2025_2 - COMPILADORES - METATURMA

AV12 - Renomeação de variáveis

Data de entrega: quarta, 5 Nov 2025, 23:59
Arquivos requeridos: driver.py, Lexer.py, Parser.py, Expression.py, Visitor.py, Asm.py (Baixar)
Tipo de trabalho: Trabalho individual

No último laboratório de programação, foi desenvolvido um gerador de código para um subconjunto de instruções RISC-V. Aquele exercício possuia uma restrição importante: assumia-se que blocos `let` sempre definem variáveis com nomes diferentes. Em outras palavras, um programa como `let x <- 1 in let x <- 2 in x end + x end` não era válido, uma vez que a variável `x` é definida duas vezes. Esta restrição, contudo, é artificial, pois o programa mostrado é perfeitamente válido quando interpretado. Seu valor, neste caso, é 3. Neste exercício iremos remover tal restrição, permitindo que o mesmo nome de variável seja reutilizado entre variáveis diferentes. Assim, as igualdades a seguir devem ser válidas:

```
let v <- 1 in let v <- 2 in v end + v end = 3
let v <- 1 in let v <- 2 in let v <- 3 in v end + v end + v end = 6
```

Para tanto, você deve criar variáveis com diferentes nomes quando gerar código para os diferentes programas. Por exemplo, o primeiro programa acima (`let v <- 1 in let v <- 2 in v end + v end`) deveria levar a alguma sequência de instruções como abaixo:

```
v_0 = addi x0 1          -- Define v_0 com o valor 1
v_1 = addi x0 2          -- Define v_1 com o valor 2
v3 = add v_1 v_0         -- Define v3 como a soma de v_0 e v_1
```

Essas três instruções correspondem ao programa `let v_0 <- 1 in let v_1 <- 2 in v_1 end + v_0 end`, que é equivalente ao programa original. Uma forma de permitir o reuso de nomes de variáveis é renomear as variáveis definidas em uma expressão, antes da geração de código. Com tal objetivo, você deverá implementar uma classe `RenameVisitor`, que modifica uma expressão (in place) para garantir que todos os blocos `let` criem variáveis com nomes diferentes. Note que as modificações devem ocorrer "*in place*", isto é, sobre a expressão original. Os métodos visit, neste caso, não precisam retornar nada. Como exemplo, a implementação abaixo deveria ser válida:

```
def visit_eql(self, exp, name_map):
    exp.left.accept(self, name_map)
    exp.right.accept(self, name_map)
```

A modificação *in place* é necessária não somente porque ela é mais eficiente, mas também porque a própria interface de uso do *driver* requer essa abordagem. O código que renomeia variáveis é invocado como abaixo:

```
def rename_variables(exp):
    ren = RenameVisitor()
    exp.accept(ren, {})
    return exp
```

Uma curiosidade sobre este exercício é que ele implementa uma forma de atribuição estática única sobre árvores de sintaxe abstrata (AST). O formato SSA (*Static Single Assignment*) é normalmente usado sobre a representação de baixo nível de programas (o grafo de fluxo de controle, por exemplo). Mas, neste exercício estamos obtendo a propriedade da definição única sobre a árvore de sintaxe abstrata.

Submetendo e Testando

Este VPL deve ser construído sobre o VPL 11. A única extensão sobre aquele exercício é a implementação de `RenameVisitor`. Para

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

```
python3 driver.py
2 + 3 # CTRL+D
5
```

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

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

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

Arquivos requeridos

driver.py

```python
import sys
from Expression import *
from Visitor import *
from Lexer import Lexer
from Parser import Parser
import Asm as AsmModule

def rename_variables(exp):
    """
    Esta funcao invoca o renomeador de variaveis. Ela deve ser usada antes do
    inicio da fase de geracao de codigo.
    """
    ren = RenameVisitor()
    exp.accept(ren, {})
    return exp

if __name__ == "__main__":
    """
    Este arquivo nao deve ser alterado, mas deve ser enviado para resolver o
    VPL. O arquivo contem o codigo que testa a implementacao do parser.
    """
    text = sys.stdin.read()
    lexer = Lexer(text)
    parser = Parser(lexer.tokens())
    exp = rename_variables(parser.parse())
    prog = AsmModule.Program({}, [])
    gen = GenVisitor()
    var_answer = exp.accept(gen, prog)
    prog.eval()
    print(f"{prog.get_val(var_answer)}")
```

