الجلسة السابعة: تحسين المترجم

الهدف من الجلسة

تصميم مترجم وقياس أداءه وفقاً لمعاييرمحددة.

التعرف على طرق تحسين المترجم.

خطوات العمل:

1. تصميم مترجم يقوم بترجمة كود يحتوي على عبارة if/else الشرطية إضافة إلى تعريف متحويلات وإسناد قيم لها.
2. قياس أداء المترجم المُصمَّم وفقاً لمعايير أداء محددة.
3. إجراء عملية optimization للمترجم المُصمَّم بطريقتين: constant foldingو dead code elimination.

1. تصميم المترجم:

سنقوم بتصميم مترجم يحقق ترجمة للكود المصدري التالي دون حدوث أخطاء:

```
int x=2*3;
if(x>3) then print("hello");
bool y=x||5;
int i=0;
if(false) then i=10;
```

نلاحظ من الكود أن المترجم يجب أن يتعرف على:

- عبارة if then.
- التصريح عن المتغيرات.
- إسناد قيم للمتحولات.

يتكون المترجم من قسمين:

1.1 **بناء الماسح:**

وفيه نحدد كل الرموز tokens المستخدمة والاستجابة لهذه الرموز.

تحديد الرموز مع الكلمات المحجوز:

```
class Main_Parser:
    def __init__(self):
        self.tokens = (
            'IF', 'THEN', 'ELSE',
            'ID', 'NUM',
            'LPAR', 'RPAR', 'SEMI',
            'LT', 'GT', 'LE', 'GE', 'EQ', 'NE',
            'OR', 'AND','INT','BOOL',
            'PLUS', 'MINUS', 'MULT', 'DIVS',
            'EQUAL','PRINT','DQ','TRUE', 'FALSE', 'INC','DEC',
            'RCPAR','LCPAR'
        )

        self.reserved = {
            'if': 'IF',
            'then': 'THEN',
            'else': 'ELSE',
            'int': 'INT',
            'bool': 'BOOL',
            'char': 'CHAR',
            'print': 'PRINT',
            'while': 'WHILE',
            'true' : 'TRUE',
            'false' :'FALSE'
        }
```

تحديد الاستجابة للرموز:

```
        t_LPAR = r'\('
        t_RPAR = r'\)'
        t_SEMI = r';'
        t_LT = r'<'
        t_GT = r'>'
        t_LE = r'<='
        t_GE = r'>='
        t_EQ = r'=='
        t_NE = r'!='
        t_OR = r'\|\|'
        t_AND = r'&&'
        t_INC=r'\+\+'
        t_DEC=r'\-\-'
        t_PLUS = r'\+'
        t_MINUS = r'-'
        t_MULT = r'\*'
        t_DIVS = r'/'
        t_EQUAL = r'='
        t_DQ= r'\"'
        t_SQ=r'\''
        t_RCPAR=r'\{'
        t_LCPAR=r'\}'
        t_ignore = ' \t'
```

```
def t_NUM(t):
    r'\d+'
    t.value = int(t.value)
    return t

def t_ID(t):
    r'[a-zA-Z_][a-zA-Z_0-9]*'
    t.type = source.getReserved().get(t.value, 'ID')
    return t
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

def t_error(t):
    print(f"Illegal character '{t.value[0]}'")
    t.lexer.skip(1)
```

1.2 **بناء المعرب**:

حيث يتم تحديد أولوية العمليات الحسابية وتحديد القواعد اللازمة لتحقيق ترجمة صحيحة.

