INSTITUTO FEDERAL
Catarinense
Campus Rio do Sul

COMPILADORES

Algoritmos

Alexandre Bauer

RESUMO

Neste trabalho, serão abordados conceitos básicos sobre compiladores. Também será demonstrado a construção de um compilador simples, onde será aplicado estes conceitos. A entrada será um texto baseado na sintaxe de algoritmos em português estruturado. Para a criação do compilador, será empregada a ferramenta de desenvolvimento para plataformas do .NET Framework, o Visual Studio 2015[1] com a construção do projeto com a utilização da linguagem de programação C#. O resultado do processo, será gerado um código de montagem na arquitetura Assembly MIPS, compatível com o simulador MARS MIPS[2].

INTRODUÇÃO

Compiladores são um grupo de programas, que tem a finalidade de codificar de forma automática, textos escritos em uma linguagem de alto nível (textos-fonte) para uma linguagem estabelecida que permita sua execução (texto-objeto) não necessariamente de baixo nível (José Neto, 2016, p. 18).

A forte motivação que acelerou o desenvolvimento de compiladores, foi a demanda por uma maior eficiência de programação (Ricarte, 2008, p. 1). Durante os anos 50, os compiladores foram apontados como programas expressamente difíceis de escrever, como por exemplo, o Compilador Fortran, consumiu 18 homens-ano para sua implementação (Backus *et al*, 1957).

Deste então, inúmeros procedimentos sistemáticos para tratamento das várias tarefas relacionadas ao processo de compilação, foram descobertas e desenvolvidas linguagens de implementação, ambientes de programação e ferramentas de software (Crespo, 1988, p. 14), que evoluíram para o nível em que hoje, as ferramentas e técnicas existentes permitam a implementação na ordem de 1 pessoa-ano, de um compilador para uma linguagem com atributos similares a do Pascal (Martins, 1994).

Determinados problemas que aparecem na construção de compiladores são problemas abertos, ou seja, que as melhores soluções atuais ainda possuem possibilidades de melhorias, e a construção de um compilador bem-sucedido requer conhecimentos em algoritmos, engenharia e planejamento (Cooper & Torczon, 2014, p. 4). No entanto, pode-se utilizar ferramentas especiais para cada fase do processo de desenvolvimento, reduzindo o esforço do programador, como o Lex que foi desenvolvido primeiramente em Unix, e teve suas variações como o Flex que gera código em C ou C++, e o Jlex ou Jflex que geram código Java (Santos & Langlois, 2018, p. 5).

Outras ferramentas como o LexSint, foi desenvolvida com intuito de apoio ao ensino de linguagens formais e compiladores, suportando a análise léxica e sintática. Tem como principal atributo a geração automática do autômato finito e da gramática livre de contexto, utilizando exemplos da estrutura do código da linguagem alvo. Também proporciona o acompanhamento, em passos intermediários, a montagem das

[1] https://visualstudio.microsoft.com/pt-br/vs/older-downloads/
[2] http://courses.missouristate.edu/KenVollmar/MARS/index.htm

produções e regras gramaticais, melhorando assim o entendimento (Alkmim & Mello, 2010).

Seu módulo gerador de gramáticas, de início irá gerar um arquivo com a estrutura pré formatada para construção das regras gramaticais, onde é possível introduzir os *tokens* e os padrões das estruturas sintáticas da linguagem, e então irá criar automaticamente as regras no formato previsto para a gramática livre de contexto (GLC). Durante a geração da GLC não é verificado se a linguagem é válida, apenas é reproduzido o conjunto de regras a partir dos exemplos, logo, possíveis erros produzidos na construção dos exemplos, serão transferidos para a GLC (Alkmim & Mello, 2010).

LINGUAGENS

As linguagens formais são utilizadas para expressar formalmente uma linguagem computacional. É utilizado uma abordagem teórica da sintaxe, que verifica as instruções do programa, e da semântica que verifica erros do programa em lógicas (Villanueva, 2008).

A sintaxe manipula símbolos, sem considerar os seus significados, mas para resolver um problema real, é necessário dar uma interpretação semântica aos símbolos, atribuindo valores a estes (número inteiro ou real, por exemplo) (Villanueva, 2008).

A análise léxica é um tipo especial da análise sintática, e está centrada nos componentes básicos da linguagem, sendo a forma de verificar determinado alfabeto. Quando se analisar uma palavra, pode-se definir através da análise léxica se existe ou não algum caractere que não faz parte do alfabeto (Villanueva, 2008).

O alfabeto é um conjunto não vazio de símbolos, representado pela letra grega sigma Σ. Um exemplo seria Σ = {α, β, λ}, sendo o alfabeto formado pelo conjunto de símbolos α, β e λ. Logo, a combinação dos símbolos de um alfabeto é denominado palavra (Villanueva, 2008).

A palavra sobre o alfabeto Σ é qualquer combinação de um número finito de símbolos de Σ, com ou sem repetição destes, como por exemplo, para o alfabeto Σ = {a, b}, são palavras aa, ab, bb, aabb, entre outras. O comprimento de uma palavra s, representada por |s|, é igual ao número de símbolos que o compõe, por exemplo a palavra aabb tem comprimento igual a 4 (|aabb| = 4) (Villanueva, 2008).

O conceito de cadeia vazia é importante na teoria das linguagens formais. Denota-se por ε a cadeia formada por uma quantidade nula de símbolos, ou seja, a cadeia que não contém nenhum símbolo, sendo formalmente |ε| = 0 (Ramos, 2008).

Uma linguagem é o conjunto de palavras formado com os símbolos de um alfabeto Σ finito e não vazio. Dado um alfabeto Σ, define-se L como uma linguagem sobre Σ se e somente se L ⊄ Σ*, sendo Σ* o conjunto formado por todas as palavras formadas com os símbolos do alfabeto Σ (Ramos, 2008).

As linguagens regulares podem ser expressas por “expressões regulares” que são sequências de símbolos obtida pela aplicação repetitiva de um conjunto de regras de formação, como por exemplo, σ(n) e s(n) representam a repetição do símbolo σ e da palavra s por “n” vezes (Villanueva, 2008).

O reconhecedor recebe uma palavra de entrada contida em uma linguagem, e dada uma linguagem deve-se indicar se a palavra é aceita ou rejeitada. Esta notação permite escrever representações finitas para linguagens formadas por um número

infinito de palavras (Villanueva, 2008).

AUTÔMATOS

Um autômato é um modelo matemático de máquinas, com entradas e saídas discretas, que reconhece um conjunto de palavras sobre um dado alfabeto. O autômato pode ser finito ou infinito, e determinístico ou não-determinístico (Villanueva, 2008).

Um autômato finito consiste de um conjunto finito de estados definido por q, e um conjunto de transições de estados definido por ʒ, que ocorrem a partir de símbolos de entrada escolhidos sobre um alfabeto Σ. O estado inicial do autômato definido por q0 é o estado que se encontra antes de ler o primeiro símbolo. Alguns estados do autômato também são definidos como estados finais ou qf (Villanueva, 2008).

Um autômato reconhece exatamente as palavras a serem processadas, que o levam de um estado para outro. Caso o autômato possua infinitos estados é definido como autômato infinito. Se para cada símbolo de entrada existe exatamente uma transição de saída de cada estado, então o autômato é determinístico, e se existirem duas ou mais transições de saída de um estado que ocorrem para um mesmo símbolo, então o autômato é definido como não determinístico (Villanueva, 2008).

