         """
125         return visitor.visit_add(self, arg)
126
127 class And(BinaryExpression):
128     """
129     This class represents the logical disjunction of two boolean expressions.
130     The evaluation of an expression of this kind is the logical AND of the two
131     subexpression's values.
132     """
133     def accept(self, visitor, arg):
134         """
135         Example:
136         >>> b1 = Bln(True)
137         >>> b2 = Bln(False)
138         >>> e = And(b1, b2)
139         >>> ev = EvalVisitor()
140         >>> e.accept(ev, None)
141         False
142
143         >>> b1 = Bln(True)
144         >>> b2 = Bln(True)
145         >>> e = And(b1, b2)
146         >>> ev = EvalVisitor()
147         >>> e.accept(ev, None)
148         True
149         """
150         return visitor.visit_and(self, arg)
151
152 class Or(BinaryExpression):
153     """
154     This class represents the logical conjunction of two boolean expressions.
155     The evaluation of an expression of this kind is the logical OR of the two
156     subexpression's values.
157     """
158     def accept(self, visitor, arg):
159         """
160         Example:
161         >>> b1 = Bln(True)
162         >>> b2 = Bln(False)
163         >>> e = Or(b1, b2)
164         >>> ev = EvalVisitor()
165         >>> e.accept(ev, None)
166         True
167
168         >>> b1 = Bln(False)
169         >>> b2 = Bln(False)
170         >>> e = Or(b1, b2)
171         >>> ev = EvalVisitor()
172         >>> e.accept(ev, None)
173         False
174         """
175         return visitor.visit_or(self, arg)
176
177 class Sub(BinaryExpression):
178     """
179     This class represents subtraction of two expressions. The acceptuation of such
180     an expression is the subtraction of the two subexpression's values.
181     """
182     def accept(self, visitor, arg):
183         """
184         Example:
185         >>> n1 = Num(3)
186         >>> n2 = Num(4)
187         >>> e = Sub(n1, n2)
188         >>> ev = EvalVisitor()
189         >>> e.accept(ev, None)
190         -1
191         """
192         return visitor.visit_sub(self, arg)
193
194 class Mul(BinaryExpression):
195     """
196     This class represents multiplication of two expressions. The acceptuation of
197     such an expression is the product of the two subexpression's values.
198     """
199     def accept(self, visitor, arg):
200         """
201         Example:
202         >>> n1 = Num(3)
203         >>> n2 = Num(4)
204         >>> e = Mul(n1, n2)
205         >>> ev = EvalVisitor()
206         >>> e.accept(ev, None)
```

```
        12
        """
        return visitor.visit_mul(self, arg)

class Div(BinaryExpression):
    """
    This class represents the integer division of two expressions. The
    acceptuation of such an expression is the integer quocient of the two
    subexpression's values.
    """
    def accept(self, visitor, arg):
        """
        Example:
        >>> n1 = Num(28)
        >>> n2 = Num(4)
        >>> e = Div(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        7
        >>> n1 = Num(22)
        >>> n2 = Num(4)
        >>> e = Div(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        5
        """
        return visitor.visit_div(self, arg)

class Leq(BinaryExpression):
    """
    This class represents comparison of two expressions using the
    less-than-or-equal comparator. The acceptuation of such an expression is a
    boolean value that is true if the left operand is less than or equal the
    right operand. It is false otherwise.
    """
    def accept(self, visitor, arg):
        """
        Example:
        >>> n1 = Num(3)
        >>> n2 = Num(4)
        >>> e = Leq(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        True
        >>> n1 = Num(3)
        >>> n2 = Num(3)
        >>> e = Leq(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        True
        >>> n1 = Num(4)
        >>> n2 = Num(3)
        >>> e = Leq(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        False
        """
        return visitor.visit_leq(self, arg)

class Lth(BinaryExpression):
    """
    This class represents comparison of two expressions using the
    less-than comparison operator. The acceptuation of such an expression is a
    boolean value that is true if the left operand is less than the right
    operand. It is false otherwise.
    """
    def accept(self, visitor, arg):
        """
        Example:
        >>> n1 = Num(3)
        >>> n2 = Num(4)
        >>> e = Lth(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        True
        >>> n1 = Num(3)
        >>> n2 = Num(3)
        >>> e = Lth(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        False
        >>> n1 = Num(4)
        >>> n2 = Num(3)
        >>> e = Lth(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        False
        """
        return visitor.visit_lth(self, arg)

class UnaryExpression(Expression):
    """
    This class represents unary expressions. A unary expression has only one
    sub-expression.
    """
    def __init__(self, exp):
        self.exp = exp

