Manutenção e Evolução de Software

Projeto Prático

Inês Silva PG55949

Filipa Gomes PG57536

Tiago Silva PG57614

Objetivos

- Linguagem AST
- Parsing de programas
- *Pretty Printing*
- Optimização e Refactorização
- Testes de Software

Linguagem AST

Estrutura da AST usando Dataclasses

```
@dataclass
class Program:
    functions: List[Function]
    global_vars: List[VarDecl]

@dataclass
class Function:
    return_type: str
    name: str
    params: List[VarDecl]
    body: Block
```

Hierarquia de Classes

Expressões (Expr):

- Literal, Variable
- BinaryOp (+, -, *, /, ==, &&, ||)
- UnaryOp (!, ++, --)
- Assignment, FunctionCall

Statements (Stmt):

- VarDecl, Block
- If, While, For
- Return, Print

Linguagem AST

Exemplo de Transformação

Código Fonte

```
int main() { int x = 5; return x * 2; }
```

↓

AST Gerada

```
Program( functions=[ Function( return_type="int",
      name="main", params=[], body=Block([
    VarDecl("int", "x", Literal(5)), Return(
BinaryOp( Variable("x"), "*", Literal(2) ) ) ]) )
                        ] )
```

Program Parsing

Parser Combinators com Parsec

Biblioteca Parsec: A biblioteca **parsec** permite construir parsers complexos através da composição de parsers menores. Principais características utilizadas:

- **@generate:** Sintaxe monádica com yield
- **Combinadores:** sepBy, optional, many, choice
- **Lexeme parsing:** Gestão de whitespace
- **ParseError:** Erros com posição

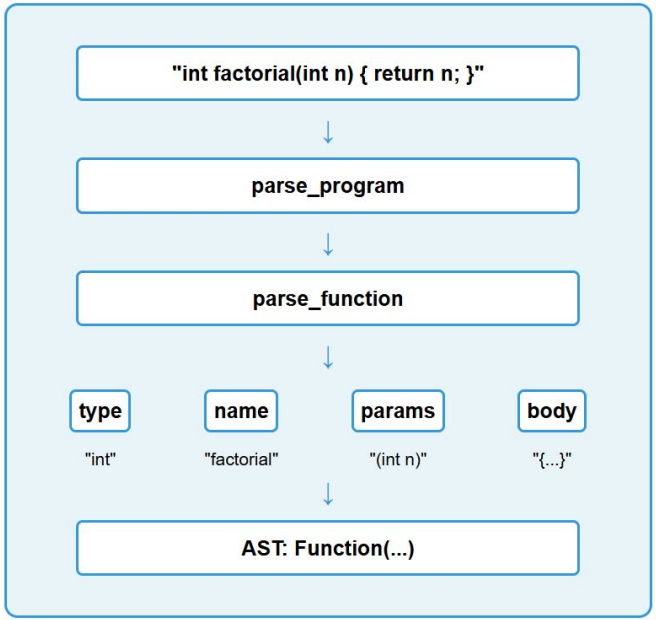
```python
def parse_code(code: str) -> Program:
    try:
        return parse_program.parse(code)
    except ParseError as e:
        raise Exception(f"Erro: {e}")

@generate
def parse_function():
    return_type = yield type_parser  # P
    name = yield ident               # a
    yield lparen                     # c
    params = yield sepBy(...)        # c
    yield rparen
    body = yield parse_block
    return Function(...)
```

Estrutura Modular com Parsec

- **Tokens:** lexeme(string('(')), regex(r'[a-zA-Z_]*')
- **Expressões:** Precedência via composição
- **Statements:** choice_parser para alternativas
- **Recursão:** Forward declarations

Program Parsing

Fluxo do parse_code



Testes do Parser

Foram realizados dois tipos de testes:

1. **Testes individuais** para cada parser (v2testParser.py)
2. **Programas completos** (testParser.py)
   a. 3 **programas válidos**(factorial, fibonacci, isPrime)
   b. 3 **programas inválidos**(ponto e vírgula em falta, parênteses desbalanceados, tipo inválido)

