Lucas Vecchete

Implementação de compilador de C- para a arquitetura ENILA

Brasil

2017, v-1.1

Lucas Vecchete

Implementação de compilador de C- para a arquitetura ENILA

Desenvolvimento de um compilador de C-, exclusivamente para a arquitetura ENILA, usando o Flex e o Bison.

Universidade Federal de São Paulo – UNIFESP

Instituto de Ciência e Tecnologia

Engenharia de Computação

Brasil

2017, v-1.1

Lista de ilustrações

Sumário

Resumo

O projeto consiste na continuação do desenvolvimento de um sistema computacional. Este projeto insere-se dentro de um projeto macro com a finalidade de desenvolvimento do sistema computacional completo, que é proposto na grade curricular do curso de engenharia de computação. Até o presente momento foi realizado o desenvolvimento de um processador, no qual sua arquitetura foi idealizada do zero, e sendo denominada de arquitetura ENILA. Essa arquitetura é uma arquitetura de 32 bits, monociclo, e com operação do tipo registrador-registrador(com LOAD e STORE). Inicialmente a arquitetura funcionava com endereçamento direto, porém, com algumas mudanças necessária para a implementação do compilador, ele foi alterado para funcionar com endereçamento relativo, utilizando uma pilha de recursão. Nessa fase do projeto foi desenvolvido o compilador de C- para a arquitetura ENILA, permitindo-se assim usar uma linguagem com um nível de abstração maior com o processador. Como será discutido nesse relatório, foi desenvolvido todas os módulos que compõem um compilador, sendo eles: analisador léxico, semântico, e sintático, gerador de código intermediário e gerador de código de máquina.

1 Introdução

Com o advento do computador proposto por Von Neumann na década de 40, era necessário escrever sequências de instruções (programas) para que o computador pudesse cumprir o seu propósito. Inicialmente esse programa era escrito em linguagem de máquina, ou seja, instruções formadas por 0 e 1. Logo o desenvolvimento com código de máquina foi substituído pela implementação em *assembly*, no qual se utilizava de um programa escrito em código de máquina que traduzia o *assembly* em código de máquina, esse programa era denominado como gerador de código de máquina(em inglês, *assembler*).(1)

Entre 1954 e 1957 o FORTRAN foi desenvolvido pela IBM assim como o seu compilador. Paralelamente Noam Chomsky avançou nos estudos de linguagens formais e autômatos, que contribuíram fortemente para o desenvolvimento dos compiladores.(1)

Pode-se definir então que compiladores são programas que pegam como entrada um programa escrito em uma linguagem fonte e traduzem para uma linguagem de destino. Geralmente, a linguagem fonte é uma linguagem de alto nível, como C ou C++, e a linguagem de destino é o código de máquina.(1)

Na estrutura de um compilador, existe alguns sub-módulos que o compõem, como o analisador léxico, analisador sintático, analisador semântico, gerador de código intermediário, e gerador de código de máquina. A figura abaixo (Figura 1) representa os passos presentes em um compilador.

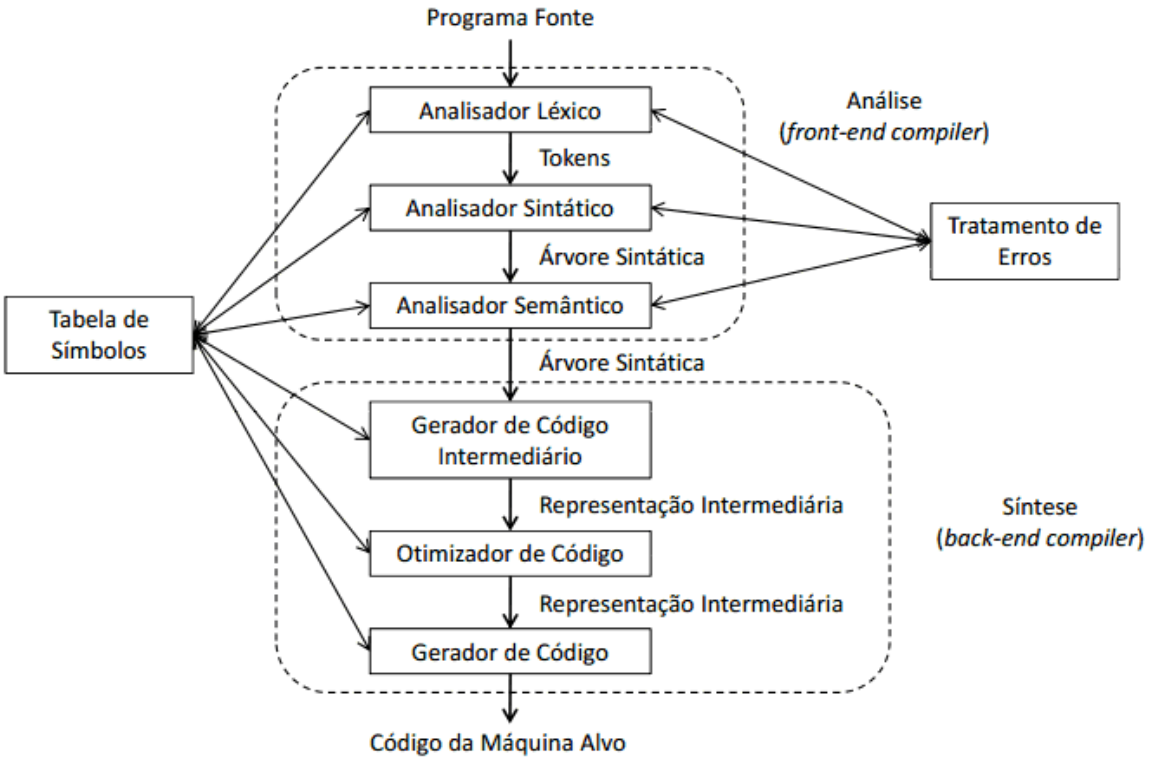

Figura 1 – Fluxo de processos em um compilador de acordo com Louden (1).

Um compilador é como um todo um programa bem complexo que pode ter facilmente de 10 mil a um 1 milhão de linhas, portanto, não é um programa fácil de desenvolver. Todavia, é extremamente importante que um profissional da área de computação saiba como compilador funciona como inteiro, por isso, justifica-se a necessidade desse projeto.

2 Fundamentação Teórica

Nessa seção serão listados e abordados todos os conceitos envolvidos no projeto.

2.1 Compiladores

O compilador é um programa escrito em uma linguagem determinada, que como dito anteriormente, converte um código fonte em um código de máquina, ou código objeto. Foi dito também, que para gerar tal código na linguagem de destino são realizadas algumas etapas. Tais etapas podem ser dividas em módulos. Sendo eles:

- *Scanner*: Também chamado de analisador léxico, tem a função de processar e analisar a entrada de linhas de caracteres. Depois produz uma sequência de símbolos chamado "símbolos léxicos"(*lexical tokens*).

- *Parser*: Também chamado de analisador sintático, analisa a sequência de entrada gerada pelo *Scanner* para determinar sua estrutura gramatical segundo uma determinada gramática formal. Durante o processo, esse módulo gera a árvore de análise sintática.

- *Semantic Analyser*: A analise 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. A análise semântica percorre a árvore sintática e relaciona os identificadores com seus dependentes de acordo com a estrutura hierárquica. Durante esse módulo, enquanto não é achado erros, as variáveis são armazenadas em uma tabela de símbolos.

- *Code Generator*: Nesse módulo é gerado uma sequência de código denominada código intermediário, que posteriormente em outras fases irá gerar o código objeto. Por ventura, esse módulo pode não existir e a compilação pode ser feita diretamente para o código objeto.

- *Target Code Generator*: A geração de código objeto é o último módulo do processo de compilação e recebe como entrada uma representação intermediaria que mapeia a linguagem objeto. Desse momento deve ser feito a seleção de registradores e um gerenciamento de memória para contantes e variáveis. Essa é uma etapa muito importante pois a produção de código objeto eficiente deve ter uma cuidadosa seleção de registradores.

A Figura abaixo(Figura 2) exemplifica o processo de compilação divido em etapas. Essas etapas correspondem a estrutura lógica de um compilador e o resultado de cada fase.

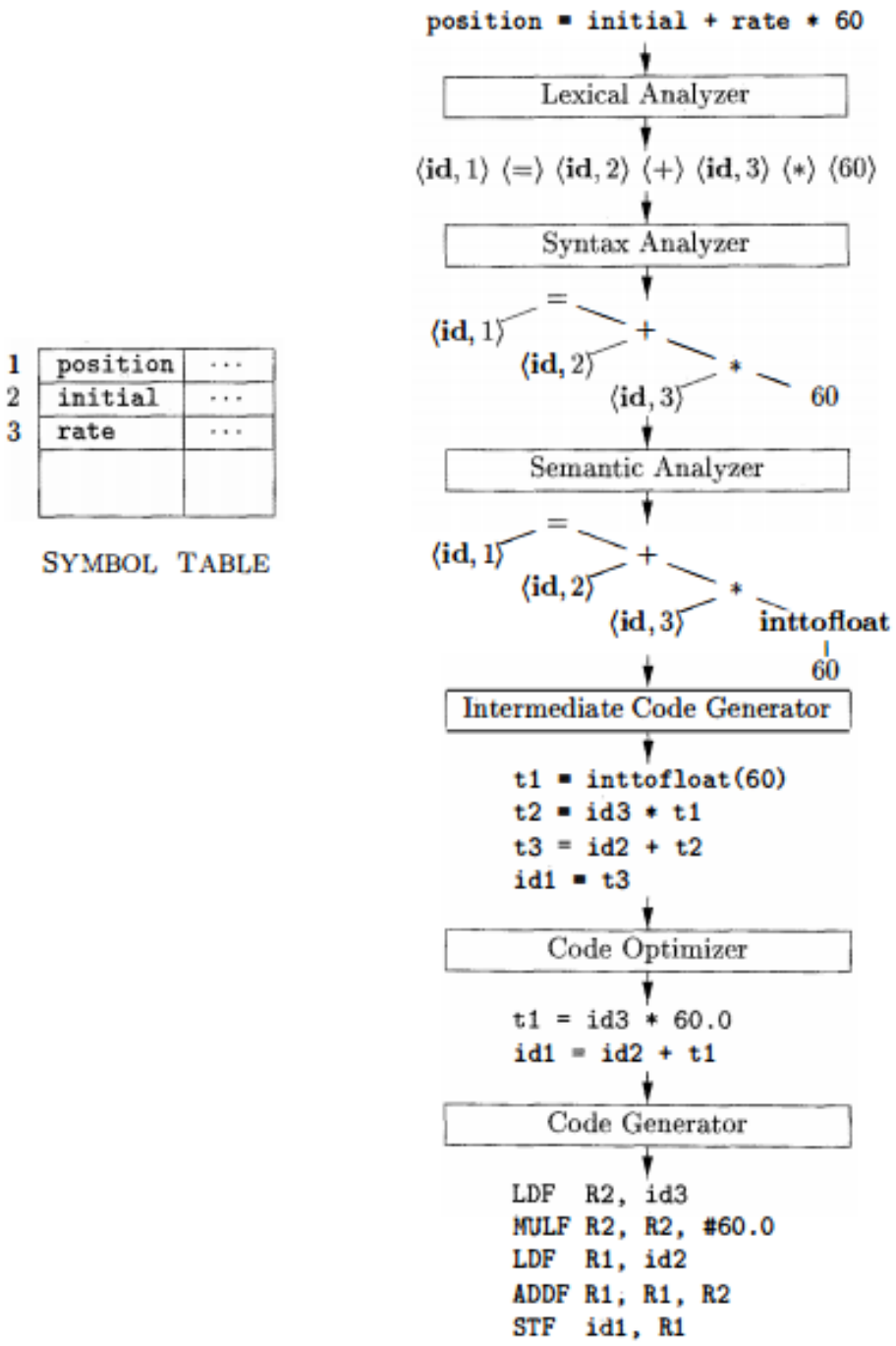

Figura 2 – Estrutura didática do funcionamento de um compilador, ilustrando todos os seus módulos.

2.1.1 Analisador Léxico

O analisador léxico é a primeira fase de um compilador. Sua tarefa principal é a de ler os caracteres de entrada e produzir uma sequencia de *tokens* que o *parser* utiliza para a análise sintática. Essa interação sumarizada esquematicamente na Figura 3 é comumente implementada fazendo-se com que o analisador léxico seja uma sub-rotina ou uma co-rotina do *parser*. Ao receber do *parser* um comando "obter o próximo *token*", o analisador léxico lê os caracteres de entrada até que possa identificar o próximo *token*.

