Scanner e Parser de MiniJava+

Bruna Becker **Pedro Lanzarini**

Instituto de Computação – Universidade Federal Fluminense (UFF)

Compiladores - A1 (2024/2) Luis Antonio Brasil Kowada

Documento de descrição da implementação.

1. Scanner

O scanner é responsável por analisar o código fonte e convertê-lo em uma sequência de tokens, que são os blocos básicos da linguagem.

1.1. Funcionalidades

- Identifica e classifica os tokens baseados na especificação léxica da linguagem MiniJava+.
- Ignora espaços em branco e comentários.
- Reporta erros caso sejam encontrados caracteres inesperados.

1.2. Implementação

O scanner está implementado no arquivo **minijava_scanner.py**. Ele utiliza expressões regulares para identificar diferentes tipos de tokens:

- **Palavras reservadas**: **class, public, static, void**, etc.
- **Identificadores**: Sequências de caracteres alfanuméricos que começam com uma letra.
- **Números**: Literais inteiros.
- **Operadores**: Incluindo **+, -, *, &&, <, >**, etc.
- **Pontuações**: Caracteres como **(,), {, }, ;**, et cetera.

Os padrões são aplicados iterativamente sobre o código de entrada, adicionando os tokens identificados a uma lista de saída. Tokens de espaços em branco e comentários são ignorados.

1.3. Exemplo

Um exemplo de entrada:

```python
class Factorial {

	public static void main(String[] a) {

	System.out.println(new Fac().ComputeFac(10));

	}
```

```
}
```

Gera os tokens:

[('RESERVED', 'class'), ('IDENTIFIER', 'Factorial'), ('PUNCTUATION', '{'), ...]

---

2. Parser

O parser utiliza a sequência de tokens gerada pelo scanner para construir a árvore sintática abstrata (AST), validando a conformidade do programa com a gramática da linguagem.

Atualizações na Gramática para Evitar Recursão à Esquerda

Para implementar o parser de forma top-down compatível com a abordagem LL(1), foi necessário ajustar a gramática original do MiniJava+ para eliminar recursões à esquerda. Abaixo estão as mudanças realizadas na gramática para garantir compatibilidade com essa abordagem.

Mudanças na Gramática

- **Expressões Aditivas**

Antes:

```Python
AEXP -> AEXP + MEXP
      | AEXP - MEXP
      | MEXP
```

Depois (recursão à esquerda eliminada):

```Python
AEXP -> MEXP AEXP'

AEXP' -> + MEXP AEXP'
```

```
       | - MEXP AEXP'
       | ε
```

- **Expressões Multiplicativas**

Antes:

```
Python
MEXP -> MEXP * SEXP
      | SEXP
```

Depois:

```
Python
MEXP -> SEXP MEXP'
MEXP' -> * SEXP MEXP'
       | ε
```

- **Expressões Relacionais**

Antes:

```
Python
REXP -> REXP < AEXP
      | REXP == AEXP
      | REXP != AEXP
```

```
      | AEXP
```

Depois:

```Python
REXP -> AEXP REXP'

REXP' -> < AEXP REXP'
       | == AEXP REXP'
       | != AEXP REXP'
       | ε
```

- **Expressões Lógicas**

Antes:

```Python
EXP -> EXP && REXP
      | REXP
```

Depois:

```Python
EXP -> REXP EXP'

EXP' -> && REXP EXP'
      | ε
```

Atualizações no Código

As mudanças acima foram incorporadas diretamente no código do parser para refletir a gramática ajustada. Abaixo estão os trechos de código que implementam as novas regras.

- **Expressões Aditivas**

No arquivo **minijava_parser.py**:

```python
def parse_aexp(self):
        """
        AEXP -> MEXP AEXP'
        AEXP' -> + MEXP AEXP'
               | - MEXP AEXP'
               | ε
        """
        left = self.parse_mexp()
        while self.current_token() and self.current_token()[1] in {"+", "-"}:
        operator = self.consume("OPERATOR")[1]
        right = self.parse_mexp()
        left = {"type": "ArithmeticOp", "operator": operator, "left": left,
"right": right}
        return left
```

- **Expressões Multiplicativas**

```python
def parse_mexp(self):
```

```
    """
    MEXP -> SEXP MEXP'
    MEXP' -> * SEXP MEXP'
           | ε
    """
    left = self.parse_sexp()
    while self.current_token() and self.current_token()[1] == "*":
        self.consume("OPERATOR", "*")
        right = self.parse_sexp()
        left = {"type": "ArithmeticOp", "operator": "*", "left": left,
"right": right}
    return left
```

- **Expressões Relacionais**

Python

```
def parse_rexp(self):
    """
    REXP -> AEXP REXP'
    REXP' -> < AEXP REXP'
           | == AEXP REXP'
           | != AEXP REXP'
           | ε
    """
    left = self.parse_aexp()
```

```
    while self.current_token() and self.current_token()[1] in {"<", "==",
"!="}:
        operator = self.consume("OPERATOR")[1]
        right = self.parse_aexp()
        left = {"type": "RelationalOp", "operator": operator, "left": left,
"right": right}
    return left
```

- **Expressões Lógicas**

```
Python
def parse_expression(self):
    """
    EXP -> REXP EXP'
    EXP' -> && REXP EXP'
          | ε
    """
    left = self.parse_rexp()
    while self.current_token() and self.current_token()[1] == "&&":
        self.consume("OPERATOR", "&&")
        right = self.parse_rexp()
        left = {"type": "LogicalAnd", "left": left, "right": right}
    return left
```

Essas mudanças foram essenciais para evitar a recursão à esquerda, garantindo que o parser top-down funcione corretamente. O código foi ajustado para refletir diretamente essas alterações, mantendo a conformidade com a especificação da linguagem MiniJava+.

2.2. Implementação

O parser está implementado no arquivo **minijava_parser.py** como uma classe chamada **MiniJavaParserLL1**. Suas principais características incluem:

- **Estrutura LL(1)**: Cada regra gramatical é implementada como um método recursivo.
- **Funções principais**:
    - **parse_program**: Lida com a estrutura geral do programa (classe principal e classes adicionais).
    - **parse_main**: Analisa a classe principal contendo o método **main**.
    - **parse_class**: Analisa declarações de classes, incluindo variáveis e métodos.
    - **parse_command**: Analisa comandos como **if**, **while**, e atribuições.

2.3. Geração da AST

A AST é representada como uma estrutura hierárquica em forma de dicionários e listas. Por exemplo, a entrada acima geraria a seguinte saída em JSON:

```python
{
    "type": "Program",
    "main_class": {
        "type": "MainClass",
        "class_name": "Factorial",
        "argument_name": "a",
        "commands": [
            {
                "type": "Print",
                "expression": {
                    "type": "MethodCall",
                    "target": {
                        "type": "NewObject",
                        "class_name": "Fac"
```

```
                    },
                    "method_name": "ComputeFac",
                    "arguments": [
                        {
                            "type": "Number",
                            "value": 10
                        }
                    ]
                }
            }
        ]
    },
    "classes": [...]
}
```

---

3. Interface Gráfica

O arquivo **ui.py** fornece uma interface gráfica para a interação com o scanner e parser, permitindo que o usuário insira qualquer código válido e é gerado uma imagem png com a sua respectiva árvore sintática.

3.1. Funcionalidades

- Entrada de código MiniJava+ via uma caixa de texto.
- Botão para gerar a árvore sintática.
- Salva a árvore como uma imagem (**syntax_tree.png**) usando a biblioteca Graphviz.

3.2. Execução

Infelizmente, o executável gerado só funciona corretamente se o diretório "{Caminho para o projeto}\bin" for adicionado manualmente ao **PATH** do sistema operacional. Durante o desenvolvimento, não conseguimos implementar um método para adicionar automaticamente este caminho ao **PATH** durante a execução do código. Por isso, incluímos abaixo a imagem gerada pelo código como uma forma de comprovar a funcionalidade e os resultados esperados. Além disso, na próxima seção, apresentamos alternativas para validar o funcionamento do código.

Caso deseje realizar o ajuste manual no **PATH**, siga os passos abaixo, dependendo do seu sistema operacional:

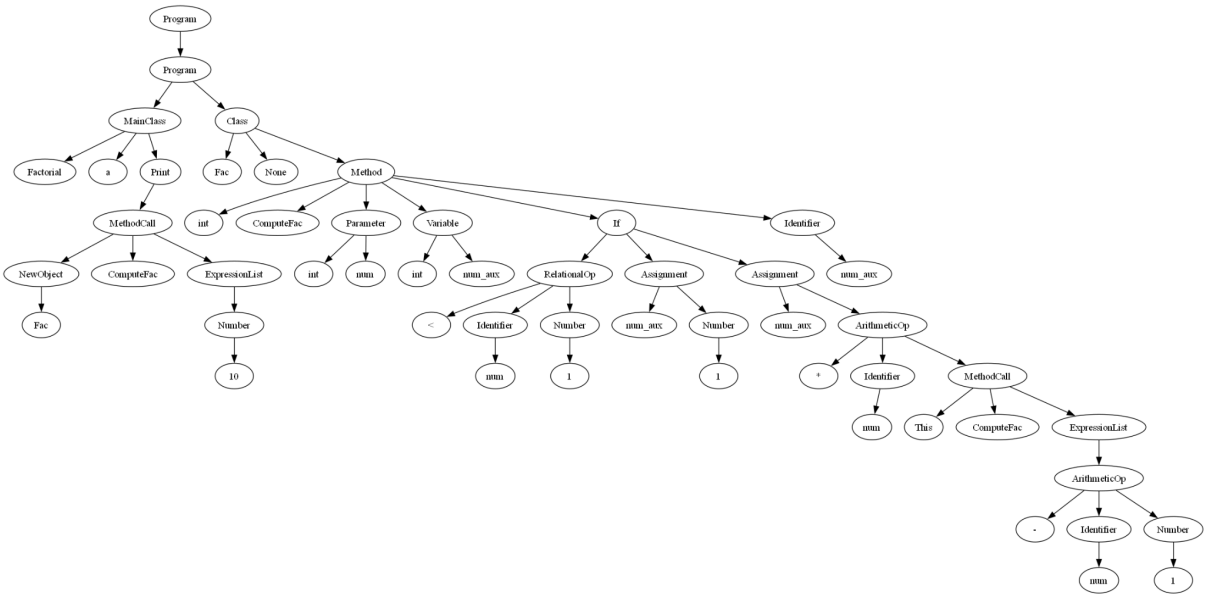
1. **Windows**:
    - Abra o **Painel de Controle** e vá em **Sistema** > **Configurações Avançadas do Sistema**.
    - Clique em **Variáveis de Ambiente**.
    - Localize a variável **PATH**, clique em **Editar**, e adicione o diretório completo onde o executável foi gerado (C:\path\to\compiladorest1\bin).
    - Salve as alterações e reinicie o terminal.

3.3. Resultado

A seguir está o código exemplo utilizado para a geração de um png com o resultado da árvore sintática.

```
Python

class Factorial {

    public static void main(String[] a) {

        System.out.println(new Fac().ComputeFac(10));

    }

}


class Fac {

    public int ComputeFac(int num) {

        int num_aux;

        if (num < 1)
```

```
            num_aux = 1;
        else
            num_aux = num * (this.ComputeFac(num - 1));
        return num_aux;
    }
}
```



---

4. Execução do Programa

A execução do scanner e parser do MiniJava+ pode ser realizada de duas formas principais: utilizando o executável fornecido para interação com a interface gráfica ou executando as partes individualmente. Abaixo estão as instruções detalhadas para cada abordagem.

4.1. Utilizando a Interface Gráfica

A interface pode ser executada tanto com o executável quanto com o script em python, dependendo da disponibilidade para a configuração do ambiente.

Utilizando o Arquivo Executável

O executável **ui.exe** localizado no diretório **compiladorest1\executavel** fornece uma interface gráfica para análise de código MiniJava+. Para executar:

1. Navegue até o diretório **compiladorest1\executavel**.
2. Execute o arquivo **ui.exe**.
3. Insira o código MiniJava+ no campo de texto exibido na interface.
4. Clique no botão "**Gerar Árvore Sintática**".
5. A árvore sintática será salva como uma imagem no mesmo diretório, com o nome **syntax_tree.png**.

Utilizando o Arquivo Python

O arquivo **ui.py** localizado no diretório **compiladorest1** fornece uma interface gráfica e pode ser executado diretamente com Python:

1. Navegue até o diretório **compiladorest1** com o Prompt de Comando.
2. Execute "**pip install graphviz**" e "**python ui.py**".
3. Insira o código MiniJava+ no campo de texto exibido na interface.
4. Clique no botão "**Gerar Árvore Sintática**".
5. A árvore sintática será salva como uma imagem no mesmo diretório, com o nome **syntax_tree.png**.

4.2. Executando Individualmente Scanner e Parser

Caso deseje executar as partes individuais do scanner e parser, é possível rodar os scripts Python diretamente. Para isso:

1. **Executando o Scanner**:
    - Abra o arquivo **minijava_scanner.py**.
    - Localize a variável **code** na seção **if __name__ == "__main__":** e substitua seu conteúdo pelo código MiniJava+ que deseja analisar.
    - Execute o script:
      python minijava_scanner.py
    - O programa exibirá os tokens gerados no terminal.
2. **Executando o Parser**:
    - Abra o arquivo **minijava_parser.py**.
    - Localize a variável **code** na seção **if __name__ == "__main__":** e substitua seu conteúdo pelo código MiniJava+ que deseja analisar.
    - Execute o script:
      python minijava_parser.py
    - O programa exibirá a árvore sintática no terminal, além de exibir a estrutura da árvore sintática formatada.