Lexer.py

```python
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
    ADD = 202
    SUB = 203
    MUL = 204
    DIV = 205
    LEQ = 206
    LTH = 207
    NEG = 208
    NOT = 209
    LPR = 210
    RPR = 211
    ASN = 212  # The assignment '<-' operator


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. 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 -- 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 (you can reuse Lab 5: Visitors)!
        """
        token = None
        return token
```

Parser.py

```
import sys

from Expression import *
from Lexer import Token, TokenType

"""
This file implements the parser of arithmetic expressions. The same rules of
precedence and associativity from Lab 5: Visitors, apply.

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

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

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

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

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

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

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

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

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

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

        >>> tk0 = Token('30', TokenType.NUM)
        >>> tk1 = Token('-', TokenType.SUB)
        >>> tk2 = Token('4', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2])
        >>> exp = parser.parse()
        >>> exp.eval(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()
        >>> exp.eval(None)
        14

```

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

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

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

        >>> tk0 = Token('not', TokenType.NOT)
        >>> tk1 = Token('4', TokenType.NUM)
        >>> tk2 = Token('<', TokenType.LTH)
        >>> tk3 = Token('4', TokenType.NUM)
        >>> parser = Parser([tk0, tk1, tk2, tk3])
        >>> exp = parser.parse()
        >>> exp.eval(None)
        True

        >>> 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()
        >>> exp.eval({})
        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()
        >>> exp.eval({})
        42
        """
        # TODO: implement this method.
        return None
```

Expression.py

```
"""
This file implements the data structures that represent Expressions. You don't
need to modify it for this assignment.
"""

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):
        """
        Variables don't need to be implemented for this exercise.
        """
        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):
        """
        booleans don't need to be implemented for this exercise.
        """
        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):
        """
        Equality doesn't need to be implemented for this exercise.
        """
        return visitor.visit_eql(self, arg)


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

```

```
    def accept(self, visitor, arg):
        """
        Example:
        >>> n1 = Num(3)
        >>> n2 = Num(4)
        >>> e = Add(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        7
        """
        return visitor.visit_add(self, arg)


class Sub(BinaryExpression):
    """
    This class represents subtraction of two expressions. The acceptuation of
    such an expression is the subtraction of the two subexpression's values.
    """

    def accept(self, visitor, arg):
        """
        Example:
        >>> n1 = Num(3)
        >>> n2 = Num(4)
        >>> e = Sub(n1, n2)
        >>> ev = EvalVisitor()
        >>> e.accept(ev, None)
        -1
        """
        return visitor.visit_sub(self, arg)


class Mul(BinaryExpression):
    """
    This class represents multiplication of two expressions. The acceptuation of
    such an expression is the product of the two subexpression's values.
    """

    def accept(self, visitor, arg):
        """
        Example:
        >>> n1 = Num(3)
        >>> n2 = Num(4)
        >>> e = Mul(n1, n2)
        >>> ev = EvalVisitor()
        >>> 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):
        """
        Comparisons don't need to be implemented for this exercise.
        """
        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):
        """
        Comparisons don't need to be implemented for this exercise.
        """
```

