Anggota:

- Arkan Ramadhan Nugraha
  NIM 241524033
- Fauzi Ismail
  NIM 241524042
- Hizba Shafwatuddin
  NIM 241524046

Kelas: 2B – D4

Tugas Kelompok Programming Language Pragmatics.

Rencana Domain DSL yang akan diimplementasi, serta Pemahaman ANTLR

DSL

1. Domain

Domain yang dimodelkan oleh DSL ini adalah "Validasi Siklus Hidup Memori Statis" (Static Memory Lifecycle Validation).

DSL ini beroperasi pada fase Implementasi (Coding) dan Verifikasi (Verification). Tujuannya adalah untuk mendefinisikan secara formal pasangan fungsi yang mengalokasikan (allocate) dan membebaskan (deallocate) memori. Dengan definisi ini, sebuah *tool* analisis statis dapat membaca kode sumber (tanpa menjalankannya) dan menganalisis semua jalur eksekusi (control flow paths) untuk menemukan orphaned memory.

Secara spesifik, domain ini berfokus pada penegakan *policy* (kebijakan) bahwa setiap *pointer* yang menerima alokasi memori *harus* menemui panggilan dealokasi yang sesuai di *setiap* jalur eksekusi yang mungkin (termasuk *exception)*

2. Entitas (Entities)

Entitas dalam domain ini sangat terfokus. Kita tidak melacak *semua* variabel, hanya *pointer* yang terkait dengan alokasi heap.

- HeapMemory: Ini adalah entitas *konseptual* utama. Ia merepresentasikan sebuah blok memori yang tidak terkelola (unmanaged) yang diperoleh dari *heap*. Entitas ini memiliki "status" (state) yang berubah selama eksekusi.

- MemoryPointer: Ini adalah representasi *konkret* di dalam kode (misal: void* ptr). Ini adalah "pegangan" (handle) ke HeapMemory. *Tool* analisis akan melacak variabel-variabel yang bertipe MemoryPointer.

Status Entitas (HeapMemory) Siklus hidup entitas ini adalah inti dari domain:

1. UNALLOCATED (Belum dialokasi): Status awal.
2. ALLOCATED (Dialokasi): Status setelah fungsi alokasi dipanggil.
3. FREED (Dibebaskan): Status setelah fungsi dealokasi dipanggil.

3. Relasi

Ini adalah "resep" atau struktur sintaksis dari DSL. Relasi ini mendefinisikan *bagaimana* kita memberi tahu *tool* analisis tentang *mana* fungsi yang mengubah status entitas HeapMemory.

Grammar utamanya adalah:

track_memory { [Blok Definisi] }

Di dalam Blok Definisi, kita memiliki relasi-relasi berikut:

- allocate_by: [List of Functions]
    - Relasi: Ini *merelasikan* satu atau lebih nama fungsi (sebagai *string*) ke aksi yang mengubah status HeapMemory dari UNALLOCATED menjadi ALLOCATED.
    - Contoh: allocate_by: ["malloc", "calloc", "MyCustomAllocator"]
- deallocate_by: [List of Functions]
    - Relasi: Ini *merelasikan* satu atau lebih nama fungsi ke aksi yang mengubah status HeapMemory dari ALLOCATED menjadi FREED.
    - Contoh: deallocate_by: ["free", "MyCustomFree"]
- transfer_ownership: [List of Functions]
    - Relasi: (Opsional, tapi penting) Ini *merelasikan* fungsi-fungsi yang "memindahkan" tanggung jawab untuk mem-free ke konteks lain (misal: memasukkan *pointer* ke dalam *struct*).
    - Contoh: transfer_ownership: ["StorePointerInGlobalList"]

4. Aturan (Rules / Constraints)

Aturan di sini bukan sesuatu yang ditulis pengguna di DSL, melainkan logika semantik (semantic logic) yang di-*hardcode* di dalam *tool* analisis. DSL berfungsi sebagai input untuk memberitahu *tool apa* yang harus dicari.