Um autômato finito determinístico é formado por uma quíntupla, A = (q, Σ, ʒ, q0, qf), sendo q o conjunto finito de estados do autômato, Σ o alfabeto ou conjunto de símbolos σ, ʒ é a função de transição de estados, q0 é o estado inicial, qf é o conjunto de estados finais (Villanueva, 2008).

Graficamente um autômato é representado por um grafo direcionado, cujas vértices representam os estados e os arcos representam a função de transição. O estado inicial é identificado por uma seta, e os estados finais são representados por vértices com linhas duplas. Conhecida a função de transição e o estado atual de um autômato, é possível determinar seu estado após processar um dado símbolo (Villanueva, 2008).

Por exemplo, suponha-se um autômato que reconhece a palavra w = aabb, sendo este dado por A = ({q0, q1, q2, q3}, {a, b}, { ʒ(q0, a) = (q1, a, R), ʒ(q1, a) = q2, ʒ(q2, b) = q3, ʒ(q3, b) = q4 }, q0, {q4}). Quando no estado inicial q0 encontrar o símbolo ‘a’ avança para o estado q1, quando no estado q1 encontrar o símbolo ‘a’ avança para o estado q2, quando no estado q2 encontrar o símbolo ‘b’ avança para o estado q3, quando no estado q3 encontrar o símbolo ‘b’ avança para o estado q4 e finaliza o reconhecimento da palavra. Na figura 1 pode ser observado o grafo do autômato.

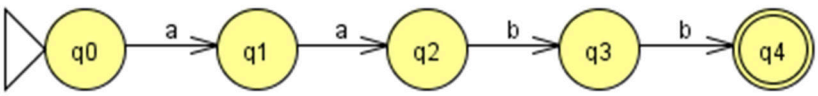

Figura 1: Exemplo de autômato (Fonte: do autor).

O autômato reconhece uma determinada linguagem através da leitura sequencial de símbolos, mudando de estado a cada leitura. A palavra é dita reconhecida se o estado alcançado após a leitura do último símbolo da palavra pertence ao conjunto de estados finais. Cada símbolo lido atualiza o estado do autômato de acordo com uma função de transição (Villanueva, 2008).

ANÁLISE SINTÁTICA

A sintaxe é a parte da gramática que estuda a disposição das palavras na frase e das frases em um discurso. A análise sintática deve reconhecer se uma frase está escrita corretamente (Sethi et al, 2008).

No processo de compilação de um programa, é nessa etapa que o processo de compilação deve reconhecer as formas do programa fonte e determinar se ele é valido ou não (Sethi et al, 2008).

Esse modelo pode ser definido utilizando gramáticas livres de contexto que representam uma gramática formal e pode ser escrita através de algoritmos fazem a derivação de todas as possíveis construções da linguagem (Sethi et al, 2008).

As derivações tem como objetivo determinar se um fluxo de palavras se encaixa na sintaxe da linguagem de programação (Sethi et al, 2008).

Alguns termos utilizados na definição de linguagens de programação (Sethi et al, 2008

- Símbolo: são os elementos mínimos que compõe uma linguagem. Na linguagem humana são as letras.
- Sentença: É um conjunto ordenado de símbolos que forma uma cadeia ou texto. Na linguagem humana são as palavras.
- Alfabeto: É um conjunto de símbolos. Na linguagem humana é o conjunto de letras {a, b, c, d, ...}
- Linguagem: É o conjunto de sentenças. Na linguagem humana são os conjuntos de palavras {compiladores, linguagem, ...}
- Gramática: É uma forma de representar as regras para formação de uma linguagem.

Trazendo esse conceito para linguagem de programação temos (Sethi et al, 2008):

- Alfabeto: {w, h, i, l, e, +, 1, 2, 3}
- Símbolos: 1, 5, +, w
- Sentença: while, 123, +1
- Linguagem: {while, 123, +1}

Dada uma gramática G e uma sentença s o objetivo do analisador sintático é verificar se a sentença s pertence a linguagem G. O analisador sintático recebe do analisador léxico a sequência de tokens que constitui a sentença s e produz uma arvore de derivação se a sentença é válida ou emite um erro se a sentença é inválida (Sethi *et al*, 2008).

O analisador léxico é desenvolvido para reconhecer cadeias de caracteres fazendo uma leitura dos caracteres e obtendo a sequência de tokens, esse analisador vê o texto como uma sequência de palavras de uma linguagem regular e reconhece ele através de autômatos finitos determinísticos e não determinísticos (Torczon & Keith, 2014).

Já o analisador sintático vê o mesmo texto como uma sequência de sentenças que deve satisfazer as regras gramaticais. É através da gramática que podemos validar expressões criadas na linguagem de programação (Torczon & Keith, 2014).

A derivação é a substituição do conjunto de símbolos não terminais por símbolos terminais começando pelo símbolo inicial, ao final desse processo o resultado é a forma como a linguagem deve assumir (Torczon & Keith, 2014).

Durante a derivação devemos aplicar as regras de produção para substituir cada símbolo não terminal por um símbolo terminal, isso permite identificar se certas cadeias de caracteres pertence a linguagem, as regras expandem todas as produções possíveis. Como resultado desse processo temos a árvore de derivação (Torczon & Keith, 2014).

A árvore de derivação é uma estrutura em formato de árvore que representa a derivação de uma sentença ou conjunto de sentenças, essa estrutura irá gerar as árvores de análise sintática que representam o programa fonte, sendo essa estrutura muito utilizada nas etapas de compilação (Torczon & Keith, 2014).

ANÁLISE SEMÂNTICA

A análise semântica é responsável por verificar aspectos relacionados ao significado das instruções, essa é a terceira etapa do processo de compilação e nesse momento ocorre a validação de uma serie regras que não podem ser verificadas nas etapas anteriores (Sethi et al, 2008).

Não é possível representar com expressões regulares ou com uma gramática livre de contexto regras como, por exemplo, que todo identificador deve ser declarado antes de ser usado. Muitas verificações devem ser realizadas com meta-informações e com elementos que estão presentes em vários pontos do código fonte, distantes uns dos outros. O analisador semântico utiliza a árvore sintática e a tabela de símbolos para fazer as analise semântica (Torczon & Keith, 2014).

As validações que não podem ser executadas pelas etapas anteriores devem ser executadas durante a análise semântica a fim de garantir que o programa fonte esteja coerente e o mesmo possa ser convertido para linguagem de máquina (Torczon & Keith, 2014).

A análise semântica percorre a árvore sintática relaciona os identificadores com seus dependentes de acordo com a estrutura hierárquica (Torczon & Keith, 2014).

Essa etapa também captura informações sobre o programa fonte para que as fases subsequentes possam gerar o código objeto, onde um importante componente da análise semântica é a verificação de tipos, nela o compilador verifica se cada operador recebe os operandos permitidos e especificados na linguagem fonte (Torczon & Keith, 2014).

ARQUITETURA MIPS

A arquitetura MIPS (*microprocessor without Interlocked Pipelined Stages*, em português, microprocessador sem estágios de pipeline interligados), foi projetado sobre a arquitetura de processadores RISC (*reduced instruction set computer*, em português, computador com conjunto de instruções reduzido). Teve seu desenvolvimento iniciado em 1984, e lançado em 1985, sendo que até o final dos anos 80, várias empresas de estações de trabalho e servidores estavam utilizando essa arquitetura (Sweetman, 1999).

No início da década de 90 foi lançado o R4000, sendo este o primeiro microprocessador de 64 bits. Já na metade da década de 90, foi lançado o R10000, sendo na época de seu lançamento, considerado a CPU com arquitetura mais sofisticada já construído (Sweetman, 1999).

