

2025_2 - COMPILADORES - METATURMA

[PAÍNEL](#) > [MINHAS TURMAS](#) > [2025_2 - COMPILADORES - METATURMA](#) > [LABORATÓRIOS DE PROGRAMAÇÃO VIRTUAL](#)
> [AV7 - INFERÊNCIA DE TIPOS](#)

[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 atravessa 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`:

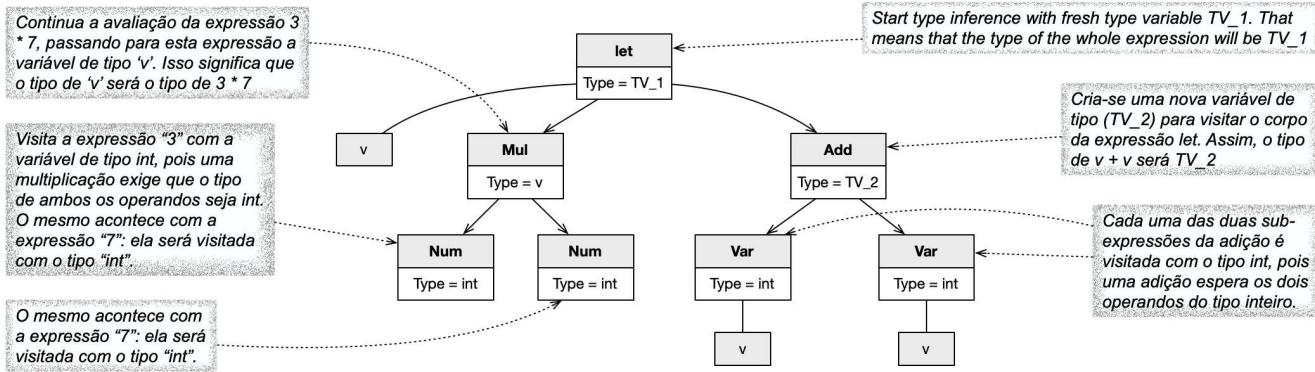


Figura 1: Como funciona a criação de variáveis de tipo para tipar uma expressão como "let v <- 3 * 7 in v + v end".

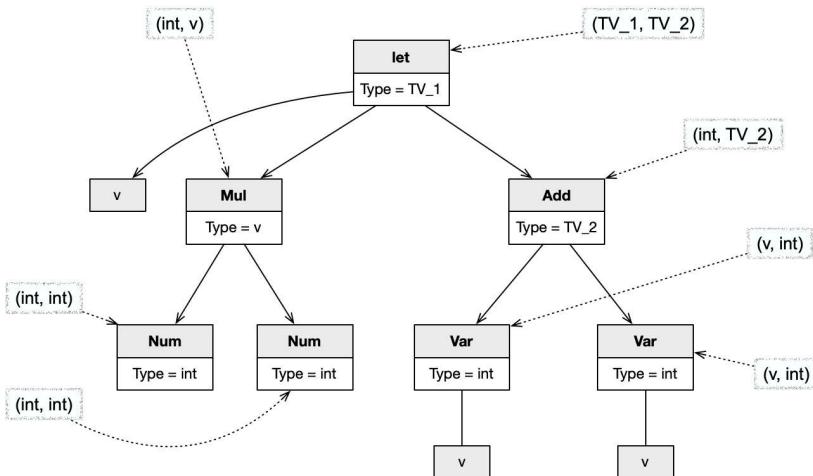


Figura 2: Lista de relações produzidas durante a visitação da expressão "let v <- 3 * 7 in v + v end"

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

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.

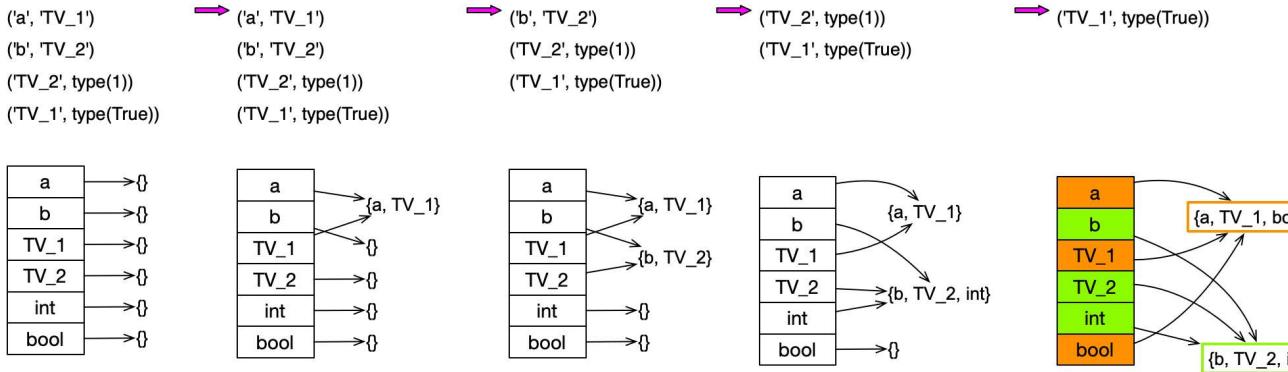
O visitor que gera relações de equivalência irá gerar o seguinte conjunto de equivalências para a expressão acima: { (int, v), (TV_1, TV_2), (int, int), (int, TV_2), (v, int), (v, int) }. Os nomes TV_1 e TV_2 são variáveis de tipos, criadas para unificar os tipos das diferentes sub-expressões que formam 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, substitua `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`

```
1 import sys
2 from Unifier import *
3 from Lexer import Lexer
4 from Parser import Parser
5
6 def print_types(type_names):
7     """
8         This method runs the type inference engine, and prints the type of
9         variables 'x', 'y' and 'z', if they exist in the test case.
10    """
11    print(f"Type(v): {type_names.setdefault('v', type(None)).__name__}")
12    print(f"Type(w): {type_names.setdefault('w', type(None)).__name__}")
13    print(f"Type(x): {type_names.setdefault('x', type(None)).__name__}")
14    print(f"Type(y): {type_names.setdefault('y', type(None)).__name__}")
15    print(f"Type(z): {type_names.setdefault('z', type(None)).__name__}")
16
17 if __name__ == "__main__":
18     """
19         Este arquivo nao deve ser alterado, mas deve ser enviado para resolver o
20         VPL. O arquivo contem o codigo que testa a implementacao do parser.
21     """
22     text = sys.stdin.read()
23     lexer = Lexer(text)
24     parser = Parser(lexer.tokens())
25     expression = parser.parse()
26     print_types(infer_types(expression))
```