```
207         """
208         return visitor.visit_lth(self, arg)
209
210
211 class UnaryExpression(Expression):
212     """
213     This class represents unary expressions. A unary expression has only one
214     sub-expression.
215     """
216
217     def __init__(self, exp):
218         self.exp = exp
219
220     @abstractmethod
221     def accept(self, visitor, arg):
222         raise NotImplementedError
223
224
225 class Neg(UnaryExpression):
226     """
227     This expression represents the additive inverse of a number. The additive
228     inverse of a number n is the number -n, so that the sum of both is zero.
229     """
230
231     def accept(self, visitor, arg):
232         """
233         Example:
234         >>> n = Num(3)
235         >>> e = Neg(n)
236         >>> ev = EvalVisitor()
237         >>> e.accept(ev, None)
238         -3
239         >>> n = Num(0)
240         >>> e = Neg(n)
241         >>> ev = EvalVisitor()
242         >>> e.accept(ev, None)
243         0
244         """
245         return visitor.visit_neg(self, arg)
246
247
248 class Not(UnaryExpression):
249     """
250     This expression represents the negation of a boolean. The negation of a
251     boolean expression is the logical complement of that expression.
252     """
253
254     def accept(self, visitor, arg):
255         """
256         No need to implement negation for this exercise, for we don't even have
257         booleans at this point.
258         """
259         return visitor.visit_not(self, arg)
260
261
262 class Let(Expression):
263     """
264     This class represents a let expression. The semantics of a let expression,
265     such as "let v <- e0 in e1" on an environment env is as follows:
266     1. Evaluate e0 in the environment env, yielding e0_val
267     2. Evaluate e1 in the new environment env' = env + {v:e0_val}
268     """
269
270     def __init__(self, identifier, exp_def, exp_body):
271         self.identifier = identifier
272         self.exp_def = exp_def
273         self.exp_body = exp_body
274
275     def accept(self, visitor, arg):
276         """
277         We don't have bindings at this point. So, nothing to be done here, for
278         this exercise.
279         """
280         return visitor.visit_let(self, arg)
```

Visitor.py

```python
import sys
from abc import ABC, abstractmethod
from Expression import *
import Asm as AsmModule


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_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


class RenameVisitor(ABC):
    """
    This visitor traverses the AST of a program, renaming variables to ensure
    that they all have different names.

    Usage:
        >>> e0 = Let('x', Num(2), Add(Var('x'), Num(3)))
        >>> e1 = Let('x', e0, Mul(Var('x'), Num(10)))
        >>> e0.identifier == e1.identifier
        True

        >>> e0 = Let('x', Num(2), Add(Var('x'), Num(3)))
        >>> e1 = Let('x', e0, Mul(Var('x'), Num(10)))
        >>> r = RenameVisitor()
        >>> e1.accept(r, {})
        >>> e0.identifier == e1.identifier
        False

        >>> x0 = Var('x')
        >>> x1 = Var('x')
        >>> e0 = Let('x', Num(2), Add(x0, Num(3)))
        >>> e1 = Let('x', e0, Mul(x1, Num(10)))
        >>> x0.identifier == x1.identifier
        True

        >>> x0 = Var('x')
        >>> x1 = Var('x')
        >>> e0 = Let('x', Num(2), Add(x0, Num(3)))
        >>> e1 = Let('x', e0, Mul(x1, Num(10)))
        >>> r = RenameVisitor()
        >>> e1.accept(r, {})
        >>> x0.identifier == x1.identifier
        False
    """
```