Quase no fim dos anos 90, a arquitetura MIPS liderava o mercado de

processadores embarcados, sendo mais de 100 milhões utilizados em dispositivos como videogames, impressoras laser, roteadores de rede, entre outros. Com isso, ultrapassou a quantia de processadores embarcados em mais de 1000 para 1, em relação a computadores de uso geral. E com esse crescimento da arquitetura MIPS em dispositivos embarcados, resultou na criação da MIPS Technologies (da Silicon Graphics) como uma empresa independente em 1998 (Sweetman, 1999).

Todas as instruções possuem comprimento de 32 bits, o que impossibilita integrar uma constante de 32 bits em uma única instrução (não haveria bits suficientes na instrução para codificar a operação e o registrador de destino), com isso, os arquitetos do MIPS determinaram um espaço de uma constante de 26 bits, para codificar o endereço de destino de um salto ou salto para sub-rotina, porém a maioria dos campos de constantes tem 16 bis, sendo necessário executar duas instruções para carregar um valor arbitrário de 32 bits, e os desvios condicionais são limitados a um intervalo de 64K instruções (Sweetman, 1999).

SIMULADOR MARS

O simulador MARS (*MIPS Assembler and Runtime Simulator*, em português, montador MIPS e simulador de tempo de execução) é uma combinação de editor de linguagem assembly, montador, simulador e depurador para o processador MIPS. Foi desenvolvido por Pete Sanderson e Kenneth Vollmar na Missouri State University (Vollmar & Sanderson, 2006).

Foi desenvolvido com intuito de ser utilizado no meio educacional, como ferramenta de aprendizagem, tentando suprir as necessidades dos alunos de graduação e instrutores, sendo útil para cursos como organização e arquitetura de computadores, programação em linguagem assembly e escrita de compiladores (Vollmar & Sanderson, 2006).

Possui uma interface de usuário baseada em Java[3], com controles e visualizações diversas, como por exemplo, uma barra deslizante para controlar a velocidade de execução, podendo ser passo único ou velocidade variável; guias selecionáveis para a visualização dos 32 registradores; opção de visualização dos valores em decimal ou hexadecimal; e inclusive um editor e montador integrado (Vollmar & Sanderson, 2006).

RESULTADOS OBTIDOS

Para desenvolvimento, foi escolhido como entrada um texto baseado na sintaxe de algoritmos em português estruturado, limitando os comandos disponíveis da gramática, como por exemplo, não será implementado registros, e os parâmetros dos procedimentos e funções não irão aceitar constantes e operações como argumentos, apenas variáveis serão aceitas.

Na figura 2 é apresentado o diagrama do processo executado pelo compilador desenvolvido, sendo a entrada é o código fonte na linguagem de algoritmos, que passará pelo processo de análise e tratamento de erros, onde o analisador léxico irá tentar gerar uma lista de tokens representando o código fonte, e se não encontrar nenhum erro, o analisador sintático irá processar essa lista, validando conforme a gramática definida para o analisador sintático ascendente SLR, que se não encontrar

[3] https://www.java.com/pt-BR/

nenhum erro, irá gerar uma lista representando uma árvore sintática, sendo então efetuada a análise semântica, que irá validar as variáveis e constantes utilizadas nos comandos, verificando tipos de dados, escopo e nomes repetidos de variáveis, e ao mesmo tempo, se não encontrar nenhum erro, irá montar a estrutura de objetos que representará a árvore sintática do código fonte.

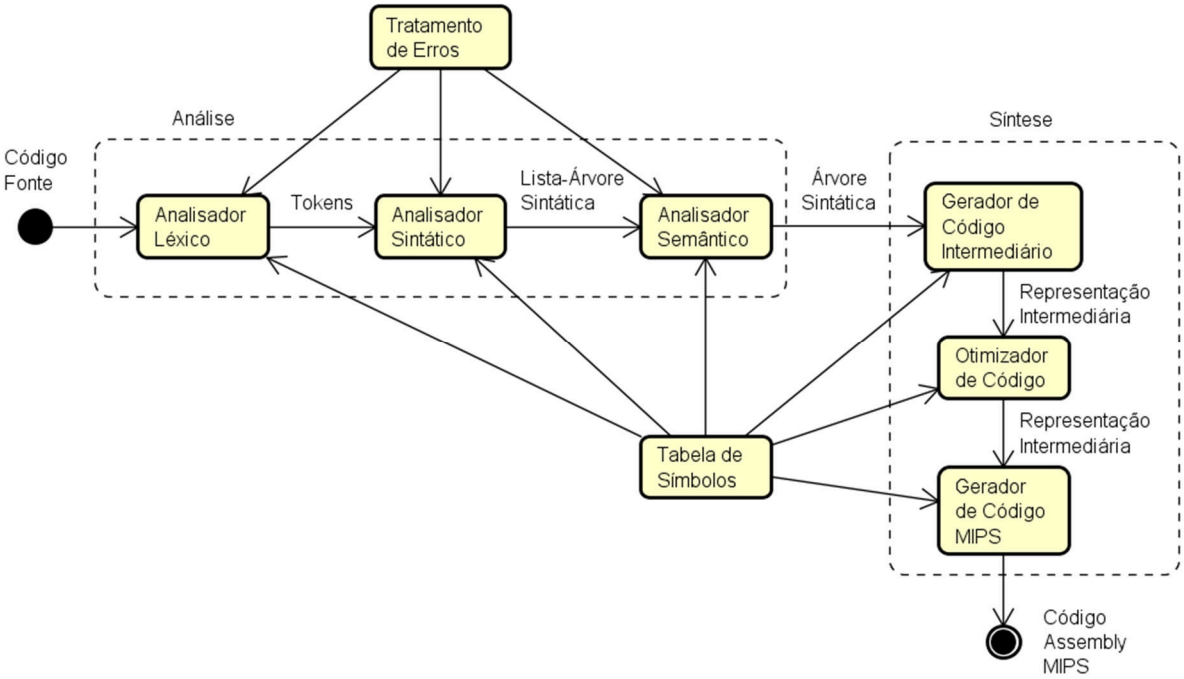

Figura 2: Diagrama processo (Fonte: do autor).

Ainda na figura 2, pode ser visto o processo de síntese da árvore sintática, que irá gerar o código intermediário, utilizando apenas o código fonte necessário, ou seja, variáveis, procedimentos ou funções que não são utilizados, não serão incluídos, já que como não são utilizados, irão apenas consumir memória do sistema. O resultado passará por algumas otimizações de memória, para então gerar o código de montagem MIPS.

Na listagem 1, está exemplificado a estrutura do texto de entrada para o algoritmo, sendo, na linha 1 a palavra reservada ‘algoritmo’ seguido de um texto representando o nome do algoritmo envolvido por aspas duplas; na linha 2 a palavra reservada ‘variáveis’, e na linha 3 as declarações das variáveis globais, composto pelo nome da variável separado pelo símbolo dois pontos e seguido pelo tipo da variável, podendo ter mais de uma variável declarada na mesma linha, deste que os nomes sejam separados por vírgula, e também mais tipos de variáveis podem ser declaradas nas linhas seguintes seguindo as mesmas regras; nas linhas 6-7, 13-14 e 20-21, são aplicadas as mesmas regras detalhadas para as linhas 2-3, exceto que essas variáveis serão locais, e acessíveis apenas dentro do escopo à que forem declaradas. Também vale ressaltar que, as declarações de variáveis são opcionais, podendo ou não existir nos pontos definidos.

