ساختار برنامه:

این برنامه از سه پوشه و دو فایل تشکیل شده است. فایل requirement.txt تمامی کتابخانه پیش نیاز برای اجرای برنامه وجود دارد. فایل example.py نمونه‌ای از نحوه استفاده از ابزار را نشان می‌دهد.

پوشه example چند نمونه از برنامه های پایتون را نشان می‌دهد که برای استخراج گراف جریان کنترلی از آن‌ها استفاده شده است ولی می‌توان از هر کد دیگر بجای این فایل‌ها استفاده کرد. پوشه output در واقع شامل فایل‌های خروجی برنامه است که به دو صورت pdf و gv که شامل ساختار متنی گراف می‌باشد و می توان در کاربردهای دیگر از آن استفاده کرد. پوشه دیگر CFG می‌باشد که تمامی ابزار و برنامه در داخل آن قرار دارد و در قسمت بعد به شرح آن خواهیم پرداخت.

شرح توابع:

اولین و بالاترین تابع برای تولید گراف جریان کنترلی تابع gen_cfg می‌باشد که ساختار آن در پایین آمده است.

```
# Input: source code
# output: control flow graph asdictionary data structure

def gen_cfg(fnsrc, remove_start_stop=True):
  # generate cgf as dictionary format
  reset_registry()
  cfg = PyCFG()
  cfg.gen_cfg(fnsrc)
  cache = dict(REGISTRY)
  if remove_start_stop:
    return {
      k: cache[k]
      for k in cache if cache[k].source()  not in {'start', 'stop'}
    }
  else:
    return cache
```

همان‌طور که در بالا مشاهده می‌شود این تابع متن کد را به عنوان ورودی دریافت کرده و خروجی آن به صورت ساختار دیکشینری در پایتون می‌باشد. ابتدا تابع reset_registery فراخوانی می‌شود تا تمامی متغیرهای عمومی برای نگه داری شناسه‌های گره‌های درخت تجریدی نحوی خالی شود تا مقادیر جدید بتواند داخل آن ثبت شود. ساختار این تابع در زیر آمده است.

```
def reset_registry():
  global REGISTRY_IDX
  global REGISTRY
  REGISTRY_IDX = 0
  REGISTRY = {}
```

سپس در ادامه شی جدیدی از کلاس pyCFG که در قسمت بعدی توضیح داده خواهد شد، ساخته می‌شود و با فراخوانی تابع gen_cfg از داخل این کلاس گراف جریان کنترلی برای کد ورودی ساخته می‌شود. و سپس دو گزینه مختلف وجود دارد که گره شروع و پایان را از گراف پاک می‌کند و در صورتی که کاربر بخواهد این گره‌ها می‌تواند در گراف باقی بماند.

حال به ساختار کلاس pyCFG می‌پردازیم. در ایتدا با فراخوانی این کلاس و ساخت شی جدید از آن یک گراف خالی با گره start ساخته می‌شود. این کار در کد پایین قابل مشاهده است.

```
class PyCFG:
  def __init__(self):
    self.founder = CFGNode(
       parents=[], ast=ast.parse('start').body[0])   # sentinel
    self.founder.ast_node.lineno   = 0
    self.functions = {}
    self.functions_node  = {}
```

سپس با فراخوانی gen_cfg با ورودی سورس کد تابع، به ترتیب چند مرحله زیر اجرا می‌شود:

```
  def gen_cfg(self, src):
    node = self.parse(src)
    nodes = self.walk(node, [self.founder])
    self.last_node  = CFGNode(parents=nodes,  ast=ast.parse('stop').body[0])
    ast.copy_location(self.last_node.ast_node,  self.founder.ast_node)
    self.update_children()
    self.update_functions()
    self.link_functions()
```

همانطور که مشاهده می‌شود ایتدا تابع parse فراخوانی می‌شود و کد تابع به آن ارسال می‌شود تا کد به شکل کامل تجزیه شود کارکرد این تابع نیز در پایین آمده است:

```
Import ast
def parse(self, src):
    return ast.parse(src)
```

در این کد ار کتابخانه ast استفاده شده است که از کتابخانه‌های استاندارد پایتون می‌باشد و کد پایتون را به صورت خودکار تجزیه می‌کند. و این کتابخانه خود تابع parse را دارد که به صورت کامل این کار را انجام می‌دهد.