```
104
105     def visit_var(self, exp, arg):
106         # TODO: Implement this method.
107         raise NotImplementedError
108
109     def visit_bln(self, exp, arg):
110         # TODO: Implement this method.
111         raise NotImplementedError
112
113     def visit_num(self, exp, arg):
114         # TODO: Implement this method.
115         raise NotImplementedError
116
117     def visit_eql(self, exp, arg):
118         # TODO: Implement this method.
119         raise NotImplementedError
120
121     def visit_add(self, exp, arg):
122         # TODO: Implement this method.
123         raise NotImplementedError
124
125     def visit_sub(self, exp, arg):
126         # TODO: Implement this method.
127         raise NotImplementedError
128
129     def visit_mul(self, exp, arg):
130         # TODO: Implement this method.
131         raise NotImplementedError
132
133     def visit_div(self, exp, arg):
134         # TODO: Implement this method.
135         raise NotImplementedError
136
137     def visit_leq(self, exp, arg):
138         # TODO: Implement this method.
139         raise NotImplementedError
140
141     def visit_lth(self, exp, arg):
142         # TODO: Implement this method.
143         raise NotImplementedError
144
145     def visit_neg(self, exp, arg):
146         # TODO: Implement this method.
147         raise NotImplementedError
148
149     def visit_not(self, exp, arg):
150         # TODO: Implement this method.
151         raise NotImplementedError
152
153     def visit_let(self, exp, arg):
154         # TODO: Implement this method.
155         raise NotImplementedError
156
157
158 class GenVisitor(Visitor):
159     """
160     The GenVisitor class compiles arithmetic expressions into a low-level
161     language.
162     """
163
164     def __init__(self):
165         self.next_var_counter = 0
166
167     def next_var_name(self):
168         self.next_var_counter += 1
169         return f"v{self.next_var_counter}"
170
171     def visit_var(self, exp, prog):
172         """
173         Usage:
174             >>> e = Var('x')
175             >>> p = AsmModule.Program({"x":1}, [])
176             >>> g = GenVisitor()
177             >>> v = e.accept(g, p)
178             >>> p.eval()
179             >>> p.get_val(v)
180             1
181         """
182         return exp.identifier
183
184     def visit_bln(self, exp, env):
185         """
186         Usage:
187             >>> e = Bln(True)
188             >>> p = AsmModule.Program({}, [])
189             >>> g = GenVisitor()
190             >>> v = e.accept(g, p)
191             >>> p.eval()
192             >>> p.get_val(v)
193             1
194
195             >>> e = Bln(False)
196             >>> p = AsmModule.Program({}, [])
197             >>> g = GenVisitor()
198             >>> v = e.accept(g, p)
199             >>> p.eval()
200             >>> p.get_val(v)
201             0
202         """
203         # TODO: Implement this method.
204         raise NotImplementedError
205
206     def visit_num(self, exp, prog):
207         """
```

```
        """
        Usage:
            >>> e = Num(13)
            >>> p = AsmModule.Program({}, [])
            >>> g = GenVisitor()
            >>> v = e.accept(g, p)
            >>> p.eval()
            >>> p.get_val(v)
            13
        """
        # TODO: Implement this method.
        raise NotImplementedError

    def visit_eql(self, exp, prog):
        """
        >>> e = Eql(Num(13), Num(13))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        1

        >>> e = Eql(Num(13), Num(10))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        0

        >>> e = Eql(Num(-1), Num(1))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        0
        """
        # TODO: Implement this method.
        raise NotImplementedError

    def visit_add(self, exp, prog):
        """
        >>> e = Add(Num(13), Num(-13))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        0

        >>> e = Add(Num(13), Num(10))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        23
        """
        # TODO: Implement this method (see the example in the lab's page).
        raise NotImplementedError

    def visit_sub(self, exp, prog):
        """
        >>> e = Sub(Num(13), Num(-13))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        26

        >>> e = Sub(Num(13), Num(10))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        3
        """
        # TODO: Implement this method.
        raise NotImplementedError

    def visit_mul(self, exp, prog):
        """
        >>> e = Mul(Num(13), Num(2))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        26

        >>> e = Mul(Num(13), Num(10))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        130
        """
        # TODO: Implement this method.
        raise NotImplementedError
```

```
        raise NotImplementedError

    def visit_div(self, exp, prog):
        """
        >>> e = Div(Num(13), Num(2))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        6

        >>> e = Div(Num(13), Num(10))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        1
        """
        # TODO: Implement this method.
        raise NotImplementedError

    def visit_leq(self, exp, prog):
        """
        >>> e = Leq(Num(3), Num(2))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        0

        >>> e = Leq(Num(3), Num(3))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        1

        >>> e = Leq(Num(2), Num(3))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        1

        >>> e = Leq(Num(-3), Num(-2))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        1

        >>> e = Leq(Num(-3), Num(-3))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        1

        >>> e = Leq(Num(-2), Num(-3))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        0
        """
        # TODO: Implement this method.
        raise NotImplementedError

    def visit_lth(self, exp, prog):
        """
        >>> e = Lth(Num(3), Num(2))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        0

        >>> e = Lth(Num(3), Num(3))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        0