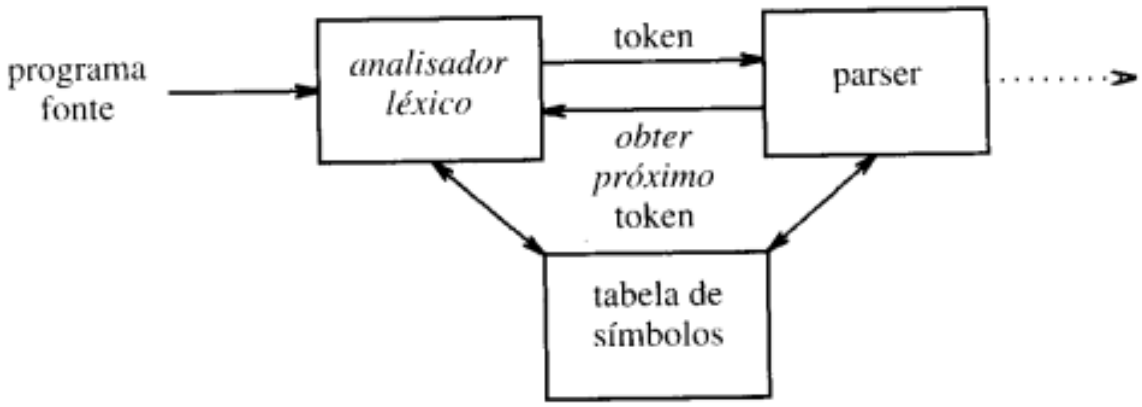

Figura 3 – Interação do analisador léxico com o parser.

Como o analisador léxico é a parte do compilador que lê o texto-fonte, também pode realizar algumas tarefas secundárias ao nível da interface com o usuário. Uma delas é a de remover do programa-fonte os comentários e espaços em branco, os últimos sob a forma de espaços, tabulações e caracteres de avanço de linha.

Uma forma simples de construir um analisador léxico é escrever um diagrama que ilustre a estrutura dos *tokens* da linguagem-fonte e então traduzi-lo manualmente num programa que os localize.

A análise léxica pode ser dividida em duas etapas, a primeira chamada de escandimento que é uma simples varredura removendo comentários e espaços em branco, e a segunda etapa, a analise léxica propriamente dita onde o texto é quebrado em lexemas.

Defini-se então três termos relacionados a implementação de um analisador léxico:

- Padrão: é a forma que os lexemas de uma cadeia de caracteres pode assumir. No caso de palavras reservadas é a sequência de caracteres que formam a palavra reservada, no caso de identificadores são os caracteres que formam os nomes das variáveis e funções.

- Lexema:é uma sequência de caracteres reconhecidos por um padrão.

- Token: é um par constituído de um nome é um valor de atributo esse ultimo opcional. O nome de um token é um símbolo que representa a unidade léxica. Por exemplo: palavras reservadas; identificadores; números, etc.

A Figura 4 abaixo mostra os exemplos de uso dos termos durante a análise léxica.

Token	Padrão	Lexema	Descrição
`<const, >`	Sequência das palavras c, o, n, s, t	const	Palavra reservada
`<while, >`	Sequência das palavras w, h, i, l, e	while, While, WHILE	Palavra reservada
`<if, >`	Sequência das palavras i, f	If, IF, iF, If	Palavra reservada
`<=, >`	<, >, <=, >=, ==, !=	==, !=	
`<numero, 18>`	Dígitos numéricos	0.6, 18, 0.009	Constante numérica
`<literal, "Olá">`	Caracteres entre ""	"Olá Mundo"	Constante literal
`<identificador, 1>`	Nomes de variáveis, funções, parâmetros de funções.	nomeCliente, descricaoProduto, calcularPreco()	Nome de variável, nome de função
`<=, >`	=	=	Comando de atribuição
`<{, >`	{, }, [,]	{, }, [,]	Delimitadores de início e fim

Figura 4 – Interação do analisador léxico com o parser.

Para implementar um analisador léxico é necessário ter uma descrição dos lexemas, então, pode-se escrever o código que irá identificar a ocorrência de cada lexema e identificar cada cadeia de caractere casando com o padrão.

Os tokens destacados anteriormente são símbolos léxicos reconhecidos através de um padrão. Os tokens podem ser divididos em dois grupos:

- Tokens simples: são tokens que não têm valor associado pois a classe do token já a descreve. Exemplo: palavras reservadas, operadores, delimitadores: <if,>, <else>, <+,>.

- Tokens com argumento: são tokens que têm valor associado e corresponde a elementos da linguagem definidos pelo programador. Exemplo: identificadores, constantes numéricas - <id, 3>, <numero, 10>, <literal, Olá Mundo> .

Um token possui a seguinte estrutura:

```
<nome-token , valor-atributo>
```

Suponha que tenhamos o seguinte trecho de código:

```
total = entrada * saida() + 2
```

O seguinte fluxo de tokens é gerado.

```
<id , total> <=, > <id , entrada> <*, > <id ,saida >, <(>, <)> <+, > <numero ,
```

Percebe-se que análise léxica é muito prematura para identificar alguns erros de compilação, veja o exemplo abaixo:

```
fi (a == "123")
```

O analisador léxico não consegue identificar o erro da instrução listada acima, pois ele não consegue identificar que em determinada posição deve ser declarado a palavra reservada if ao invés de fi. Essa verificação somente é possível ser feita na análise sintática.

Porem é importante ressaltar que o compilador deve continuar o processo de compilação afim de encontrar o maior número de erros possível.

Uma situação comum de erro léxico é a presença de caracteres que não pertencem a nenhum padrão conhecido da linguagem. Nesse caso o analisador léxico deve sinalizar um erro informando a posição desse caractere.

Expressões regulares são um mecanismo importante para especificar os padrões de lexemas, pois são uma forma simples e flexível de identificar cadeias de caracteres em palavras. As expressões regulares estão diretamente relacionadas a autômatos finitos não determinístico e são uma alternativa amigável para criar notações de NFA. São utilizadas por editores de texto, linguagem de programação, programas utilitários, IDE de desenvolvimento e compiladores e seu padrões são independentes de linguagem de programação.

Essas expressões são escritas em uma linguagem formal que pode ser interpretada por um processador de expressão regular que examina o texto e identifica partes que casam com a especificação dada, são muito utilizadas para validar entradas de dados, fazer buscas, e extrair informações de textos. As expressões regulares não validam dados, apenas verificam se um texto está em uma determinado padrão.

As expressões regulares são formadas por metacarateres que definem padrões para obter o casamento entre uma expressão regular e um texto. Metacaracteres, são caracteres

que tem um significado especial na expressão regular. Na Figura 5 vemos um exemplo com os principais metacaracteres.

Meta	Descrição	Exemplo
.	Curinga	Qualquer caractere
[]	Lista	Qualquer caractere incluído no conjunto
	Lista negada	Qualquer caractere não incluído no conjunto
\d	Dígito	O mesmo que [0-9]
\D	Não-digito	O mesmo que [0-9]
\s	Caracteres em branco	
\S	Caracteres em diferentes de branco	
\D	Alfanumérico	O mesmo que [a-zA-Z0-9_]
\W	Não-alfanumérico	
\	Escape	Faz com que o caracteres não sejam avaliados na Regex
(...)	Grupo	É usado para criar um agrupamento de expressões
|	OU	casa | bonita – pode ser casa ou bonita

Figura 5 – Principais metacaracteres.

Existe também os quantificadores, que são tipos de metacaracteres que definem um número permitido de repetições na expressão regular. A Figura abaixo(Figura 6), mostra quais são os quantificadores.

Expressão	Descrição	Exemplo
{n}	Exatamente n ocorrências	
{n,m}	No mínimo n ocorrências e no máximo m	
{n,}	No mínimo n ocorrências	
{,n}	No máximo n ocorrências	
?	0 ou 1 ocorrência	car?ro – caro ou carro.
+	1,ou mais ocorrência	ca*ro –carro, carrro, carrrro, nunca será caro.
*	0 ou mais ocorrência	ca*ro – caro, carro, carro, carrrro

Figura 6 – Metacaracteres quantificadores.

Para realizar uma análise léxica pode-se utilizar um gerador de análise léxica que automatiza o processo de criação do autômato finito e o processo de reconhecimento de sentenças regulares a partir da especificação de expressões regulares. Essa ferramentas são comumente chamadas de lex.

A Figura abaixo (Figura 7) é uma representação de implementação de um autômato finito.

```
letter: set of (a .. z);
digit: set of (0 .. 9);
ident := null;        /* inicialização */
number := null;
begin
        car:= getchar;
        while car = ' '  do  car:= getchar;
        if car in letter
        then  while car in (letter or digit) do  begin
                                                      ident := ident || car;
                                                      car := getchar
                                                end
        else   if car in digit
               then  while car in digit do  begin
                                                      number := number || car;
                                                      car := getchar
                                                end
               else  if car = '('
                     then ....
end
```

Figura 7 – Exemplo de implementação de um autômato.

Embora no exemplo seja simples implementar um analisador léxico, essa tarefa podem ser muito trabalhosa, como essa complexidade é frequente na evolução de uma linguagem de programação surgiriam ferramentas que apoiam esse tipo de desenvolvimento.

Existem diversas implementações para gerar analisadores léxicos para diferentes linguagens de programação. E o ponto de partida para a criar uma especificação usando a linguagem lex é criar uma especificação de expressões regulares que descrevem os itens léxicos que são aceitos.

Este arquivo é composto por até três seções:

- Declarações: Nessa seção se se encontram as declarações de variáveis que representam definições regulares dos lexemas.

- Regras de Tradução: Nessa seção são vinculada regras que correspondentes a ações em cada expressão regular valida na linguagem.

- Procedimentos Auxiliares: Esta é a terceira e última seção do arquivo de especificação. Nela são colocadas as definições de procedimentos necessários para a realização das ações especificadas ou auxiliares ao analisador léxico.

2.1.2 Analisador Sintático

A sintaxe é a parte da gramática que estuda a disposição das palavras na frase e das frases em um discurso. Essa etapa no processo de compilação deve reconhecer as forma do programa fonte e determinar se ele é valido ou não.

No modelo de compilador que esta sendo descrito, o analisador sintático, ou *parser*, obtém uma cadeia de tokens preveniente do analisador léxico, como mostrado na Figura 8, e verifica se a mesma pode ser gerada pela gramática da linguagem-fonte. Espera-se que o analisador sintático relate quaisquer erros de sintaxe de uma forma inteligível. De também se recuperar dos erros que ocorram mais comumente, a fim de poder continuar processando o resto de sua entrada.

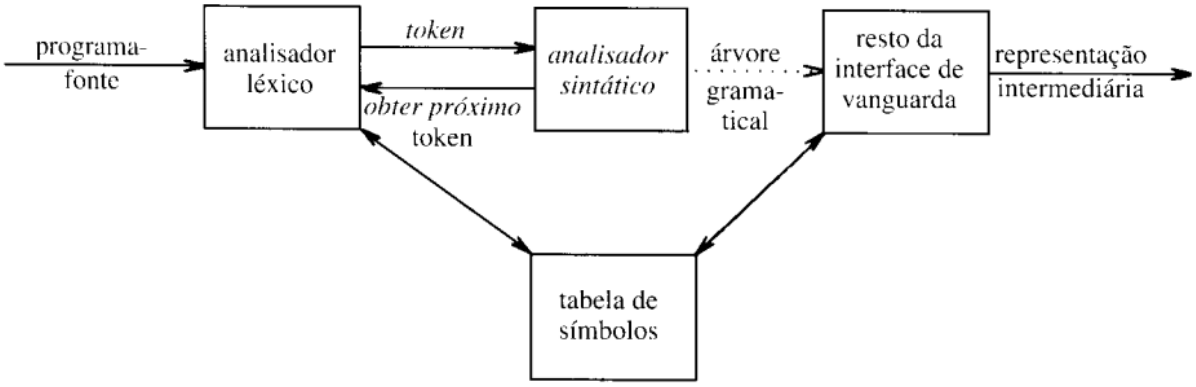

Figura 8 – Posição de um analisador sintático num modelo de compilador.

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. As derivações tem como objetivo determinar se um fluxo de palavras se encaixa na sintaxe da linguagem de programação.

As linguagens de programação em geral pertencem a uma categoria chamada de Linguagens Livres de Contexto. Umas das formas de representar essas linguagens é através de Gramáticas Livres de Contexto que são a base para a construção de analisadores sintáticos. Elas são utilizadas para especificar as regras sintáticas de uma linguagem de programação, uma linguagem regular pode ser reconhecida por uma automato finitos

determinísticos e não determinísticos, já uma Gramatica Livre de Contexto pode ser reconhecida por um automato de pilha.

Uma gramática descreve naturalmente como é possível fazer construções no programa. Veja o exemplo de um comando if-else em Pascal que deve ter a seguinte forma.

```
"if (expressao) then declaracao else declaracao ;"
```

Essa mesma forma em uma Gramática Livre de Contexto pode ser expressada da seguinte maneira:

```
declaracao -> if ( expressao ) then declaracao else declaracao ;
```

A definição formal de uma gramática livre de contexto pode ser representada através dos seguintes componentes:

```
G = (N, T, P, S)
```

Onde:

- N – Conjunto finito de símbolos não terminais.
- T – Conjunto finito de símbolos terminais.
- P – Conjunto de regras de produções.
- S – Símbolo inicial da gramática.

As regras de produção são representadas da seguinte forma:

```
{A} -> {\alpha}
```

Onde:

- A é uma variável - simbolo não terminal.
- -> representa a produção.
- P – Conjunto de regras de produções.
- α é a combinação símbolos terminais e não terminais que representam a forma como uma string vai ser formada.

Para realizar a derivação deve-se substituir o 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.

Durante a derivação devemos aplicar as regras de produção para substituir cada simbolo não terminal por um simbolo terminal, isso permite identificar se certa cadeias de caracteres pertence a linguagem, as regras expandem todas as produções possíveis. Como resultado desse processo temos a árvore de derivação.