A declaração de procedimentos pode ser vista na linha 5, sendo iniciado pela palavra reservada ‘procedimento’, seguida pelo nome do procedimento, e entre parênteses as declarações dos parâmetros de entrada, valendo para estes as

mesmas regras usadas para declaração de variáveis, exceto que se for necessário declarar mais tipos, deverão serem separados pelo símbolo ponto e vírgula. Já a declaração de funções segue as mesmas regras utilizadas para procedimentos, exceto que deve iniciar com a palavra reservada 'função', e após o fechamento de parênteses, deverá ser seguido pelo símbolo dois pontos e o tipo de retorno da função.

```
1  algoritmo "nome algoritmo"
2  variáveis
3     declaração_variáveis_globais: tipo_variável;
4
5  procedimento nome_procedimento(declaração_parâmetros: tipo_parâmetro)
6  variáveis
7     declaração_variáveis_locais: tipo_variável;
8  inicio
9     comandos;
10 fim;
11
12 função nome_função(declaração_parâmetros: tipo_parâmetro): tipo_retorno
13 variáveis
14    declaração_variáveis_locais: tipo_variável;
15 inicio
16    comandos;
17    retorne valor_retorno;
18 fim;
19
20 variáveis
21    declaração_variáveis_locais: tipo_variável;
22 inicio
23    comandos;
24 fim.
```

Listagem 1: Estrutura Algoritmo (Fonte: do autor).

Ainda na listagem 1, as palavras reservadas 'início' e 'fim' demarcam a região onde serão inseridos os comandos do programa, sendo estes as entradas e saídas de usuário, operações aritméticas e lógicas, comandos condicionais, e laços de repetição. Ainda vale ressaltar, que o procedimento principal do programa, não possui explicitamente um nome ou palavra reservada, ele deve ser colocado ao final do programa, após a declaração de variáveis, procedimentos e funções, como pode ser visto entre as linhas 20 e 24.

A análise léxica é executada por um autômato finito determinístico (AFD), que irá percorrer o texto do programa fonte, identificando e criando uma lista com os tokens encontrados. Cada token encontrado será armazenado em uma classe, juntamente com as informações do lexema que representa o token, a linha onde foi encontrado, e o índice inicial absoluto iniciado em zero, da posição do primeiro caracter do lexema no texto.

Na figura 3, pode ser observado um fragmento do diagrama do autômato finito determinístico da análise léxica. O estado q0 é o estado inicial do autômato, o estado q1 é o estado final para identificadores (como variáveis e nomes de procedimentos e funções) e seu token é representado pelo valor 'ID'; Se em q0 e encontrar um espaço em branco, vai para q2 que é um estado final, e representa o token 'ESPACO'; Se em q0 encontrar uma aspa dupla, vai para q3 e permanece neste até encontrar outra aspas dupla, sendo encontrada uma constantes de texto, e representada pelo token 'CONST_TEXTO'; Se em q0 e encontrar um número, vai para o estado final q5 e permanece enquanto ser seguido por mais números, e se encontrar um caracter diferente de um ponto irá representar um valor numérico constante inteiro pelo token

‘CONST_INT’, ou se encontrar um ponto irá para q6, que será esperado um valor numérico que poderá ser seguido por mais números, indo para o estado final q7 e representado uma constante numérica real (ponto flutuante) pelo token ‘CONST_REAL’, ou se em q6 encontrar um ponto irá para q8, que irá esperar um valor numérico indo para o estado final q9, que poderá ser seguido por mais números, representando uma faixa de índices de vetores, representado pelo token ‘CONST_FAIXA_VET’.

Se em q0 encontrar o símbolo de maior (>), vai para o estado final q24 representando o token ‘MAIOR’, mas se seguido pelo símbolo de igual (=) irá para o estado final q25, representando então o token ‘MAIOR_IGUAL’. Se em q0 encontrar o caracter ‘o’ irá para o estado q172, onde se encontrar qualquer letra ou número, exceto a letra ‘u’, irá para o estado final q1 (ID), e se encontrar a letra ‘u’ vai para o estado final q173, identificando o operador lógico ‘ou’, que irá representado pelo token ‘OU’, exceto se seguido por letra ou número, onde então irá para o estado q1.

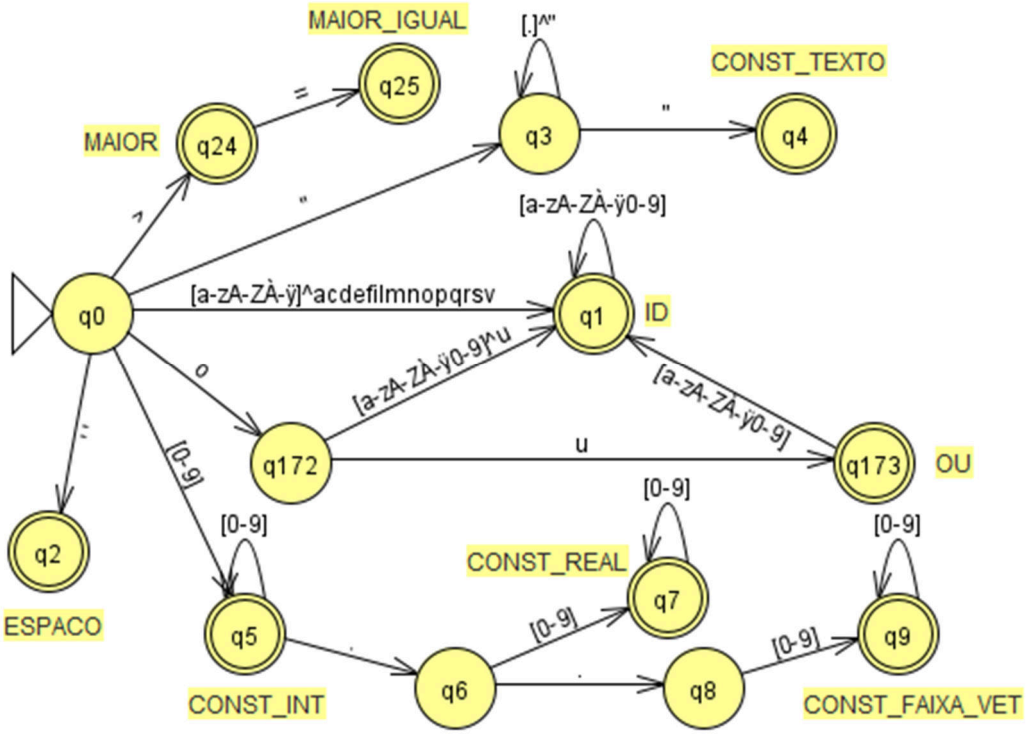

Figura 3: Diagrama AFD Parcial (Fonte: do autor).

Para a análise sintática, foi desenvolvido um analisador sintático ascendente SLR, que irá identificar a gramática especificada para o texto fonte do algoritmo. Para isso, foi desenvolvido uma classe estática que irá receber o analisador léxico e obter os tokens deste, verificando se estão de acordo com a gramática definida.