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

class Neg(UnaryExpression):
    """
```

```
    This expression represents the additive inverse of a number. The additive
    inverse of a number n is the number -n, so that the sum of both is zero.
    """
    def accept(self, visitor, arg):
        """
        Example:
        >>> n = Num(3)
        >>> e = Neg(n)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        -3
        >>> n = Num(0)
        >>> e = Neg(n)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        0
        """
        return visitor.visit_neg(self, arg)

class Not(UnaryExpression):
    """
    This expression represents the negation of a boolean. The negation of a
    boolean expression is the logical complement of that expression.
    """
    def accept(self, visitor, arg):
        """
        Example:
        >>> t = Bln(True)
        >>> e = Not(t)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        False
        >>> t = Bln(False)
        >>> e = Not(t)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        True
        """
        return visitor.visit_not(self, arg)

class Let(Expression):
    """
    This class represents a let expression. The semantics of a let expression,
    such as "let v <- e0 in e1" on an environment env is as follows:
    1. Evaluate e0 in the environment env, yielding e0_val
    2. Evaluate e1 in the new environment env' = env + {v:e0_val}
    """
    def __init__(self, identifier, exp_def, exp_body):
        self.identifier = identifier
        self.exp_def = exp_def
        self.exp_body = exp_body
    def accept(self, visitor, arg):
        """
        Example:
        >>> e = Let('v', Num(42), Var('v'))
        >>> ev = EvalVisitor()
        >>> e.accept(ev, {})
        42

        >>> e = Let('v', Num(40), Let('w', Num(2), Add(Var('v'), Var('w'))))
        >>> ev = EvalVisitor()
        >>> e.accept(ev, {})
        42

        >>> e = Let('v', Add(Num(40), Num(2)), Mul(Var('v'), Var('v')))
        >>> ev = EvalVisitor()
        >>> e.accept(ev, {})
        1764
        """
        return visitor.visit_let(self, arg)

class IfThenElse(Expression):
    """
    This class represents a conditional expression. The semantics an expression
    such as 'if B then E0 else E1' is as follows:
    1. Evaluate B. Call the result ValueB.
    2. If ValueB is True, then evalute E0 and return the result.
    3. If ValueB is False, then evaluate E1 and return the result.
    Notice that we only evaluate one of the two sub-expressions, not both. Thus,
    "if True then 0 else 1 div 0" will return 0 indeed.
    """
    def __init__(self, cond, e0, e1):
        self.cond = cond
        self.e0 = e0
        self.e1 = e1
    def accept(self, visitor, arg):
        """
        Example:
        >>> e = IfThenElse(Bln(True), Num(42), Num(30))
        >>> ev = EvalVisitor()
        >>> e.accept(ev, {})
        42

        >>> e = IfThenElse(Bln(False), Num(42), Num(30))
        >>> ev = EvalVisitor()
        >>> e.accept(ev, {})
        30
        """
        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


class CtrGenVisitor(Visitor):
    """
    This visitor creates constraints for a type-inference engine. Basically,
    it traverses the abstract-syntax tree of expressions, producing pairs like
    (type0, type1) on the way. A pair like (type0, type1) indicates that these
    two type variables are the same.

    Examples:
        >>> e = Let('v', Num(40), Let('w', Num(2), Add(Var('v'), Var('w'))))
        >>> ev = CtrGenVisitor()
        >>> sorted([str(ct) for ct in e.accept(ev, ev.fresh_type_var())])
        ["('TV_1', 'TV_2')", "('TV_2', 'TV_3')", "('v', <class 'int'>)", "('w', <class 'int'>)", "(<class 'int'>, 'TV_3')", "(<clas
    """

    def __init__(self):
        self.fresh_type_counter = 0

    def fresh_type_var(self):
        """
        Create a new type var using the current value of the fresh_type_counter.
        Two successive calls to this method will return different type names.
        Notice that the name of a type variable is always TV_x, where x is
        some integer number. That means that probably we would run into
```

```
        errors if someone declares a variable called, say, TV_1 or TV_2, as in
        "let TV_1 <- 1 in TV_1 end". But you can assume that such would never
        happen in the test cases. In practice, we should define a new class
        to represent type variables. But let's keep the implementation as
        simple as possible.