A árvore de derivação uma estrutura em formato de árvore que representa a derivação de uma sentença ou conjunto de sentenças, essa estrutura ira gera a árvores de analise sintática que representa o programa fonte, e é o resultado da analise sintática, essa estrutura facilita é muito utilizada nas etapas seguida da compilação. É importante ressaltar que a árvore de análise sintática esta diretamente relacionada a existência de derivações.

Outra maneira de representar linguagens livres de contexto, é usando a forma de Backus-Naur, que é muito semelhante as GLCs mas possuem duas importantes diferenças quando comparamos com GLCs. A Figura 9 mostra como é um BNF.

```
<expression>  ::=  <expression> + <term>
              |    <expression> - <term>
              |    <term>
<term>        ::=  <term> * <factor>
              |    <term> / <factor>
              |    <factor>
<factor>      ::=  number
              |    name
              |    ( <expression> )
```

Figura 9 – Exemplo de um BNF.

Um analisador utiliza-se de uma GLC, gerando uma arvore de derivações. Essa árvore é chamada de "arvore de análise sintática".

Existem três tipos gerais de analisadores sintáticos. Os métodos universais de análise sintática, tais como o algoritmo de Cocke-Younger-Kasami e o de Earley, podem tratar qualquer gramática. Esses métodos, entretanto, são muito ineficientes para se usar em um compilador. Os métodos mais comumente usados nos compiladores são classificados

como *top-down* ou *bottom-up*. Como indicado por seus nomes, os analisadores sintáticos *top-down* constroem árvores do topo (raiz), para o fundo(folhas), enquanto que os *bottom-up* começam pelas folhas e trabalham árvore acima até a raiz. Em ambos os casos, a entrada é varrida da esquerda para a direita, um símbolo de cada vez.

Os métodos de análise sintática mais eficientes, tanto *top-down* quanto *bottom-up*, trabalham somente em determinadas subclasses de gramáticas, mas várias dessas subclasses de gramáticas, mas várias dessas subclasses, como as das gramáticas LL e LR, são suficientemente expressivas para descrever a maioria das construções sintáticas das linguagens de programação. Os analisadores implementados manualmente trabalham frequentemente com gramaticas LL. Os da classe mais ampla das gramáticas LR são usualmente construídos através de ferramentas automatizadas.

Assim como a análise semântica, existem ferramentas que implementam automaticamente um módulo de análise sintática baseado na GML da linguagem e suas produções, no qual permite-se utilizar várias linguagens de programação para auxiliar na montagem do compilador.

2.1.3 Analisador Semântico

Não é possível representar com expressões regulares ou com uma gramática livre de contexto regras como: 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.

A analise 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.

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.

A análise semântica percorre a árvore sintática e relaciona os identificadores com seus dependentes de acordo com a estrutura hierárquica.

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

Um exemplo que ilustra muito bem essa etapa de validação de tipos é a atribuição de objetos de tipos ou classe diferentes. Em alguns casos, o compilador realiza a conversão

automática de um tipo para outro que seja adequado à aplicação do operador. Por exemplo a expressão.

```
var s: String;
s := 2 + '2';
```

Os principais erros semânticos são:

- Variável não declarada no escopo.
- Variável sendo utilizada como um tipo diferente do que ela foi declarada.
- Função sendo chamada e não declarada.
- Tipo de retorno da função.

2.1.4 Gerador de código intermediário

A tradução do código de alto nível para o código do processador está associada a traduzir para a linguagem-alvo a representação da árvore gramatical obtida para as diversas expressões do programa. Embora tal atividade possa ser realizada para a árvore completa após a conclusão da análise sintática, em geral ela é efetivada através das ações semânticas associadas à aplicação das regras de reconhecimento do analisador sintático. Este procedimento é denominado tradução dirigida pela sintaxe.

Em geral, a geração de código não se dá diretamente para a linguagem assembly do processador-alvo. Por conveniência, o analisador sintático gera código para uma máquina abstrata, com uma linguagem próxima ao assembly, porém independente de processadores específicos. Em uma segunda etapa da geração de código, esse código intermediário é traduzido para a linguagem assembly desejada. Dessa forma, grande parte do compilador é reaproveitada para trabalhar com diferentes tipos de processadores.

Várias técnicas e várias tarefas se reúnem sob o nome de Otimização. Alguns autores da literatura consideram que, para qualquer critério de qualidade razoável, é impossível construir um programa "otimizador", isto é, um programa que recebe como entrada um programa P e constrói um programa P' equivalente que é o melhor possível, segundo o critério considerado. O que se pode construir são programas que melhoram outros programas, de maneira que o programa P' é, na maioria das vezes, melhor, segundo o critério especificado do que o programa P original. A razão para essa impossibilidade é a mesma que se encontra quando se estuda, por exemplo em LFA (Linguagens Formais e Autômatos), o "problema da parada": um programa (procedimento, máquina de Turing, etc.) não pode obter informação suficiente sobre todas as possíveis formas de execução de outro programa (procedimento, máquina de Turing, etc.).

Duas formas usuais para a representação dentro de um código intermediário são a notação posfixa e o código de três endereços.

A notação tradicional para expressões aritméticas, que representa uma operação binária na forma $x + y$, ou seja, com o operador entre seus dois operandos, é conhecida como notação infixa. Uma notação alternativa para esse tipo de expressão é a notação posfixa, também conhecida como notação polonesa , na qual o operador é expresso após seus operandos.

O atrativo da notação posfixa é que ela dispensa o uso de parênteses. Por exemplo, as expressões:

```
a*b+c;
a*(b+c);
(a+b)*c;
(a+b)*(c+d);
```

Seriam representadas nesse tipo de notação respectivamente como:

```
a b * c +
a b c + *
a b + c *
a b + c d + *
```

Instruções de desvio em código intermediário usando a notação posfixa assumema forma:

```
L jump
x y L jcc
```

Expressões em formato intermediário usando a notação posfixa podem ser eficientemente avaliadas em máquinas baseadas em pilhas, também conhecidas como máquinas de zero endereços. Nesse tipo de máquinas, operandos são explicitamente introduzidos e retirados do topo da pilha por instruções push e pop, respectivamente. Além disso, a aplicação de um operador retira do topo da pilha seus operandos e retorna ao topo da pilha o resultado de sua aplicação.

Por exemplo, a avaliação da expressão a*(b+c) em uma máquina baseada em pilha poderia ser traduzida para o código:

```
push a
push b
push c
add
mult
```

A outra notação, o código de três endereços, é composto por uma sequência de instruções envolvendo operações binárias ou unárias e uma atribuição. O nome "três endereços"está associado à especificação, em uma instrução, de no máximo três variáveis: duas 129 Geração de código e otimização para os operadores binários e uma para o resultado. Assim, expressões envolvendo diversas operações são decompostas nesse código em uma série de instruções, eventualmente com a utilização de variáveis temporárias introduzidas na tradução. Dessa forma, obtém-se um código mais próximo da estrutura da linguagem assembly e, consequentemente, de mais fácil conversão para a linguagem-alvo.

Uma possível especificação de uma linguagem de três endereços envolve quatro tipos básicos de instruções: expressões com atribuição, desvios, invocação de rotinas e acesso indexado e indireto.

A representação interna das instruções em códigos de três endereços dá-se na forma de armazenamento em tabelas com quatro ou três colunas. Na abordagem que utiliza quádruplas (as tabelas com quatro colunas), cada instrução é representada por uma linha na tabela com a especificação do operador, do primeiro argumento, do segundo argumento e do resultado. Por exemplo, a tradução da expressão $a = b + c * d$; resultaria no seguinte trecho da tabela da Figura 10.

	operador	arg 1	arg 2	resultado
1	*	c	d	_t1
2	+	b	_t1	a

Figura 10 – Exemplo de notação de três endereços.

Percebe-se que essa última notação tem uma semelhança muito grande com a estrutura de uma instrução em assembly.

2.1.5 Gerador de código objeto

Esse módulo do compilador utiliza a representação produzida pelo gerador de código intermediário e traduz efetivamente para a arquitetura de destino.

Para se gerar o código objeto, precisa-se saber como a arquitetura de destino funciona, considerando modo de endereçamento e operação, quantidade de memória disponível, além das próprias instruções da arquitetura.

3 Materiais

3.0.1 Processador ENILA

Como linguagem de destino para o meu compilador, será usado a linguagem de máquina do processador ENILA. A arquitetura ENILA foi desenvolvida no laboratório de arquitetura e organização de computadores. O processador foi baseado em uma arquitetura RISC, com implementação de Harvard. Algumas considerações prévias são importantes:

- Cada instrução do processador é uma palavra de 32 bits, contendo uma métrica semelhante, mesmo se forrem de tipos diferentes.
- Possui um banco de registradores com 40 registradores de 16 bits.
- Possui uma cache(ou pode ser entendido como memória principal) de 130KBytes, com cada palavra tendo 2 bytes.
- A arquitetura terá somente endereçamento por registrador. Sendo necessário uma instrução para gravar na memória principal, e uma para carregar da memória (Load/Store).
- O processador possui uma pilha de recursão que permite até 16 chamadas de função sequencialmente.
- O endereçamento para a cache é feita com um endereço relativo a posição da memória da pilha de recursão.

A Figura 11 representa quais tipos de instrução a arquitetura possui.

Tipos de Instrução		
Tipo	Abreviação	Descrição
Logical and Arithmectical Instructions	LAI	Instruções que realizam operações aritméticas ou lógicas. Utiliza endereçamento por gravação direta, e, adicionalmente, pode realizar operações com o conteúdo dos registos ou os números enviados juntamente com- afirmação.
Data Transfer Instructions	DT	Instruções que realizam transferências de dados entre estruturas de armazenamento de dados diferentes.
Control Transfer Instructions	CT	Instrução que desvia o fluxo do programa seguindo alguma comparação ou algo do tipo.
Control Instruction	CI	Instrução que muda as configurações do sistema, ou estados do mesmo.

Figura 11 – Tipos de instrução da ENILA.

O conjunto de instruções da arquitetura é o descrito na Figura 12.

Instruções					
Tipo	**Assembly**	**Descrição**	**Tipo**	**Modo**	**Op**
CT	NOP	NOP	00		0000
CT	JMP	Pula para uma posição do program	00		0001
CT	JPE	Pula se for igual.	00	xx	0010
CT	JPNE	Pula se não fo igual.	00	xx	0011
CT	JPL	Pula se um valor for menor que o outro.	00	xx	0100
CT	JPG	Pula se um valor for maior que o outro.	00	xx	0101
CT	JPLE	Pula se um valor for maior que o outro.	00	xx	0110
CT	JPGE	Pula se um valor for maior que o outro.	00	xx	0111
DT	SRVALUE	Grava um valor a um registrador.	01	00	0000
DT	LOAD	Carrega um dado da memória para o registrador.	01	00 ou 11	0001
DT	STORE	Store data in memory	01	-	0010
DT	REGCOPY	Copia o registrador para outra posição	01	00	0011
DT	GET_IN	Pega um input da chave e grava no registrador $3	01	00	0100
LAI	OR	Porta lógica OR	10	xx	0000
LAI	AND	Porta lógica AND	10	xx	0001
LAI	NOR	Porta lógica NOR	10	xx	0010
LAI	NAND	Porta lógica NAND	10	xx	0011
LAI	XOR	Porta lógica XOR	10	xx	0100
LAI	ADD	Op Adição	10	xx	0101
LAI	SUB	Op Subtração	10	xx	0110
LAI	MUL	Multiplicação sem sinal	10	xx	0111
LAI	DIV	Divisão sem sinal	10	xx	1000
LAI	SHR	Desloca bit para a direita	10	xx	1001
LAI	SHL	Desloca bit para a esquerda	10	xx	1010
SCI	RST	Reinicia o microcontrolador.	11	-	0000
CI	PUSH.R	Empilha endereço na pilha de recurssão.	11	-	0001
CI	POP.R	Desempilha endereço rel. da pilha de recursão	11	-	0010
CI	PUSH.PC	Empilha PC na pilha de recurssão.	11	-	0011
CI	POP.PC	Desempilha PC da pilha de recursão	11	-	0100

Figura 12 – Instruções da ENILA.

As instruções que tem o campo modo com "XX", permitem usar um imediato. A Figura 13 descreve como funciona esse comportamento.

O *datapath* do processador está anexado ao projeto.

Modo	Description
00	Primeiro operando e o segundo são endereços do registrador.
01	Primeiro operando é endereço do registrador e o segundo é imediato.
10	Primeiro operando é imediato e o segundo é endereço do registrador.
11	Primeiro operando e o segundo são imediatos.

Figura 13 – Modos com que as instruções com modo xx podem se comportar.

4 Desenvolvimento

O projeto foi dividido entre os 5 módulos vistos na fundamentação teórica. Analisador Léxico, Sintático, Semântico e Gerador de Código Intermediário.