`Lexer.py`

```

1 import sys
2 import enum
3
4
5 class Token:
6     """
7         This class contains the definition of Tokens. A token has two fields: its
8         text and its kind. The "kind" of a token is a constant that identifies it
9         uniquely. See the TokenType to know the possible identifiers (if you want).
10        You don't need to change this class.
11    """
12    def __init__(self, tokenText, tokenKind):
13        # The token's actual text. Used for identifiers, strings, and numbers.
14        self.text = tokenText
15        # The TokenType that this token is classified as.
16        self.kind = tokenKind
17
18
19 class TokenType(enum.Enum):
20     """
21         These are the possible tokens. You don't need to change this class at all.
22     """
23     EOF = -1 # End of file
24     NLN = 0 # New line
25     WSP = 1 # White Space
26     COM = 2 # Comment
27     NUM = 3 # Number (integers)
28     STR = 4 # Strings
29     TRU = 5 # The constant true
30     FLS = 6 # The constant false
31     VAR = 7 # An identifier
32     LET = 8 # The 'let' of the let expression
33     INX = 9 # The 'in' of the let expression
34     END = 10 # The 'end' of the let expression
35     EQL = 201 # x = y
36     ADD = 202 # x + y
37     SUB = 203 # x - y
38     MUL = 204 # x * y
39     DIV = 205 # x / y
40     LEQ = 206 # x <= y
41     LTH = 207 # x < y
42     NEG = 208 # ~x
43     NOT = 209 # not x
44     LPR = 210 # (
45     RPR = 211 # )
46     ASN = 212 # The assignment '<->' operator
47     ORX = 213 # x or y
48     AND = 214 # x and y
49     IFX = 215 # The 'if' of a conditional expression
50     THN = 216 # The 'then' of a conditional expression
51     ELS = 217 # The 'else' of a conditional expression
52
53
54 class Lexer:
55
56     def __init__(self, source):
57         """
58             The constructor of the lexer. It receives the string that shall be
59             scanned.
60             TODO: You will need to implement this method.
61         """
62         pass
63
64     def tokens(self):
65         """
66             This method is a token generator: it converts the string encapsulated
67             into this object into a sequence of Tokens. Notice that this method
68             filters out three kinds of tokens: white-spaces, comments and new lines.
69
70             Examples:
71
72             >>> l = Lexer("1 + 3")
73             >>> [tk.kind for tk in l.tokens()]
74             [<TokenType.NUM: 3>, <TokenType.ADD: 202>, <TokenType.NUM: 3>]
75
76             >>> l = Lexer('1 * 2\n')
77             >>> [tk.kind for tk in l.tokens()]
78             [<TokenType.NUM: 3>, <TokenType.MUL: 204>, <TokenType.NUM: 3>]
79
80             >>> l = Lexer('1 * 2 -- 3\n')
81             >>> [tk.kind for tk in l.tokens()]
82             [<TokenType.NUM: 3>, <TokenType.MUL: 204>, <TokenType.NUM: 3>]
83
84             >>> l = Lexer("1 + var")
85             >>> [tk.kind for tk in l.tokens()]
86             [<TokenType.NUM: 3>, <TokenType.ADD: 202>, <TokenType.VAR: 7>]
87
88             >>> l = Lexer("let v <- 2 in v end")
89             >>> [tk.kind.name for tk in l.tokens()]
90             ['LET', 'VAR', 'ASN', 'NUM', 'INX', 'VAR', 'END']
91
92             token = self.getToken()
93             while token.kind != TokenType.EOF:
94                 if token.kind != TokenType.WSP and token.kind != TokenType.COM \
95                     and token.kind != TokenType.NLN:
96                     yield token
97                 token = self.getToken()
98
99     def getToken(self):
100         """
101             Return the next token.
102             TODO: Implement this method!
103         """