إضافة الأولويات:

```
    # Precedence rules
    self.precedence = (
        ('right', 'EQUAL'),
        ('left', 'OR'),
        ('left', 'AND'),
        ('left', 'LT', 'GT', 'LE', 'GE', 'EQ', 'NE'),
        ('left', 'PLUS', 'MINUS'),
        ('left', 'MULT', 'DIVS'),
    )
```

كتابة القواعد:

```
    def p_S(p):
        '''S : ST'''
        print("Input accepted.")
        p[0] = p[1]
    def p_ST_exp_assign(p):
        'ST : TYPE ID EQUAL E4 SEMI'
        p[0]=('assign',p[2],p[4])

    def p_ST_TYPE(p):
        '''TYPE : INT
                | BOOL'''
        p[0]=('type_assign',p[1])

    def p_E4(p):
        'E4 : ST1'
        p[0]=('assign_statement',p[1])

    def p_ST_if_then_else(p):
        'ST : IF LPAR E2 RPAR THEN ST1 SEMI ELSE ST1 SEMI'
        p[0] = ('if_then_else', p[3], p[6], p[9])
```

```
def p_ST_if_then(p):
    'ST : IF LPAR E2 RPAR THEN ST1 SEMI'
    p[0] = ('if_then', p[3], p[6])

def p_ST1_statement(p):
    'ST1 : ST'
    p[0] = p[1]
def p_E_true(p):
    'E2 : TRUE'
    p[0] = True
def p_E_false(p):
    'E2 : FALSE'
    p[0] = False
def p_ST1_expression(p):
    'ST1 : E'
    p[0] = p[1]
def p_ST1_print(p):
    'ST1 : PRINT LPAR DQ E3 DQ RPAR'
    p[0] = ('print st',p[1],p[4])
def p_E3(p):
    'E3 : E'
    p[0] = ('string',p[1])
```

```
def p_E_assignment(p):
    'E : ID EQUAL E'
    p[0] = ('assign', p[1], p[3])

def p_E_binop(p):
    '''E : E PLUS E
         | E MINUS E
         | E MULT E
         | E DIVS E
         | E LT E
         | E GT E
         | E LE E
         | E GE E
         | E EQ E
         | E NE E
         | E OR E
         | E AND E'''
    p[0] = ('binop', p[2], p[1], p[3])

def p_E_negative(p):
    'E : MINUS E'
    p[0] = ('neg', p[2])
```

```
def p_E_negative(p):
    'E : MINUS E'
    p[0] = ('neg', p[2])

def p_E_id(p):
    'E : ID'
    p[0] = ('id', p[1])

def p_E_num(p):
    'E : NUM'
    p[0] = ('num', p[1])

def p_E2_binop(p):
    '''E2 : E LT E
          | E GT E
          | E LE E
          | E GE E
          | E EQ E
          | E NE E
          | E OR E
          | E AND E'''
    p[0] = ('binop', p[2], p[1], p[3])

def p_E2_id(p):
    'E2 : ID'
    p[0] = ('id', p[1])

def p_E2_num(p):
    'E2 : NUM'
    p[0] = ('num', p[1])

def p_error(p):
    if p:
        print(f"Syntax error at '{p.value}'")
    else:
        print("Syntax error at EOF")
```

عند تنفيذ البرنامج وكتابة 'C' يظهر الخرج التالي، وهو دلالة على ترجمة الكود المصدري السابق كاملةً:

```
Test 1: int x=2*3;
Input accepted.

Test 2: if(x>3) then print("hello");
Input accepted.

Test 3: bool y=x||5;
Input accepted.

Test 4: int i=0;
Input accepted.

Test 5: if(false) then i=10;
Input accepted.
```

عند تنفيذ البرنامج وكتابة 'U' سيقوم بتنفيذ unit testing على مجموعة من الأسطر البرمجية، وسنتطرق لذلك فيما بعد.

2. تقييم تصميم المترجم:

التصميم الجيد للمترجم يجب أن يحقق خرج صحيح من أجل كل سطر برمجي في الكود المصدري، كما يجب أن يحقق أداء جيد من ناحية سرعة الترجمة والذاكرة المستهلكة وسرعة التنفيذ.