4.1 Analisador Léxico

O analisador léxico foi feito utilizando Flex, e primeiramente definiu-se qual a gramatica do C- e seus símbolos. Tem-se então a seguinte convenção léxica para o C-:

```
palavras-chave:
else if int return void while
simbolos especiais:
+ - * / < <= > >= == != = ; , ( ) [ ] { }
Identificadores:
ID = letra+
Numeros:
NUM = digito+
Comentarios entre /* */
```

Com isso, conseguimos definir as expressões regulares que reconhecem os tokens:

```
digito [0-9]
number {digito}+
letra [a-zA-Z]
ID {letra}+({letra}|{digito})*
newline \n
carriage \r
whitespace [ \t]+
other [^0-9a-zA-Z;/=\-"+""*""("")""\n"\[\]\,\{\}\<\>|!=\==\<=\>=]
```

Com o Flex é possivel usar a linguagem C para criar funções auxiliares como contar a quantidade de tokens e até mesmo retornar erros. Então para o compilador, foi feito o seguinte código complementar:

```
%%

"if"            {if(ScannerFeedback) printf("IF\n");return IF;}
"else"          {if(ScannerFeedback) printf("ELSE\n");return ELSE
   ;}
"int"                   {if(ScannerFeedback) printf("INT\n");
   return INT;}
```

```
"return"                {if(ScannerFeedback) printf("RETURN\n");
   return RETURN;}
"void"                  {if(ScannerFeedback) printf("VOID\n");
   return VOID;}
"while"                 {if(ScannerFeedback) printf("WHILE\n");
   return WHILE;}
"="             {if(ScannerFeedback) printf("ASSIGN\n");return
   ASSIGN;}
"=="            {if(ScannerFeedback) printf("EQUAL\n");return
   EQUAL;}
"!="                    {if(ScannerFeedback) printf("NEQUAL\n");
   return NEQUAL;}
"<"             {if(ScannerFeedback) printf("LESS\n");return LESS
   ;}
"<="                    {if(ScannerFeedback) printf("LESSEQ\n");
   return LESSEQ;}
">"                             {if(ScannerFeedback) printf("GREAT
   \n");return GREAT;}
">="                    {if(ScannerFeedback) printf("GREATEQ\n");
   return GREATEQ;}
"+"             {if(ScannerFeedback) printf("ADD\n");return ADD;}
"-"             {if(ScannerFeedback) printf("SUB\n");return SUB;}
"*"             {if(ScannerFeedback) printf("MULT\n");return MULT
   ;}
"/"             {if(ScannerFeedback) printf("DIV\n");return DIV;}
"("             {if(ScannerFeedback) printf("LPARENTS\n");return
   LPARENTS;}
")"             {if(ScannerFeedback) printf("RPARENTS\n");return
   RPARENTS;}
";"             {if(ScannerFeedback) printf("SEMICOLON\n");return
   SEMICOLON;}
","                             {if(ScannerFeedback) printf("COMMA
   \n");return COMMA;}
"["                             {if(ScannerFeedback) printf("
   LSQRBRA\n");return LSQRBRA;}
"]"                             {if(ScannerFeedback) printf("
   RSQRBRA\n");return RSQRBRA;}
"{"                             {if(ScannerFeedback) printf("
   LCURBRA\n");return LCURBRA;}
"}"                             {if(ScannerFeedback) printf("
   RCURBRA\n");return RCURBRA;}
{number}        {if(ScannerFeedback) printf("NUMERO \n"); strcpy(
   ids,yytext);return NUMERO;}
{ID}            {if(ScannerFeedback) printf("ID\n"); strcpy(ids,
   yytext); return ID;}
{whitespace}    {/*tava dando problema no newline*/}
{newline}               { lineno++; if(ScannerFeedback) printf("\t
```

```
   %d\n", yylineno);}
{carriage}              { }
"/*"            {       char c, d;
                        c = input();
                        do
                        {       if (c == EOF) break;
                                d = input();
                                if (c == '\n')
                                        lineno++;
                                if (c == '*' && d == '/') break;
                                c = d;
                  } while (1);
                }
{other}                 {printf("Lexical error on line: %d\n",
   yylineno); return ERROR;}
<<EOF>>         return(0);
%%
```

Esse código em especifico acima retorna o retorno das identificações. O resto do código estará no Apêndice.

Importante destacar que no analisador léxico, ao identificar um token, é armazenado em uma variável global o numero da linha deste token.

4.2 Analisador Sintático

Para se elaborar a análise sintática foi utilizado o Yacc/Bison para poder gerar a árvore a partir do BNF e das regras de construção.

O gerador de analisador sintático funciona de maneira similar o gerador de analisador léxico. Existe um *header* no qual define-se as funções externas e define-se as estruturas, a seção com as regras em formato BNF e ações em C executadas quando a regra é reconhecida, e uma seção no fim com as funções e procedimentos.

Primeiramente precisa-se definir a estrutura de árvore e quais os seus campos. Abaixo está como ficou a estrutura do nó de árvore de análise sintática em C:

```
/* ExpType is used for type checking */
typedef enum {Void,Integer} ExpType;
typedef enum {Call,FuncDecl,VarDecl,Var,VectorDecl,VectorPos,
   FuncAttrVar, FuncAttrVector} IDType;

typedef struct treeNode
   { struct treeNode * child[MAXCHILDREN];//maximo de filhos do no
     struct treeNode * sibling;//ponteiro para o no irmao
     int lineno;
         int codeGen;
```

```
    NodeKind nodekind;
    union { StmtKind stmt; ExpKind exp;} kind;
    union { int op;
            int val;
            char * name; } attr;//tipo de atributo do no se eh
               nome, um token ou valor
  ExpType type; /* for type checking of exps */ //preenche so
     quando DeclType
      IDType id_type;
      char * scope;
      int scope_number;
} TreeNode;
```

Definiu-se o valor máximo de "filhos"que cada nó poderá ter como 3. Cada nó possui um "irmão". Importante ressaltar que na árvore tem um campo de armazena o valor se for inteiro ou o nome se for um ID. Um campo chamado $scope_number$ grava o numero de escopo que é atribuído.

O nó pode ser do tipo $StmtK$ ou $ExpK$. $StmtK$ são nós que referenciam um *statement*, como por exemplo If,$While$, $Assign$, $Return$. Já $ExpK$ pode ser operação, número ou ID(declaração de função ou variável, chamada de função, ou apenas a variável mesmo). O caso do ID é um caso com várias situações, por isso cada nó que seja uma ID pode ter um valor de $IDType$ diferente. Esses valores diferenciam uma ID em, chamada de função($Call$), declaração de função ($FuncDecl$),declaração de variável ($VarDecl$), variável (Var), declaração de vetor($VectorDecl$), variável do tipo vetor($VectorPos$), parâmetro de função do tipo variável simples ($FuncAttrVar$) e parâmetro de função do tipo vetor ($FuncAttrVector$).

Perceba que cada ID também terá um valor de $ExpType$ no qual cada valor representa um tipo de dado. Existem dois tipos somente no C-, inteiro ($Integer$) ou *void*($Void$).

Para saber em qual linha ocorreu um determinado nó, tem-se na estrutura do nó da árvore a variável *lineno* que durante as derivações da árvore guarda qual o número da linha de acordo com a informação dos tokens fornecidos pela análise léxica.

Então foi escrito o BNF do C- juntamente com as rotinas em C para criar a árvore de análise sintática. Segue abaixo a definição do BNF do C-:

```
programa -> declaracao-lista
declaracao-lista -> declaracao-lista declaracao | declaracao
declaracao -> var-declaracao | fun-declaracao
var-declaracao -> tipo-especificador ID ; | tipo-especificador ID [ NUM ] ;
tipo-especificador -> int | void | float
fun-declaracao -> tipo-especificador ID ( params ) composto-decl
```

```
params -> param-lista | void
param-lista -> param-lista , param | param
param -> tipo-especificador ID | tipo-especificador ID [ ]
composto-decl -> { local-declaracoes statement-lista }
local-declaracoes -> local-declaracoes var-declaracao | vazio
statement-lista -> statement-lista statement | vazio
statement -> expressao-decl | composto-decl | selecao-decl |
             iteracao-decl | retorno-decl
expressao-decl -> expressao ; | ;
selecao-decl -> if ( expressao ) statement |
                if ( expressao ) statement else statement
iteracao-decl -> while ( expressao ) statement
retorno-decl -> return ; | return expressao;
expressao -> var = expressao | simples-expressao
var -> ID | ID [ expressao ]
simples-expressao -> soma-expressao relacional soma-expressao |
                     soma-expressao
relacional -> <= | < | > | >= | == | !=
soma-expressao -> soma-expressao soma termo | termo
soma -> + | -
termo -> termo mult fator | fator
mult -> * | /
fator -> ( expressao ) | var | ativacao | NUM
ativacao -> ID ( args )
args -> arg-lista | vazio
arg-lista -> arg-lista , expressao | expressao
```

Vamos utilizar as derivações do BNF para gerar a arvore.

```
declaracao_lista: declaracao_lista declaracao
                                    {
                                            if (productionFeedBack)
                                               printf("
                                               declaracao_lista->
                                               declaracao_lista
                                               declaracao .\n");
                                            YYSTYPE t = $1;
                                            if (t!=NULL){
                                                  while(t->sibling
                                                     != NULL) t = t
                                                     ->sibling;
                                            t->sibling = $2;
                                            $$ = $1;
```

```
                                        }else{
                                                $$ = $2;
                                        }
                                }
                        |       declaracao
                                {
                                        if (productionFeedBack)
                                            printf("
                                            declaracao_lista->
                                            declaracao .\n");
                                        $$ = $1;
                                }
```

Abaixo coloca-se as declarações como irmão.

```
programa: declaracao_lista
                                {
                                if (productionFeedBack) printf("
                                    programa-> declaracao_lista .\n
                                    ");
                                savedTree = $1;
                                }
```

Nessa derivação percebe-se que a declaração pode ser uma declaração de variável ou de função. Assim, é inserido a derivação na raiz.

```
declaracao: var_declaracao
                                {
                                        if (productionFeedBack)
                                            printf("declaracao->
                                            var_declaracao .\n");
                                        $$ = $1;
                                }
                        | fun_declaracao
                                {
                                        if (productionFeedBack)
                                            printf("declaracao->
                                            fun_declaracao .\n");
                                        $$ = $1;
                                        savedFunction = "Global";
                                }
```

No Apêndice desse relatório é possível achar o código com todas as derivações e qual o comportamento para gerar a árvore de análise sintática. Pode-se resumir e ilustrar o comportamento desse trecho de código com as seguintes especificações para a criação da árvore:

- Quando existe uma sequencia de declarações (sendo que o BNF só permite que as

declarações ocorram no inicio da função ou do código caso seja uma declaração global ou declaração de função) os nó com o tipo da função (*Integer* ou *Void*) são irmãos entre si, com o ID no nó abaixo de cada um, e o restante do código abaixo de cada ID ou como irmão do "tipo"no final. A Figura 11 ilustra a situação.

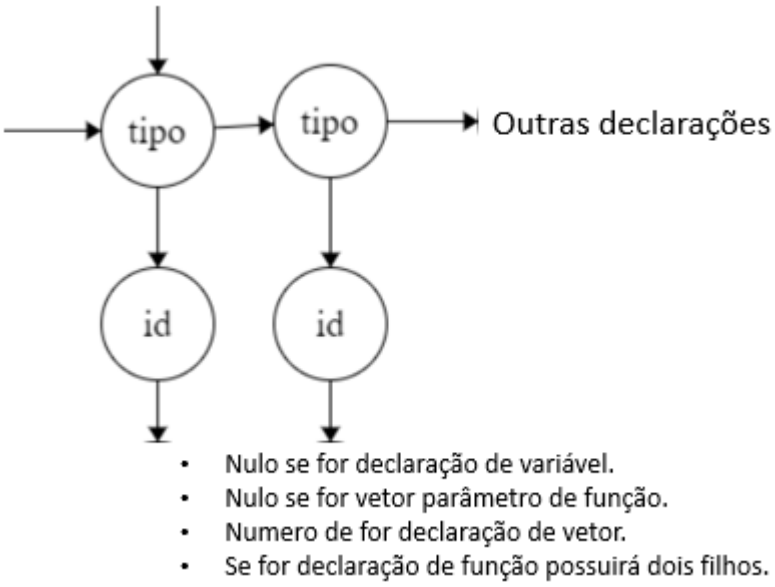

Figura 14 – Estrutura da arvore com as declarações.

- Se a declaração for uma declaração de função, o seu primeiro filho abaixo do ID conterá uma "lista"com as declarações seguindo o mesmo padrão apresentado acima, e o segundo filho possuirá o resto do código.