Na listagem 2 pode ser observado um fragmento da gramática (linhas 1 à 3) e das reduções (linhas 5 à 8), estando destacado o comando ‘leia’ para entrada do usuário. O comando inicia com a palavra reservada ‘leia’ (token ‘LEIA’), seguida pelo símbolo de abertura de parênteses (token ‘ABRE_PAR), pelo terminal <LISTA_CMD_LEIA_VAR>, pelo símbolo fecha parênteses (token ‘FECHA_PAR’), e pelo símbolo ponto e vírgula (token ‘PONTO_VIRGULA’). O terminal <LISTA_CMD_LEIA_VAR> pode ser uma redução do terminal <CMD_LEIA_VAR>

(linha 5) ou uma redução de um terminal <LISTA_CMD_LEIA_VAR>, seguido por uma vírgula (token 'VIRGULA') e um terminal <CMD_LEIA_VAR> (linha 6). Já o terminal <CMD_LEIA_VAR> pode ser uma redução de um identificador (linha 7, token 'ID'), ou uma redução de uma variável do tipo vetor (linha 8).

```
1 <CMD_LEIA>           ::= LEIA ABRE_PAR <LISTA_CMD_LEIA_VAR> FECHA_PAR PONTO_VIRGULA
2 <LISTA_CMD_LEIA_VAR> ::= <CMD_LEIA_VAR> | <LISTA_CMD_LEIA_VAR> VIRGULA <CMD_LEIA_VAR>
3 <CMD_LEIA_VAR>       ::= ID | ID ABRE_COL <LST_ATRIB_VET_INDEX> FECHA_COL
4
5 <LISTA_CMD_LEIA_VAR> -> <CMD_LEIA_VAR>
6 <LISTA_CMD_LEIA_VAR> -> <LISTA_CMD_LEIA_VAR> VIRGULA <CMD_LEIA_VAR>
7 <CMD_LEIA_VAR> -> ID
8 <CMD_LEIA_VAR> -> ID ABRE_COL <LST_ATRIB_VET_INDEX> FECHA_COL
```

Listagem 2: Fragmento da gramática e das reduções (Fonte: do autor).

Como visto, o comando 'leia' pode obter do usuário um valor por vez, ou uma sequência de valores que serão armazenados nas variáveis informadas nos parâmetros do comando.

Ainda durante a análise sintática, irá ser criada simultaneamente uma lista de representação da árvore de análise sintática, essa lista é uma variação da lista de tokens gerada pelo analisador léxico, porém além das informações do token (lexema, token, linha, posição, etc.), também irá possuir as informações de escopo, do grupo à qual o token pertence, e o tipo de redução do grupo. As informações de escopo e de grupo são utilizadas para indicar como deverá ser construído os segmentos da árvore de análise sintática (AAS).

A AAS é construída utilizando como entrada, a lista de representação da árvore de análise sintática gerada durante a análise semântica, e as classes de tipos dos segmentos da árvore de análise sintática. Na figura 4 pode ser observado um fragmento do diagrama de classes desses segmentos.

A classe 'Tipo' implementa a interface 'ITipo', e possui um método estático para construção automática de segmentos da AAS, que utiliza os tokens dos comandos e o tipo de redução executada durante a análise sintática (obtidos da lista de representação da AAS). Também possui métodos genéricos de validação semântica compartilhados entre as demais classes que efetuam sua herança, sendo um método que valida os nomes das variáveis declaradas no escopo local em procedimentos, funções e na função principal, e dos nomes dos parâmetros nos procedimentos e funções. E outro método que valida as variáveis nos comandos dentro dos procedimentos, funções e da função principal, verificando se os tipos de dados são compatíveis com a operação em que são utilizados.

A classe 'Algoritmo' é a classe raiz da AAS, e é composto por uma função principal, e pelas listas de variáveis globais e de comandos, que podem ser vazias. A lista de comandos se refere apenas à procedimentos e funções, outros tipos não são utilizados neste caso.

As classes de procedimentos, funções e da função principal, são segmentos que sucedem a raiz da AAS, estes armazenando as listas de variáveis locais (pode ser vazia) e a lista de comandos (exige pelo menos um comando). Nos procedimentos e funções, também são armazenados os parâmetros de entrada destes.

As variáveis podem ser de qualquer um dos tipos primitivos (lógico, inteiro, real ou caracter), podendo ser definido apenas um vetor por declaração, e para variáveis simples (utiliza a classe 'DeclaraçãoSimples') pode ser definido uma ou várias

variáveis simultaneamente do mesmo tipo primitivo.

Os comandos de procedimentos, funções e da função principal, são comandos para gestão do ciclo do programa, sendo laços de repetição, seleções condicionais se-senão, atribuições (lógicas, aritméticas e de texto), leitura de entrada do usuário, e escrita de saída para o usuário.

A leitura de entrada é feita pela classe 'Leia', e aceita um ou vários parâmetros, sendo que para parâmetros do tipo vetor, se for informado apenas o nome do vetor, irá solicitar a entrada do valor para todos os registros do vetor, mas se for informado o índice do registro no vetor, apenas irá solicitar o valor para o índice informado, da mesma forma que faria com uma variável simples.

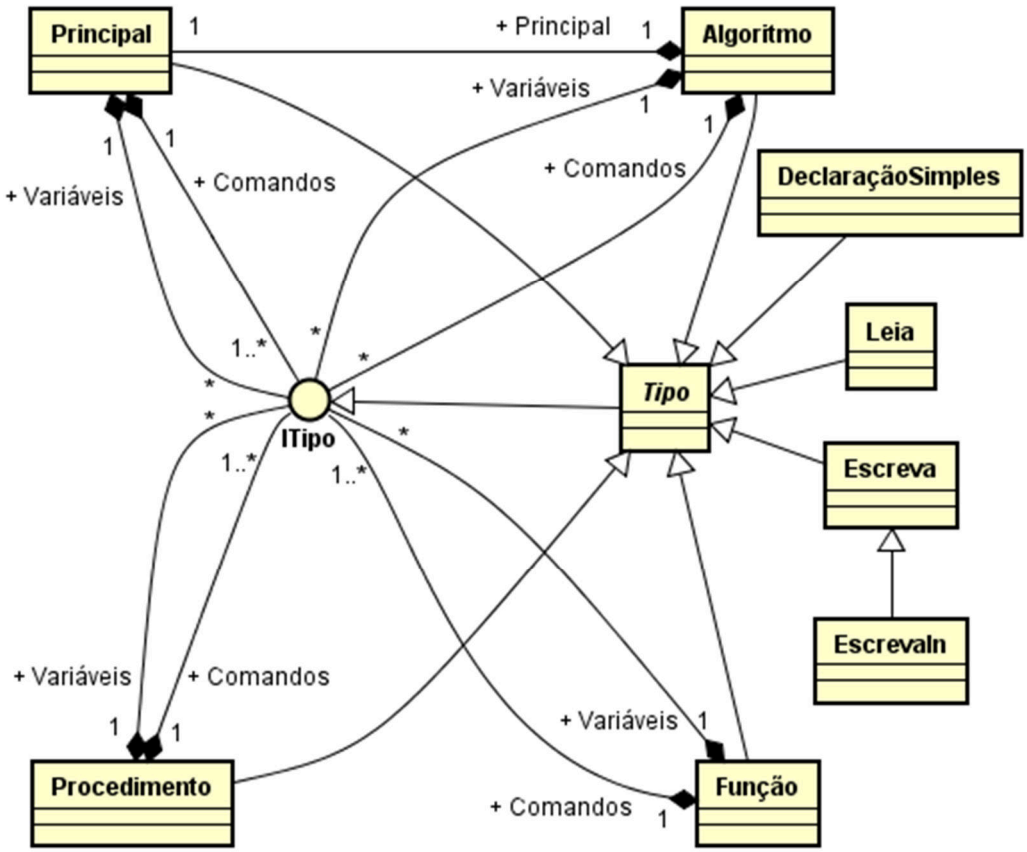

Figura 4: Fragmento diagrama de classes dos tipos (Fonte: do autor).

A escrita de saída é feita pelas classes 'Escreva' ou 'Escrevaln', e aceitam um ou vários parâmetros, sendo que para parâmetros do tipo vetor, se for informado apenas o nome do vetor, irá escrever o valor de todos os registros do vetor, separando os valores com vírgula, e agrupando entre abre e fecha chaves os valores em cada uma das dimensões, mas se for informado o índice do registro no vetor, apenas irá escrever o valor para o índice informado, da mesma forma que faria com uma variável simples. A classe 'Escrevaln' é uma extensão da classe 'Escreva', e tem como única diferença, adicionar a quebra de linha ao final da escrita da saída.

O código de montagem MIPS, é gerado dinamicamente por cada um dos segmentos da AAS, e é armazenado em uma estrutura desenvolvida para armazenar, manipular e otimizar o código de montagem, sendo responsável por isso a classe 'MipsClass'.

Inicialmente para a geração do código de montagem MIPS, é criado uma instância da classe 'MipsClass' com a passagem do algoritmo (raiz) da AAS. E após isso, pode ser gerado o resultado pela chamada do método 'ObterCódigo()', que irá processar e gerar os comandos resultantes de cada segmento, e retornar um texto representando o código fonte de entrada no formato de montagem MIPS.

As expressões aritméticas possuem tratamento de precedência, sendo da maior para a menor, parênteses, multiplicação, divisão, resto, soma e subtração. No caso da existência de parênteses, irá extrair a expressão entre os parênteses e chamar o método de geração de forma recursiva, e definindo uma variável temporária na seção de dados do código MIPS. As variáveis temporárias geradas substituem a expressão entre os parênteses à que se referem. Por exemplo: seja o caso a expressão $a * (b + c)$, a expressão $(b + c)$ será substituída pela variável temporária $tmpX\_Y$, onde X é o tipo da variável resultante e Y a posição no código fonte do abre parênteses à que a variável se refere, e essa variável temporária será calculada primeiro e o seu resultado será armazenado na memória, após isso, será calculado o restante da expressão. De forma resumida, primeiro $tmpX\_Y = b + c$, e depois $a * tmpX\_Y$, e se obtém o resultado final.

Para as operações aritméticas são utilizados três registradores temporários inteiros e três de ponto flutuante, sendo, $t0 ou $f4 para o resultado ou o primeiro termo ou acumulador, $t1 ou $f5 para o segundo termo ou segundo acumulador, e $t2 ou $f6 para o terceiro termo. O tipo de resultado esperado é verificado antes de iniciar a montagem de instruções MIPS, sendo se for um resultado inteiro, irá utilizar os registradores $t0, $t1 e $t2, e se for de ponto flutuante, irá utilizar os registradores $f4, $f5 e $f6. Por exemplo, para a expressão de valores inteiros $a + b$, o registador $t0 irá receber o valor da variável a, e o registrador $t1 irá receber o valor da variável b, sendo em seguida adicionada a instrução de adição, entre os registradores $t0 e $t1 e o resultado armazenado em $t0.

Na figura 5 pode ser observado a janela principal do programa desenvolvido, com um código fonte já carregado. Palavras reservadas são formatadas automaticamente, assim como valores numéricos e comentários. Durante a execução da análise do código fonte, os erros encontrados em qualquer uma das partes do processo serão destacados e os erros serão descritos no campo de informações de saída. Se forem encontrados erros durante uma etapa de análise, o processo não irá executar o seguinte, por exemplo, se as análise léxica e sintática passarem, mas durante a análise semântica for encontrado erros, o processo irá parar ao final desta, e não será iniciada a geração de código. Já na figura 6 e na listagem 3, são apresentados a AAS e parte do código MIPS resultante deste código.

A AAS na figura 6, representa detalhadamente o código fonte analisado, destacando quantidade de variáveis declaradas, quantidade de procedimentos e funções, quantidade de comandos dentro de cada escopo, além de estruturar visivelmente a hierarquia dos componentes da árvore.

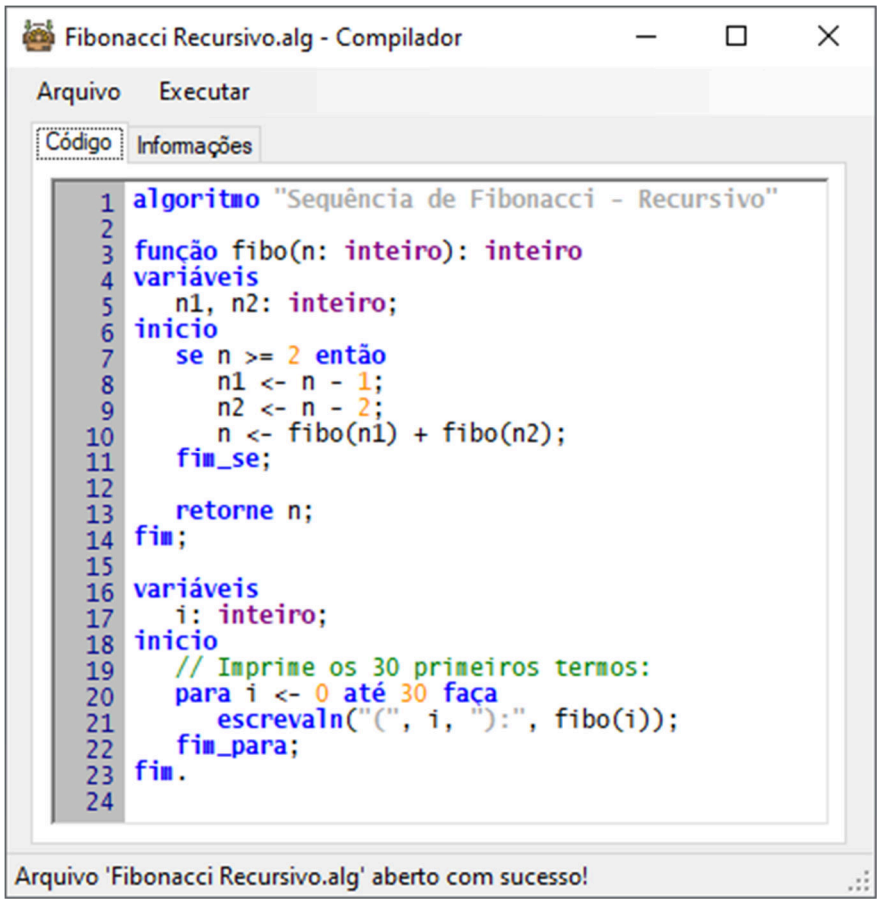

Figura 5: Janela principal do programa (Fonte: do autor).

Na listagem 3, está um trecho do código MIPS gerado a partir do código fonte na figura 5, demonstrando o resultado da geração da função que está entre as linhas 3 e 14 (figura 5). Na linha 16 do código MIPS, está a etiqueta utilizada para chamada da função 'fibo(n)', nas linhas 17 e 18 é deslocado o ponteiro da pilha (registrador $sp) em 4 posições e armazenado o endereço de retorno da chamada da função, que está no registrador $ra, sendo importante salvar esse endereço na pilha, para evitar problemas em casos que se tenha procedimentos ou funções aninhadas, pois a cada chamada o endereço de retorno no registrador $ra será atualizado, e o endereço de retorno da função anterior será perdido, causando erro e travamento do sistema.