بعض معايير الأداء:

1. صحة الخرج: للتحقق من صحة خرج المترجم الذي صممناه يمكن اختبار كود مصدري كامل أو إجراء unit testing على مجموعة اختبارات شاملة.
2. زمن ترجمة الكود المصدري من قبل المترجم الذي قمنا بتصميمه.
3. كمية الذاكرة المستهلكة أثناء عملية الترجمة.

2.1 اختبار صحة خرج المترجم:

وهنا نريد أن نتأكد من التصميم الذي قمنا بتطبيقه يؤدي الغرض المطلوب منه.

1. سنقوم باختبار كود كامل وهوما ظهر معنا عند تنفيذ البرنامج وكتابة 'C' في الفقرة السابقة.
2. الاختبار عن طريق الـ unit testing لمجموعة من الأسطر البرمجية وهي كالتالي:

```
test_cases=["if (x < 5) then y = 10;",
"if (x < 5 then y = 10;",
"if (x < 5)  y = 10;",
"if (x < 5) then y = 10; else y=1;",
"int x = 1;",
"int x = 1",
"x = 1;"
]
```

عند تنفيذ البرنامج وكتابة 'U' نحصل على الخرج التالي:

```
testing for if (x < 5) then y = 10;
Input accepted.
testing for if (x < 5 then y = 10;
Syntax error at 'then'
testing for if (x < 5)  y = 10;
Syntax error at 'y'
testing for if (x < 5) then y = 10; else y=1;
Input accepted.
testing for int x = 1;
Input accepted.
testing for int x = 1
Syntax error at EOF
testing for x = 1;
Syntax error at 'x'
```

الكود الخاص بالاختبار:

```python
class UnitTestingClass(unittest.TestCase):
    def __init__(self, methodName='runTest'):  # ADD DEFAULT PARAMETER
        super().__init__(methodName)  # MUST CALL SUPER INIT
        self.parser = IfElseParserWrapper()
    def test_valid_if_then(self,code):
        res=self.parser.parse(code, optimize=False)
        self.assertIsNotNone(res)
    def test_valid_if_then_else(self,code):
        res=self.parser.parse(code, optimize=False)
        self.assertIsNotNone(res)
    def test_valid_assert(self,code):
        res=self.parser.parse(code, optimize=False)
        self.assertIsNotNone(res)

    @unittest.expectedFailure
    def test_missing_bracket_if_then(self,code):
        res=self.parser.parse(code, optimize=False)
        self.assertIsNone(res)#none or syntax error
    @unittest.expectedFailure
    def test_missing_then_if_then(self,code):
        res=self.parser.parse(code, optimize=False)
        self.assertIsNone(res)

    @unittest.expectedFailure
    def test_no_semi(self,code):
            res=self.parser.parse(code, optimize=False)
            self.assertIsNone(res)
    @unittest.expectedFailure
    def test_false_assertion(self,code):
            res=self.parser.parse(code, optimize=False)
            self.assertIsNone(res)
```

2.2 زمن ترجمة الكود:

هنا سنميز حالة قياس زمن انتهاء عملية الترجمة كاملةً وبين زمن تنفيذ كل عملية.

سنركز على زمن عملية الترجة كاملةً.

يقوم الكود التالي بتحديد نقطة بداية لقياس الزمن ثم البدء بعملية الترجمة،بعد الانتهاء نحدد لحظة النهاية.

ولمعرفة وزمن الترجمة كاملةً سنقوم بحساب الفرق بين زمن بداية الترجمة زمن الانتهاء.

سنكرر هذا القياس عدد من المرات ولذلك بسبب احتمال تأثر عملية الترجمة بعمليات أخرى تحدث في نفس الوقت مما قد يزيد من وقت الترجمة.

```
for i in range(self.test_iterations):
    start_time = time.perf_counter()# set a starting time
    result = getattr(self.parser, parse_method)(input_data) ## Returns the 'parse' method this method is in the if_else compiler file
    end_time = time.perf_counter()# set parsing end time
    elapsed = (end_time - start_time) * 1000  # measure the difference between start and end time and Convert to milliseconds
    times.append(elapsed)
```