        >>> e = Lth(Num(2), Num(3))
        >>> p = AsmModule.Program({}, [])
        >>> g = GenVisitor()
        >>> v = e.accept(g, p)
        >>> p.eval()
        >>> p.get_val(v)
        1
        """
        # TODO: Implement this method.
        raise NotImplementedError
```

```
414
415     def visit_neg(self, exp, prog):
416         """
417         >>> e = Neg(Num(3))
418         >>> p = AsmModule.Program({}, [])
419         >>> g = GenVisitor()
420         >>> v = e.accept(g, p)
421         >>> p.eval()
422         >>> p.get_val(v)
423         -3
424
425         >>> e = Neg(Num(0))
426         >>> p = AsmModule.Program({}, [])
427         >>> g = GenVisitor()
428         >>> v = e.accept(g, p)
429         >>> p.eval()
430         >>> p.get_val(v)
431         0
432
433         >>> e = Neg(Num(-3))
434         >>> p = AsmModule.Program({}, [])
435         >>> g = GenVisitor()
436         >>> v = e.accept(g, p)
437         >>> p.eval()
438         >>> p.get_val(v)
439         3
440         """
441         # TODO: Implement this method.
442         raise NotImplementedError
443
444     def visit_not(self, exp, prog):
445         """
446         >>> e = Not(Bln(True))
447         >>> p = AsmModule.Program({}, [])
448         >>> g = GenVisitor()
449         >>> v = e.accept(g, p)
450         >>> p.eval()
451         >>> p.get_val(v)
452         0
453
454         >>> e = Not(Bln(False))
455         >>> p = AsmModule.Program({}, [])
456         >>> g = GenVisitor()
457         >>> v = e.accept(g, p)
458         >>> p.eval()
459         >>> p.get_val(v)
460         1
461
462         >>> e = Not(Num(0))
463         >>> p = AsmModule.Program({}, [])
464         >>> g = GenVisitor()
465         >>> v = e.accept(g, p)
466         >>> p.eval()
467         >>> p.get_val(v)
468         1
469
470         >>> e = Not(Num(-2))
471         >>> p = AsmModule.Program({}, [])
472         >>> g = GenVisitor()
473         >>> v = e.accept(g, p)
474         >>> p.eval()
475         >>> p.get_val(v)
476         0
477
478         >>> e = Not(Num(2))
479         >>> p = AsmModule.Program({}, [])
480         >>> g = GenVisitor()
481         >>> v = e.accept(g, p)
482         >>> p.eval()
483         >>> p.get_val(v)
484         0
485         """
486         # TODO: Implement this method.
487         raise NotImplementedError
488
489     def visit_let(self, exp, prog):
490         """
491         Usage:
492             >>> e = Let('v', Not(Bln(False)), Var('v'))
493             >>> p = AsmModule.Program({}, [])
494             >>> g = GenVisitor()
495             >>> v = e.accept(g, p)
496             >>> p.eval()
497             >>> p.get_val(v)
498             1
499
500             >>> e = Let('v', Num(2), Add(Var('v'), Num(3)))
501             >>> p = AsmModule.Program({}, [])
502             >>> g = GenVisitor()
503             >>> v = e.accept(g, p)
504             >>> p.eval()
505             >>> p.get_val(v)
506             5
507
508             >>> e0 = Let('x', Num(2), Add(Var('x'), Num(3)))
509             >>> e1 = Let('y', e0, Mul(Var('y'), Num(10)))
510             >>> p = AsmModule.Program({}, [])
511             >>> g = GenVisitor()
512             >>> v = e1.accept(g, p)
513             >>> p.eval()
514             >>> p.get_val(v)
515             50
516         """
```

```
517         # TODO: Implement this method.
518         raise NotImplementedError
```

Asm.py

```
"""
This file contains the implementation of a simple interpreter of low-level
instructions. The interpreter takes a program, represented as an array of
instructions, plus an environment, which is a map that associates variables with
values. The following instructions are recognized:

    * add rd, rs1, rs2: rd = rs1 + rs2
    * addi rd, rs1, imm: rd = rs1 + imm
    * mul rd, rs1, rs2: rd = rs1 * rs2
    * sub rd, rs1, rs2: rd = rs1 - rs2
    * xor rd, rs1, rs2: rd = rs1 ^ rs2
    * xori rd, rs1, imm: rd = rs1 ^ imm
    * div rd, rs1, rs2: rd = rs1 // rs2 (signed integer division)
    * slt rd, rs1, rs2: rd = (rs1 < rs2) ? 1 : 0 (signed comparison)
    * slti rd, rs1, imm: rd = (rs1 < imm) ? 1 : 0

This file uses doctests all over. To test it, just run python 3 as follows:
"python3 -m doctest Asm.py". The program uses syntax that is excluive of
Python 3. It will not work with standard Python 2.
"""

import sys
from collections import deque
from abc import ABC, abstractmethod


class Program:
    """
    The 'Program' is a list of instructions plus an environment that associates
    names with values, plus a program counter, which marks the next instruction
    that must be executed. The environment contains a special variable x0,
    which always contains the value zero.
    """

    def __init__(self, env, insts):
        self.__env = env
        self.__insts = insts
        self.pc = 0
        self.__env["x0"] = 0

    def get_inst(self):
        if self.pc >= 0 and self.pc < len(self.__insts):
            inst = self.__insts[self.pc]
            self.pc += 1
            return inst
        else:
            return None

    def add_inst(self, inst):
        self.__insts.append(inst)

    def set_pc(self, pc):
        self.pc = pc

    def set_val(self, name, value):
        self.__env[name] = value

    def get_val(self, name):
        """
        The register x0 always contains the value zero:

        >>> p = Program({}, [])
        >>> p.get_val("x0")
        0
        """
        if name in self.__env:
            return self.__env[name]
        else:
            sys.exit("Def error")

    def print_env(self):
        for name, val in sorted(self.__env.items()):
            print(f"{name}: {val}")

    def print_insts(self):
        for inst in self.__insts:
            print(inst)

    def eval(self):
        """
        This function evaluates a program until there is no more instructions to
        evaluate.

        Example:
            >>> insts = [Add("x0", "b0", "b1"), Sub("x1", "x0", "b2")]
            >>> p = Program({"b0":2, "b1":3, "b2": 4}, insts)
            >>> p.eval()
            >>> p.print_env()
            b0: 2
            b1: 3
            b2: 4
            x0: 5
            x1: 1
        """
        inst = self.get_inst()
        while inst:
            inst.eval(self)
            inst = self.get_inst()


def max(a, b):
    """
    This example computes the maximum between a and b.
```

```
104
105     Example:
106         >>> max(2, 3)
107         3
108
109         >>> max(3, 2)
110         3
111
112         >>> max(-3, -2)
113         -2
114
115         >>> max(-2, -3)
116         -2
117     """
118     p = Program({}, [])
119     p.set_val("rs1", a)
120     p.set_val("rs2", b)
121     p.add_inst(Slt("t0", "rs2", "rs1"))
122     p.add_inst(Slt("t1", "rs1", "rs2"))
123     p.add_inst(Mul("t0", "t0", "rs1"))
124     p.add_inst(Mul("t1", "t1", "rs2"))
125     p.add_inst(Add("rd", "t0", "t1"))
126     p.eval()
127     return p.get_val("rd")
128
129
130 def distance_with_acceleration(V, A, T):
131     """
132     This example computes the position of an object, given its velocity (V),
133     its acceleration (A) and the time (T), assuming that it starts at position
134     zero, using the formula D = V*T + (A*T^2)/2.
135
136     Example:
137         >>> distance_with_acceleration(3, 4, 5)
138         65
139     """
140     p = Program({}, [])
141     p.set_val("rs1", V)
142     p.set_val("rs2", A)
143     p.set_val("rs3", T)
144     p.add_inst(Addi("two", "x0", 2))
145     p.add_inst(Mul("t0", "rs1", "rs3"))
146     p.add_inst(Mul("t1", "rs3", "rs3"))
147     p.add_inst(Mul("t2", "rs2", "t1"))
148     p.add_inst(Div("t2", "t2", "two"))
149     p.add_inst(Add("rd", "t0", "t2"))
150     p.eval()
151     return p.get_val("rd")
152
153
154 class Inst(ABC):
155     """
156     The representation of instructions. Every instruction refers to a program
157     during its evaluation.
158     """
159
160     def __init__(self):
161         pass
162
163     @abstractmethod
164     def get_opcode(self):
165         raise NotImplementedError
166
167     @abstractmethod
168     def eval(self, prog):
169         raise NotImplementedError
170
171
172 class BinOp(Inst):
173     """
174     The general class of binary instructions. These instructions define a
175     value, and use two values.
176     """
177
178     def __init__(self, rd, rs1, rs2):
179         assert isinstance(rd, str) and isinstance(rs1, str) and isinstance(rs2, str)
180         self.rd = rd
181         self.rs1 = rs1
182         self.rs2 = rs2
183
184     def __str__(self):
185         op = self.get_opcode()
186         return f"{self.rd} = {op} {self.rs1} {self.rs2}"
187
188
189 class BinOpImm(Inst):
190     """
191     The general class of binary instructions where the second operand is an
192     integer constant. These instructions define a value, and use one variable
193     and one immediate constant.
194     """
195
196     def __init__(self, rd, rs1, imm):
197         assert isinstance(rd, str) and isinstance(rs1, str) and isinstance(imm, int)
198         self.rd = rd
199         self.rs1 = rs1
200         self.imm = imm
201
202     def __str__(self):
203         op = self.get_opcode()
204         return f"{self.rd} = {op} {self.rs1} {self.imm}"
205
206
```