```

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

Parser.py

```

1 import sys
2
3 from Expression import *
4 from Lexer import Token, TokenType
5
6 """
7 This file implements the parser of logic and arithmetic expressions.
8
9 Precedence table:
10    1: not ~ ()
11    2: * /
12    3: + -
13    4: < <= >= >
14    5: =
15    6: and
16    7: or
17    8: if-then-else
18
19 Notice that not 2 < 3 must be a type error, as we are trying to apply a boolean
20 operation (not) onto a number.
21
22 References:
23     see https://www.engr.mun.ca/~theo/Misc/exp\_parsing.htm#classic
24 """
25
26 class Parser:
27     def __init__(self, tokens):
28         """
29             Initializes the parser. The parser keeps track of the list of tokens
30             and the current token. For instance:
31             """
32         self.tokens = list(tokens)
33         self.cur_token_idx = 0 # This is just a suggestion!
34
35     def parse(self):
36         """
37             Returns the expression associated with the stream of tokens.
38
39             Examples:
40             >>> parser = Parser([Token('123', TokenType.NUM)])
41             >>> exp = parser.parse()
42             >>> ev = EvalVisitor()
43             >>> exp.accept(ev, None)
44             123
45
46             >>> parser = Parser([Token('True', TokenType.TRU)])
47             >>> exp = parser.parse()
48             >>> ev = EvalVisitor()
49             >>> exp.accept(ev, None)
50             True
51
52             >>> parser = Parser([Token('False', TokenType.FLS)])
53             >>> exp = parser.parse()
54             >>> ev = EvalVisitor()
55             >>> exp.accept(ev, None)
56             False
57
58             >>> tk0 = Token('~', TokenType.NEG)
59             >>> tk1 = Token('123', TokenType.NUM)
60             >>> parser = Parser([tk0, tk1])
61             >>> exp = parser.parse()
62             >>> ev = EvalVisitor()
63             >>> exp.accept(ev, None)
64             -123
65
66             >>> tk0 = Token('3', TokenType.NUM)
67             >>> tk1 = Token('*', TokenType.MUL)
68             >>> tk2 = Token('4', TokenType.NUM)
69             >>> parser = Parser([tk0, tk1, tk2])
70             >>> exp = parser.parse()
71             >>> ev = EvalVisitor()
72             >>> exp.accept(ev, None)
73             12
74
75             >>> tk0 = Token('3', TokenType.NUM)
76             >>> tk1 = Token('*', TokenType.MUL)
77             >>> tk2 = Token('~', TokenType.NEG)
78             >>> tk3 = Token('4', TokenType.NUM)
79             >>> parser = Parser([tk0, tk1, tk2, tk3])
80             >>> exp = parser.parse()
81             >>> ev = EvalVisitor()
82             >>> exp.accept(ev, None)
83             -12
84
85             >>> tk0 = Token('30', TokenType.NUM)
86             >>> tk1 = Token('/', TokenType.DIV)
87             >>> tk2 = Token('4', TokenType.NUM)
88             >>> parser = Parser([tk0, tk1, tk2])
89             >>> exp = parser.parse()
90             >>> ev = EvalVisitor()
91             >>> exp.accept(ev, None)
92             7
93
94             >>> tk0 = Token('3', TokenType.NUM)
95             >>> tk1 = Token('+', TokenType.ADD)
96             >>> tk2 = Token('4', TokenType.NUM)
97             >>> parser = Parser([tk0, tk1, tk2])
98             >>> exp = parser.parse()
99             >>> ev = EvalVisitor()
100            >>> exp.accept(ev, None)
101            7
102
103            >>> tk0 = Token('30', TokenType.NUM)

```

```

104     >>> tk1 = Token('-', TokenType.SUB)
105     >>> tk2 = Token('4', TokenType.NUM)
106     >>> parser = Parser([tk0, tk1, tk2])
107     >>> exp = parser.parse()
108     >>> ev = EvalVisitor()
109     >>> exp.accept(ev, None)
110     26
111
112     >>> tk0 = Token('2', TokenType.NUM)
113     >>> tk1 = Token('*', TokenType.MUL)
114     >>> tk2 = Token('(', TokenType.LPR)
115     >>> tk3 = Token('3', TokenType.NUM)
116     >>> tk4 = Token('+', TokenType.ADD)
117     >>> tk5 = Token('4', TokenType.NUM)
118     >>> tk6 = Token(')', TokenType.RPR)
119     >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6])
120     >>> exp = parser.parse()
121     >>> ev = EvalVisitor()
122     >>> exp.accept(ev, None)
123     14
124
125     >>> tk0 = Token('4', TokenType.NUM)
126     >>> tk1 = Token('==', TokenType.EQL)
127     >>> tk2 = Token('4', TokenType.NUM)
128     >>> parser = Parser([tk0, tk1, tk2])
129     >>> exp = parser.parse()
130     >>> ev = EvalVisitor()
131     >>> exp.accept(ev, None)
132     True
133
134     >>> tk0 = Token('4', TokenType.NUM)
135     >>> tk1 = Token('<=', TokenType.LEQ)
136     >>> tk2 = Token('4', TokenType.NUM)
137     >>> parser = Parser([tk0, tk1, tk2])
138     >>> exp = parser.parse()
139     >>> ev = EvalVisitor()
140     >>> exp.accept(ev, None)
141     True
142
143     >>> tk0 = Token('4', TokenType.NUM)
144     >>> tk1 = Token('<', TokenType.LTH)
145     >>> tk2 = Token('4', TokenType.NUM)
146     >>> parser = Parser([tk0, tk1, tk2])
147     >>> exp = parser.parse()
148     >>> ev = EvalVisitor()
149     >>> exp.accept(ev, None)
150     False
151
152     >>> tk0 = Token('not', TokenType.NOT)
153     >>> tk1 = Token('(', TokenType.LPR)
154     >>> tk2 = Token('4', TokenType.NUM)
155     >>> tk3 = Token('<', TokenType.LTH)
156     >>> tk4 = Token('4', TokenType.NUM)
157     >>> tk5 = Token(')', TokenType.RPR)
158     >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5])
159     >>> exp = parser.parse()
160     >>> ev = EvalVisitor()
161     >>> exp.accept(ev, None)
162     True
163
164     >>> tk0 = Token('true', TokenType.TRU)
165     >>> tk1 = Token('or', TokenType.ORX)
166     >>> tk2 = Token('false', TokenType.FLS)
167     >>> parser = Parser([tk0, tk1, tk2])
168     >>> exp = parser.parse()
169     >>> ev = EvalVisitor()
170     >>> exp.accept(ev, None)
171     True
172
173     >>> tk0 = Token('true', TokenType.TRU)
174     >>> tk1 = Token('and', TokenType.AND)
175     >>> tk2 = Token('false', TokenType.FLS)
176     >>> parser = Parser([tk0, tk1, tk2])
177     >>> exp = parser.parse()
178     >>> ev = EvalVisitor()
179     >>> exp.accept(ev, None)
180     False
181
182     >>> tk0 = Token('let', TokenType.LET)
183     >>> tk1 = Token('v', TokenType.VAR)
184     >>> tk2 = Token('<', TokenType.ASN)
185     >>> tk3 = Token('42', TokenType.NUM)
186     >>> tk4 = Token('in', TokenType.INX)
187     >>> tk5 = Token('v', TokenType.VAR)
188     >>> tk6 = Token('end', TokenType.END)
189     >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6])
190     >>> exp = parser.parse()
191     >>> ev = EvalVisitor()
192     >>> exp.accept(ev, {})
193     42
194
195     >>> tk0 = Token('let', TokenType.LET)
196     >>> tk1 = Token('v', TokenType.VAR)
197     >>> tk2 = Token('<', TokenType.ASN)
198     >>> tk3 = Token('21', TokenType.NUM)
199     >>> tk4 = Token('in', TokenType.INX)
200     >>> tk5 = Token('v', TokenType.VAR)
201     >>> tk6 = Token('+', TokenType.ADD)
202     >>> tk7 = Token('v', TokenType.VAR)
203     >>> tk8 = Token('end', TokenType.END)
204     >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6, tk7, tk8])
205     >>> exp = parser.parse()
206     >>> ev = EvalVisitor()
207     ...
```

```