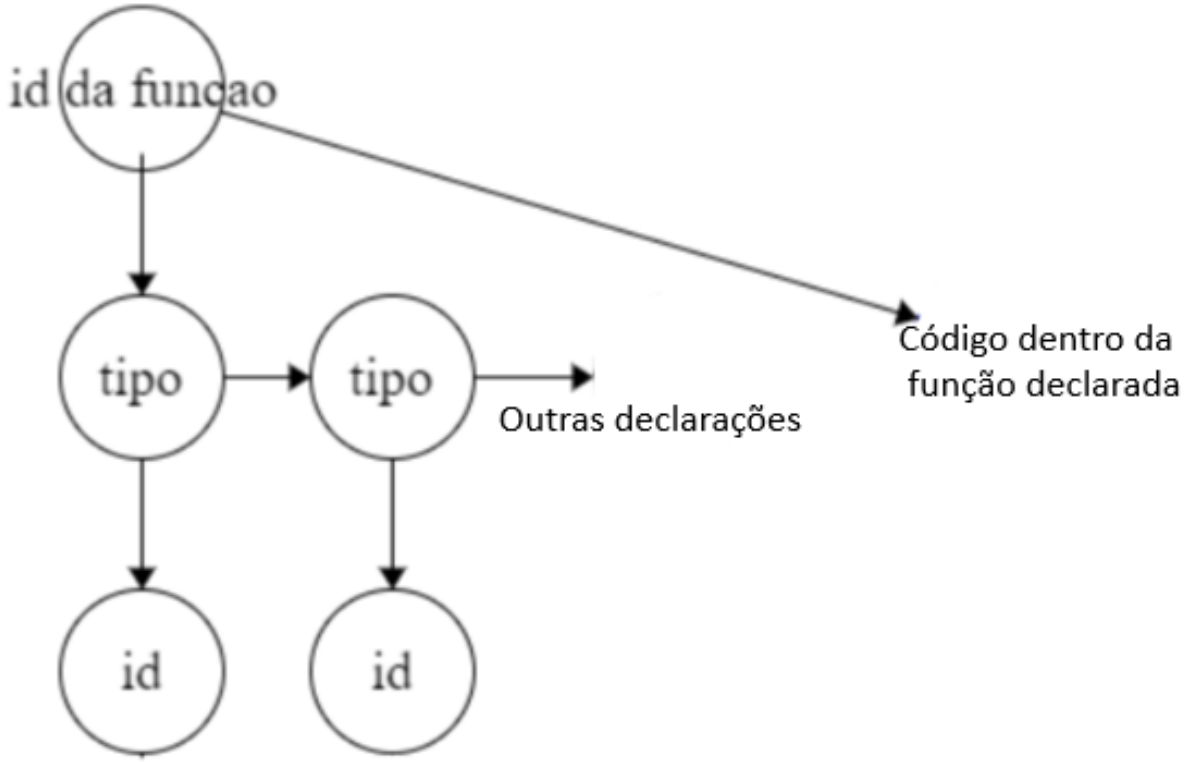

Figura 15 – Declarações de funções.

- Toda expressão que estiver na mesma "identação"do código deverá ser uma "irmã"da outra. Ou seja, todas as expressões que estiverem em um mesmo escopo deverão estar conectadas polo nó irmão.

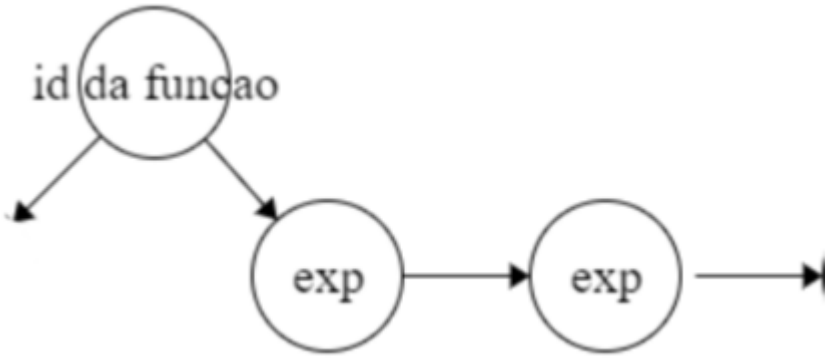

Figura 16 – Expressões em um mesmo escopo.

Quando entra em um if ou *while* é encarado como troca de escopo, e essa regra se mantém.

- Quando há um if, a expressão condicional entra no primeiro filho, a expressão caso a condição seja verdadeira entra no segundo filho, e caso seja falso e tenha um else entra no terceiro filho.

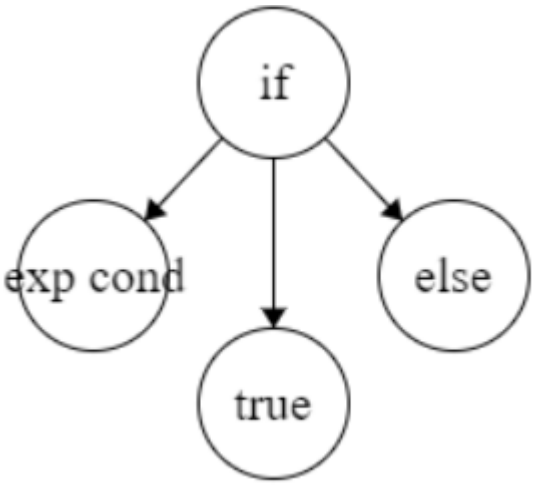

Figura 17 – Estrutura da arvore com if.

- No caso de uma operação, a operação entra como a raiz, o primeiro filho é a variável de destino, e o segundo filho é a expressão fonte.

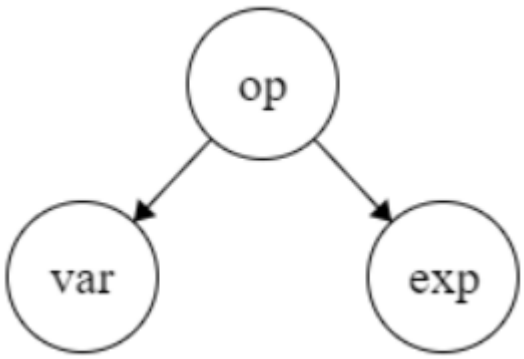

Figura 18 – Estrutura do nó da árvore que contém uma operação.

- Se o nó for um *While* então a expressão condicional entra no primeiro filho e expressão, caso seja verdade, no segundo filho.

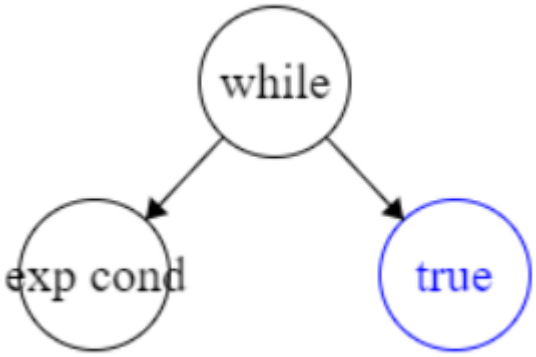

Figura 19 – Estrutura do nó da árvore que contém um *while*.

- Quando o nó for um *Return* então a expressão de retorno será o primeiro filho.

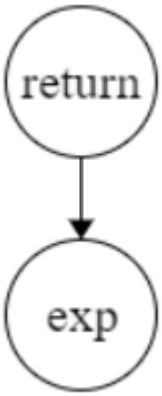

Figura 20 – Estrutura do nó da árvore que contém um *return*.

- Se o nó for uma chamada de função então a "lista"de argumentos ficará no primeiro filho do nó, conforme a Figura 21 ilustra.

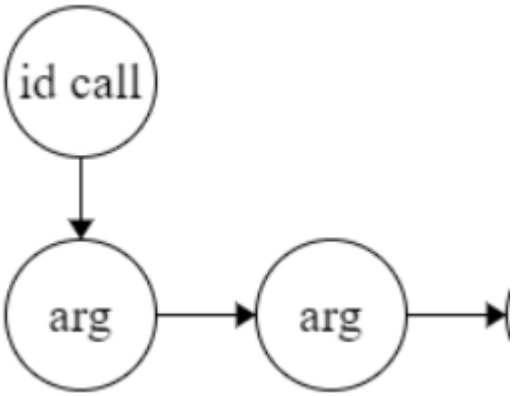

Figura 21 – Estrutura do nó da árvore que contém uma chamada de função.

Com todas essas regras em mente pode-se então criar a árvore de análise sintática diretamente nas produções. Para o compilador desenvolvido, por exemplo, para o código abaixo:

```
int gcd(int u, int v)
{ if (v==0) return u;
else return gcd(v, u-u/v*v);
}
int input(void){
        int x;
        return x;
}
void output(int x){

}
void main(void)
{ int x; int y;
x = input(); y = input();
output(gcd(x,y));
}
```

Obtém, com este compilador, a árvore de análise sintática conforma mostrado na Figura 22.

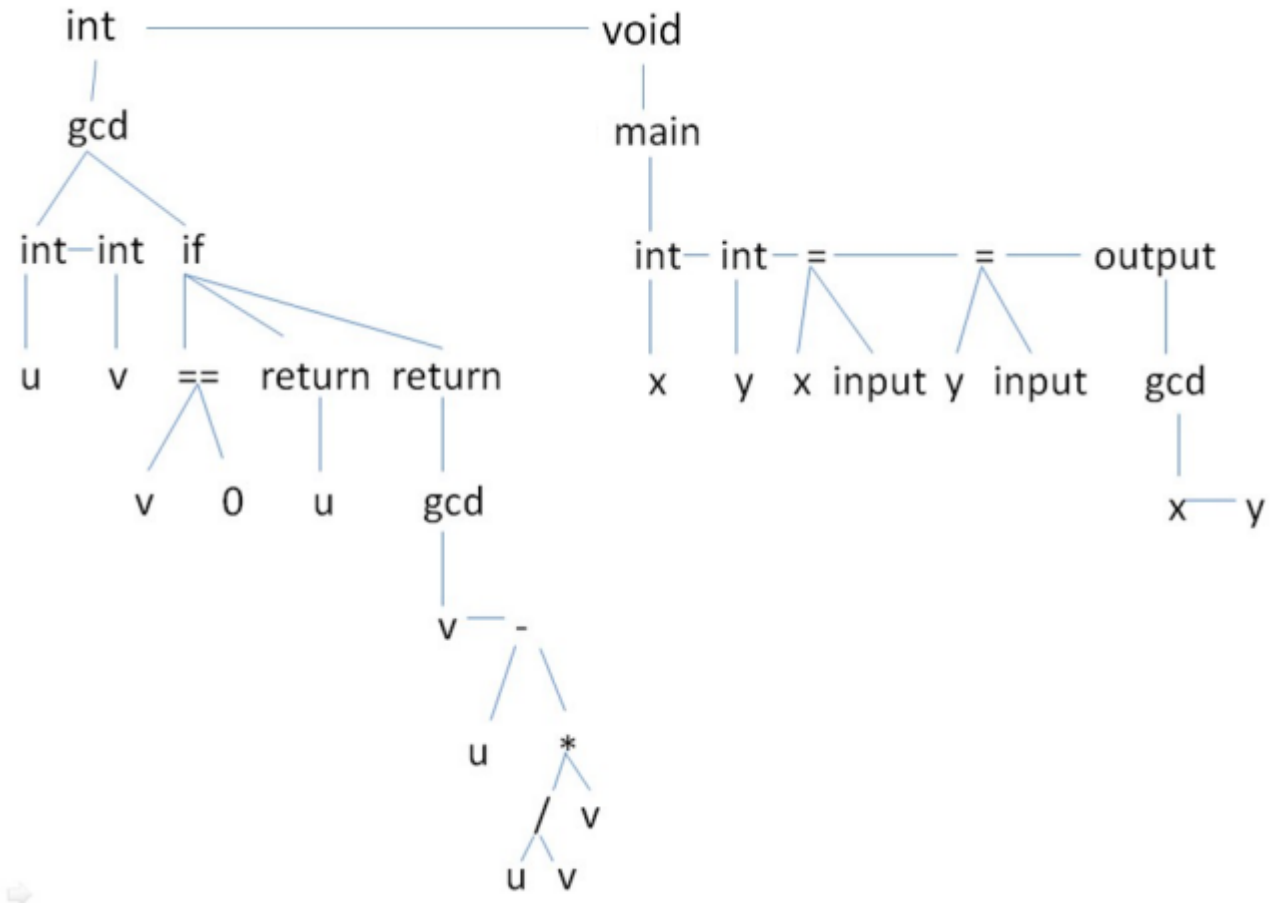

Figura 22 – Estrutura da arvore com as expressões.

Perceba que essa árvore é ligeiramente diferente da árvore proposta pela disciplina de Compiladores. Essa árvore possui todas as declarações de função encadeadas como uma lista. Isso foi feito para garantir uma consistência, pois assim, as regras de formação da árvore podem ser aplicadas para todos os casos, veja que se encaixa na regra das expressões de um mesmo contexto. Nesse caso, especificamente, a declaração *gcd* e *main* estão no mesmo contexto global.

Após gerar a árvore, percebeu que houve alguns problemas com os escopos e e tipos dos IDs, então criou-se funções para percorrer a árvore atualizando esses parâmetros. Sabe-se que esse não é o modelo ideal, e que foi uma alternativa a curto prazo para essa disciplina, podendo ocorrer mudanças nos próximos semestres. Nessa segunda passagem, é dado um "número de escopo"($scope_number$) para cada ID. Quando a ID está no escopo global, o número é 0; quando a ID está em qualquer outro escopo ela tem múltiplos de 200 como numero de escopo (o múltiplo depende da ordem de aparição da função) e esse escopo onde ela está inserida é chamado escopo base;para cada *while* é adicionado 20 ao numero de escopo, e para cada if é adicionado 1.