در مرحله‌ی بعد تابع walk از کلاس pyCFG فراخوانی می‌شود و تجزیه شده‌ی کد به عنوان اولین ورودی این تابع خواهد بود و دومین ورودی این تابع والدهای این گره در گراف می‌باشد و در هنگام ساخت کلاس تعرف شده است که founder بوده و در اولین اجرا به گره start اشاره دارد. و ساختار walk به صورت زیر می‌باشد:

```
  def walk(self, node, myparents):
    fname = "on_%s" % node.__class__.__name__.lower()
```

```
    if hasattr(self, fname):
      fn = getattr(self, fname)
      v = fn(node, myparents)
      return v
    else:
      return myparents
```

این تابع ابتدا چک می‌کند که کدام یک از دستورات زبان پایتون استفاده شده است این کار را با تابع hasattr انجام می‌دهد و اگر تابع مربوط به آن در کلاس ما تعریف شده باشد با تابع getattr آن تابع فراخوانی می‌شود. در انتهای گزارش چند مورد از توابعی که در مواجهه با دستورات فراخوانی می‌شوند آورده می‌شود.

مراحل بعدی در gen_cfg به این صورت می‌باشد که با copy_location ابتدا گره فرزند و والد فعلی را ذخیره می‌کند سپس لیست فرزندان و والدها را آپدیت می‌کند که این دو با توابع update_function و update_children انجام می‌شود. سپس تابع link_functions باعث خواهد شد تا این فرآیند به صورت بازگشتی برای گره‌های تابع جدید که ازداخل تابع فعلی فراخوانی شده است تکرار شود به این صورت که فرزندها به عنوان والد برای مرحله بعد در نظر گرفته شده و سپس با پیدا کردن فرزندان این والدهای جدید از درخت ast به ساخت گراف جریان کنترلی ادامه می‌دهیم. کد مربوط به این تابع در زیر آمده است.

```
def link_functions(self):
  for _, node in REGISTRY.items():
    if node.calls:
      for calls in node.calls:
        if calls in self.functions:
          enter, exit = self.functions[calls]
          enter.add_parent(node)
          if node.children:
            # # until we link the functions up, the node
            # # should only have succeeding node in text as
            # # children.
            # assert(len(node.children)  == 1)
            # passn = node.children[0]
            # # We require a single pass statement after every
            # # call (which means no complex expressions)
            # assert(type(passn.ast_node)  == ast.Pass)
            # # unlink the call statement
            assert node.calllink  > -1
            node.calllink  += 1
            for i in node.children:
              i.add_parent(exit)
            # passn.set_parents([exit])
            # ast.copy_location(exit.ast_node,  passn.ast_node)
            # #for c in passn.children:  c.add_parent(exit)
            # #passn.ast_node  = exit.ast_node
```

نحوه نصب و اجرا:

در ابتدا برای نصب پیش نیازهای برنامه یک محیط مجازی برای پایتون ایجاد می‌کنیم. در دومین گام محیط را فعال می‌کنیم. در سومین گام نیازمندی‌های برنامه در محیط فعال شده نصب می‌شود که این پیش نیازها در فایل requirement.txt لیست شده است. در آخرین گدم فایل example که شی cfg در آن ایجاد شده و توابع آن فراخوانی شده است را اجرا می‌کنیم.

1. Virtualenv venv
2. .\venv\scripts\activate
3. pip install –r requirement.txt
4. python3 example.py

نحوه استفاده:

در فایل example.py ما با ایجاد یک شی cfg شروع می‌کنیم. ورودی های این شی برای ساختار ابتدایی عبارت است ازمسیر ورودی برنامه پایتونی که می‌خواهیم گراف جریان کنترلی آن را استخراج کنیم دومین ورودی نام فایل پایتونی است. سومین ورودی نام تابعی است که می‌خواهیم گراف جریان کنتری آن را استخراج کنیم به کار می‌آید و آخرین ورودی مسیری است که خروجی های توابع این شی می‌خواهیم در آن قرار دهیم وارد می‌شود.