```
20/
208
209
210 >>> exp.accept(ev, {})
211
212 42
213
214 >>> tk0 = Token('if', TokenType.IFX)
215 >>> tk1 = Token('2', TokenType.NUM)
216 >>> tk2 = Token('<', TokenType.LTH)
217 >>> tk3 = Token('3', TokenType.NUM)
218 >>> tk4 = Token('then', TokenType.THN)
219 >>> tk5 = Token('1', TokenType.NUM)
220 >>> tk6 = Token('else', TokenType.ELS)
221 >>> tk7 = Token('2', TokenType.NUM)
222 >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5, tk6, tk7])
223 >>> exp = parser.parse()
224 >>> ev = EvalVisitor()
225 >>> exp.accept(ev, None)
226
227 1
228
229
230 >>> tk0 = Token('if', TokenType.IFX)
231 >>> tk1 = Token('false', TokenType.FLS)
232 >>> tk2 = Token('then', TokenType.THN)
233 >>> tk3 = Token('1', TokenType.NUM)
234 >>> tk4 = Token('else', TokenType.ELS)
235 >>> tk5 = Token('2', TokenType.NUM)
236 >>> parser = Parser([tk0, tk1, tk2, tk3, tk4, tk5])
237 >>> exp = parser.parse()
238 >>> ev = EvalVisitor()
239 >>> exp.accept(ev, None)
240
241 2
242
243 """
244
245 # TODO: implement this method.
246
247 return None
```

Expression.py

```

1 """
2 This file implements the data structure that represents the logic and
3 arithmetic expressions in our language.
4
5 IMPORTANT: There is no need to change this file to solve this VPL!
6 """
7 from abc import ABC, abstractmethod
8 from Visitor import *
9
10 class Expression(ABC):
11 @abstractmethod
12 def accept(self, visitor, arg):
13 raise NotImplementedError
14
15 class Var(Expression):
16 """
17 This class represents expressions that are identifiers. The value of an
18 identifier is the value associated with it in the environment table.
19 """
20 def __init__(self, identifier):
21 self.identifier = identifier
22 def accept(self, visitor, arg):
23 """
24 Example:
25 >>> e = Var('var')
26 >>> ev = EvalVisitor()
27 >>> e.accept(ev, {'var': 42})
28 42
29
30 >>> e = Var('v42')
31 >>> ev = EvalVisitor()
32 >>> e.accept(ev, {'v42': True, 'v31': 5})
33 True
34 """
35 return visitor.visit_var(self, arg)
36
37 class Bln(Expression):
38 """
39 This class represents expressions that are boolean values. There are only
40 two boolean values: true and false. The acceptuation of such an expression is
41 the boolean itself.
42 """
43 def __init__(self, bln):
44 self.bln = bln
45 def accept(self, visitor, arg):
46 """
47 Example:
48 >>> e = Bln(True)
49 >>> ev = EvalVisitor()
50 >>> e.accept(ev, None)
51 True
52 """
53 return visitor.visit_bln(self, arg)
54
55 class Num(Expression):
56 """
57 This class represents expressions that are numbers. The acceptuation of such
58 an expression is the number itself.
59 """
60 def __init__(self, num):
61 self.num = num
62 def accept(self, visitor, arg):
63 """
64 Example:
65 >>> e = Num(3)
66 >>> ev = EvalVisitor()
67 >>> e.accept(ev, None)
68 3
69 """
70 return visitor.visit_num(self, arg)
71
72 class BinaryExpression(Expression):
73 """
74 This class represents binary expressions. A binary expression has two
75 sub-expressions: the left operand and the right operand.
76 """
77 def __init__(self, left, right):
78 self.left = left
79 self.right = right
80
81 @abstractmethod
82 def accept(self, visitor, arg):
83 raise NotImplementedError
84
85 class Eql(BinaryExpression):
86 """
87 This class represents the equality between two expressions. The acceptuation
88 of such an expression is True if the subexpressions are the same, or false
89 otherwise.
90 """
91 def accept(self, visitor, arg):
92 """
93 Example:
94 >>> n1 = Num(3)
95 >>> n2 = Num(4)
96 >>> e = Eql(n1, n2)
97 >>> ev = EvalVisitor()
98 >>> e.accept(ev, None)
99 False
100
101 >>> n1 = Num(3)
102 >>> n2 = Num(3)
103 >>> 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)
207 12

```