Nas linhas 20 a 25, é executado o teste condicional maior ou igual entre a variável n e a constante 2 (linha 7 na figura 5), para isso, a variável n é armazenada no registrador $t3, e a constante 2 no registrador $t0, onde a instrução 'sge' (*set greater equal than*, ou em português, seta se maior ou igual a) irá testar os valores nos registradores $t3 e $t0 e salvar o resultado em $t0, sendo se $t3 for maior ou igual a $t0, $t0 irá receber 1, caso contrario 0. Após o valor no registrador $t0 é movido para $t4, e a instrução 'beq' (*branch equal*, ou em português, ramificar se igual) irá testar se o valor em $t4 é igual a 1, e se for irá para a posição da etiqueta 'seBlocoA_121' (linha 27), caso contrário, irá continuar a execução e encontrar na linha 26 a instrução 'j' (*jump*, ou em português, pular) para a etiqueta 'seFim_121' (linha 89).

Já entre as linhas 28 até 87, está o conteúdo do bloco A (ou bloco verdadeiro) do comando se-senão (linhas 8 a 10 na figura 5), onde nas linhas 28 a 31 e 33 a 36 estão sendo calculados os valores de 'n1' e 'n2' respectivamente, e nas linhas 38 a

87 é calculado o valor de ‘n’. O resultado de ‘n’ é a soma da função ‘fibo’ sendo chamada recursivamente, onde ‘fibo(n1)’ está entre as linhas 38 a 60, e ‘fibo(n2)’ está entre as linhas 63 a 83. Nas linhas 38 e 39, é transferido o valor da variável ‘n1’ para o parâmetro ‘n’ da chamada ‘fibo(n1)’, e nas linhas 63 e 64, é transferido o valor da variável ‘n2’ para o parâmetro ‘n’ da chamada ‘fibo(n2)’.

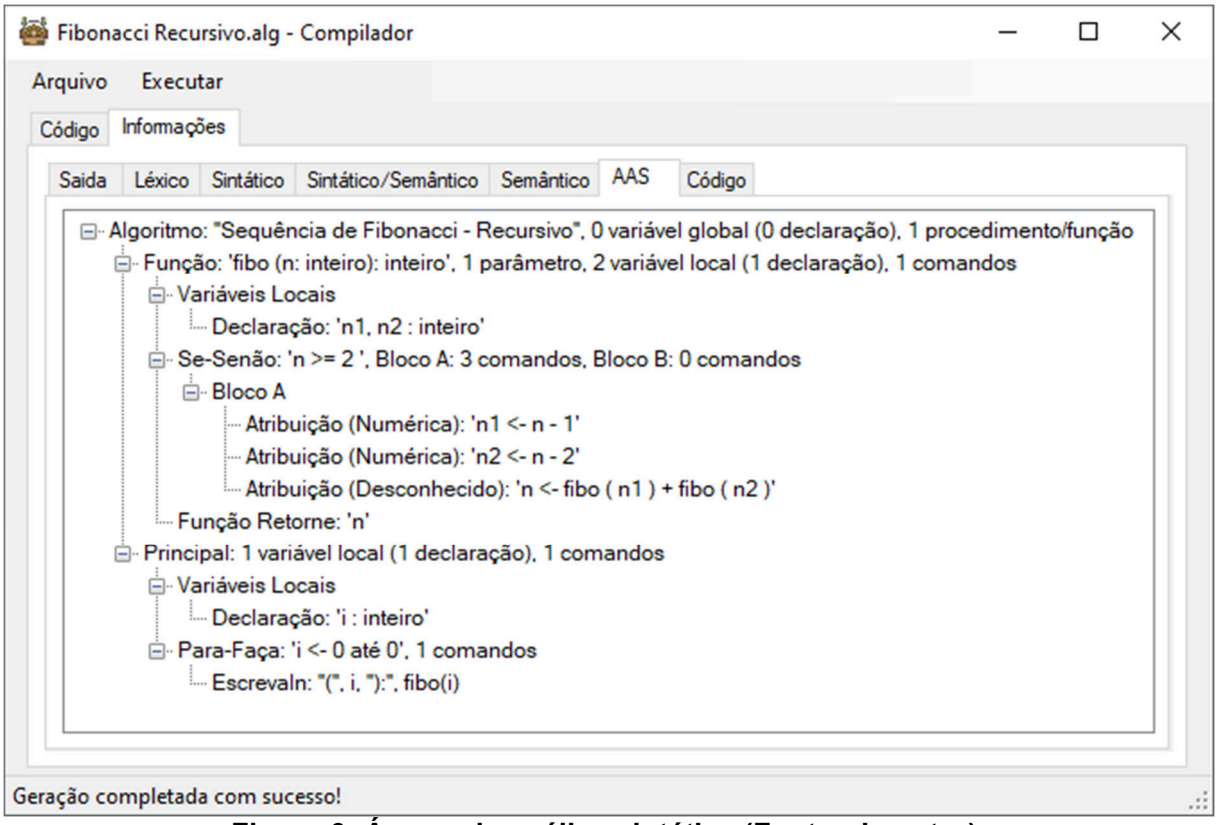

Figura 6: Árvore de análise sintática (Fonte: do autor).

Nas chamadas recursivas é necessário salvar e restaurar as informações de contexto, para isso foi utilizado a pilha do sistema, sendo implementado algumas regras de gestão da pilha, onde é reservado o espaço necessário para armazenar o contexto naquele instante, e ainda deve ficar disponível pelo menos dez bytes na pilha, sendo se não houver espaço suficiente, o sistema irá gerar um erro de estouro de pilha e terminar a execução.

Isso pode ser visto entre as linhas 40 a 60 na listagem 3, onde na linha 40 é carregado a quantidade mínima requerida de bytes disponíveis na pilha (calculado durante a geração do código MIPS), e em seguida a instrução ‘blt’ (*branch lower than*, ou em português, ramificar se menor que) testa essa quantia com o valor de endereço do ponteiro da pilha, que representa a quantidade de bytes na pilha, e se a quantidade na pilha for menor que a necessária, gera o erro de estouro de pilha finalizando a execução. Já na linha 42, é reservado o espaço necessário para armazenar o contexto na pilha, sendo nas linhas 43 a 50 o contexto armazenado na pilha, e então na linha 51 é executado a instrução ‘jal’ (*jump and link*, ou em português, saltar e vincular) para a etiqueta ‘fibo’ (linha 16), onde é efetuado a chamada da função e o endereço para retorno é armazenado no registrador $ra. Após retornar da execução da função, o contexto é restaurado pelas instruções das linhas 52 até 59, e na linha 60 é restituído