A arvore do código é gerada em um arquivo chamado "*output.txt*, e ela tem o formato por identação, onde cadda identação representa um filho. Esse formato não foi o melhor possível para *debug*, mas foi útil para visualizar casos específicos. Para exemplificar, segue abaixo a árvore de análise sintática gerada pelo analisador para o mesmo código *gcd* mostrado acima.

```
Tipo de ID: int
        Id: gcd do escopo Global do tipo decl funcao.
            Tipo de ID: int
                Id: u do escopo gcd do tipo param func var.
            Tipo de ID: int
                Id: v do escopo gcd do tipo param func var.
            if
                Op: ==
                    Id: v do escopo gcd do tipo variavel.
                    Numero: 0
                return
                    Id: u do escopo gcd->if0 do tipo variavel.
                return
                    Id: gcd do escopo gcd->else0 do tipo cham func.
                        Id: v do escopo gcd->else0 do tipo variavel.
                        Op: -
                            Id: u do escopo gcd->else0 do tipo variavel.
                            Op: *
```

```
                                    Op: /
                                        Id: u do escopo gcd->else0 do tipo
                                            variavel.
                                        Id: v do escopo gcd->else0 do tipo
                                            variavel.
                                    Id: v do escopo gcd->else0 do tipo
                                        variavel.
Tipo de ID: int
    Id: input do escopo Global do tipo decl funcao.
        Tipo de ID: int
            Id: x do escopo input do tipo decl var.
        return
            Id: x do escopo input do tipo variavel.
Tipo de ID: void
    Id: output do escopo Global do tipo decl funcao.
        Tipo de ID: int
            Id: x do escopo output do tipo param func var.
Tipo de ID: void
    Id: main do escopo Global do tipo decl funcao.
        Tipo de ID: int
            Id: x do escopo main do tipo decl var.
        Tipo de ID: int
            Id: y do escopo main do tipo decl var.
         =
            Id: x do escopo main do tipo variavel.
            Id: input do escopo main do tipo cham func.
         =
            Id: y do escopo main do tipo variavel.
            Id: input do escopo main do tipo cham func.
        Id: output do escopo main do tipo cham func.
            Id: gcd do escopo main do tipo cham func.
                Id: x do escopo main do tipo variavel.
                Id: y do escopo main do tipo variavel.
```

Perceba que foram adicionados as funções *input* e *output* na declaração, pois se não faz isso o analisador semântico, que é abordado na próxima sessão identifica erro. Para mais informações do código do analisador sintático, veja o Apêndice.

4.3 Analisador Semântico

Durante a análise semântica é gerada uma tabela de símbolos que guarda as informações das IDs que estão no código. Informações como nome da variável, nó da árvore da variável, tipo de ID, escopo da variável, numero do escopo, linhas que a ID apareceu, número total de aparições, e tamanho (caso seja um vetor).

Como era de se esperar, essa tabela é um vetor de estruturas. A definição dessa estrutura de dado pode ser vista no código abaixo:

```
typedef struct symbol{
        char *ID;
        IDType id_type;
        char *data_type;
        int index;
        char *scope;
        int lines[50];
        int top;
        int mem_add; //memLoc
        int size;
        int im_add;      // lineLocAssembly
        TreeNode *node;
        struct symbol *nxt;
} Symbol;

typedef struct {
  Symbol *begin;
}SymList;

SymList HashVec[hash_size];
```

Resumidamente pode-se simplificar a definição dessa estrutura como um vetor de listas.

Para saber qual posição acessar é utilizado uma função de *hash*, e determina-se o valor máximo do *hash* como sendo 211, consequentemente a tabela tem 211 posições. A função que calcula o *hash* é descrita com a seguinte função:

```
int hash_calc(char *nameID) {
        int key = 0;
        int i;
        for(i = 0; i < strlen(nameID)+1; i++) {
                key = ((key << 4) + nameID[i])%hash_size;
        }
        if(TabGenFeedBack) printf("Chave calculada: %d - %s\n",key
           ,nameID);
        return key;
}
```

Durante a análise semântica são realizadas três passagens na árvore. Na primeira passagem são verificados os seguintes erros:

- A 'main' nao pode ser chamada recursivamente.
- Função x não declarada.
- Falta o atributo x do tipo y na chamada z.
- Excesso de atributo x do tipo y na chamada z.
- Existe uma variável x declarada como Void.
- Existe um id_type x declarado com o mesmo nome.
- Existe um vetor x declarado como Void.
- O(a) id_type está sendo utilizado porém não foi declarado(a).
- O(a) x é vetor quando era para ser variável.
- O(a) x é variável quando era para ser vetor.
- O vetor está sendo acessado em uma posição maior que a declarada.
- O vetor está sendo acessado em uma posição maior que a declarada.

Após essa primeira verificação, são verificadas as operações, e então verifica o seguinte erro:

- A operação x está sendo realizada entre dois operandos de tipos diferentes de dados. y com tipo y_tipo e z com tipo z_tipo.

Por último são verificados os $Assign$, ou simplesmente =. Os seguintes erros podem ser retornados nessa terceira passagem:

- Está associando uma função que retorna 'void' para uma variável.
- O valor retornado sera truncado (tipos diferentes de dados).
- Não deveria existir return para a função do tipo void.

Todas os erros, caso ocorram, são notificados devidamente, juntamente com a linha do código onde ocorreu o erro.

Na primeira passagem pela arvore na análise semântica, verifica-se os IDs apenas, e portanto, quando não é acusado erro, insere-se o ID como um símbolo na tabela de símbolos com todos os atributos discutidos no início da sessão.

Depois de terminado e ter tudo funcionando, verificou-se que tomar os if, $else$ e $while$ como troca de escopo foi uma péssima ideia. Ao idealizar a análise sintática não foi atentado para o fato de que todas as declarações ocorrem sempre no inicio da função ou como variável global. Dessa forma, não faz diferença tomar uma variável dentro de um $statement$ desses como em outro escopo. Em C isso ocorre, pois pode-se declarar uma variável dentro de um if, e ela estar somente no contexto do if .

Para corrigir isso, ao final realiza-se uma passagem pela tabela de símbolos inteira, retirando esses escopos, e considerando o escopo base. Por exemplo, se uma variável x está no escopo $main->if0$, essa passagem transforma esse escopo em $main$ apenas.

Depois realiza-se outra passagem pela tabela para unir as variáveis que possuem um mesmo escopo em um único símbolo. Essa passagem serve para corrigir o resultado da última passagem, onde irá gerar dois símbolos com o mesmo tipo, nome, escopo, porém em posições diferentes.

Essas duas passagens a mais são claramente desnecessárias, mas foi uma forma rápida de resolver os problemas. Tendo em vista disso sabe-se que será preciso resolver esse problema de forma mais eficaz.

Para mais detalhes de implementação veja o Apendice com os códigos.

4.4 Geração de Código Intermediário

Para realizar o código intermediário foi o estipulado como notação o código de três endereços. Tal notação foi escolhida por se aproximar do utilizado pelo código de máquina da arquitetura.

Para gerar o código de máquina, percorre-se a árvore de análise sintática por inteiro novamente, verificando os nós do tipo $StmtK$ e do tipo $ExpK$.

Nessa fase não são realizados gerenciamentos de memória tão profundos, o único gerenciamento realizado é o de temporários para as operações. Nesse caso, toda expressão formada por operações sempre leva para um temporário e depois esse temporário é dado um $assign$ a uma variável. Esse valor de temporário é incrementado sempre, pois seu numero exato não importa, basta saber que nunca existirão dois temporário pendentes, ou seja, para serem "utilizados". Esse conhecimento será importante para gerar o código de máquina.

O código intermediário possui os seguintes acrônimos, com a suas descrições respectivamente:

- $fundef$:Representa a declaração de função, sempre será apresentado como $(fundef, ID, -, -)$.

- *add*:Representa a operação de soma, sempre será apresentado como (*add*, *operando*1, *operando*2, *destin*
  Os operandos podem ser uma ID, um temporário, representado por $TEMP - X$,
  ou um retorno de função , representado por RET.

- *sub*:Representa a operação de subtração, sempre será apresentado como (*sub*, *operando*1, *operando*2, *de*
  Os operandos podem ser uma ID, um temporário, representado por $TEMP - X$,
  ou um retorno de função , representado por RET.

- *mul*:Representa a operação de multiplicação, sempre será apresentado como (*sub*, *operando*1, *operando*
  Os operandos podem ser uma ID, um temporário, representado por $TEMP - X$,
  ou um retorno de função , representado por RET.

- *div*:Representa a operação de divisão, sempre será apresentado como (*sub*, *operando*1, *operando*2, *desti*
  Os operandos podem ser uma ID, um temporário, representado por $TEMP - X$,
  ou um retorno de função , representado por RET.

- *eq*:Representa a operação de "igual a", sempre será apresentado como (*eq*, *operando*1, *operando*2, *destin*
  Os operandos podem ser uma ID, um temporário, representado por $TEMP - X$,
  ou um retorno de função , representado por RET.

- *neq*:Representa a operação de "diferente de", sempre será apresentado como (*neq*, *operando*1, *operando*
  Os operandos podem ser uma ID, um temporário, representado por $TEMP - X$,
  ou um retorno de função , representado por RET.

- *lt*:Representa a operação de "menor que", sempre será apresentado como (*lt*, *operando*1, *operando*2, *dest*
  Os operandos podem ser uma ID, um temporário, representado por $TEMP - X$,
  ou um retorno de função , representado por RET.

- *gt*:Representa a operação de "maior que", sempre será apresentado como (*gt*, *operando*1, *operando*2, *des*
  Os operandos podem ser uma ID, um temporário, representado por $TEMP - X$,
  ou um retorno de função , representado por RET.

- *leq*:Representa a operação de "menor ou igual que, sempre será apresentado como (*leq*, *operando*1, *operando*2, *destino*). Os operandos podem ser uma ID, um temporário, representado por $TEMP - X$, ou um retorno de função , representado por RET.

- *geq*:Representa a operação de "maior ou igual que, sempre será apresentado como (*geq*, *operando*1, *operando*2, *destino*). Os operandos podem ser uma ID, um temporário, representado por $TEMP - X$, ou um retorno de função , representado por RET.

- $assign$:Representa a atribuição de um valor a outro, ou seja, o operador =. Sempre será apresentado como $(assign, destino, origem, -)$. Os operandos de destino obrigatoriamente são IDs. Enquanto a origem pode ser uma ID, um temporário, representado por $TEMP - X$, ou um retorno de função , representado por RET.

- $begin - args$:Representa que será iniciado uma chamada de função assim como o envio de seus argumentos. Sempre será apresentado como $(begin - args, -, -, -)$.

- arg:Representa que é um argumento a ser salvo para a chamada de função que vier a acontecer. Sempre será apresentado como $(arg, operando1, -, -)$.O operando pode ser uma ID, um temporário, representado por $TEMP - X$, ou um retorno de função , representado por RET.

- $funcal$: Representa a própria chamada de função. Sempre será apresentado como $(funcal, ID, -, -)$.

- ret: Representa o retorno da função. Sempre será apresentado como $(ret, operando1, -, -)$.O operando pode ser uma ID, um temporário, representado por $TEMP - X$, ou um retorno de função , representado por RET.

- $if - false$: Representa um *jump* condicional. Sempre será apresentado como $(if - false, TEMP - X, LX, -)$. Assim, essa representação diz que se o resultado da operação for falso então pula para a label LX.

- jmp: Representa um *jump* . Sempre será apresentado como $(jmp, LX, -, -)$. Assim, essa representação diz que se é para pular para a label LX.

- lb: Representa uma *label* . Sempre será apresentado como $(lb, LX, -, -)$.

Depois de definidos os acrônimos, a seguir apresenta-se algumas regras para a construção do código intermediário:

- Toda chamada de função terá como rotina padrão as seguintes instruções $begin-args$, arg(na quantidade de variáveis que são passadas como parâmetro) e $funcal$. Percebe-se que se existir uma expressão que define um argumento, essa expressão estará incluída entre o $begin - args$ e o $funcal$.

- Todo if ou $while$ do código terá como dupla as instruções eq,neq,leq,geq,lt, mt com a $if - false$ logo em seguida.

- Logo após a funcal, se o a chamada de função atribuir um valor a alguma variável, então atribui-se o RET, que é o temporário de retorno.

Um ponto importante de se destacar é de que não haverá as instruções *eq*,*neq*,*leq*,*geq*,*lt*, *mt*, assim como $if - false$, dentro da "rotina"de chamada de função.

Assim como feito anteriormente, se gerarmos o código intermediário para o código "gcd", usado nas sessões acima, obteremos o seguinte resultado:

```
 0: (fundef,gcd,_,_)
 1: (eq,v,0,_TEMP0_)
 2: (if_false,_TEMP0_,L0,_)
 3: (ret,u,_,_)
 4: (jmp,L1,_,_)
 5: (label,L0,_,_)
 6: (begin_args,_,_,_)
 7: (arg,v,_,_)
 8: (div,u,v,_TEMP1_)
 9: (mul,_TEMP1_,v,_TEMP2_)
10: (sub,u,_TEMP2_,_TEMP3_)
11: (arg,_TEMP3_,_,_)
12: (funcal,gcd,_,_)
13: (ret,_RET_,_,_)
14: (label,L1,_,_)
15: (fundef,input,_,_)
16: (ret,x,_,_)
17: (fundef,output,_,_)
18: (fundef,main,_,_)
19: (funcal,input,_,_)
20: (assign,x,_RET_,_)
21: (funcal,input,_,_)
22: (assign,y,_RET_,_)
23: (begin_args,_,_,_)
24: (begin_args,_,_,_)
25: (arg,x,_,_)
26: (arg,y,_,_)
27: (funcal,gcd,_,_)
28: (arg,_RET_,_,_)
29: (funcal,output,_,_)
```

Para mais informações de implementação consulte o Apendice.

4.5 Geração de Código Objeto

Nessa sessão apresenta a fase final de compilação do código, onde é analisado a tabela de símbolos gerada no analisador semântico e o código intermediário a fim de se gerar o código para a arquitetura ENILA.