```

20/
208 12
209 """
210 return visitor.visit_mul(self, arg)
211
211 class Div(BinaryExpression):
212 """
213 This class represents the integer division of two expressions. The
214 acceptuation of such an expression is the integer quotient of the two
215 subexpression's values.
216 """
217 def accept(self, visitor, arg):
218 """
219 Example:
220 >>> n1 = Num(28)
221 >>> n2 = Num(4)
222 >>> e = Div(n1, n2)
223 >>> ev = EvalVisitor()
224 >>> e.accept(ev, None)
225 7
226
226 >>> n1 = Num(22)
227 >>> n2 = Num(4)
228 >>> e = Div(n1, n2)
229 >>> ev = EvalVisitor()
230 >>> e.accept(ev, None)
231 5
232 """
233
233 return visitor.visit_div(self, arg)
234
235 class Leq(BinaryExpression):
236 """
237 This class represents comparison of two expressions using the
238 less-than-or-equal comparator. The acceptuation of such an expression is a
239 boolean value that is true if the left operand is less than or equal the
240 right operand. It is false otherwise.
241 """
242 def accept(self, visitor, arg):
243 """
244 Example:
245 >>> n1 = Num(3)
246 >>> n2 = Num(4)
247 >>> e = Leq(n1, n2)
248 >>> ev = EvalVisitor()
249 >>> e.accept(ev, None)
250 True
251
252 >>> n1 = Num(3)
253 >>> n2 = Num(3)
254 >>> e = Leq(n1, n2)
255 >>> ev = EvalVisitor()
256 >>> e.accept(ev, None)
257 True
258
259 >>> n1 = Num(4)
260 >>> n2 = Num(3)
261 >>> e = Leq(n1, n2)
262 >>> ev = EvalVisitor()
263 >>> e.accept(ev, None)
264 False
265 """
265
266 return visitor.visit_leq(self, arg)
266
267 class Lth(BinaryExpression):
268 """
269 This class represents comparison of two expressions using the
270 less-than comparison operator. The acceptuation of such an expression is a
271 boolean value that is true if the left operand is less than the right
272 operand. It is false otherwise.
273 """
273 def accept(self, visitor, arg):
274 """
275 Example:
276 >>> n1 = Num(3)
277 >>> n2 = Num(4)
278 >>> e = Lth(n1, n2)
279 >>> ev = EvalVisitor()
280 >>> e.accept(ev, None)
281 True
282
283 >>> n1 = Num(3)
284 >>> n2 = Num(3)
285 >>> e = Lth(n1, n2)
286 >>> ev = EvalVisitor()
287 >>> e.accept(ev, None)
288 False
289
290 >>> n1 = Num(4)
291 >>> n2 = Num(3)
292 >>> e = Lth(n1, n2)
293 >>> ev = EvalVisitor()
294 >>> e.accept(ev, None)
295 False
295 """
295
295 return visitor.visit_lth(self, arg)
296
297 class UnaryExpression(Expression):
298 """
299 This class represents unary expressions. A unary expression has only one
300 sub-expression.
301 """
302 def __init__(self, exp):
303 self.exp = exp
304
305 @abstractmethod
306 def accept(self, visitor, arg):
307 raise NotImplementedError
308
309 class Neg(UnaryExpression):
310 """

```

```

311 This expression represents the additive inverse of a number. The additive
312 inverse of a number n is the number -n, so that the sum of both is zero.
313 """
314 def accept(self, visitor, arg):
315 """
316 Example:
317 >>> n = Num(3)
318 >>> e = Neg(n)
319 >>> ev = EvalVisitor()
320 >>> e.accept(ev, None)
321 -3
322 >>> n = Num(0)
323 >>> e = Neg(n)
324 >>> ev = EvalVisitor()
325 >>> e.accept(ev, None)
326 0
327 """
328 return visitor.visit_neg(self, arg)
329
330 class Not(UnaryExpression):
331 """
332 This expression represents the negation of a boolean. The negation of a
333 boolean expression is the logical complement of that expression.
334 """
335 def accept(self, visitor, arg):
336 """
337 Example:
338 >>> t = Bln(True)
339 >>> e = Not(t)
340 >>> ev = EvalVisitor()
341 >>> e.accept(ev, None)
342 False
343 >>> t = Bln(False)
344 >>> e = Not(t)
345 >>> ev = EvalVisitor()
346 >>> e.accept(ev, None)
347 True
348 """
349 return visitor.visit_not(self, arg)
350
351 class Let(Expression):
352 """
353 This class represents a let expression. The semantics of a let expression,
354 such as "let v <- e0 in e1" on an environment env is as follows:
355 1. Evaluate e0 in the environment env, yielding e0_val
356 2. Evaluate e1 in the new environment env' = env + {v:e0_val}
357 """
358 def __init__(self, identifier, exp_def, exp_body):
359 self.identifier = identifier
360 self.exp_def = exp_def
361 self.exp_body = exp_body
362 def accept(self, visitor, arg):
363 """
364 Example:
365 >>> e = Let('v', Num(42), Var('v'))
366 >>> ev = EvalVisitor()
367 >>> e.accept(ev, {})
368 42
369
370 >>> e = Let('v', Num(40), Let('w', Num(2), Add(Var('v'), Var('w'))))
371 >>> ev = EvalVisitor()
372 >>> e.accept(ev, {})
373 42
374
375 >>> e = Let('v', Add(Num(40), Num(2)), Mul(Var('v'), Var('v')))
376 >>> ev = EvalVisitor()
377 >>> e.accept(ev, {})
378 1764
379 """
380 return visitor.visit_let(self, arg)
381
382 class IfThenElse(Expression):
383 """
384 This class represents a conditional expression. The semantics an expression
385 such as 'if B then E0 else E1' is as follows:
386 1. Evaluate B. Call the result ValueB.
387 2. If ValueB is True, then evaluate E0 and return the result.
388 3. If ValueB is False, then evaluate E1 and return the result.
389 Notice that we only evaluate one of the two sub-expressions, not both. Thus,
390 "if True then 0 else 1 div 0" will return 0 indeed.
391 """
392 def __init__(self, cond, e0, e1):
393 self.cond = cond
394 self.e0 = e0
395 self.e1 = e1
396 def accept(self, visitor, arg):
397 """
398 Example:
399 >>> e = IfThenElse(Bln(True), Num(42), Num(30))
400 >>> ev = EvalVisitor()
401 >>> e.accept(ev, {})
402 42
403
404 >>> e = IfThenElse(Bln(False), Num(42), Num(30))
405 >>> ev = EvalVisitor()
406 >>> e.accept(ev, {})
407 30
408 """
409 return visitor.visit_ifThenElse(self, arg)

```