Setelah *tool* membaca file memory.policy di atas, ia akan menerapkan aturan-aturan berikut pada kode sumber:

- Aturan 1: Kebocoran Memori (Memory Leak)
  - Deskripsi: Jika sebuah MemoryPointer (misal: ptr) yang statusnya ALLOCATED (karena ia menerima nilai dari malloc) mencapai akhir dari cakupannya (misal: akhir fungsi) tanpa pernah melewati fungsi deallocate_by (misal: free(ptr)), ini adalah PELANGGARAN.
  - Logika: state(ptr) == ALLOCATED di End-of-Scope -> Error: Memory Leak.
- Aturan 2: Pembebasan Ganda (Double Free)
  - Deskripsi: Jika sebuah MemoryPointer yang statusnya sudah FREED (karena free(ptr) sudah dipanggil) diproses lagi oleh fungsi deallocate_by (misal: free(ptr) dipanggil lagi).
  - Logika: state(ptr) == FREED DAN call(deallocate_by, ptr) -> Error: Double Free.
- Aturan 3: Penggunaan Setelah Dibebaskan (Use After Free)
  - Deskripsi: Jika sebuah MemoryPointer yang statusnya sudah FREED diakses (dibaca/ditulis).
  - Logika: state(ptr) == FREED DAN call(access, ptr) -> Error: Use After Free.
- Aturan 4: Pembebasan Ilegal (Invalid Free)
  - Deskripsi: Jika sebuah MemoryPointer yang statusnya UNALLOCATED (misal: *pointer* ke *stack*, atau *pointer NULL*) diproses oleh fungsi deallocate_by.
  - Logika: state(ptr) == UNALLOCATED DAN call(deallocate_by, ptr) -> Error: Invalid Free.

5. Contoh Bahasa

```
// File: memory.policy
//
// Relasi utama: memulai blok pelacakan
track_memory {

  // Relasi: memetakan string "malloc" dan "calloc"
  // ke aksi 'alokasi' (mengubah state ke ALLOCATED)
```

```
  allocate_by: [
    "malloc",
    "calloc",
    "realloc" // realloc juga bisa mengalokasi
  ]

  // Relasi: memetakan string "free"
  // ke aksi 'dealokasi' (mengubah state ke FREED)
  deallocate_by: [
    "free",
    "realloc" // realloc juga bisa mem-free memori lama
  ]
}
```

Pemahaman Antlr

Laporan Pemahaman ANTLR Berdasarkan Proyek Kalkulator Simpel

Pendahuluan

Pada tugas ini, saya membuat sebuah proyek kalkulator sederhana menggunakan ANTLR untuk mempelajari konsep dasar parsing dan pembuatan compiler/interpreter. Proyek ini dibantu oleh AI (GPT dan Gemini) untuk memahami dan mengimplementasikan ANTLR.

Apa itu ANTLR?

ANTLR (ANother Tool for Language Recognition) adalah sebuah framework open-source yang digunakan untuk membangun parser, compiler, interpreter, dan translator untuk bahasa pemrograman atau bahasa domain khusus (DSL). ANTLR memudahkan proses pembuatan grammar dan menghasilkan kode parser secara otomatis.€€

Langkah-langkah Pengerjaan Proyek Kalkulator

1. Membuat Grammar Kalkulator
    a. Saya mendefinisikan grammar untuk operasi matematika dasar (penjumlahan, pengurangan, perkalian, pembagian).
    b. Grammar ditulis dalam format .g4 yang dipahami oleh ANTLR.
2. Generate Parser dan Lexer

    a. Dengan perintah ANTLR, grammar diubah menjadi kode sumber parser dan lexer (biasanya dalam Java, Python, atau bahasa lain yang didukung).
3. Implementasi Evaluasi Ekspresi
    a. Parser dan lexer hasil generate digunakan untuk membaca dan mengeksekusi ekspresi matematika.
    b. Hasil parsing dievaluasi untuk mendapatkan hasil perhitungan.
4. Testing
    a. Kita mencoba berbagai ekspresi matematika untuk memastikan parser dan evaluator berjalan sesuai harapan.