A arquitetura ENILA já possuía uma pilha de recursão implementada, tendo instruções que manipulam a pilha livremente. Dessa forma, ao gerar o código de máquina já tem-se como vantagem o fato de a memória ser relativa. Ou seja, a troca de contexto é feita automaticamente no hardware.

Uma técnica utilizada foi processar o código intermediário utilizando uma tabela de memória e de registradores virtual. Como o endereçamento de memória é relativo, é possível percorrer o código fazendo essa análise.

Então foram utilizadas três estruturas auxiliares básicas:

- Tabela de memória.
- Tabela de registradores.
- Pilha de registradores livres.

A tabela de memória possui os seguintes campos:

- Nome da variável.
- Número de ocorrências no escopo (número de vezes que a variável é utilizada).
- Numero do registrador atribuído a variável.
- Tamanho da variável no escopo. (Maior que 1 se for vetor)
- Primeira linha de ocorrência da variável no escopo.

A tabela de registradores possui os seguintes campos:

- Posição da tabela de variáveis, da variável que está ocupando o registrador.
- Campo que diz se o registrador está livre ou não.

A pilha possui um único campo que é o número do registrador livre.

A tabela de memória é dinamicamente alocada. Por padrão é inicializada com nenhuma posição. Enquanto a tabela de registradores é alocada estaticamente pois é

conhecido o número de registradores da arquitetura, que são 40 registradores. E a pilha como era de se esperar é inicializada com nada nela.

Antes de compreender o algoritmo por trás da geração de código de máquina, é importante destacar que os registradores não são todos de uso geral. Ao se pensar no algoritmo decidiu-se reservar 8 registradores para uso exclusivo do compilador para diversas operações. Abaixo estão os registradores reservados e uma breve introdução para a sua importância:

Registrador	Nome	Função
$0	Return	Serve para armazenar o valor de retorno de uma função.
$1	Result	Esse registrador é utilizado pelo compilador para sequências de instruções que são executadas em série e são sempre iguais.
$2	Operand_1	Todo primeiro operando de qualquer operação é carregado para esse registrador.
$3	Operand_2	Todo segundo operando de qualquer operação é carregado para esse registrador.
$4	Temp_1	O compilador gerencia para qual registrador vai ser associado os temporários gerados no código intermediário
$5	Temp_2	
$6	Base_Vetor	Aqui será armazenado o endereço base do vetor pelo próprio compilador.
$7	Reserved	Reservado para futuras implementações
$8	GPR 8	Registradores de uso gerais, são utilizados para armazenar os valores da memória.
...	...	
$39	GPR 39	

Figura 23 – Descrição dos registradores reservados pelo compilador.

Algumas outras considerações precisam ser levadas em consideração. Foram implementadas rotinas dinâmicas que simulam o armazenamento e carregamento da memória.

Existe um apontador que indica em que registrador deve ser armazenado a memória. Esse apontador incrementa cada vez que alguma instrução de LOAD aramzena um dado na posição do apontador. Essa decisão foi baseada na ideia de FIFO(First In First Out). Porém nem sempre um dado da memória irá ser carregado nessa posição do apontador, pois se houve um STORE que liberou um espaço no registrador fora do local aonde está sendo apontado, irá empilhar esse valor na pilha de registradores livres. Então ao dar LOAD de uma posição de memória, o compilador irá verificar qual registrador está livre,

dando prioridade para a pilha de registradores livres e depois para o apontador (ao dar LOAD e existir um registrador na pilha, o registrador de destino é desempilhado).

Para cada LOAD e STORE realizado, as informações das tabelas são atualizadas. Por exemplo, ao executar um LOAD da posição de memória 1 para o registrador $8, é adicionado na tabela de memória que na posição 1 está o registrador 8 atrelado, além disso é decrementado o número de ocorrências da variável. Na tabela de registradores é colocado a posição de memória que está sendo utilizado e que o registrador está em uso. No caso do STORE os valores são zerados, deixando o registrador livre.

Importante salientar que nem sempre é decrementado o número de ocorrências da variável, no compilador existem situações em que LOADs e STOREs são realizados para garantir a coerência de dados entre escopos. Portanto o valor só é decrementado quando o LOAD provém de uma instrução que necessita que determinada variável esteja no banco de registradores.

Tendo definido então as premissas básicas de alocação de memória, inicializa-se a análise do algoritmo utilizado. O algoritmo inicia com a procura da $fundef$ com ID "main"no código intermediário.

Ao achar a "main", é construída uma tabela de memória com as vaiáveis globais. Logo depois é construída uma tabela com as variáveis do escopo.

É atribuído inicialmente um valor de registrador para cada posição de memória, como se tivesse existido um LOAD. Porém não é realizado LOAD uma vez que não possui nada na memória (isso para o caso dos testes realizados no momento). No futuro se precisar, basta tirar essa rotina.

Ao executar esses passos iniciais basta seguir o passo a passo descrito na linha de código abaixo:

```
        while(code_int!=NULL){
                Quad_Cell quad = code_int->quad;
                switch(quad.op){
                case BEGIN_ARGS:
                        code_int = Generate_Assembly_for_ARGS(
                           code_int->next);
                        //FUNCAL
                        Store_ALL_Reg_Table();
                        Store_Global_Reg_Table_nextScope();
                        Op1 = AlocaOperand(MEM,16,
                           reserved_mem_total);
                        InsereAssembly(I_PUSH_R,Op1,NULL,NULL);//
                           PUSH.R reserved_mem_total
                        printf("PUSH.R %d\n", reserved_mem_total);
                        Op1 = AlocaOperand(MEM,8, contador+2);
                        InsereAssembly(I_PUSH_PC,Op1,NULL,NULL);//
```

```
                    PUSH.PC contador+2
                printf("PUSH.PC %d \n", contador+2);


                Op1 = AlocaOperand(LAB,32,
                    convertString_to_label(funcalString));
                InsereAssembly(I_JMP,Op1,NULL,NULL);//JMP
                    value
                printf("JMP %d\n", convertString_to_label(
                    funcalString));

                InsereAssembly(I_POP_R,NULL,NULL,NULL);//
                    POP.R
                printf("POP.R \n");
                Store_Global_Reg_Table_prevScope();
                Load_ALL_Mem_Table();

            break;
            case INTCODE_ADD:

                returnComponente(quad,1);
                Load_Operand(c,2);
                returnComponente(quad,2);
                Load_Operand(c,3);
                returnComponente(quad,3);
                if (c->type == TEMPORARIO){
                        int reg_temp = returnFree_Temp();
                        Op1 = AlocaOperand(REG,8, 2);
                        Op2 = AlocaOperand(REG,8, 3);
                        Op3 = AlocaOperand(REG,8, reg_temp
                            );
                        InsereAssembly(I_ADD,Op1,Op2,Op3);
                            //ADD $2 $3 $temp
                        printf("ADD $2 $3 $%d\n", reg_temp
                            );
                        tabela_registradores[2].
                            flag_utilizado = 0;
                        tabela_registradores[3].
                            flag_utilizado = 0;
                        tabela_registradores[reg_temp].
                            flag_utilizado = 1;
                        tabela_registradores[reg_temp].var
                             = c->value;
                }

            break;
            case INTCODE_SUB:
```

```
                returnComponente(quad,1);
                Load_Operand(c,2);
                returnComponente(quad,2);
                Load_Operand(c,3);
                returnComponente(quad,3);
                if (c->type == TEMPORARIO){
                        int reg_temp = returnFree_Temp();
                        Op1 = AlocaOperand(REG,8, 2);
                        Op2 = AlocaOperand(REG,8, 3);
                        Op3 = AlocaOperand(REG,8, reg_temp
                            );
                        InsereAssembly(I_SUB,Op1,Op2,Op3);
                            //SUB $2 $3 $temp
                        printf("SUB $2 $3 $%d\n", reg_temp
                            );
                        tabela_registradores[2].
                            flag_utilizado = 0;
                        tabela_registradores[3].
                            flag_utilizado = 0;
                        tabela_registradores[reg_temp].
                            flag_utilizado = 1;
                        tabela_registradores[reg_temp].var
                             = c->value;
                }

        break;
        case INTCODE_MUL:

                returnComponente(quad,1);
                Load_Operand(c,2);
                returnComponente(quad,2);
                Load_Operand(c,3);
                returnComponente(quad,3);
                if (c->type == TEMPORARIO){
                        int reg_temp = returnFree_Temp();
                        Op1 = AlocaOperand(REG,8, 2);
                        Op2 = AlocaOperand(REG,8, 3);
                        Op3 = AlocaOperand(REG,8, reg_temp
                            );
                        InsereAssembly(I_MUL,Op1,Op2,Op3);
                            //MUL $2 $3 $temp
                        printf("MUL $2 $3 $%d\n", reg_temp
                            );
                        tabela_registradores[2].
                            flag_utilizado = 0;
                        tabela_registradores[3].
```

```
                                flag_utilizado = 0;
                            tabela_registradores[reg_temp].
                                flag_utilizado = 1;
                            tabela_registradores[reg_temp].var
                                = c->value;
                    }

            break;
            case INTCODE_DIV:

                    returnComponente(quad,1);
                    Load_Operand(c,2);
                    returnComponente(quad,2);
                    Load_Operand(c,3);
                    returnComponente(quad,3);
                    if (c->type == TEMPORARIO){
                            int reg_temp = returnFree_Temp();
                            Op1 = AlocaOperand(REG,8, 2);
                            Op2 = AlocaOperand(REG,8, 3);
                            Op3 = AlocaOperand(REG,8, reg_temp
                                );
                            InsereAssembly(I_DIV,Op1,Op2,Op3);
                                //DIV $2 $3 $temp
                            printf("DIV $2 $3 $%d\n", reg_temp
                                );
                            tabela_registradores[2].
                                flag_utilizado = 0;
                            tabela_registradores[3].
                                flag_utilizado = 0;
                            tabela_registradores[reg_temp].
                                flag_utilizado = 1;
                            tabela_registradores[reg_temp].var
                                = c->value;
                    }

            break;
            case INTCODE_LT:

                    returnComponente(quad,1);
                    Load_Operand(c,2);
                    returnComponente(quad,2);
                    Load_Operand(c,3);
                    code_int = code_int->next;

                    Store_ALL_Reg_Table();
                    quad = code_int->quad;
                    returnComponente(quad,2);
```

```
                        if (c->type == L_LABEL){
                                Op1 = AlocaOperand(REG,8, 2);
                                Op2 = AlocaOperand(REG,8, 3);
                                Op3 = AlocaOperand(LAB,8, c->value
                                    );
                                InsereAssembly(I_JPGE,Op1,Op2,Op3)
                                    ;//JPG $2 3 c->value
                                printf("JPGE $2 $3 %d\n", c->value
                                    );
                                tabela_registradores[2].
                                    flag_utilizado = 0;
                                tabela_registradores[3].
                                    flag_utilizado = 0;
                        }

                break;
                case INTCODE_LEQ:

                        returnComponente(quad,1);
                        Load_Operand(c,2);
                        returnComponente(quad,2);
                        Load_Operand(c,3);
                        code_int = code_int->next;

                        Store_ALL_Reg_Table();
                        quad = code_int->quad;
                        returnComponente(quad,2);
                        if (c->type == L_LABEL){
                                Op1 = AlocaOperand(REG,8, 2);
                                Op2 = AlocaOperand(REG,8, 3);
                                Op3 = AlocaOperand(LAB,8, c->value
                                    );
                                InsereAssembly(I_JPG,Op1,Op2,Op3);
                                    //JPGE $2 3 c->value
                                printf("JPG $2 $3 %d\n", c->value)
                                    ;
                                tabela_registradores[2].
                                    flag_utilizado = 0;
                                tabela_registradores[3].
                                    flag_utilizado = 0;
                        }

                break;
                case INTCODE_GT:

                        returnComponente(quad,1);
                        Load_Operand(c,2);
```

```
                        returnComponente(quad,2);
                        Load_Operand(c,3);
                        code_int = code_int->next;

                        Store_ALL_Reg_Table();
                        quad = code_int->quad;
                        returnComponente(quad,2);
                        if (c->type == L_LABEL){
                                Op1 = AlocaOperand(REG,8, 2);
                                Op2 = AlocaOperand(REG,8, 3);
                                Op3 = AlocaOperand(LAB,8, c->value
                                    );
                                InsereAssembly(I_JPLE,Op1,Op2,Op3)
                                    ;//JPL $2 3 c->value
                                printf("JPLE $2 $3 %d\n", c->value
                                    );
                                tabela_registradores[2].
                                    flag_utilizado = 0;
                                tabela_registradores[3].
                                    flag_utilizado = 0;
                        }

                break;
                case INTCODE_GEQ:

                        returnComponente(quad,1);
                        Load_Operand(c,2);
                        returnComponente(quad,2);
                        Load_Operand(c,3);
                        code_int = code_int->next;

                        Store_ALL_Reg_Table();
                        quad = code_int->quad;
                        returnComponente(quad,2);
                        if (c->type == L_LABEL){
                                Op1 = AlocaOperand(REG,8, 2);
                                Op2 = AlocaOperand(REG,8, 3);
                                Op3 = AlocaOperand(LAB,8, c->value
                                    );
                                InsereAssembly(I_JPL,Op1,Op2,Op3);
                                    //JPLE $2 3 c->value
                                printf("JPL $2 $3 %d\n", c->value)
                                    ;
                                tabela_registradores[2].
                                    flag_utilizado = 0;
                                tabela_registradores[3].
                                    flag_utilizado = 0;
```

```
                }

            break;
            case INTCODE_EQ:
                returnComponente(quad,1);
                Load_Operand(c,2);
                returnComponente(quad,2);
                Load_Operand(c,3);
                code_int = code_int->next;

                Store_ALL_Reg_Table();
                quad = code_int->quad;
                returnComponente(quad,2);
                if (c->type == L_LABEL){
                    Op1 = AlocaOperand(REG,8, 2);
                    Op2 = AlocaOperand(REG,8, 3);
                    Op3 = AlocaOperand(LAB,8, c->value
                        );
                    InsereAssembly(I_JMPNE,Op1,Op2,Op3
                        );//JMPNE $2 3 c->value
                    printf("JMPNE $2 $3 %d\n", c->
                        value);
                    tabela_registradores[2].
                        flag_utilizado = 0;
                    tabela_registradores[3].
                        flag_utilizado = 0;
                }

            break;
            case INTCODE_NEQ:
                returnComponente(quad,1);
                Load_Operand(c,2);
                returnComponente(quad,2);
                Load_Operand(c,3);
                code_int = code_int->next;

                Store_ALL_Reg_Table();
                quad = code_int->quad;
                returnComponente(quad,2);
                if (c->type == L_LABEL){
                    Op1 = AlocaOperand(REG,8, 2);
                    Op2 = AlocaOperand(REG,8, 3);
                    Op3 = AlocaOperand(LAB,8, c->value
                        );
                    InsereAssembly(I_JPE,Op1,Op2,Op3);
                        //JPE $2 3 c->value
                    printf("JPE $2 $3 %d\n", c->value)
```

```
                        ;
                    tabela_registradores[2].
                        flag_utilizado = 0;
                    tabela_registradores[3].
                        flag_utilizado = 0;
            }

        break;
        case INTCODE_ASSIGN:
            returnComponente(quad,2);
            Load_Operand(c,3);
            returnComponente(quad,1);
            Assign_Operand(c,3);

        break;
        case INTCODE_RET:
            if (quad.addr_01.kind != Vazio){
                returnComponente(quad,1);
                Load_Operand(c,0);

                Store_ALL_Reg_Table();
                Op1 = AlocaOperand(LAB,32, -1);
                InsereAssembly(I_JMP,Op1,NULL,NULL
                    );//JMP c->value
                printf("JMP %d\n", -1);

            }else{
                Store_ALL_Reg_Table();
                Op1 = AlocaOperand(LAB,32, -1);
                InsereAssembly(I_JMP,Op1,NULL,NULL
                    );//JMP c->value
                printf("JMP %d\n", -1);

            }
        break;
        case JMP:

            Store_ALL_Reg_Table();
            returnComponente(quad,1);
            if(c->type == L_LABEL){
                Op1 = AlocaOperand(LAB,32, c->
                    value);
                Op2 = NULL;
                Op3 = NULL;
                InsereAssembly(I_JMP,Op1,Op2,Op3);
                    //JMP c->value
                printf("JMP %d\n", c->value);
```

```
                }
                Load_ALL_Mem_Table();

            break;
            case LABEL:

                Store_ALL_Reg_Table();
                returnComponente(quad,1);
                if(c->type == L_LABEL){
                    Op1 = AlocaOperand(LAB,32, c->
                        value);
                    Op2 = NULL;
                    Op3 = NULL;
                    InsereAssembly(I_LABEL,Op1,Op2,Op3
                        );//LB c->value
                    printf("LB %d\n", c->value);
                }
                Load_ALL_Mem_Table();

            break;
            case FUNCAL:

                Store_ALL_Reg_Table();
                Store_Global_Reg_Table_nextScope();
                Op1 = AlocaOperand(MEM,16,
                    reserved_mem_total);
                InsereAssembly(I_PUSH_R,Op1,NULL,NULL);//
                    PUSH.R reserved_mem_total
                printf("PUSH.R %d\n", reserved_mem_total);
                Op1 = AlocaOperand(MEM,8, contador+2);
                InsereAssembly(I_PUSH_PC,Op1,NULL,NULL);//
                    PUSH.PC contador+2
                printf("PUSH.PC %d \n", contador+2,
                    contador);
                returnComponente(quad,1);
                if(c->type == VARIAVEL){
                    Op1 = AlocaOperand(LAB,32,
                        convertString_to_label(c->var))
                        ;
                    InsereAssembly(I_JMP,Op1,NULL,NULL
                        );//JMP value
                    printf("JMP %d\n",
                        convertString_to_label(c->var))
                        ;
                }
                InsereAssembly(I_POP_R,NULL,NULL,NULL);//
                    POP.R
```

```
                    printf("POP.R \n");
                    Store_Global_Reg_Table_prevScope();
                    Load_ALL_Mem_Table();

                break;
                }
                code_int = code_int->next;
        }
```