```

1 import sys
2 from abc import ABC, abstractmethod
3 from Expression import *
4
5
6 class Visitor(ABC):
7 """
8 The visitor pattern consists of two abstract classes: the Expression and the
9 Visitor. The Expression class defines on method: 'accept(visitor, args)'.
10 This method takes in an implementation of a visitor, and the arguments that
11 are passed from expression to expression. The Visitor class defines one
12 specific method for each subclass of Expression. Each instance of such a
13 subclass will invoke the right visiting method.
14 """
15
16 @abstractmethod
17 def visit_var(self, exp, arg):
18 pass
19
20 @abstractmethod
21 def visit_bln(self, exp, arg):
22 pass
23
24 @abstractmethod
25 def visit_num(self, exp, arg):
26 pass
27
28 @abstractmethod
29 def visit_eql(self, exp, arg):
30 pass
31
32 @abstractmethod
33 def visit_and(self, exp, arg):
34 pass
35
36 @abstractmethod
37 def visit_or(self, exp, arg):
38 pass
39
40 @abstractmethod
41 def visit_add(self, exp, arg):
42 pass
43
44 @abstractmethod
45 def visit_sub(self, exp, arg):
46 pass
47
48 @abstractmethod
49 def visit_mul(self, exp, arg):
50 pass
51
52 @abstractmethod
53 def visit_div(self, exp, arg):
54 pass
55
56 @abstractmethod
57 def visit_leq(self, exp, arg):
58 pass
59
60 @abstractmethod
61 def visit_lth(self, exp, arg):
62 pass
63
64 @abstractmethod
65 def visit_neg(self, exp, arg):
66 pass
67
68 @abstractmethod
69 def visit_not(self, exp, arg):
70 pass
71
72 @abstractmethod
73 def visit_let(self, exp, arg):
74 pass
75
76 @abstractmethod
77 def visit_ifThenElse(self, exp, arg):
78 pass
79
80
81 class CtrGenVisitor(Visitor):
82 """
83 This visitor creates constraints for a type-inference engine. Basically,
84 it traverses the abstract-syntaxis tree of expressions, producing pairs like
85 (type0, type1) on the way. A pair like (type0, type1) indicates that these
86 two type variables are the same.
87
88 Examples:
89 >>> e = Let('v', Num(40), Let('w', Num(2), Add(Var('v'), Var('w'))))
90 >>> ev = CtrGenVisitor()
91 >>> sorted([str(ct) for ct in e.accept(ev, ev.fresh_type_var())])
92 ["('TV_1', 'TV_2')", "('TV_2', 'TV_3')", "('v', <class 'int'>)", "('w', <class 'int'>)", "<class 'int'>, 'TV_3')", "(<class 'int'>, 'TV_3')"
93 """
94
95 def __init__(self):
96 self.fresh_type_counter = 0
97
98 def fresh_type_var(self):
99 """
100 Create a new type var using the current value of the fresh_type_counter.
101 Two successive calls to this method will return different type names.
102 Notice that the name of a type variable is always TV_x, where x is
103 some integer number. That means that probably we would run into

```

```

104 errors if someone declares a variable called, say, TV_1 or TV_2, as in
105 "let TV_1 <- 1 in TV_1 end". But you can assume that such would never
106 happen in the test cases. In practice, we should define a new class
107 to represent type variables. But let's keep the implementation as
108 simple as possible.
109
110 Example:
111 >>> ev = CtrGenVisitor()
112 >>> [ev.fresh_type_var(), ev.fresh_type_var()]
113 ['TV_1', 'TV_2']
114
115 self.fresh_type_counter += 1
116 return f"TV_{self.fresh_type_counter}"
117
118 """
119 The CtrGenVisitor class creates constraints that, once solved, will give
120 us the type of the different variables. Every accept method takes in
121 two arguments (in addition to self):
122
123 exp: is the expression that is being analyzed.
124 type_var: that is a name that works as a placeholder for the type of the
125 expression. Whenever we visit a new expression, we create a type variable
126 to represent its type (you can do that with the method fresh_type_var).
127 The only exception is the type of Var expressions. In this case, the type
128 of a Var expression is the identifier of that expression.
129
130
131 def visit_var(self, exp, type_var):
132 """
133 Example:
134 >>> e = Var('v')
135 >>> ev = CtrGenVisitor()
136 >>> e.accept(ev, ev.fresh_type_var())
137 {'v', 'TV_1'}
138
139 return {(exp.identifier, type_var)}
140
141 def visit_bln(self, exp, type_var):
142 """
143 Example:
144 >>> e = Bln(True)
145 >>> ev = CtrGenVisitor()
146 >>> e.accept(ev, ev.fresh_type_var())
147 {(<class 'bool'>, 'TV_1')}
148
149 return {(type(True), type_var)}
150
151 def visit_num(self, exp, type_var):
152 """
153 Example:
154 >>> e = Num(1)
155 >>> ev = CtrGenVisitor()
156 >>> e.accept(ev, ev.fresh_type_var())
157 {(<class 'int'>, 'TV_1')}
158
159 return {(type(1), type_var)}
160
161 def visit.eql(self, exp, type_var):
162 """
163 Example:
164 >>> e = Eql(Num(1), Bln(True))
165 >>> ev = CtrGenVisitor()
166 >>> sorted([str(ct) for ct in e.accept(ev, ev.fresh_type_var())])
167 ["(<class 'bool'>, 'TV_1')", "(<class 'bool'>, 'TV_2')", "(<class 'int'>, 'TV_2')"]
168
169 Notice that if we have repeated constraints, they only appear once in
170 the set of constraints (after all, it's a set!). As an example, we
171 would have two occurrences of the pair (TV_2, int) in the following
172 example:
173 >>> e = Eql(Num(1), Num(2))
174 >>> ev = CtrGenVisitor()
175 >>> sorted([str(ct) for ct in e.accept(ev, ev.fresh_type_var())])
176 ["(<class 'bool'>, 'TV_1')", "(<class 'int'>, 'TV_2')"]
177
178 # TODO: Implement this method!
179 raise NotImplementedError
180
181 def visit_and(self, exp, type_var):
182 """
183 Example:
184 >>> e = And(Bln(False), Bln(True))
185 >>> ev = CtrGenVisitor()
186 >>> sorted([str(ct) for ct in e.accept(ev, ev.fresh_type_var())])
187 ["(<class 'bool'>, 'TV_1')", "(<class 'bool'>, <class 'bool'>)"]
188
189 In the above example, notice that we ended up getting a trivial
190 constraint, e.g.: (<class 'bool'>, <class 'bool'>). That's alright:
191 don't worry about these trivial constraints at this point. We can
192 remove them from the set of constraints later on, when we try to
193 solve them.
194
195 # TODO: Implement this method!
196 raise NotImplementedError
197
198 def visit_or(self, exp, type_var):
199 """
200 Example:
201 >>> e = Or(Bln(False), Bln(True))
202 >>> ev = CtrGenVisitor()
203 >>> sorted([str(ct) for ct in e.accept(ev, ev.fresh_type_var())])
204 ["(<class 'bool'>, 'TV_1')", "(<class 'bool'>, <class 'bool'>)"]
205
206 # TODO: Implement this method!
207
208

```

```

20/
208
209 raise NotImplementedError
210
211 def visit_add(self, exp, type_var):
212 """
213 Example:
214 >>> e = Add(Num(1), Num(2))
215 >>> ev = CtrGenVisitor()
216 >>> sorted([str(ct) for ct in e.accept(ev, ev.fresh_type_var())])
217 [("<class 'int'>", 'TV_1'), ("<class 'int'>", <class 'int'>)"]
218
219 # TODO: Implement this method!
220 raise NotImplementedError
221
222 def visit_sub(self, exp, type_var):
223 """
224 Example:
225 >>> e = Sub(Num(1), Num(2))
226 >>> ev = CtrGenVisitor()
227 >>> sorted([str(ct) for ct in e.accept(ev, ev.fresh_type_var())])
228 [("<class 'int'>", 'TV_1'), ("<class 'int'>", <class 'int'>)"]
229
230 # TODO: Implement this method!
231 raise NotImplementedError
232
233 def visit_mul(self, exp, type_var):
234 """
235 Example:
236 >>> e = Mul(Num(1), Num(2))
237 >>> ev = CtrGenVisitor()
238 >>> sorted([str(ct) for ct in e.accept(ev, ev.fresh_type_var())])
239 [("<class 'int'>", 'TV_1'), ("<class 'int'>", <class 'int'>)"]
240
241 # TODO: Implement this method!
242 raise NotImplementedError
243
244 def visit_div(self, exp, type_var):
245 """
246 Example:
247 >>> e = Div(Num(1), Num(2))
248 >>> ev = CtrGenVisitor()
249 >>> sorted([str(ct) for ct in e.accept(ev, ev.fresh_type_var())])
250 [("<class 'int'>", 'TV_1'), ("<class 'int'>", <class 'int'>)"]
251
252 # TODO: Implement this method!
253 raise NotImplementedError
254
255 def visit_leq(self, exp, type_var):
256 """
257 Example:
258 >>> e = Leq(Num(1), Num(2))
259 >>> ev = CtrGenVisitor()
260 >>> sorted([str(ct) for ct in e.accept(ev, ev.fresh_type_var())])
261 [("<class 'bool'>", 'TV_1'), ("<class 'int'>", <class 'int'>)"]
262
263 # TODO: Implement this method!
264 raise NotImplementedError
265
266 def visit_lth(self, exp, type_var):
267 """
268 Example:
269 >>> e = Lth(Num(1), Num(2))
270 >>> ev = CtrGenVisitor()
271 >>> sorted([str(ct) for ct in e.accept(ev, ev.fresh_type_var())])
272 [("<class 'bool'>", 'TV_1'), ("<class 'int'>", <class 'int'>)"]
273
274 # TODO: Implement this method!
275 raise NotImplementedError
276
277 def visit_neg(self, exp, type_var):
278 """
279 Example:
280 >>> e = Neg(Num(1))
281 >>> ev = CtrGenVisitor()
282 >>> sorted([str(ct) for ct in e.accept(ev, ev.fresh_type_var())])
283 [("<class 'int'>", 'TV_1'), ("<class 'int'>", <class 'int'>)"]
284
285 # TODO: Implement this method!
286 raise NotImplementedError
287
288 def visit_not(self, exp, type_var):
289 """
290 Example:
291 >>> e = Not(Bln(True))
292 >>> ev = CtrGenVisitor()
293 >>> sorted([str(ct) for ct in e.accept(ev, ev.fresh_type_var())])
294 [("<class 'bool'>", 'TV_1'), ("<class 'bool'>", <class 'bool'>)"]
295
296 # TODO: Implement this method!
297 raise NotImplementedError
298
299 def visit_let(self, exp, type_var):
300 """
301 Example:
302 >>> e = Let('v', Num(42), Var('v'))
303 >>> ev = CtrGenVisitor()
304 >>> sorted([str(ct) for ct in e.accept(ev, ev.fresh_type_var())])
305 [("TV_1", 'TV_2'), ("v", 'TV_2'), ("<class 'int'>, 'v')"]
306
307 # TODO: Implement this method!
308 raise NotImplementedError
309
310 def visit_ifThenElse(self, exp, type_var):
311 """
312 Example:

```

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

Unifier.py