Program Parsing

Exemplos de testes

1. **Individuais**

```
✓ parse_comparison funciona :  BinaryOp(left=BinaryOp(left=Variable(name
='i'), op='*', right=Literal(value=2)), op='>', right=Literal(value=8))
✓ parse_expr funciona com int:  Literal(value=42)
✓ parse_expr funciona com variável:  Variable(name='variavel')
✓ parse_expr funciona com parênteses:  Literal(value=42)
2. TESTANDO PARSE_VAR_DECL
✓ Declaração de variável parseada com sucesso: VarDecl
  Tipo: int
  Nome: i
  Init: 0
✓ Declaração de variável parseada com sucesso: VarDecl
  Tipo: bool
  Nome: a
  Init: (expressão)

TESTANDO PARSE_IF
✓ Statement if parseado com sucesso: If
  Tem condição: BinaryOp(left=Variable(name='x'), op='>', right=Literal(va
lue=0))
  Tem ramo then: Block(statements=[Return(value=Variable(name='x'))])
  Tem ramo else: Block(statements=[Return(value=Literal(value=0))])
```

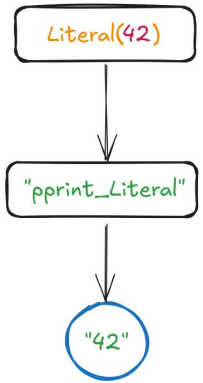
2. **Programas Completos**

```
=== TESTE DO PARSER DA LINGUAGEM LANG ===
Testagem de programas válidos:
Programa 1 (Fatorial) parseado com sucesso!
Número de funções: 2
Funções: ['factorial', 'main']

Programa 2 (Fibonacci) parseado com sucesso!
Número de funções: 1
Funções: ['main']

Programa 3 (Verificação de primo) parseado com sucesso!
Número de funções: 2
Funções: ['isPrime', 'main']

====================================
Testagem de programas inválidos:
ERRO: Programa 4 (falta ponto e vírgula) foi parseado sem erros!
ERRO: Programa 5 (parênteses desbalanceados) foi parseado sem erros!
ERRO: Programa 6 (tipo inválido) foi parseado sem erros!
====================================
```

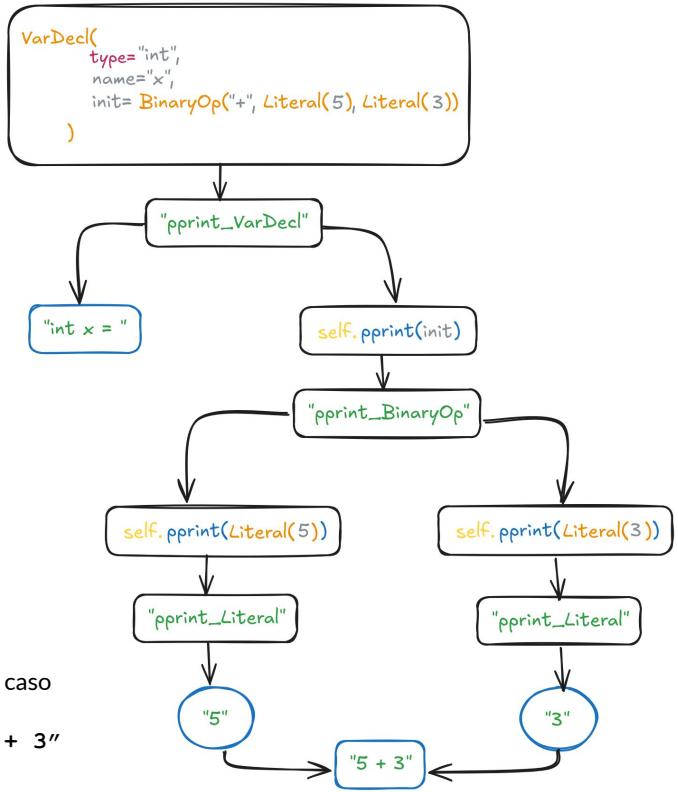
Pretty Printing

```python
def pprint(self, node: Any, parent_prec: int = 0) -> str:
    method = 'pprint_' + node.__class__.__name__
    if hasattr(self, method):
        return getattr(self, method)(node)
    else:
        raise NotImplementedError(...)
```

Gerador dinâmico de código a partir de uma AST



Exemplo de *flow* para o caso de `Literal()`
Output: **"42"**



Exemplo de *flow* para o caso de `VarDecl(...)`
Output: **"int x = 5 + 3"**

Optimização e Refactorização

Optimização:

- `5 + 3` → `8`
- `x + 0` → `x`, `x * 1` → `x`, `x * 0` → `0`
- `x && true` → `x`, `x || false` → `x`

Refactoring:

- `x == true` → `x`, `x == false` → `!x`
- `if true then ...` → just `...`

Utilidades

- `names(ast)`
- `instructions(ast)`
- `code_smells(ast)`

Optimização e Refactorização

Optimização:

Antes

```
int main() {
  bool flag = true;
  bool otherFlag = false;
  int a = (2 + 3) * (4 - 1);
  int b = x + 0;
  int c = 1 * y;
  int d = z * 0;
  if (true) {
    a = 42;
  } else {
    a = 0;
  }
  int e = flag == true;
  int f = otherFlag == false;
  return a + b + c + d + e + f;
}
```

Depois

```
Optimized:


int main() {
  bool flag = true;
  bool otherFlag = false;
  int a = 15;
  int b = x;
  int c = y;
  int d = 0;
  if (true) {
    a = 42;
  } else {
    a = 0;
  }
  int e = flag == true;
  int f = otherFlag == false;
  return a + b + c + d + e + f;
}
```

Optimização + Refactorização:

Depois

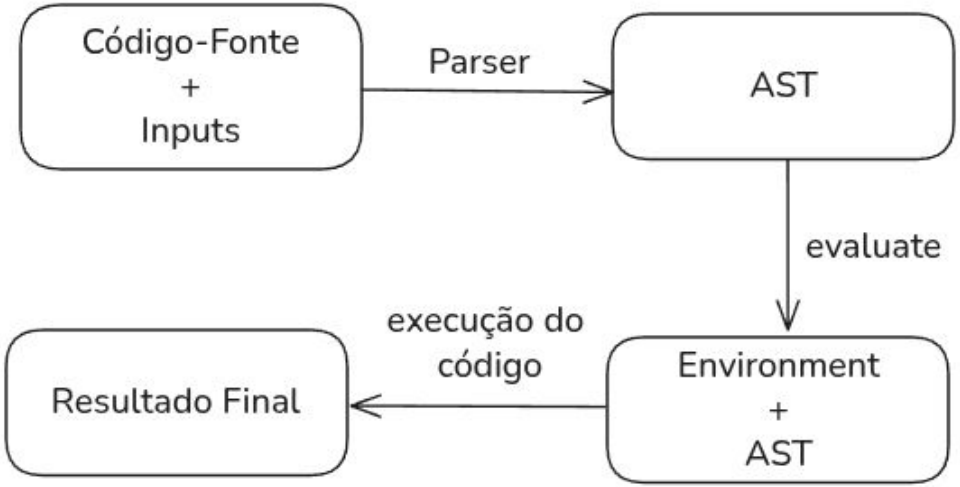
```
Optimized+Refactored:


int main() {
  bool flag = true;
  bool otherFlag = false;
  int a = 15;
  int b = x;
  int c = y;
  int d = 0;
  {
    a = 42;
  }
  int e = flag;
  int f = !otherFlag;
  return a + b + c + d + e + f;
}
```

Testes de Software

Esta fase foi dividida em 4 subfases principais:

- Evaluate
- Testes Unitários
- Mutantes
- Instrumentação

Evaluate



- Avaliação de expressões aritméticas, lógicas e atribuições.
- Controle de fluxo implementado: if, while, for, return.
- Cria um Environment que armazena as variáveis e os seus valores