بعد الانتهاء من عملية القياس سنقوم بإجراء عمليات إحصائية للحصول على الأداء العام وفقاً لمجموعة من التوابع الموجوجة في مكتبة أساسية تدعى **numpy** المُختصرة ب np وفقاً للكود التالي:

```
# Statistical analysis
times_array = np.array(times, dtype=np.float64)

self.results['time'] = {
    'raw_times': times_array.tolist(),
    'mean': float(np.mean(times_array)),
    'median': float(np.median(times_array)),
    'stdev': float(np.std(times_array, ddof=1)) if len(times_array) > 1 else 0.0,
    'min': float(np.min(times_array)),
    'max': float(np.max(times_array)),
    'total': float(np.sum(times_array))
}
```

المصفوفة time_array لحفظ الزمن المستخدم لترجمة الكود كاملةً .

Mean: المتوسط الحسابي لكل دورة ترجمة للكود.

Median: القيمة الوسطى بين كل الأزمنة.

Stdev: الانحراف المعياري.

Min: القيمة الصغرى للأزمنة.

Max: القيمة العظمى للأزمنة.

نتيجة تنفيذ الكود من أجل 10 تكرارات:

```
Input accepted.

  Time Statistics:
   Mean:      0.17 ms
   Median:    0.17 ms
   Min:       0.16 ms
   Max:       0.18 ms
   Std Dev:   0.01 ms
   Total:     1.66 ms

==================================================
PERFORMANCE SUMMARY
==================================================
  Parsing Time: 0.17 ms (avg)
```

2.3 حجم الذاكرة المستهلك أثناء عملية الترجمة:

سنبدأ بعملية تنظيف للذاكرة اعتماداً على garbage collector.

- ثم البدء بعملية مراقبة الذاكرة.
- قياس حالة الذاكرة قبل البدء بعملية الترجمة مثل حالة الأغراض وحجمها والـ call stack.
- البدء بالترجمة.
- قياس حالة الذاكرة بعد الانتهاء من عملية الترجمة.
- سنحتفظ بقيمة الـ peak memory وهي أكبر من الذاكرة التي يحتاجها المترجم لعملية الترجمة، وقيمة top allocations وهي الجزء الذي استهلك أكبر مقدار من الذاكرة.

عند تنفيذ الكود نحصل على الخرج التالي:

```
  Top allocations:
    - C:\Users\DELL\AppData\Local\Programs\Python\Python313\Lib\site-packages\ply\lex.py:320 - 0.51 KB
    - C:\Users\DELL\AppData\Local\Programs\Python\Python313\Lib\tracemalloc.py:560 - 0.32 KB
    - C:\Users\DELL\AppData\Local\Programs\Python\Python313\Lib\tracemalloc.py:423 - 0.32 KB

==================================================
PERFORMANCE SUMMARY
==================================================
 Peak Memory:  0.00 KB
```

تحسين المترجم

يوجد العديد من الطرق لتحسين المترجم وسنذكرمنها طريقتين:

Constant folding:

وهي طريقة لتحسين الكود عن طريق تعويض العمليات (ومنها الحسابية) التي تجري على الثوابت بناتج الحساب:

مثال:

$$x = 2 * 3 ==> 6$$

Dead Code elimination:

تحسين المترجم عن طريق إزالة الكود الذي لا يفضي إلى نتيجة، مثال:

If(false) then y=1;

بما أن الكود لتنينفذ أبداً فهو كود ميت.

عند تطبيق عمليات التحسين نحصل على الخرج التالي:

مثال constant folding:

```
Original:
assign
  x
  assign_statement
    binop
      *
      num
        2
      num
        3
Input accepted.

Optimized:
assign
  x
  assign_statement
    num
      6

 optimized
```

مثال code elimination:

```
Test 5: if(false) then i=10;
Input accepted.

Original:
if_then
  False
  assign
    i
    num
      10
Input accepted.

Optimized:
```

انتهت الجلسة