```

1 from Expression import *
2 from Visitor import *
3 import sys
4
5
6 def unify(constraints, sets):
7 """
8 This function unifies all the type variables in the list of constraints;
9 thus, producing a set of unifiers.
10
11 Example:
12 >>> sets = unify([('a', type(1))], {})
13 >>> integers = sets[type(1)] - {type(1)}
14 >>> sorted(integers)
15 ['a']
16
17 >>> sets = unify([(type(1), 'b'), ('a', type(1))], {})
18 >>> integers = sets[type(1)] - {type(1)}
19 >>> sorted(integers)
20 ['a', 'b']
21
22 >>> sets = unify([(type(True), 'b'), ('a', type(1))], {})
23 >>> booleans = sets[type(True)] - {type(True)}
24 >>> sorted(booleans)
25 ['b']
26
27 >>> sets = unify([(type(True), 'b'), ('a', type(1))], {})
28 >>> integers = sets[type(1)] - {type(1)}
29 >>> sorted(integers)
30 ['a']
31
32 >>> sets = unify([('a', 'TV_1'), ('b', 'TV_2'), ('TV_2', type(1)), ('TV_1', type(1))], {})
33 >>> integers = sets[type(1)] - {type(1)}
34 >>> sorted(integers)
35 ['TV_1', 'TV_2', 'a', 'b']
36
37 Notice that at this stage, we still allow sets with invalid types. For
38 instance, the set associated with 'b' in the example below will contain
39 four elements, namely: {<class 'bool'>, <class 'int'>, 'b', 'a'}:
40 >>> sets = unify([(type(True), 'b'), ('a', type(1)), ('a', 'b')], {})
41 >>> len(sets['b'])
42 4
43
44 # TODO: Implement this method!
45 raise NotImplemented
46
47
48
49 def name_sets(sets):
50 """
51 This method replaces type sets with "canonical type names". A canonical
52 type name is the name of a type set. For instance, the type set
53 {'a', 'b', type(int)} has the canonical name type(int)
54
55 Notice that this method produces two types of error messages:
56 * Polymorphic type: if any canonical type set is empty
57 * Ambiguous type: if any canonical type set contains more than one element.
58 In both cases, if any of these errors happen, the program should stop with
59 the following error message: 'Type error'
60
61 Example:
62 >>> sets = name_sets({'a': {'a', 'b', type(1)}, 'b': {'a', 'b', type(1)}})
63 >>> [sets['a'], sets['b']]
64 [<class 'int'>, <class 'int'>]
65
66 >>> sets = name_sets({'a': {'a', type(1)}, 'b': {'b', type(True)}})
67 >>> [sets['a'], sets['b']]
68 [<class 'int'>, <class 'bool'>]
69
70
71 # TODO: Implement this method!
72 raise NotImplemented
73
74
75
76 def infer_types(expression):
77 """
78 This method maps all the program variables to type names. We have
79 implemented this method for you. This implementation might help you to
80 understand how the other two methods, unify and name_sets are meant to
81 work.
82
83 Example:
84 >>> e = Let('v', Num(42), Var('v'))
85 >>> type_names = infer_types(e)
86 >>> type_names['v']
87 <class 'int'>
88
89 >>> e = Let('v', Num(1), Let('y', Var('v'), Var('y')))
90 >>> type_names = infer_types(e)
91 >>> [type_names['v'], type_names['y']]
92 [<class 'int'>, <class 'int'>]
93
94 >>> e0 = Let('v', Num(1), Let('y', Var('v'), Var('y')))
95 >>> e1 = IfThenElse(Lth(e0, Num(2)), Bln(True), Bln(False))
96 >>> e2 = Let('w', e1, And(Var('w'), Var('w')))
97 >>> type_names = infer_types(e2)
98 >>> [type_names['v'], type_names['w'], type_names['y']]
99 [<class 'int'>, <class 'bool'>, <class 'int'>]
100
101 ev = CtrGenVisitor()
102 constraints = list(expression.accept(ev, ev.fresh_type_var()))
103 type_sets = unify(constraints, {})

```

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

[VPL](#)