```
class Add(BinOp):
    """
    add rd, rs1, rs2: rd = rs1 + rs2

    Example:
        >>> i = Add("a", "b0", "b1")
        >>> str(i)
        'a = add b0 b1'

        >>> p = Program(env={"b0":2, "b1":3}, insts=[Add("a", "b0", "b1")])
        >>> p.eval()
        >>> p.get_val("a")
        5
    """

    def eval(self, prog):
        rs1 = prog.get_val(self.rs1)
        rs2 = prog.get_val(self.rs2)
        prog.set_val(self.rd, rs1 + rs2)

    def get_opcode(self):
        return "add"


class Addi(BinOpImm):
    """
    addi rd, rs1, imm: rd = rs1 + imm

    Example:
        >>> i = Addi("a", "b0", 1)
        >>> str(i)
        'a = addi b0 1'

        >>> p = Program(env={"b0":2}, insts=[Addi("a", "b0", 3)])
        >>> p.eval()
        >>> p.get_val("a")
        5
    """

    def eval(self, prog):
        rs1 = prog.get_val(self.rs1)
        prog.set_val(self.rd, rs1 + self.imm)

    def get_opcode(self):
        return "addi"


class Mul(BinOp):
    """
    mul rd, rs1, rs2: rd = rs1 * rs2

    Example:
        >>> i = Mul("a", "b0", "b1")
        >>> str(i)
        'a = mul b0 b1'

        >>> p = Program(env={"b0":2, "b1":3}, insts=[Mul("a", "b0", "b1")])
        >>> p.eval()
        >>> p.get_val("a")
        6
    """

    def eval(self, prog):
        rs1 = prog.get_val(self.rs1)
        rs2 = prog.get_val(self.rs2)
        prog.set_val(self.rd, rs1 * rs2)

    def get_opcode(self):
        return "mul"


class Sub(BinOp):
    """
    sub rd, rs1, rs2: rd = rs1 - rs2

    Example:
        >>> i = Sub("a", "b0", "b1")
        >>> str(i)
        'a = sub b0 b1'

        >>> p = Program(env={"b0":2, "b1":3}, insts=[Sub("a", "b0", "b1")])
        >>> p.eval()
        >>> p.get_val("a")
        -1
    """

    def eval(self, prog):
        rs1 = prog.get_val(self.rs1)
        rs2 = prog.get_val(self.rs2)
        prog.set_val(self.rd, rs1 - rs2)

    def get_opcode(self):
        return "sub"


class Xor(BinOp):
    """
    xor rd, rs1, rs2: rd = rs1 ^ rs2

    Example:
        >>> i = Xor("a", "b0", "b1")
        >>> str(i)
        'a = xor b0 b1'
```