        Example:
            >>> ev = CtrGenVisitor()
            >>> [ev.fresh_type_var(), ev.fresh_type_var()]
            ['TV_1', 'TV_2']
        """
        self.fresh_type_counter += 1
        return f"TV_{self.fresh_type_counter}"

    """
    The CtrGenVisitor class creates constraints that, once solved, will give
    us the type of the different variables. Every accept method takes in
    two arguments (in addition to self):

    exp: is the expression that is being analyzed.
    type_var: that is a name that works as a placeholder for the type of the
    expression. Whenever we visit a new expression, we create a type variable
    to represent its type (you can do that with the method fresh_type_var).
    The only exception is the type of Var expressions. In this case, the type
    of a Var expression is the identifier of that expression.
    """

    def visit_var(self, exp, type_var):
        """
        Example:
            >>> e = Var('v')
            >>> ev = CtrGenVisitor()
            >>> e.accept(ev, ev.fresh_type_var())
            {('v', 'TV_1')}
        """
        return {(exp.identifier, type_var)}

    def visit_bln(self, exp, type_var):
        """
        Example:
            >>> e = Bln(True)
            >>> ev = CtrGenVisitor()
            >>> e.accept(ev, ev.fresh_type_var())
            {(<class 'bool'>, 'TV_1')}
        """
        return {(type(True), type_var)}

    def visit_num(self, exp, type_var):
        """
        Example:
            >>> e = Num(1)
            >>> ev = CtrGenVisitor()
            >>> e.accept(ev, ev.fresh_type_var())
            {(<class 'int'>, 'TV_1')}
        """
        return {(type(1), type_var)}

    def visit_eql(self, exp, type_var):
        """
        Example:
            >>> e = Eql(Num(1), Bln(True))
            >>> ev = CtrGenVisitor()
            >>> sorted([str(ct) for ct in e.accept(ev, ev.fresh_type_var())])
            ["(<class 'bool'>, 'TV_1')", "(<class 'bool'>, 'TV_2')", "(<class 'int'>, 'TV_2')"]

        Notice that if we have repeated constraints, they only appear once in
        the set of constraints (after all, it's a set!). As an example, we
        would have two occurrences of the pair (TV_2, int) in the following
        example:
            >>> e = Eql(Num(1), Num(2))
            >>> ev = CtrGenVisitor()
            >>> sorted([str(ct) for ct in e.accept(ev, ev.fresh_type_var())])
            ["(<class 'bool'>, 'TV_1')", "(<class 'int'>, 'TV_2')"]
        """
        # TODO: Implement this method!
        raise NotImplementedError

    def visit_and(self, exp, type_var):
        """
        Example:
            >>> e = And(Bln(False), Bln(True))
            >>> ev = CtrGenVisitor()
            >>> sorted([str(ct) for ct in e.accept(ev, ev.fresh_type_var())])
            ["(<class 'bool'>, 'TV_1')", "(<class 'bool'>, <class 'bool'>)"]

        In the above example, notice that we ended up getting a trivial
        constraint, e.g.: (<class 'bool'>, <class 'bool'>). That's alright:
        don't worry about these trivial constraints at this point. We can
        remove them from the set of constraints later on, when we try to
        solve them.
        """
        # TODO: Implement this method!
        raise NotImplementedError

    def visit_or(self, exp, type_var):
        """
        Example:
            >>> e = Or(Bln(False), Bln(True))
            >>> ev = CtrGenVisitor()
            >>> sorted([str(ct) for ct in e.accept(ev, ev.fresh_type_var())])
            ["(<class 'bool'>, 'TV_1')", "(<class 'bool'>, <class 'bool'>)"]
        """
        # TODO: Implement this method!
```