Pemahaman yang Didapat

- Grammar: Grammar adalah aturan yang mendefinisikan struktur bahasa. Di ANTLR, grammar menentukan bagaimana input dipecah menjadi token dan bagaimana token tersebut diatur menjadi struktur yang bermakna.
- Lexer & Parser: Lexer memecah input menjadi token, parser menyusun token menjadi pohon sintaks (parse tree).
- Parse Tree: Parse tree digunakan untuk mengekstrak makna dari input, misalnya menghitung hasil ekspresi matematika.
- Error Handling: ANTLR menyediakan mekanisme penanganan error yang memudahkan debugging grammar dan input yang tidak valid.

Refleksi Penggunaan AI (GPT & Gemini)

- AI sangat membantu dalam memahami konsep grammar, debugging, dan memberikan contoh implementasi.
- AI juga membantu menjelaskan error dan solusi secara cepat, sehingga proses belajar lebih efisien.

Kesimpulan

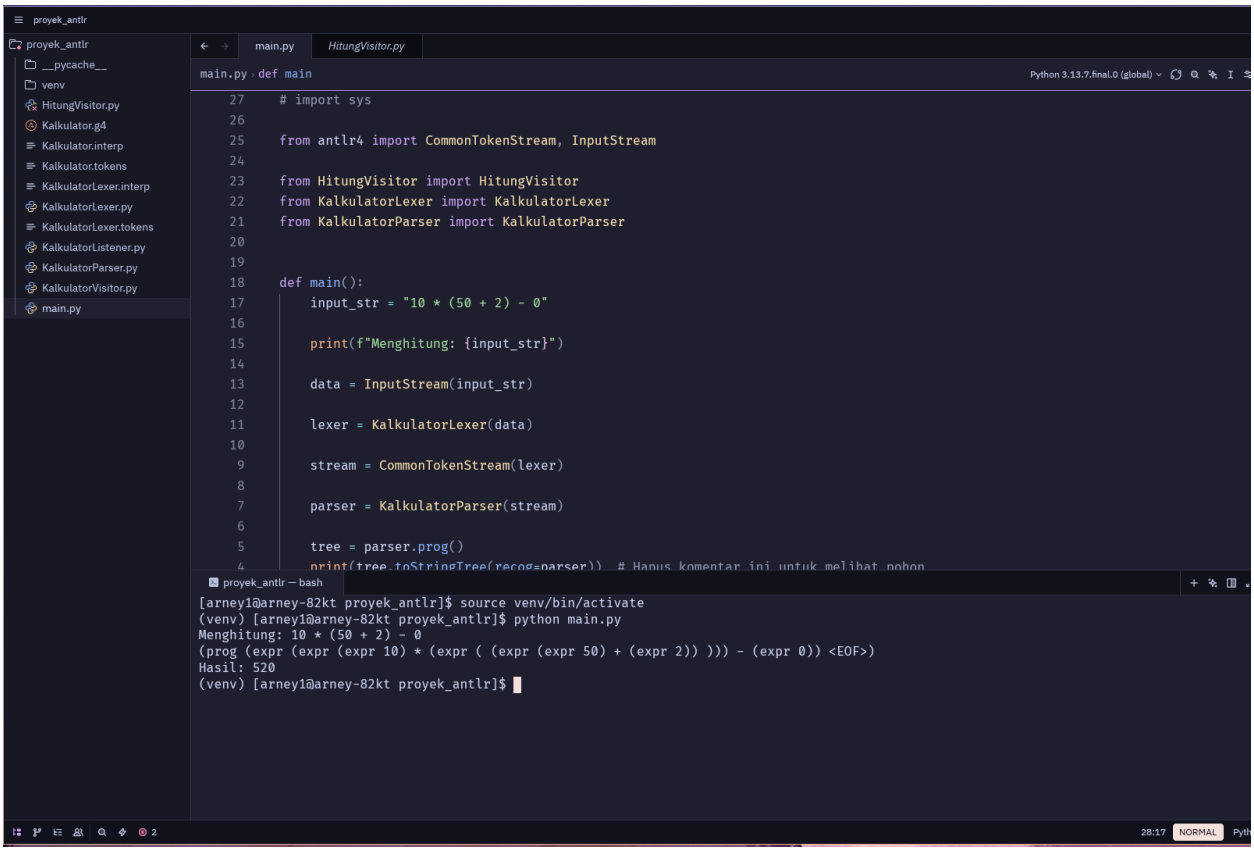
Dengan proyek kalkulator kecil ini, saya memahami dasar penggunaan ANTLR untuk parsing dan evaluasi ekspresi. Pengalaman ini membuka wawasan tentang bagaimana bahasa pemrograman dan interpreter bekerja di balik layar.