```
        >>> p = Program(env={"b0":2, "b1":3}, insts=[Xor("a", "b0", "b1")])
        >>> p.eval()
        >>> p.get_val("a")
        1
    """

    def eval(self, prog):
        rs1 = prog.get_val(self.rs1)
        rs2 = prog.get_val(self.rs2)
        prog.set_val(self.rd, rs1 ^ rs2)

    def get_opcode(self):
        return "xor"


class Xori(BinOpImm):
    """
    xori rd, rs1, imm: rd = rs1 ^ imm

    Example:
        >>> i = Xori("a", "b0", 10)
        >>> str(i)
        'a = xori b0 10'

        >>> p = Program(env={"b0":2}, insts=[Xori("a", "b0", 3)])
        >>> p.eval()
        >>> p.get_val("a")
        1
    """

    def eval(self, prog):
        rs1 = prog.get_val(self.rs1)
        prog.set_val(self.rd, rs1 ^ self.imm)

    def get_opcode(self):
        return "xori"


class Div(BinOp):
    """
    div rd, rs1, rs2: rd = rs1 // rs2 (signed integer division)
    Notice that RISC-V does not have an instruction exactly like this one.
    The div operator works on floating-point numbers; not on integers.

    Example:
        >>> i = Div("a", "b0", "b1")
        >>> str(i)
        'a = div b0 b1'

        >>> p = Program(env={"b0":8, "b1":3}, insts=[Div("a", "b0", "b1")])
        >>> p.eval()
        >>> p.get_val("a")
        2
    """

    def eval(self, prog):
        rs1 = prog.get_val(self.rs1)
        rs2 = prog.get_val(self.rs2)
        prog.set_val(self.rd, rs1 // rs2)

    def get_opcode(self):
        return "div"


class Slt(BinOp):
    """
    slt rd, rs1, rs2: rd = (rs1 < rs2) ? 1 : 0 (signed comparison)

    Example:
        >>> i = Slt("a", "b0", "b1")
        >>> str(i)
        'a = slt b0 b1'

        >>> p = Program(env={"b0":2, "b1":3}, insts=[Slt("a", "b0", "b1")])
        >>> p.eval()
        >>> p.get_val("a")
        1

        >>> p = Program(env={"b0":3, "b1":3}, insts=[Slt("a", "b0", "b1")])
        >>> p.eval()
        >>> p.get_val("a")
        0

        >>> p = Program(env={"b0":3, "b1":2}, insts=[Slt("a", "b0", "b1")])
        >>> p.eval()
        >>> p.get_val("a")
        0
    """

    def eval(self, prog):
        rs1 = prog.get_val(self.rs1)
        rs2 = prog.get_val(self.rs2)
        prog.set_val(self.rd, 1 if rs1 < rs2 else 0)

    def get_opcode(self):
        return "slt"


class Slti(BinOpImm):
    """
    slti rd, rs1, imm: rd = (rs1 < imm) ? 1 : 0
    (signed comparison with immediate)
```

```
414     Example:
415         >>> i = Slti("a", "b0", 0)
416         >>> str(i)
417         'a = slti b0 0'
418 
419         >>> p = Program(env={"b0":2}, insts=[Slti("a", "b0", 3)])
420         >>> p.eval()
421         >>> p.get_val("a")
422         1
423 
424         >>> p = Program(env={"b0":3}, insts=[Slti("a", "b0", 3)])
425         >>> p.eval()
426         >>> p.get_val("a")
427         0
428 
429         >>> p = Program(env={"b0":3}, insts=[Slti("a", "b0", 2)])
430         >>> p.eval()
431         >>> p.get_val("a")
432         0
433     """
434 
435     def eval(self, prog):
436         rs1 = prog.get_val(self.rs1)
437         prog.set_val(self.rd, 1 if rs1 < self.imm else 0)
438 
439     def get_opcode(self):
440         return "slti"
```

VPL