```
        raise NotImplementedError

    def visit_add(self, exp, type_var):
        """
        Example:
            >>> e = Add(Num(1), Num(2))
            >>> ev = CtrGenVisitor()
            >>> sorted([str(ct) for ct in e.accept(ev, ev.fresh_type_var())])
            ["(<class 'int'>, 'TV_1')", "(<class 'int'>, <class 'int'>)"]
        """
        # TODO: Implement this method!
        raise NotImplementedError

    def visit_sub(self, exp, type_var):
        """
        Example:
            >>> e = Sub(Num(1), Num(2))
            >>> ev = CtrGenVisitor()
            >>> sorted([str(ct) for ct in e.accept(ev, ev.fresh_type_var())])
            ["(<class 'int'>, 'TV_1')", "(<class 'int'>, <class 'int'>)"]
        """
        # TODO: Implement this method!
        raise NotImplementedError

    def visit_mul(self, exp, type_var):
        """
        Example:
            >>> e = Mul(Num(1), Num(2))
            >>> ev = CtrGenVisitor()
            >>> sorted([str(ct) for ct in e.accept(ev, ev.fresh_type_var())])
            ["(<class 'int'>, 'TV_1')", "(<class 'int'>, <class 'int'>)"]
        """
        # TODO: Implement this method!
        raise NotImplementedError

    def visit_div(self, exp, type_var):
        """
        Example:
            >>> e = Div(Num(1), Num(2))
            >>> ev = CtrGenVisitor()
            >>> sorted([str(ct) for ct in e.accept(ev, ev.fresh_type_var())])
            ["(<class 'int'>, 'TV_1')", "(<class 'int'>, <class 'int'>)"]
        """
        # TODO: Implement this method!
        raise NotImplementedError

    def visit_leq(self, exp, type_var):
        """
        Example:
            >>> e = Leq(Num(1), Num(2))
            >>> ev = CtrGenVisitor()
            >>> sorted([str(ct) for ct in e.accept(ev, ev.fresh_type_var())])
            ["(<class 'bool'>, 'TV_1')", "(<class 'int'>, <class 'int'>)"]
        """
        # TODO: Implement this method!
        raise NotImplementedError

    def visit_lth(self, exp, type_var):
        """
        Example:
            >>> e = Lth(Num(1), Num(2))
            >>> ev = CtrGenVisitor()
            >>> sorted([str(ct) for ct in e.accept(ev, ev.fresh_type_var())])
            ["(<class 'bool'>, 'TV_1')", "(<class 'int'>, <class 'int'>)"]
        """
        # TODO: Implement this method!
        raise NotImplementedError

    def visit_neg(self, exp, type_var):
        """
        Example:
            >>> e = Neg(Num(1))
            >>> ev = CtrGenVisitor()
            >>> sorted([str(ct) for ct in e.accept(ev, ev.fresh_type_var())])
            ["(<class 'int'>, 'TV_1')", "(<class 'int'>, <class 'int'>)"]
        """
        # TODO: Implement this method!
        raise NotImplementedError

    def visit_not(self, exp, type_var):
        """
        Example:
            >>> e = Not(Bln(True))
            >>> ev = CtrGenVisitor()
            >>> sorted([str(ct) for ct in e.accept(ev, ev.fresh_type_var())])
            ["(<class 'bool'>, 'TV_1')", "(<class 'bool'>, <class 'bool'>)"]
        """
        # TODO: Implement this method!
        raise NotImplementedError

    def visit_let(self, exp, type_var):
        """
        Example:
            >>> e = Let('v', Num(42), Var('v'))
            >>> ev = CtrGenVisitor()
            >>> sorted([str(ct) for ct in e.accept(ev, ev.fresh_type_var())])
            ["('TV_1', 'TV_2')", "('v', 'TV_2')", "(<class 'int'>, 'v')"]
        """
        # TODO: Implement this method!
        raise NotImplementedError

    def visit_ifThenElse(self, exp, type_var):
        """
        Example:
```

```
310         Example:
311             >>> e = IfThenElse(Bln(True), Num(42), Num(30))
312             >>> ev = CtrGenVisitor()
313             >>> sorted([str(ct) for ct in e.accept(ev, ev.fresh_type_var())])
314             ["('TV_1', 'TV_2')", "(<class 'bool'>, <class 'bool'>)", "(<class 'int'>, 'TV_2')"]
315         """
316         # TODO: Implement this method!
317         raise NotImplementedError
```

Unifier.py

```
from Expression import *
from Visitor import *
import sys


def unify(constraints, sets):
    """
    This function unifies all the type variables in the list of constraints;
    thus, producing a set of unifiers.