Kalkulator.g4

```
grammar Kalkulator;

prog: expr EOF;

expr
    : expr op=(MUL | DIV) expr  # MulDiv
    | expr op=(ADD | SUB) expr  # AddSub
    | INT                       # Angka
    | '(' expr ')'              # Kurung
    ;

MUL : '*' ;
DIV : '/' ;
ADD : '+' ;
SUB : '-' ;
INT : [0-9]+ ;

WS  : [ \t\r\n]+ -> skip ;

```

HitungVisitor.py › class **HitungVisitor** › def visitAngka

```python
from KalkulatorVisitor import KalkulatorVisitor
from KalkulatorParser import KalkulatorParser

class HitungVisitor(KalkulatorVisitor):
    def visitProg(self, ctx):
        return self.visit(ctx.expr())

    def visitAngka(self, ctx):
        return int(ctx.INT().getText())

    def visitKurung(self, ctx):
        return self.visit(ctx.expr())

    def visitAddSub(self, ctx):
        kiri = self.visit(ctx.expr(0))
        kanan = self.visit(ctx.expr(1))

        if ctx.op.type == KalkulatorParser.ADD:
            return kiri + kanan
        else:
            return kiri - kanan

    def visitMulDiv(self, ctx):
        kiri = self.visit(ctx.expr(0))
        kanan = self.visit(ctx.expr(1))

        if ctx.op.type == KalkulatorParser.MUL:
            return kiri * kanan
        else:
            if kanan == 0:
                raise ZeroDivisionError("Tidak bisa membagi dengan nol")
            return kiri / kanan

```


proyek_antlr
proyek_antlr
__pycache__
venv
HitungVisitor.py
Kalkulator.g4
Kalkulator.interp
Kalkulator.tokens
KalkulatorLexer.interp
KalkulatorLexer.py
KalkulatorLexer.tokens
KalkulatorListener.py
KalkulatorParser.py
KalkulatorVisitor.py
main.py
main.py
HitungVisitor.py
main.py › def main
Python 3.13.7.final.0 (global)
27 # import sys
26
25 from antlr4 import CommonTokenStream, InputStream
24
23 from HitungVisitor import HitungVisitor
22 from KalkulatorLexer import KalkulatorLexer
21 from KalkulatorParser import KalkulatorParser
20
19
18 def main():
17 input_str = "10 * (50 + 2) - 0"
16
15 print(f"Menghitung: {input_str}")
14
13 data = InputStream(input_str)
12
11 lexer = KalkulatorLexer(data)
10
9 stream = CommonTokenStream(lexer)
8
7 parser = KalkulatorParser(stream)
6
5 tree = parser.prog()
4 print(tree.toStringTree(recog=parser)) # Hapus komentar ini untuk melihat pohon
proyek_antlr — bash
[arney1@arney-82kt proyek_antlr]$ source venv/bin/activate
(venv) [arney1@arney-82kt proyek_antlr]$ python main.py
Menghitung: 10 * (50 + 2) - 0
(prog (expr (expr (expr 10) * (expr ((expr (expr 50) + (expr 2))))) - (expr 0)) <EOF>)
Hasil: 520
(venv) [arney1@arney-82kt proyek_antlr]$
28:17
NORMAL