o espaço que havia sido reservado anteriormente. Esse mesmo processo se repete entre as linhas 65 a 83.

```
15 # Função: 'fibo (n: inteiro): inteiro'
16 fibo:
17   addiu $sp,$sp,-4
18   sw $ra,0($sp)
19
20   lw $t0,prInt_60_n
21   move $t3,$t0
22   li $t0,2
23   sge $t0,$t3,$t0
24   move $t4,$t0
25   beq $t4,1,seBlocoA_121
26   j seFim_121
27 seBlocoA_121:
28   lw $t0,prInt_60_n
29   li $t1,1
30   sub $t0,$t0,$t1
31   sw $t0,varInt_94_n1
32
33   lw $t0,prInt_60_n
34   li $t1,2
35   sub $t0,$t0,$t1
36   sw $t0,varInt_98_n2
37
38   lw $t5,varInt_94_n1
39   sw $t5,prInt_60_n
40   li $t5,30
41   blt $sp,$t5,fimErroStackPointer
42   addiu $sp,$sp,-20
43   sw $t0,0($sp)
44   sw $t1,4($sp)
45   lw $t5,varInt_94_n1
46   sw $t5,8($sp)
47   lw $t5,varInt_98_n2
48   sw $t5,12($sp)
49   lw $t5,prInt_60_n
50   sw $t5,16($sp)
51   jal fibo
52   lw $t0,0($sp)
53   lw $t1,4($sp)
54   lw $t5,8($sp)
55   sw $t5,varInt_94_n1
56   lw $t5,12($sp)
57   sw $t5,varInt_98_n2
58   lw $t5,16($sp)
59   sw $t5,prInt_60_n
60   addiu $sp,$sp,20
61   move $t0,$s0
62
63   lw $t5,varInt_98_n2
64   sw $t5,prInt_60_n
65   li $t5,26
66   blt $sp,$t5,fimErroStackPointer
67   addiu $sp,$sp,-16
68   sw $t0,0($sp)
69   lw $t5,varInt_94_n1
70   sw $t5,4($sp)
71   lw $t5,varInt_98_n2
72   sw $t5,8($sp)
73   lw $t5,prInt_60_n
74   sw $t5,12($sp)
75   jal fibo
76   lw $t0,0($sp)
77   lw $t5,4($sp)
78   sw $t5,varInt_94_n1
79   lw $t5,8($sp)
80   sw $t5,varInt_98_n2
81   lw $t5,12($sp)
82   sw $t5,prInt_60_n
83   addiu $sp,$sp,16
84   move $t1,$s0
85
86   add $t0,$t0,$t1
87   sw $t0,prInt_60_n
88
89 seFim_121:
90
91   lw $t0,prInt_60_n
92   move $s0,$t0
93   lw $ra,0($sp)
94   addiu $sp,$sp,4
95   jr $ra
```

Listagem 3: Fragmento código MIPS (Fonte: do autor).

Para funções que possuem retorno de valores, como é o caso da função ‘fibo(n)’, é utilizado o registrador $s0 para armazenar esse valor de retorno entre as chamadas das funções. Nas linhas 61 e 84 na listagem 3, pode ser observado que o valor no registrador $s0 é transferido para os registradores $t0 e $t1, compondo assim a operação de soma dos resultados de ‘fibo(n1)’ e ‘fibo(n2)’, realizada pela instrução ‘add’ (adicionar) na linha 86, e com o resultado armazenado no registrador $t0, sendo após transferido para a variável ‘n’.

Ainda na listagem 3, ao fim da função ocorre o comando de retorno da função (corresponde à linha 13 da figura 5), onde na linha 91 o valor da variável ‘n’ é armazenada no registrador $t0, e em seguida transferida para o registrador $s0, estando agora o valor de retorno da função disponível quando ocorrer o salto de retorno da chamada da função. Na linha 93 é restaurado da pilha o endereço de retorno da chamada da função para o registrador $ra, e em seguida restituído o espaço utilizado na pilha para o sistema. E na linha 95 a instrução ‘jr’ (*jump register*, ou em português, saltar para registro) irá retornar a execução para o endereço de retorno da chamada da função armazenada no registrador $ra.

CONCLUSÕES

Neste trabalho foi apresentado um compilador simples, onde o código fonte de entrada é um texto baseado na sintaxe de algoritmos em português estruturado, e a saída, é gerado do código de montagem na arquitetura Assembly MIPS, compatível com o simulador MARS MIPS.

Todas as etapas do processo possuem saídas detalhadas individualmente, sendo útil como ferramenta de aprendizado, sobre como cada etapa do processo de compilação é executado, e como irá responder caso seja encontrado erros durante qualquer etapa de análise (léxica, sintática ou semântica).

O código fonte de entrada, foi limitado para reduzir a complexidade durante o

desenvolvimento, para isso, não foi implementado suporte a registros, os procedimentos e funções apenas aceitam como argumentos de entrada variáveis únicas (operações aritméticas ou lógicas e constantes não serão aceitos).

Posteriormente, pretende-se implementar suporte para registros, e também para operações e constante nos argumentos de procedimentos e funções. A integração de um simulador para a arquitetura de montagem MIPS também é prevista, o que iria remover a necessidade de um simulador de terceiros, como a utilização do simulador MARS MIPS.

O código fonte do compilador desenvolvido, além da documentação completa da gramática, reduções e as tabelas de estados sintáticos, está disponibilizado de forma livre para referências e consultas na plataforma GitHub[4].

REFERÊNCIAS

Alkmim, G. P., & Mello, B. A. **Ferramenta de apoio às fases iniciais do ensino de linguagens formais e compiladores**. Anais do Simpósio Brasileiro de Informática na Educação, 21 (pp. 1-4), João Pessoa, 2010. Disponível em: <http://ojs.sector3.com.br/index.php/sbie/article/view/1529>, acesso em: 20/11/2022.

Backus, J. W., Beeber, R. J., Best, S., Goldberg, R., Haibt, L. M., Herrick, H. L., Nelson, R. A., Sayre, D., Sheridan, P. B., Stern, H., Ziller, I., Hughes, R. A., & Nutt, R. (1957). **The FORTRAN automatic coding system. February**. In Proceedings Western Joint Computer Conference, 57 (pp. 188-198), New York. doi: 10.1145/1455567.1455599

Cooper, K., & Torczon, L. **Engineering a compiler**. Elsevier, 2ª edição, 2011.

Crespo, R. G. **Processadores de Linguagens: da concepção à implementação**. Lisboa: Instituto Superior Técnica, 1998.

José Neto, J. **Introdução à Compilação**. Rio de Janeiro, Elsevier, 1ª edição, 2016.

Martins, J. P. **Introdução à Programação usando o Pascal**. Lisboa: McGraw-Hill Portugal, 1994.

Ramos, M. V. M.; **Linguagens Formais e Autômatos**. 22/04/2008. Disponível em: <http://gege.fct.unesp.br/docentes/dmec/olivete/lfa/arquivos/Apostila.pdf>. Acesso em: 20/11/2022.

Ricarte, I. **Introdução à compilação**. Rio de Janeiro: Elsevier, 1ª edição, 2008.

Santos, P. R., & Langlois, T. (2018). **Compiladores: da teoria à prática**. Rio de Janeiro: LTC; 1ª edição, 2018.

Sethi, R. Ullman, J. D.; Lam, M. S.; **Compiladores: Princípios, Técnicas e Ferramentas**. 2. ed. São Paulo: Pearson Addison Wesley, 2008.

Sweetman, D. **See MIPS Run**. Morgan Kaufmann Publishers, 1ª edição, 1999.

Torczon, L.; Keith, C..; **Construindo Compiladores**. 2ª. ed. Rio de Janeiro: Elsevier, 2014.

Villanueva, J. M. M.; **Teoria de Linguagens Formais e Autômatos**. 2016. Disponível em: <http://www.cear.ufpb.br/juan/wp-content/uploads/2016/08/Aula-4a-Linguagens-

[4] https://github.com/alexandrebaua/compiladores2022-atividade08

Formais-e-Autômatos.pdf>. Acesso em: 20/11/2022.

Vollmar, K.; Sanderson, P.; **MARS: an education-oriented MIPS assembly language simulator**. 2006. Disponível em: <https://dl.acm.org/doi/abs/10.1145/1121341.1121415?casa_token=kPqmIHNeaVkAAAAA:4rK2F2R3QlXwduuJZ2wq_iDSXG4QMqWIu_yGDapHjL8EIy3u_eJVPTCZQKa7_sfhIGPRQGXZNYZF>. Acesso em: 23/11/2022.