    Example:
        >>> sets = unify([('a', type(1))], {})
        >>> integers = sets[type(1)] - {type(1)}
        >>> sorted(integers)
        ['a']

        >>> sets = unify([(type(1), 'b'), ('a', type(1))], {})
        >>> integers = sets[type(1)] - {type(1)}
        >>> sorted(integers)
        ['a', 'b']

        >>> sets = unify([(type(True), 'b'), ('a', type(1))], {})
        >>> booleans = sets[type(True)] - {type(True)}
        >>> sorted(booleans)
        ['b']

        >>> sets = unify([(type(True), 'b'), ('a', type(1))], {})
        >>> integers = sets[type(1)] - {type(1)}
        >>> sorted(integers)
        ['a']

        >>> sets = unify([('a', 'TV_1'), ('b', 'TV_2'), ('TV_2', type(1)), ('TV_1', type(1))], {})
        >>> integers = sets[type(1)] - {type(1)}
        >>> sorted(integers)
        ['TV_1', 'TV_2', 'a', 'b']

    Notice that at this stage, we still allow sets with invalid types. For
    instance, the set associated with 'b' in the example below will contain
    four elements, namely: {<class 'bool'>, <class 'int'>, 'b', 'a'}:
        >>> sets = unify([(type(True), 'b'), ('a', type(1)), ('a', 'b')], {})
        >>> len(sets['b'])
        4
    """
    # TODO: Implement this method!
    raise NotImplementedError



def name_sets(sets):
    """
    This method replaces type sets with "canonical type names". A canonical
    type name is the name of a type set. For instance, the type set
    {'a', 'b', type(int)} has the canonical name type(int)

    Notice that this method produces two types of error messages:
    * Polymorphic type: if any canonical type set is empty
    * Ambiguous type: if any canonical type set contains more than one element.
    In both cases, if any of these errors happen, the program should stop with
    the following error message: 'Type error'

    Example:
        >>> sets = name_sets({'a': {'a', 'b', type(1)}, 'b': {'a', 'b', type(1)}})
        >>> [sets['a'], sets['b']]
        [<class 'int'>, <class 'int'>]

        >>> sets = name_sets({'a': {'a', type(1)}, 'b': {'b', type(True)}})
        >>> [sets['a'], sets['b']]
        [<class 'int'>, <class 'bool'>]
    """

    # TODO: Implement this method!
    raise NotImplementedError



def infer_types(expression):
    """
    This method maps all the program variables to type names. We have
    implemented this method for you. This implementation might help you to
    understand how the other two methods, unify and name_sets are meant to
    work.

    Example:
        >>> e = Let('v', Num(42), Var('v'))
        >>> type_names = infer_types(e)
        >>> type_names['v']
        <class 'int'>

        >>> e = Let('v', Num(1), Let('y', Var('v'), Var('y')))
        >>> type_names = infer_types(e)
        >>> [type_names['v'], type_names['y']]
        [<class 'int'>, <class 'int'>]

        >>> e0 = Let('v', Num(1), Let('y', Var('v'), Var('y')))
        >>> e1 = IfThenElse(Lth(e0, Num(2)), Bln(True), Bln(False))
        >>> e2 = Let('w', e1, And(Var('w'), Var('w')))
        >>> type_names = infer_types(e2)
        >>> [type_names['v'], type_names['w'], type_names['y']]
        [<class 'int'>, <class 'bool'>, <class 'int'>]
    """
    ev = CtrGenVisitor()
    constraints = list(expression.accept(ev, ev.fresh_type_var()))
    type_sets = unify(constraints, {})
```

```
104       return name_sets(type_sets)
```

VPL