Universidade Federal Fluminense
Instituto de Computação
Departamento de Ciência da Computação

Relatório do Trabalho I

Professor: **Luis Antonio Brasil Kowada**

Grupo: **Lucas Silveira Serrano e Matheus Marques Barros**

Niterói - RJ
2024.2

Sumário

1 Introdução

Neste trabalho, abordamos o desenvolvimento de um scanner e de um parser para MiniJava+, fundamentais para o processo de tradução de código fonte. O scanner é responsável por identificar e categorizar os tokens no código-fonte, enquanto o parser interpreta a sequência de tokens, verificando sua correção sintática de acordo com a gramática da linguagem.

A implementação do scanner foi baseada em uma abordagem de análise lexical, que inclui a leitura do código-fonte, identificação de palavras-chave, operadores e delimitadores, e geração de uma tabela de tokens. Para o parser, adotamos uma abordagem top-down, onde consumimos cada símbolo da entrada e construímos uma árvore sintática a partir da raiz até as folhas. Para garantir a correção e a eficiência do parser, foram feitas alterações na gramática original, incluindo a eliminação de recursões à esquerda.

Além disso, a biblioteca `Graphviz` foi utilizada tanto para estruturar a árvore sintática quanto para gerar uma imagem visual que facilita a compreensão da análise sintática realizada.

2 Scanner

A primeira etapa do algoritmo consiste em realizar a análise léxica por meio do Scanner. Assim, a partir de um código em **MiniJava+**, são extraídos e categorizados os tokens para a análise sintática que será vista posteriormente.

2.1 Abordagem do Scanner

Antes de nos aprofundarmos na implementação do Scanner, é importante contextualizar sua lógica. Basicamente, este programa lê um código em **MiniJava+** como uma variável 'string'. Em seguida, são removidas as linhas de comentários, pois estes não são do interesse de um compilador (não são tokens). O conteúdo 'limpo' do código é examinado uma posição por vez. Quando um token é identificado, é criada uma tupla contendo seu tipo e seu valor. Esta tupla é adicionada à lista que será retornada pelo Scanner. A identificação de comentários e tokens é realizada com o auxílio da biblioteca Regex, a qual será explicada posteriormente.

2.2 Entrada e saída

O programa **scanner.py** espera receber como argumento o nome do arquivo contendo o código na linguagem **MiniJava+**. A saída deste programa é uma lista de tuplas contendo o tipo do token e seu valor.

2.3 Implementação do Scanner

Visando melhorar tanto a legibilidade quanto a manutenibilidade do nosso algoritmo, organizamos cada programa em classes com funções específicas para cada tarefa. A função de inicialização (__**init**__) relaciona os tipos dos tokens com suas respectivas expressões regulares, permitindo a identificação por meio da biblioteca Regex. A função **get_tokens**

recebe o conteúdo do código com os comentários já removidos (retorno da função **remove_comments**) e lista cada token na ordem em que aparece, identificando seu tipo por meio do Regex.

2.4 Regex

A biblioteca **re** do Python, também conhecida como Regex (Expressões Regulares), é uma ferramenta poderosa para realizar buscas e manipulações de texto de maneira eficiente. Ela permite a definição de padrões que podem ser usados para encontrar sequências de caracteres em uma string, verificar correspondências e até mesmo substituí-las. A função **re.match** é uma das principais ferramentas utilizadas neste programa para a identificação de tokens. Ela tenta casar um padrão no início de uma string ou em uma parte específica dela. Caso o padrão seja encontrado, **re.match** retorna um objeto de correspondência contendo as informações sobre o trecho da string que corresponde ao padrão. Caso contrário, retorna `None`.

Na função **get_tokens**, o comando **re.match** é utilizado para verificar se um token correspondente pode ser extraído a partir da posição atual da string de entrada.

3 Parser

Após a implementação do scanner, foi gerada uma tabela de tokens durante a análise léxica, a qual será consumida pelo parser na análise sintática para a construção de uma árvore sintática.

3.1 Implementação do Parser

Para uma melhor organização, assim como no caso do scanner, foi desenvolvida a classe `MiniJavaParser`, responsável por consumir cada token da tabela de tokens e verificar sua correção. Além disso, caso todos os tokens sejam validados corretamente, a classe utiliza a biblioteca `Graphviz` tanto para a estruturação da árvore quanto para a geração de um arquivo `tree.png`, que contém uma representação gráfica da árvore sintática.

Primeiramente, foram implementadas funções na classe para lidar com diferentes aspectos da leitura e manipulação da tabela de tokens, incluindo:

- Retornar o token atual sem consumir.
- Consumir o token atual e avançar para o próximo.
- Verificar se o token atual corresponde à categoria e valor.
- Garantir que o token atual é o esperado; caso contrário, gera um erro.

Para lidar com as regras da `EBNF`, foi criada uma função correspondente para cada símbolo não terminal presente na gramática. Vale ressaltar que a `EBNF` original precisou ser ajustada em algumas regras para eliminar recursões à esquerda. Por exemplo, a regra para expressões lógicas foi modificada da seguinte forma:

$$\text{Regra original: } \text{EXP} \rightarrow \text{EXP} \,\&\&\, \text{REXP} \mid \text{REXP}$$

$$\text{Regra atual: } \text{EXP} \rightarrow \text{REXP}\ \text{EXP\_AUX}$$

$$\text{Regra adicional: } \text{EXP\_AUX} \rightarrow \,\&\&\, \text{REXP}\ \text{EXP\_AUX} \mid \varepsilon$$

3.2 Exemplo de saída

Agora, para exemplificar a saída, vamos gerar a árvore sintática correspondente ao seguinte código escrito em MiniJava+:

Código em MiniJava+

```
class Factorial{
    public static void main(String[] a){
        n = 1 + 2 * 3;
    }
}
```

Árvore Sintática

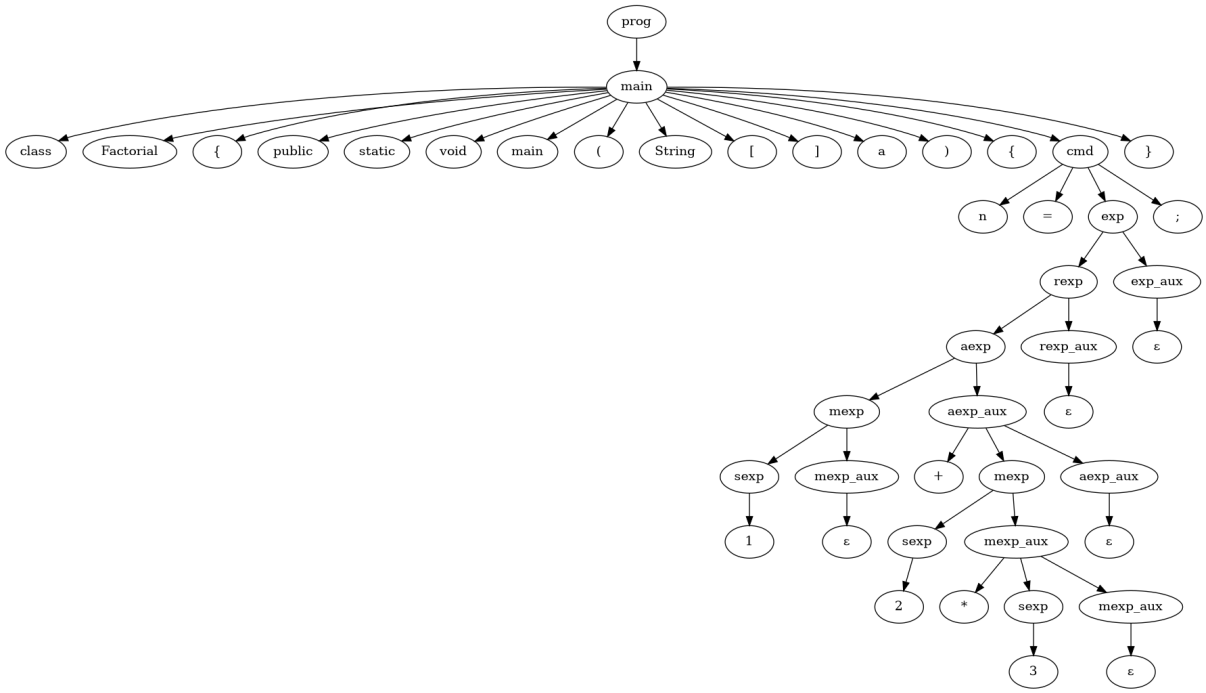

Figura 1: Árvore sintática gerada para o código em MiniJava+.