Testes Unitários

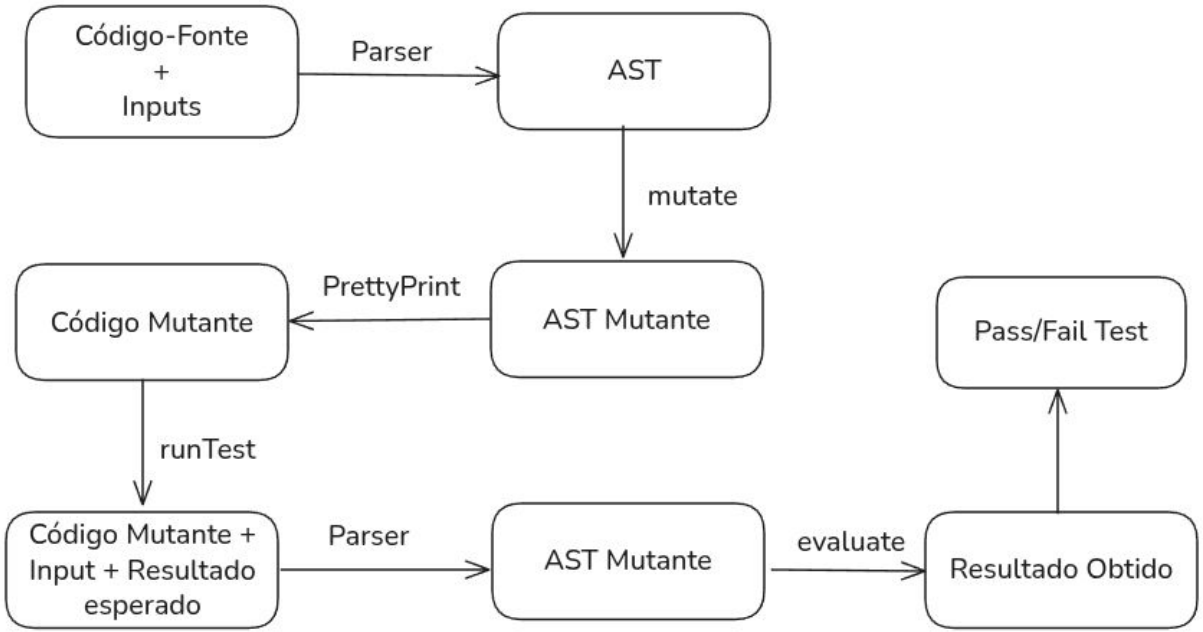
runTest:

- Recebe código, inputs e resultados esperados
- Faz uso da função evaluate e compara o resultado obtido com o esperado

runTestSuite:

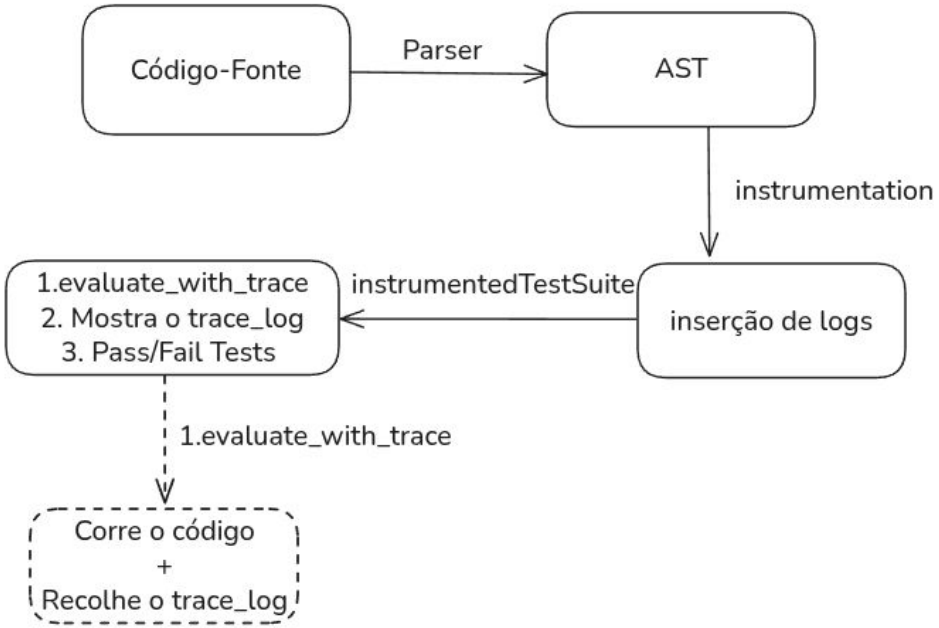
- itera sobre todos os casos de teste

Mutantes



- Procura nós AST em que pode fazer mutações (operações binárias, if's, returns, variáveis)
- Número de mutações é aleatória

Instrumentação



- Criação de uma nova função evaluate que recolhe os logs
- Permite correr os testes unitários e perceber o caminho percorrido pelo código