Relatório Final de Compiladores

Vinícius Takeo Friedrich Kuwaki

2 de Outubro de 2020

1 Introdução

Esse relatório busca esclarecer o que se foi feito durante o trabalho final da disciplina de Compiladores. As ferramentas utilizadas foram: Flex, para a análise léxica, Bizon, para a análise sintática, Jasmin para a geração de código intermediário e DrawIO para a geração dos diagramas apresentados nesse relatório.

O compilador implementou apenas o básico dos requisitos impostos a ele. A análise léxica foi totalmente concluída. Já com relação a análise sintática, o compilador pecou em alguns pontos, por exemplo, a inclusão de múltiplas funções, o que será discutido mais a frente. Também não foi implementado uma tabela de símbolos e nem recursos para uma análise semântica robusta. A forma como os problemas descritos foram contornados será apresentada nas seções seguintes. Porém é suficiente dizer que o compilador implementou o básico: variáveis inteira funcionais para cálculos, operação de adição (as demais eram apenas seguir a mesma lógica e mudar os operandos no Jasmin), laços de seleção e repetição e funções. Arrays e matrizes, assim como strings não foram implementadas. A seguir serão explicados detalhes da implementação e execução dos exemplos deixados pelo autor.

2 Análise Léxica

A tabela 1 apresenta os tokens que a linguagem reconhece. Os tokens reconhecidos nessa etapa são jogados para o analisador sintático. Ele por sua vez analisa as regras da gramática e gera uma saída a ser interpretada pelo Jasmin.

Tabela 1: Dicionário Léxico do compilador

	Token	Tipo	Observação
0	main	Palavra-reservada	Parte essencial do compilador, sem ela o analisador sintático aponta erro e suspende a tradução
1	== > <	Operadores comparativos	Apenas os essenciais, o programador tem a capacidade de tratar todos os casos com apenas esses três
2	=	Operador aritmético Operador atribuição	Atribuição de valores a variáveis
3	+ - * /	Operadores aritméticos	Quatro operações primárias, a partir dessas é possível realizar qualquer cálculo matemático
4	int float bool string	Palavras-reservadas	Tipos de dados da linguagem
5	if else	Palavras-reservadas	Referentes a estruturas de seleção
6	for while	Palavras-reservadas	Referentes a estruturas de seleção
7	print read	Palavras-reservadas	Referentes a entradas e saída de dados
8	TRUE FALSE	Palavras-reservadas	Referentes aos tipos de dados booleanos
9	[0-9]*	Digitos	Referentes a tipos de dados inteiros (0 e 1 podem ser atribuídos a booleanos)
10	[0-9]*.[0-9]*	Digitos	Referentes a tipos de dados reais
11	[a-zA-Z]*	Identificadores	Referentes a nomes de variáveis ou a strings
12	Qualquer texto entre aspas duplas	Literais	Referentes a strings, qualquer texto entre aspas duplas tem as suas aspas duplas inicias e finais removidas quando o analisador léxico envia os tokens para o sintático
13	return	Palavra-reservada	Sem ela dentro de uma função (a main inclusa) o analisador sintático aponta erro e suspende a tradução
14	append	Operador	Operador de concatenação

3 Análise Sintática

Utilizando a ferramenta Bizon, já integrada com o Flex, é possível escrever o compilador utilizando linguagem C ou C++. Para realizar a tradução, era necessário utilizar o Assembly Java (Jasmin). Com isso em mente, é necessário se gerar um arquivo externo ao qual é interpretado pela JVM, utilizando da sintaxe do Jasmin. Para isso, criou-se um arquivo modelo **model.j**, do qual o compilador copia todo o conteúdo dele ao arquivo de saída **out.j** que será interpretado pela JVM. A figura 1 apresenta esse esquema

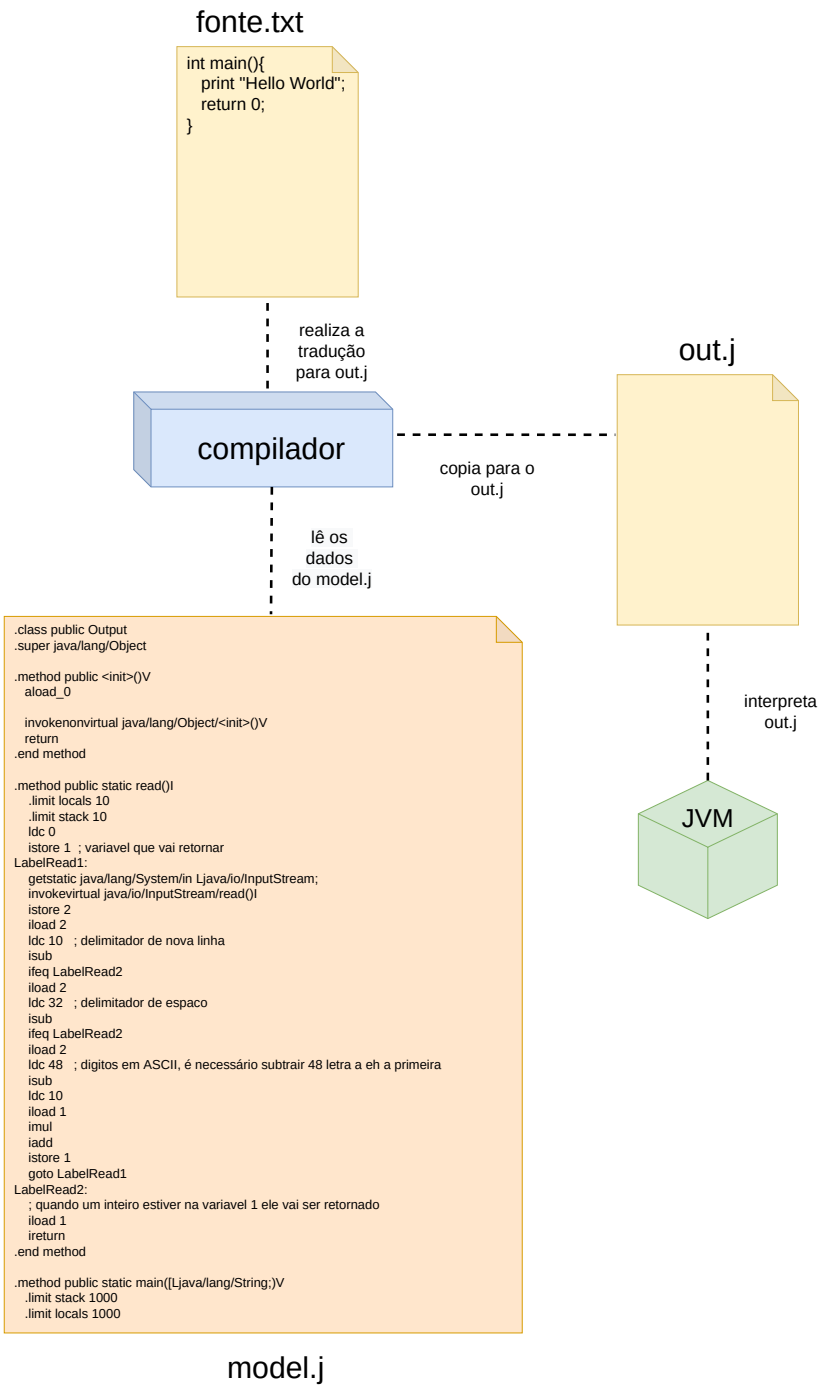

Figura 1: Esquema da tradução e interpretação do código-fonte

O arquivo modelo contém apenas o cabeçalho do método main e mais uma função de entrada de dados. Tal função realiza apenas a leitura de inteiros. Vale a menção aqui, de que devido a questões de tempo, não foi possível implementar todas as funcionalidades do compilador, mas as ideias teóricas serão expostas. Os conceitos foram implementados para variáveis inteiras, logo a função de leitura apenas lê números inteiros.

3.1 Inicio da Árvore Sintática

A partir da regra inicial, o analisador sintático desce até as folhas e depois retorna para a regra inicial. Se houverem regras sintáticas que foram satisfeitas a partir dos tokens dados pelo análisador léxico, a análise sintática é finalizada e um código intermediário é gerado.

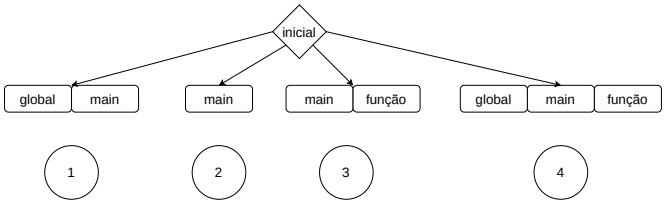

Figura 2: Inicio da árvore sintática.

Para a maioria dos casos de teste que serão apresentados a seguir, a regra utilizada da figura 2 foi a 2. Onde a partir do **inicial** percorre apenas o ramo **main**.

3.2 Hello World

Agora será apresentado o caso de teste mais básico da programação: **Hello World**. Para executar esse caso de teste, entre no diretório **compilador** e execute o comando **make hello**, esse comando vai compilar o código expresso na figura 1, representado por **fonte.txt**, porém nos arquivos submetidos ele se encontra em **casos/helloworld.txt**. Para executar o código, basta executar **make run**.

O compilador adiciona ao arquivo **out.j** (com o conteúdo de model.j já adicionado a ele) o trecho de código traduzido a seguir:

```
    getstatic java/lang/System/out Ljava/io/PrintStream;
    ldc "Hello World"
    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
    return
.end method
```

O exemplo do **Hello World** é o mais trivial de todos, visto que a tradução é quase direta, não são necessários artifícios muito complexos, visto a falta da tabela de símbolos. O compilador apenas empilha o Hello World e após isso invoca o print.

3.3 Entrada de dados e Declaração de Variáveis

Já o exemplo que será comentado agora, o nível de complexidade aumenta um pouco. O exemplo a ser narrado aqui pode ser executado utilizando **make entrada** e **make run**.

```
int main(){

    print "Digite o valor de a:";

    int a;
    read a;
```

```
    print a;

    print "Digite o valor de b:";

    int b;
    read b;
    print b;

    return 0;
}
```

Para o uso de variáveis, como mencionado anteriormente, não há uma tabela de simbolos, porém uma estrutura pobre em formato de lista que armazena o nome das variáveis (apenas inteiras). Para a tradução desse exemplo foi gerado o seguinte código:

```
.method public static main([Ljava/lang/String;)V
    .limit stack 1000
    .limit locals 1000

    getstatic java/lang/System/out Ljava/io/PrintStream;
    ldc "Digite o valor de a:"
    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
    ldc 0
    istore 1
    invokestatic Output.read()I
    istore 1
    iload 1
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(I)V

    getstatic java/lang/System/out Ljava/io/PrintStream;
    ldc "Digite o valor de b:"
    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
    ldc 0
    istore 2
    invokestatic Output.read()I
    istore 2
    iload 2
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(I)V

    return
.end method
```

Cada vez que uma nova variável é declarada, ela é adicionada a lista de variáveis do compilador. Sua posição nessa lista representa o número do seu registrador correspondente no Jasmin. Para a leitura foi utilizada a função read já mencionada anteriormente que é copiada do arquivo **model.j**. Quando o compilador precisa encontrar qual é o número do registrador dessa variável, uma função em C busca na lista de variáveis qual

é a posição dessa variável nessa lista. O compilador então empilha o valor que estava dentro do registrador, usando o comando **iload X**, onde X é o valor retornado pela função em C. Após o valor estar empilhado, basta invocar a função em C que joga no **output.j** os comandos para a impressão.

3.4 Operações Aritméticas

Para as operações aritméticas, também foram considerados apenas inteiros. Também devido a similaridade dos comandos e falta de tempo, apenas operações de soma conseguem ser realizadas, mas como no Jasmin o que muda é o nome do comando, alterar o código desse compilador para contemplar essas operações não é algo nem um pouco complicado. O exemplo a ser narrado a seguir, realiza a soma de dois inteiros.

```
int main(){

    int a = 3;
    int b = 8;

    int c = a + b;
    print c;

    return 0;
}
```

Para tal, nas regras que identificam a declaração dos inteiros, é possível realizar uma operação na hora da declaração, como no caso da linha **int c = a + b**. Também foram incluidos os casos de:

variável + constante
variável + variável
constante + constante
constante + variável

Durante a tradução, primeiro o compilador empilha os valores das constantes ou variáveis e depois realiza a operação entre eles, no caso narrado: **iadd**. Quando o analisador sintático começa a retornar, após ter descido pela árvore, ele então atribui o valor a variável. Ao final, o mesmo trecho de código do print é vinculado ao final, e o resulto é o que se segue:

```
.method public static main([Ljava/lang/String;)V
    .limit stack 1000
    .limit locals 1000

    ldc 0
    istore 1
    ldc 3
    istore 1

    ldc 0
    istore 2
    ldc 8
    istore 2

    ldc 0
```

```
    istore 3
    iload 1
    iload 2
    iadd
    istore 3
    iload 3
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(I)V

    return
.end method
```

Para executar o exemplo basta executar os comandos: **make soma** e **make run**.

3.5 Estrutura de Repetição

Para os comandos de estrutura de repetição, foi necessário, incluir um contador para os labels, já que linguagens de baixo nivel, tal como o Jasmin e o próprio Assembly utilizam a rotulagem para pular trechos de código. Sabendo disso, foi introduzido o contador. Quando o analisador sintático está descendo e percebe que o token **if** foi reconhecido, ele realiza a operação lógica associada ao comando **if**. Veja o exemplo um exemplo a ser narrado (para rodá-lo basta executar **make if** e **make run**).

```
int main(){

    int i = 10;

    if (i<50):
    {
        print "O valor eh menor que 50";

    }else:{
        print "O valor eh maior ou igual a 50 ";
    }

    return 0;
}
```

A parte que nos interessa é o trecho após o **if**. Como a instrução é **i < 50**, o compilador traduz da seguinte forma: o valor da constante é carregado para a pilha, usando o comando **ldc**, após, o valor da variável é carregado também, usando o comando **iload**. Por fim, utiliza-se o comando **isub** para subtrair esses dois valores, o resultado é jogado na pilha do Jasmin. Depois, como no exemplo narrado é destacado uma comparação de menor, é utilizado o comando o **ifge**, que verifica se o valor da pilha é maior ou igual a 0. Caso sim o Jasmin desvia para o rótulo especificado. Caso contrário ele continua, o que corresponde a entrar no **if**. Porém, quando acabam os comandos do **if**, é colocado um comando **goto** para que o comando do else seja evitado. Veja a tradução desse exemplo:

```
.method public static main([Ljava/lang/String;)V
.limit stack 1000
.limit locals 1000
```

```
    ldc 0
    istore 1
    ldc 10
    istore 1

    iload 1
    ldc 50
    isub
    ifge Label10
    getstatic java/lang/System/out Ljava/io/PrintStream;
    ldc "O valor eh menor que 50"
    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
    goto Label11
Label10:
    getstatic java/lang/System/out Ljava/io/PrintStream;
    ldc "O valor eh maior ou igual a 50 "
    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
    Label11:
    return
.end method
```

Para as operações lógicas, apenas dois comandos foram implementados:

identificador <inteiro
identificador == inteiro

Mas, novamente, o conceito básico foi explorado, bastava extender para as outras possibilidades se houvesse tempo. Com relação ao comando **if/else**, a funcionalidade única linha não foi implementada, apenas a utilizando chaves foi. Por exemplo, o trecho de código abaixo não funciona:

```
if(x<3):
    print x;
```

Enquanto que, o comando a seguir funciona:

```
if(x<3):{
    print x
}
```

Também, alguns podem perguntar o porquê do label começar em 10. Mas isso foi um problema enfrentado pela escolha de se usar C. Por algum motivo, ao transformar um inteiro para string e consequentemente jogar no arquivo, o valor 0 não funcionava, mas para qualquer valor acima de 0 era possível. Por isso as variáveis no Jasmin começam em 1, tirando as funções, mas isso sera mencionado mais a frente. Também laços de repetição e seleção começam em 10, para evitar esse tipo de problema.

3.6 Estrutura de Repetição - For

Para o for, foi implementou-se de forma similar ao **if**, já que uma estrutura de repetição é basicamente uma repetição com if. Para rodar o exemplo a ser narrado, utilize **make contador** e **make run**. O trecho de código do exemplo é o seguinte:

```
int main(){
    print "Imprimindo os valores de 1 a 9, com passo 1:";
    for (g:1,10,1):{
        print g;
    }
    return 0;
}
```

Para as estruturas de repetição do tipo for, fora criado um contador global chamado **fors**, que também inicia em 10, tal como o labels citado anteriormente. Para o for, exitem alguns passos importantes. Primeiramente, a variável do for (g no exemplo de código) é criada, tal como feito para uma declaração simples. É adicionada a lista de variáveis, empilha a constante que está após o dois pontos e depois atribui ela ao registrador. Após isso, o rótulo é adicionado, **For** concatenado com o valor do contador **fors**. Após isso, o analisador sintático desce para as funcionalidades do for (o seu escopo) e continua jogando no output a tradução dos comandos. Depois de colocar no output todo o escopo do **for**, o analisador sintático então carrega o valor da variável e adiciona o valor do passo. Após empilha o valor atual do registrador e do limitante do for, realizando a operação de subtração **isub**. Se o valor resultante for menor ou igual a zero, ele desvia para o fim, se não ele volta para o rótulo do for. Veja a tradução a seguir do trecho de código do contador:

```
.method public static main([Ljava/lang/String;)V
    .limit stack 1000
    .limit locals 1000

        getstatic java/lang/System/out Ljava/io/PrintStream;
        ldc "Imprimindo os valores de 1 a 9, com passo 1:"
        invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
        ldc 0
        istore 1
        ldc 1
        istore 1

For10:
        iload 1
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap
        invokevirtual java/io/PrintStream/println(I)V

        iinc 1 1
        ldc 10
        iload 1
        isub
        ifle Fim11
        goto For10
Fim11:
        return
.end method
```

Assim como no **if**, não foi incluido a opção de linha única, embora a análise sintática não aponte erro, nada será traduzido caso o comando seja especificado. Por exemplo, o trecho de código a seguir não irá funcionar:

```
for (g:1,10,1):
```

```
    print g;
```

Enquanto que:

```
for (g:1,10,1):{
    print g;
}
```

Irá funcionar.

3.7 Estrutura de Repetição - While

Já o while, o comando é muito próximo do for, a única diferença é que a comparação é realizada no começo, é isso que diferencia o **while** do **do while**, entretanto o **do while** não estava no escopo dessa linguagem. Para rodar o exemplo a seguir, utilize **make while** e **make run**.

```
int main(){

    int i = 1;
    print "Digite um valor maior que 9:";

    while (i<10):{
        read i;
        print i;
    }

    return 0;

}
```

Para gerar essa diferença, impactando na tradução, primeiramente, ao reconhecer o token **while**, o compilador gera uma saída no output com um rótulo while. Para esse rótulo também foi incluído um contador específico. Após isso, antes de entrar no escopo, a sequência de comandos que se segue é a do if. Empilha o valor de i, o da constante e subtrai-se. Se for maior ou igual vai para fora do while, isto é, vai para um rótulo no final da sequência (com o valor do contador somado a um). Caso contrário, continua no escopo. A seguir, está o trecho de código traduzido:

```
.method public static main([Ljava/lang/String;)V
    .limit stack 1000
    .limit locals 1000

    ldc 0
    istore 1
    ldc 1
    istore 1

    getstatic java/lang/System/out Ljava/io/PrintStream;
    ldc "Digite um valor maior que 9:"
    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
While10:
```

```
    iload 1
    ldc 10
    isub
    ifge Label11
    invokestatic Output.read()I
    istore 1
    iload 1
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(I)V

    goto While10
Label11:
    return
.end method
```

Novamente é necessário ressaltar que o while de um comando só não foi implementado, assim como no if e no for mencionados anteriormente. O while também pode apresentar instabilidades se utilizado dentro de outro while, testes a fundo não foram realizados. Porém, como ele utiliza uma variável para controlar o rótulo, é muito provável que tal evento aconteça. Uma tabela de simbolos ou uma pilha poderia resolver tal problema.

3.8 Funções

A implementação de funções foi a mais complexa e a que o resultado menos agrada. Primeiramente que esse compilador, na forma que a sua gramática foi elaborada permite apenas uma função além da main sendo declarada. Tendo em vista essa limitação imposta pela gramática, e tendo em vista a forma como o Jasmin trabalho com funções, é apresentado o exemplo de uma função de soma:

```
int main(){

    int a;
    int b;

    print "Digite o valor de a:";
    read a;

    print "Digite o valor de b:";
    read b;

    print "Chamando a funcao que computa a soma de a e b...";
    int x = inc(a,b);

    print "O resultado eh:";
    print x;
    return 0;
}

int inc (int a , int b){
    return a+b;
}
```

Levando em consideração que as funções podem ficar em qualquer lugar do código no Jasmin, então, a função é colocada no output no momento de sua declaração. Devido também a falta de um analisador semântico bem definido, ficou a cargo do programador chamar a função pelo menos nome que ela foi declarada, mais isso poderia ser resolvido colocando apenas uma string global para controlar o nome da função, já que só é possível uma no código inteiro.

Seguindo então o exemplo narrado, na linha em que a função é chamada, primeiro o analisador sintático cria a variável **x** e depois desde para o reconhecimento da chamada da função. Como dois parâmetros são passado, e o compilador até tal estágio abrange apenas tipos inteiros, então o tradutor se encarrega de colocar no output a chamada da função pelo seu nome, passando **n** I's, de acordo com a quantidade de parâmetros passados, esses podendo serem variáveis ou constantes. Veja o comando sendo invocado para a função inc:

```
invokestatic Output/inc(II)I
```

Como o tipo de retorno é inteiro, há um I após os parênteses. O valor retornado é então jogado na pilha, e o registrador da variável x então faz o **istore**.

O compilador lidou com as variáveis de escopo da função de forma bem peculiar, devido a falta de uma tabela de símbolos, que deveria ter essa responsabilidade. A forma encontrada foi então, limpar a lista de variáveis, já que como é possível ver na figura 2, após realizada a análise sintática da função **main**, as informações referentes a main não precisam ser mais armazenadas, logo, a lista de variáveis é limpa. Com isso, o processo de compilação dentro da função segue da mesma forma que era feito dentro da main. A seguir, segue o código traduzido do exemplo:

```
.method public static main([Ljava/lang/String;)V
    .limit stack 1000
    .limit locals 1000

        ldc 0
        istore 1
        ldc 0
        istore 2
        getstatic java/lang/System/out Ljava/io/PrintStream;
        ldc "Digite o valor de a:"
        invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
        invokestatic Output.read()I
        istore 1
        getstatic java/lang/System/out Ljava/io/PrintStream;
        ldc "Digite o valor de b:"
        invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
        invokestatic Output.read()I
        istore 2
        getstatic java/lang/System/out Ljava/io/PrintStream;
        ldc "Chamando a funcao que computa a soma de a e b..."
        invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
        ldc 0
        istore 3
        iload 2
        iload 1
        invokestatic Output/inc(II)I
        istore 3
```

```
        getstatic java/lang/System/out Ljava/io/PrintStream;
        ldc "O resultado eh:"
        invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
        iload 3
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap
        invokevirtual java/io/PrintStream/println(I)V

        return
.end method
.method public static inc(II)I
        .limit stack 5
        .limit locals 5
        iload 1
        iload 0
        iadd
        ireturn
.end method
```

Não foram testas as chamadas de funções várias vezes, mas devido a forma de implementação (utilizando um contador para calcular a quantidade I's a serem colocados na chamada) é muito provável que o número de I's siga exponencialmente a cada chamada. Talvez a abordagem para resolver tal problema seja limpar a variável auxiliar após cada chamada.

4 Análise Semântica e Tabela de Símbolos

Assim como discutido anteriormente, uma análise semântica robusta não foi implementada, tal como uma tabela de símbolos também não. Talvez a dificuldade que tenha levado a isso seja o fato da manipulação de estruturas de dados em C (o ambiente utilizado para a construção do compilador) seja muito pobre em estruturas de dados. Para uma boa tabela de símbolos, a utilização de uma estrutura de hash ou lista encadeada seja ideal. Para a análise semântica, analisar detalhes de lógica e auxiliar o programador, não foram muito o foco desse projeto, visto a quantidade de pormenores que faltaram serem resolvidos.

5 Considerações Finais

Embora o trabalho tenha dado uma boa base a respeito da construção de compiladores, o resultado não atingiu as expectativas do autor. Quem sabe para um projeto futuro, o desenvolvimento de um compilador utilizando o C++ seja considerado. A falta de estruturas de dados em C e as questões da dificuldade de se trabalhar com strings em C, dificultam o desenvolvimento de um compilador robusto, mas não o tornam impossível. O ambiente do Jasmin também proporcionou uma tradução simplificada, embora que a questão de tempo foi um empecilho na entrega de um trabalho concluído e 100% funcional.