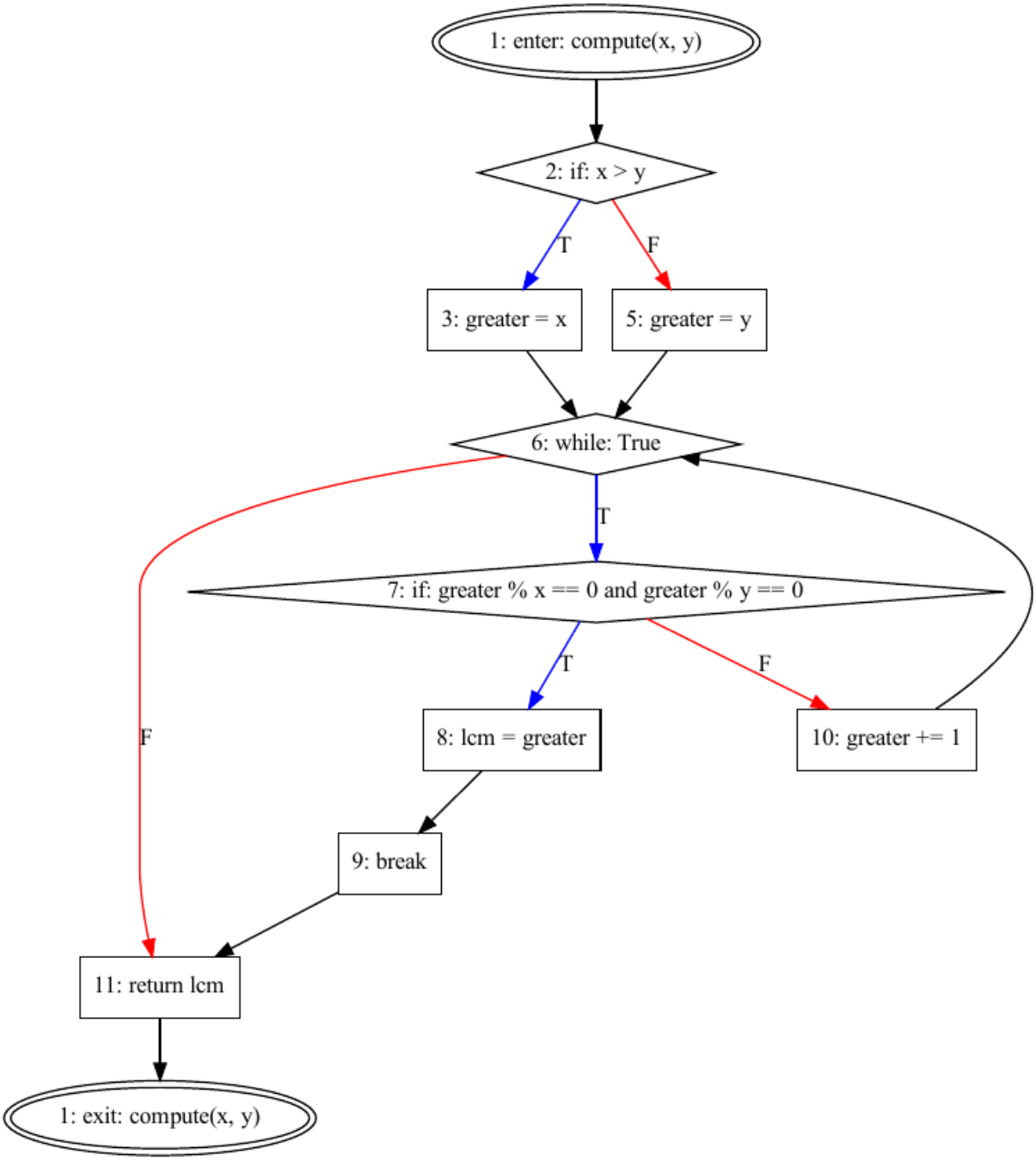
```
example = cfg("./example", "lcm.py", "compute", "./output")
example.extract_pdf()
example.extract_dot()
example.show_graph()
```

مثال از ورودی و خروجی برنامه:

مثال۱:

```
def compute(x: int, y: int):
  if x > y:
    greater = x
  else:
    greater = y
  while (True):
    if((greater % x == 0) and (greater % y == 0)):
      lcm = greater
      break
    greater += 1
  return lcm
```

گراف جریان کنترلی مربوط به کد بالا در تصویر زیر و به عنوان خروجی برنامه دیده می‌شود:



مثال ۲:

```
def check_triangle(a:  int, b: int, c: int) -> str:
    if a == b:
        if a == c:
            if b == c:
                return "Equilateral"
            else:
                return "Isosceles"
        else:
            return "Isosceles"
    else:
        if b != c:
            if a == c:
                return "Isosceles"
            else:
                return "Scalene"
        else:
            return "Isosceles"
```

گراف جریان کنترلی مربوط به کد بالا در تصویر زیر که خروجی برنامه هست: