Analisador Semântico

Nome: *Arthur Henrique Sousa Cruz*
Matrícula: *201611484* ***Turma:*** *10A*

Nome: *Breno Gomes Cardoso*
Matrícula: *201611490* ***Turma:*** *10A*

Nome: *João Pedro Teodoro Silva*
Matrícula: *201610990* ***Turma:*** *10A*

Nome: *Pedro Silveira Lopes*
Matrícula: *201611481* ***Turma:*** *10A*

Prof. Rafael Serapilha Durelli
2018/01

Universidade Federal de Lavras
Departamento de Ciência da Computação

Conteúdo

1 Introdução

A primeira parte deste trabalho consiste em implementar um analisador léxico para a linguagem **j–**. O analisador léxico de uma linguagem tem como objetivo analisar se uma entrada, como um código, por exemplo, está de acordo com as regras impostas pela gramática daquela linguagem e produzir um fluxo de *tokens*.

Caso a entrada contenha algum erro léxico, o analisador deve indicar o erro, sua linha e sua coluna, o que facilita a manutenção de códigos.

A segunda parte deste trabalho consiste em implementar um analisador sintático para a linguagem **j–**. O analisador sintático de uma linguagem tem como objetivo analisar o fluxo de *tokens* gerado pelo analisador léxico para verificar se a sequência de *tokens* é válida de acordo com a linguagem.

Caso a entrada contenha algum erro sintático, o analisador deve indicar o erro, sua linha e sua coluna, o que facilita a manutenção de códigos.

Na terceira parte, foi implementado um analisador semântico para a mesma linguagem. Esse analisador faz checagem do escopo e tipo das variáveis, além de verificar se as operações estão semanticamente corretas.

Assim como nas outras análises, em caso de erro ele será indicado mostrando a linha e a coluna onde ocorreu.

2 Analisador Léxico

2.1 Descrição do Trabalho e Estratégias de Solução

Para implementar o analisador léxico, foi criado um gerador de código "regex-to-code" que, a partir de uma expressão regular, consegue gerar um programa que aceita uma entrada caso ela esteja de acordo com aquela expressão e rejeita caso contrário. O programa gerado analisa de forma recursiva cada caractere de cada lexema, porém sem identificar o fim do lexema. Assim, caso fosse dado como entrada dois lexemas válidos separados por um espaço em branco, o programa não conseguiria identificá-los.

No entanto, para alcançar o objetivo do analisador, o gerador foi modificado para gerar códigos que conseguem separar os lexemas encontrados no arquivo e criar seus respectivos *tokens*. Assim, dando como entrada a expressão regular da linguagem j–, que define as palavras-chave, identificadores, operadores, separadores, etc., foi possível gerar um programa que cria o fluxo de *tokens*, o qual será utilizado pelo analisador sintático.

O programa lê um arquivo e separa suas linhas e, para cada linha, cria os lexemas identificados verificando o tipo de *token* que cada lexema pertence e, caso seja um identificador ou uma constante, o lexema é adicionado em uma tabela de símbolos. Caso o lexema não se enquadre a nenhum tipo de *token* é identificado como erro e reportado ao usuário.

Assim como o código do analisador, foi gerado também um AFD minimizado que determina as palavras aceitas pela linguagem, mostrado na Figura 1. Neste autômato é usado o termo *anything* no alfabeto para representar um caractere qualquer que esteja de acordo com a expressão regular $(ESC \mid \sim (\ \backslash \mid ' \mid LF \mid CR))$, se *anything* for precedido e sucedido por aspas simples, ou $(ESC \mid \sim (\ \backslash \mid " \mid LF \mid CR))$, se *anything* for precedido e sucedido

por aspas duplas, onde ESC são os símbolos de escape \\, \", \', \n, \t, \r, \b, \f.

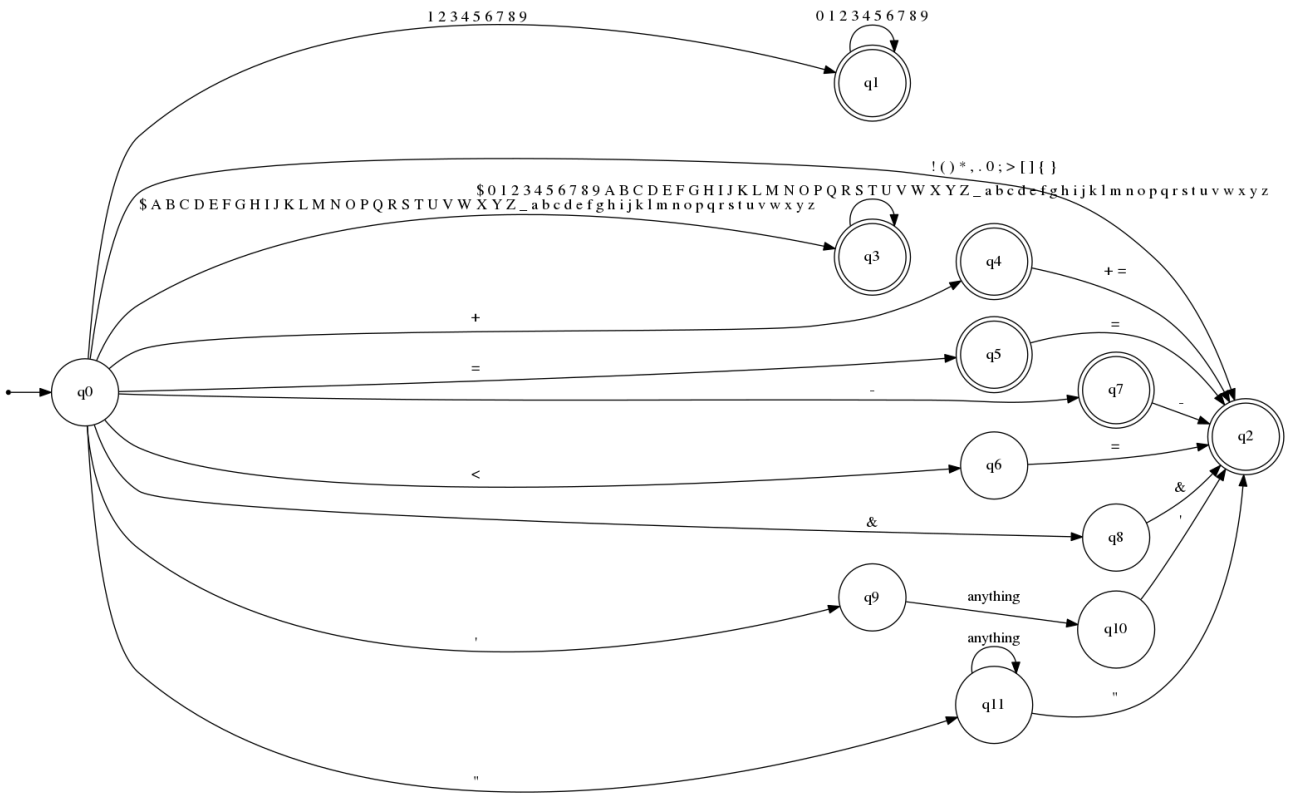


Figura 1: Grafo da linguagem J– gerada pelo *regex-to-code*

2.2 Testes e Resultados

Para verificar a corretude do analisador, foram criados dois testes: Um que gera todos os tipos de *tokens* e outro com 5 erros diferentes.

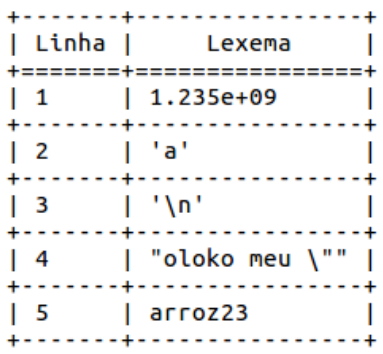
O teste com todos os tipos de *tokens* e sem erros é mostrado abaixo:

```
= == > ++ && <= ! - -- + += * //operadores

, . [] {} () ; //separadores

abstract extends int protected this boolean false //palavras chave
new public true char import null return void
class if package static while else
instanceof private super

1234567890 //int literal
'a' '\n' // char literal
"oloko meu \""

arroz23 //identificador
```

Cujos resultados foram os seguintes:

```
Lexema: =                       Linha:   1   Coluna:   1   Tipo do Token: OpAtribuicao
Lexema: ==                      Linha:   1   Coluna:   3   Tipo do Token: OpIgualdade
Lexema: >                       Linha:   1   Coluna:   6   Tipo do Token: OpMaior
Lexema: ++                      Linha:   1   Coluna:   8   Tipo do Token: OpIncremento
Lexema: &&                      Linha:   1   Coluna:  11   Tipo do Token: OpAnd
Lexema: <=                      Linha:   1   Coluna:  14   Tipo do Token: OpMenorIgual
Lexema: !                       Linha:   1   Coluna:  17   Tipo do Token: OpNot
Lexema: -                       Linha:   1   Coluna:  19   Tipo do Token: OpMenos
Lexema: --                      Linha:   1   Coluna:  21   Tipo do Token: OpDecremento
Lexema: +                       Linha:   1   Coluna:  24   Tipo do Token: OpSoma
Lexema: +=                      Linha:   1   Coluna:  26   Tipo do Token: OpSomaAtribuicao
Lexema: *                       Linha:   1   Coluna:  29   Tipo do Token: OpMultiplicacao
Lexema: ,                       Linha:   3   Coluna:   1   Tipo do Token: SepVirgula
Lexema: .                       Linha:   3   Coluna:   3   Tipo do Token: SepPonto
Lexema: [                       Linha:   3   Coluna:   5   Tipo do Token: SepAbreColchetes
Lexema: ]                       Linha:   3   Coluna:   6   Tipo do Token: SepFechaColchetes
Lexema: {                       Linha:   3   Coluna:   8   Tipo do Token: SepAbreChaves
Lexema: }                       Linha:   3   Coluna:   9   Tipo do Token: SepFechaChaves
Lexema: (                       Linha:   3   Coluna:  11   Tipo do Token: SepAbreParenteses
Lexema: )                       Linha:   3   Coluna:  12   Tipo do Token: SepFechaParenteses
Lexema: ;                       Linha:   3   Coluna:  14   Tipo do Token: SepPontoVirgula
Lexema: abstract                Linha:   5   Coluna:   1   Tipo do Token: PCAbstract
Lexema: extends                 Linha:   5   Coluna:  10   Tipo do Token: PCExtends
Lexema: int                     Linha:   5   Coluna:  18   Tipo do Token: PCInt
Lexema: protected               Linha:   5   Coluna:  22   Tipo do Token: PCProtected
Lexema: this                    Linha:   5   Coluna:  32   Tipo do Token: PCThis
Lexema: boolean                 Linha:   5   Coluna:  37   Tipo do Token: PCBoolean
Lexema: false                   Linha:   5   Coluna:  45   Tipo do Token: PCFalse
Lexema: new                     Linha:   6   Coluna:   1   Tipo do Token: PCNew
Lexema: public                  Linha:   6   Coluna:   5   Tipo do Token: PCPublic
Lexema: true                    Linha:   6   Coluna:  12   Tipo do Token: PCTrue
Lexema: char                    Linha:   6   Coluna:  17   Tipo do Token: PCChar
Lexema: import                  Linha:   6   Coluna:  22   Tipo do Token: PCImport
Lexema: null                    Linha:   6   Coluna:  29   Tipo do Token: PCNull
Lexema: return                  Linha:   6   Coluna:  34   Tipo do Token: PCReturn
Lexema: void                    Linha:   6   Coluna:  41   Tipo do Token: PCVoid
Lexema: class                   Linha:   7   Coluna:   1   Tipo do Token: PCClass
Lexema: if                      Linha:   7   Coluna:   7   Tipo do Token: PCIf
Lexema: package                 Linha:   7   Coluna:  10   Tipo do Token: PCPackage
Lexema: static                  Linha:   7   Coluna:  18   Tipo do Token: PCStatic
Lexema: while                   Linha:   7   Coluna:  25   Tipo do Token: PCWhile
Lexema: else                    Linha:   7   Coluna:  31   Tipo do Token: PCElse
Lexema: instanceof              Linha:   8   Coluna:   1   Tipo do Token: PCInstanceOf
Lexema: private                 Linha:   8   Coluna:  12   Tipo do Token: PCPrivate
Lexema: super                   Linha:   8   Coluna:  20   Tipo do Token: PCSuper
Lexema: 1234567890              Linha:  10   Coluna:   1   Tipo do Token: IntLiteral
Lexema: 'a'                     Linha:  11   Coluna:   1   Tipo do Token: CharLiteral
Lexema: '\n'                    Linha:  11   Coluna:   5   Tipo do Token: CharLiteral
Lexema: "oloko meu \""          Linha:  12   Coluna:   1   Tipo do Token: StringLiteral
Lexema: arroz23                 Linha:  14   Coluna:   1   Tipo do Token: Identificador
```

Figura 2: Resultado Teste 1



Linha	Lexema
1	1.235e+09
2	'a'
3	'\n'
4	"oloko meu \""
5	arroz23

Figura 3: Tabela de Símbolos do Teste 1

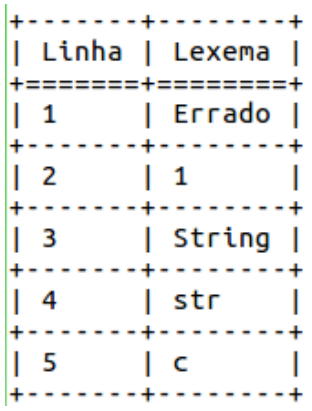
O teste com os erros léxicos é mostrado abaixo:

```
class Errado {
   int @45367 = 000;
   int 12psodk = 1;
   String str = "\";
   char c = ’aspodkapsodk’;
}
```

Cujos resultados foram os seguintes:

```
Lexema: class                   Linha:   2   Coluna:   1   Tipo do Token: PCClass
Lexema: Errado                  Linha:   2   Coluna:   7   Tipo do Token: Identificador
Lexema: {                       Linha:   2   Coluna:  14   Tipo do Token: SepAbreChaves
Lexema: int                     Linha:   3   Coluna:   2   Tipo do Token: PCInt
Lexema: =                       Linha:   3   Coluna:  13   Tipo do Token: OpAtribuicao
Lexema: ;                       Linha:   3   Coluna:  18   Tipo do Token: SepPontoVirgula
Lexema: int                     Linha:   4   Coluna:   2   Tipo do Token: PCInt
Lexema: =                       Linha:   4   Coluna:  14   Tipo do Token: OpAtribuicao
Lexema: 1                       Linha:   4   Coluna:  16   Tipo do Token: IntLiteral
Lexema: ;                       Linha:   4   Coluna:  17   Tipo do Token: SepPontoVirgula
Lexema: String                  Linha:   5   Coluna:   2   Tipo do Token: Identificador
Lexema: str                     Linha:   5   Coluna:   9   Tipo do Token: Identificador
Lexema: =                       Linha:   5   Coluna:  13   Tipo do Token: OpAtribuicao
Lexema: char                    Linha:   6   Coluna:   2   Tipo do Token: PCChar
Lexema: c                       Linha:   6   Coluna:   7   Tipo do Token: Identificador
Lexema: =                       Linha:   6   Coluna:   9   Tipo do Token: OpAtribuicao
Lexema: ;                       Linha:   6   Coluna:  24   Tipo do Token: SepPontoVirgula
Lexema: }                       Linha:   7   Coluna:   1   Tipo do Token: SepFechaChaves
Lexema: @45367                  Linha:   3   Coluna:   6   Erro: Sintaxe inválida
Lexema: 000                     Linha:   3   Coluna:  15   Erro: Sintaxe inválida
Lexema: 12psodk                 Linha:   4   Coluna:   6   Erro: Sintaxe inválida
Lexema: "\";                    Linha:   5   Coluna:  15   Erro: Sintaxe inválida
Lexema: 'aspodkapsodk'          Linha:   6   Coluna:  10   Erro: Sintaxe inválida
```

Figura 4: Resultado Teste 2



Linha	Lexema
1	Errado
2	1
3	String
4	str
5	c

Figura 5: Tabela de Símbolos do Teste 2

3 Analisador Sintático

3.1 Descrição do Trabalho e Estratégias de Solução

Para implementar o analisador sintático, foi utilizada a gramática da linguagem **j–** para criar um *parser* descendente, que a partir de um símbolo inicial, realiza derivações até encontrar o fluxo de *tokens* de entrada. Foi utilizada a abordagem recursiva devido a sua facilidade de implementação, mas com um custo em complexidade. Caso haja erros, o *parser* continua a analisar o código e mostra todos os erros que encontrou no final.

No analisador sintático com descendência recursiva, é gerada uma função para cada símbolo não-terminal da gramática da linguagem e dentro dessa função o *token* é analisado e outras funções são chamadas se necessário.

Existem basicamente dois tipos de erros:

- Quando o analisador espera um *token* específico mas o fluxo de *tokens* acabou.
- Quando o analisador espera um *token* específico mas encontra outro *token*.

O segundo caso pode gerar muitos erros, já que o analisador procura por um *token* e não o encontra ele pode interpretar o resto do código como erros.

O analisador também constrói a árvore sintática para dada entrada. Essa árvore é uma estrutura de dados onde os nós internos são os símbolos não-terminais utilizados na derivação e os nós folhas são os símbolos terminais (*tokens*). Cada vez que uma função é chamada, o símbolo não-terminal que a representa é adicionado na árvore. Analogamente, quando um *token* é derivado, ele também é inserido.

3.2 Testes e Resultados

Para verificar a corretude do analisador, foram criados três testes: Um para mostrar a árvore sintática, um com dez erros diferentes e outro correto com uma grande variedade de *tokens*.

O teste da árvore sintática é mostrado abaixo:

```
package a.a;
import b;
public class arroz {
}
```

Ele gerou a seguinte árvore:

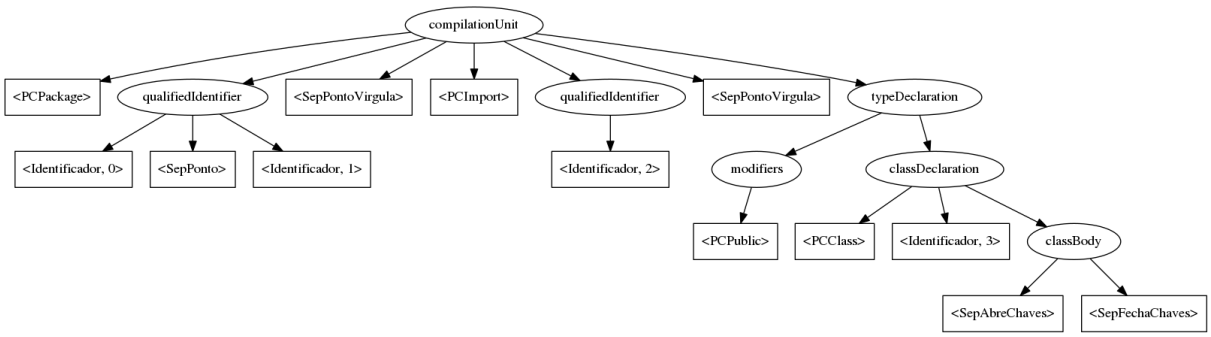


Figura 6: Árvore sintática gerada pelo analisador.

O segundo teste, que mostra um código com erros sintáticos foi o seguinte:

```
package import; // erro aqui
import java.lang.Integer;
import java.lang.System;

public class Series extends { // erro aqui
    public static int ARITHMETIC = 1;
    public static int GEOMETRIC = 2;
    private int a; // first term
    private int d; // common sum or multiple
    private int n; // number of terms
    public Series() {
        this(1, 1, 10);
    }
    public Series(int a, int d,) { // erro aqui
        this(a) = a;
        this(d) = d;
        this(n) = n;
    }
    public int computeSum(int kind) {
        int sum = a, t = a, i = n;
        while (i-- > 1) {
            if (kind == ARITHMETIC) {
                t += d;
            } else if (kind == GEOMETRIC) {
                t = t * d;
```

```
            }
            sum += t;
        }
        return sum;
    }
    public static void main(String[] args) {
        int a = Integer.parseInt(args[0]);
        int d = Integer.parseInt(args[1]);
        int n = Integer.parseInt(args[2]);
        Series s = new Series(a, d, n);
        System.out.println("Arithmetic sum = "
        + s.computeSum(Series.ARITHMETIC));
        System.out.println("Geometric sum = "
        + s.computeSum(Series.GEOMETRIC));
    }
    public int computeSum(int kind) {
        int sum = a, t = a, i = n;
        while (i > 1) {
            if (kind == ARITHMETIC) {
                t =+= d; //erro aqui
            } else if (kind == GEOMETRIC) {
                t = t * d *; // erro aqui
            }
            sum += t++; // erro aqui
        }
        sum //erro aqui
    }
    public a static void main(String[] args) { //erro aqui
        int a = Integer.parseInt(args[]); //erro aqui
        int d = Integer.parseInt(args[1]);
        int n = Integer.parseInt(args[2]);
        Series s = new Series(a, d, n);
        System.out.println("Arithmetic sum = "
        + s.computeSum(Series.ARITHMETIC));
        System.out.println("Geometric sum = "
        + s.computeSum(Series.GEOMETRIC));
    }
//erro aqui
```

Que gerou a seguinte saída:

```
Esperado o token <Identificador> porm foi encontrado 'import', a.k.a <PCImport>
Linha: 1
Coluna 9
Esperado o token <SepPontoVirgula> porm foi encontrado 'import', a.k.a <PCImport>
Linha: 1
Coluna 9
Esperado o token <Identificador> porm foi encontrado ';', a.k.a <SepPontoVirgula>
Linha: 1
Coluna 15
Esperado o token <Identificador> porm foi encontrado '{', a.k.a <SepAbreChaves>
Linha: 5
Coluna 29
Esperado o token <Identificador> porm foi encontrado ')', a.k.a
    <SepFechaParenteses>
Linha: 14
Coluna 32
Esperado o token <Identificador> porm foi encontrado ')', a.k.a
    <SepFechaParenteses>
Linha: 14
Coluna 32
Esperado o token <Identificador> porm foi encontrado '+=', a.k.a
    <OpSomaAtribuicao>
Linha: 45
Coluna 20
Esperado o token <SepPontoVirgula> porm foi encontrado 'd', a.k.a <Identificador>
Linha: 45
Coluna 23
Esperado o token <Identificador> porm foi encontrado ';', a.k.a <SepPontoVirgula>
Linha: 47
Coluna 28
Esperado o token <SepPontoVirgula> porm foi encontrado '}', a.k.a
    <SepFechaChaves>
Linha: 48
Coluna 13
Esperado o token <SepPontoVirgula> porm foi encontrado '++', a.k.a <OpIncremento>
Linha: 49
Coluna 21
Esperado o token <Identificador> porm foi encontrado ';', a.k.a <SepPontoVirgula>
Linha: 49
Coluna 23
Esperado o token <SepPontoVirgula> porm foi encontrado '}', a.k.a
    <SepFechaChaves>
Linha: 50
Coluna 9
Esperado o token <SepPontoVirgula> porm foi encontrado '}', a.k.a
    <SepFechaChaves>
```

```
Linha: 52
Coluna 5
Esperado o token <SepAbreParenteses> porm foi encontrado 'static', a.k.a
    <PCStatic>
Linha: 53
Coluna 14
Esperado o token <SepAbreChaves> porm foi encontrado 'static', a.k.a <PCStatic>
Linha: 53
Coluna 14
Esperado o token <Identificador> porm foi encontrado ']', a.k.a
    <SepFechaColchetes>
Linha: 54
Coluna 39
Esperado o token <SepFechaColchetes> porm foi encontrado ')', a.k.a
    <SepFechaParenteses>
Linha: 54
Coluna 40
Esperado o token <SepFechaChaves> antes do fim do arquivo.
```

O último teste, que mostra uma grande variedade de *tokens* é o teste abaixo:

```
package ufla;
import sil;

public abstract class Aluno extends Cuba{
    private int i = 0;
    protected static void foo(int a, String b, Sil d){
        int a = 0;
        char a = 1;
        boolean a = true;
        boolean[][][][] a = new boolean[]{a = s};
        boolean b = new int[i + 1];
        this(i) = 0;
        a = null;

        if(a == b > ++i && false <= !(true) && -2 > 3-- + "abc" += 4 * 3) {
            while(a == 'b') {
                return a;
            }

        } else {
            super.get(a, b, c);
        }
    }
    public static void main(String args) {
        System.out.println("YEAH, YEAH!!");
    }
}
```

A árvore sintática gerada por esse teste foi omitida desse relatório por ser muito grande.

4 Analisador Semântico

4.1 Descrição do Trabalho e Estratégias de Solução

Para implementar o analisador semântico, foi criada uma tabela *hash* auxiliar onde foram armazenados os dados do escopo e do tipo das variáveis. Com o auxílio dessa tabela, o programa percorre o fluxo de *tokens* verificando se o escopo do identificador é válido e se as operações são permitidas para seu tipo.

A chave para a tabela *hash* é o próprio lexema da variável junto ao identificador do escopo (um valor inteiro positivo). Para que uma variável seja considerada existente no escopo, deve existir uma instância na tabela auxiliar cuja chave é igual ao lexema da variável combinado com o identificador do seu escopo ou de algum escopo inferior.

As verificações de tipo foram limitadas para alguns casos selecionados, sendo eles operações aritméticas, de atribuição e *booleanas*, para facilitar a implementação do analisador. Também

foram colocadas algumas restrições de tipos para as operações:

- Operações aritméticas só podem ser feitas com inteiros.
- Operações de ordem só podem ser feitas com caracteres e inteiros
- Operações de equivalência só podem ser feitas com valores do mesmo tipo.
- Operações de atribuição só podem ocorrer entre uma variável e uma expressão que resulta em um valor do mesmo tipo.

Após a análise semântica foi feito o gerador de código de três endereços para operações de atribuição e aritméticas. Para simplificar, apenas as instruções que estiverem na função *main* são traduzidas para código de 3 endereços e estruturas complexas como estruturas de repetição e condição não são traduzidas para código intermediário.

Para realizar essa tradução, foi usada uma estrutura de árvore sintática como a presente no livro do dragão, como mostra a figura 7. Com essa estrutura, basta percorrer a árvore do nível mais baixo (operações que devem ser realizadas primeiro) ao mais alto, armazenando o resultado das operações dos nós intermediários em variáveis temporárias.

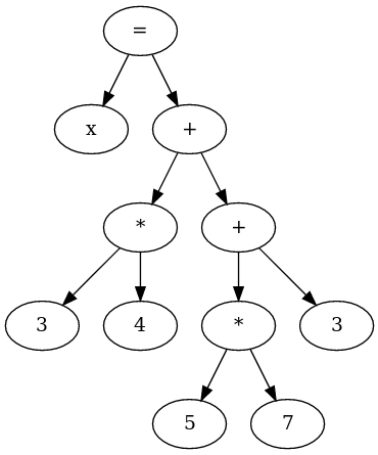


Figura 7: Exemplo de árvore sintáticaseguinte a estrutura mostrada no livro do dragão.

4.2 Testes e Resultados

O código a seguir é um exemplo de teste para geração de código intermediário:

```
public static void main() {
    int x = 3 * 4 + 5 * 7 + 3 ;
}
```

Que gerou o seguinte código:

```
t0 = 5 * 7
t1 = t0 + 3
t2 = 3 * 4
t3 = t2 + t1
x = t3
```

5 Conclusão

A implementação do analisador possibilitou uma melhor compreensão de como um compilador indica os erros léxicos presentes em um programa escrito em uma determinada linguagem e, também, como autômatos têm uma relação profunda com compiladores.

Além disso, foi possível perceber que construir um analisador léxico seria muito trabalhoso e repetitivo sem o "regex-to-code", que também facilitou a criação do fluxo de *tokens* e da tabela de símbolos, elementos importantes para as outras fases da compilação.

No analisador sintático a maior dificuldade foi o "trabalho braçal", já que não foi utilizado nenhum gerador de código para criá-lo e a gramática da linguagem é muito grande, resultando em um programa com mais de 1000 linhas de código. Apesar disso, a semelhança entre as funções facilitou a reutilização de código.

No analisador semântico pôde-se notar a amplitude de problemas semânticos que devem ser tratados nos compiladores e quão difícil se torna pensar em todos os possíveis casos de erros que necessitam ser encontrados e mostrados ao programador.

6 Bibliografia

Referências

[1] Slides disponibilizados pelo professor no Campus Virtual

[2] AHO, Alfred V. et al. Compiladores: princípios, técnicas e ferramentas . 2. ed. São Paulo, SP: Pearson Addison Wesley, c2008. x, 634 p. ISBN 9788588639249.