Como todo o código foi armazenado em uma estrutura, ao final de tudo precisou-se traduzir usando a tabela referenciada na descrição do processador.

Importante salientar que não precisou-se alterar a arquitetura e nem o conjunto de instruções.

Para mais informações basta olhar no Apendice.

5 Exemplos

5.1 Fibonacci

Na matemática, a Sucessão de Fibonacci (também Sequência de Fibonacci), é uma sequência de números inteiros, começando normalmente por 0 e 1, na qual, cada termo subsequente corresponde a soma dos dois anteriores. A sequência recebeu o nome do matemático italiano Leonardo de Pisa, mais conhecido por Fibonacci , que descreveu, no ano de 1202, o crescimento de uma população de coelhos, a partir desta. Tal sequência já era no entanto, conhecida na antiguidade.

O código em alto nível da sequencia é dado por:

```
int fibonacci(int n){
    int c;
    int next;
    int first;
    int second;
    first = 0;
    second = 1;
    c = 0;
    while(c <= n){
        if(c <= 1)
            next = c;
        else{
            next = first + second;
            first = second;
            second = next; /* Estava second = first */
        }
        c = c + 1;
    }
    return next;
}
void main(void){
  int n;
  int saida;
  n = 3;
  saida = fibonacci(n);
}
```

O código assembly gerado é dado por:

```
SRVALUE 3 $3
REGCOPY $3 $8
REGCOPY $8 $3
```

```
STORE $8 0
STORE $3 2
STORE $9 1
PUSH_R 2
PUSH_PC 9
JMP 15
POP_R
LOAD 1 $8
REGCOPY $0 $3
REGCOPY $3 $8
STORE $8 1
JMP 98
LOAD 0 $8
LOAD 1 $9
LOAD 2 $10
LOAD 3 $11
LOAD 4 $12
SRVALUE 0 $3
REGCOPY $3 $11
SRVALUE 1 $3
REGCOPY $3 $12
SRVALUE 0 $3
REGCOPY $3 $9
STORE $8 0
STORE $9 1
STORE $10 2
STORE $11 3
STORE $12 4
LOAD 0 $8
LOAD 1 $9
LOAD 2 $10
LOAD 3 $11
LOAD 4 $12
REGCOPY $9 $2
REGCOPY $8 $3
STORE $8 0
STORE $9 1
STORE $10 2
STORE $11 3
STORE $12 4
JPG $2 $3 93
LOAD 1 $8
REGCOPY $8 $2
SRVALUE 1 $3
STORE $8 1
JPG $2 $3 64
LOAD 1 $8
```

```
REGCOPY $8 $3
LOAD 2 $9
REGCOPY $3 $9
STORE $8 1
STORE $9 2
JMP 81
LOAD 1 $8
LOAD 2 $9
LOAD 3 $10
LOAD 4 $11
STORE $8 1
STORE $9 2
STORE $10 3
STORE $11 4
LOAD 1 $8
LOAD 2 $9
LOAD 3 $10
LOAD 4 $11
REGCOPY $10 $2
REGCOPY $11 $3
ADD $2 $3 $4
REGCOPY $4 $3
REGCOPY $3 $9
REGCOPY $11 $3
REGCOPY $3 $10
STORE $10 3
REGCOPY $9 $3
REGCOPY $3 $11
STORE $11 4
STORE $8 1
STORE $9 2
LOAD 1 $8
LOAD 2 $9
REGCOPY $8 $2
SRVALUE 1 $3
ADD $2 $3 $4
REGCOPY $4 $3
REGCOPY $3 $8
STORE $8 1
STORE $9 2
JMP 31
LOAD 2 $8
STORE $8 2
LOAD 2 $8
REGCOPY $8 $0
STORE $8 2
POP_PC
```

```
POP_PC
NOP
```

E o código de máquina já convertido para o formato do FPGA é:

```
I_mem[0] = {8'b01000000,16'd3,8'd3};
I_mem[1] = {8'b01000011,8'd3,8'd0,8'd8};
I_mem[2] = {8'b01000011,8'd8,8'd0,8'd3};
I_mem[3] = {8'b01000010,8'd8,16'd0};
I_mem[4] = {8'b01000010,8'd3,16'd2};
I_mem[5] = {8'b01000010,8'd9,16'd1};
I_mem[6] = {8'b11000001,16'd2,8'd0};
I_mem[7] = {8'b11000011,8'd9,16'd0};
I_mem[8] = {8'b00000001,16'd0,8'd15};
I_mem[9] = {8'b11000010,24'd0};
I_mem[10] = {8'b01000001,16'd1,8'd8};
I_mem[11] = {8'b01000011,8'd0,8'd0,8'd3};
I_mem[12] = {8'b01000011,8'd3,8'd0,8'd8};
I_mem[13] = {8'b01000010,8'd8,16'd1};
I_mem[14] = {8'b00000001,16'd0,8'd98};
I_mem[15] = {8'b01000001,16'd0,8'd8};
I_mem[16] = {8'b01000001,16'd1,8'd9};
I_mem[17] = {8'b01000001,16'd2,8'd10};
I_mem[18] = {8'b01000001,16'd3,8'd11};
I_mem[19] = {8'b01000001,16'd4,8'd12};
I_mem[20] = {8'b01000000,16'd0,8'd3};
I_mem[21] = {8'b01000011,8'd3,8'd0,8'd11};
I_mem[22] = {8'b01000000,16'd1,8'd3};
I_mem[23] = {8'b01000011,8'd3,8'd0,8'd12};
I_mem[24] = {8'b01000000,16'd0,8'd3};
I_mem[25] = {8'b01000011,8'd3,8'd0,8'd9};
I_mem[26] = {8'b01000010,8'd8,16'd0};
I_mem[27] = {8'b01000010,8'd9,16'd1};
I_mem[28] = {8'b01000010,8'd10,16'd2};
I_mem[29] = {8'b01000010,8'd11,16'd3};
I_mem[30] = {8'b01000010,8'd12,16'd4};
I_mem[31] = {8'b01000001,16'd0,8'd8};
I_mem[32] = {8'b01000001,16'd1,8'd9};
I_mem[33] = {8'b01000001,16'd2,8'd10};
I_mem[34] = {8'b01000001,16'd3,8'd11};
I_mem[35] = {8'b01000001,16'd4,8'd12};
I_mem[36] = {8'b01000011,8'd9,8'd0,8'd2};
I_mem[37] = {8'b01000011,8'd8,8'd0,8'd3};
I_mem[38] = {8'b01000010,8'd8,16'd0};
I_mem[39] = {8'b01000010,8'd9,16'd1};
I_mem[40] = {8'b01000010,8'd10,16'd2};
I_mem[41] = {8'b01000010,8'd11,16'd3};
I_mem[42] = {8'b01000010,8'd12,16'd4};
```

```
I_mem[43] = {8'b00000101,8'd2,8'd3,8'd93};
I_mem[44] = {8'b01000001,16'd1,8'd8};
I_mem[45] = {8'b01000011,8'd8,8'd0,8'd2};
I_mem[46] = {8'b01000000,16'd1,8'd3};
I_mem[47] = {8'b01000010,8'd8,16'd1};
I_mem[48] = {8'b00000101,8'd2,8'd3,8'd64};
I_mem[49] = {8'b01000001,16'd1,8'd8};
I_mem[50] = {8'b01000011,8'd8,8'd0,8'd3};
I_mem[51] = {8'b01000001,16'd2,8'd9};
I_mem[52] = {8'b01000011,8'd3,8'd0,8'd9};
I_mem[53] = {8'b01000010,8'd8,16'd1};
I_mem[54] = {8'b01000010,8'd9,16'd2};
I_mem[55] = {8'b00000001,16'd0,8'd81};
I_mem[56] = {8'b01000001,16'd1,8'd8};
I_mem[57] = {8'b01000001,16'd2,8'd9};
I_mem[58] = {8'b01000001,16'd3,8'd10};
I_mem[59] = {8'b01000001,16'd4,8'd11};
I_mem[60] = {8'b01000010,8'd8,16'd1};
I_mem[61] = {8'b01000010,8'd9,16'd2};
I_mem[62] = {8'b01000010,8'd10,16'd3};
I_mem[63] = {8'b01000010,8'd11,16'd4};
I_mem[64] = {8'b01000001,16'd1,8'd8};
I_mem[65] = {8'b01000001,16'd2,8'd9};
I_mem[66] = {8'b01000001,16'd3,8'd10};
I_mem[67] = {8'b01000001,16'd4,8'd11};
I_mem[68] = {8'b01000011,8'd10,8'd0,8'd2};
I_mem[69] = {8'b01000011,8'd11,8'd0,8'd3};
I_mem[70] = {8'b10000101,8'd2,8'd3,8'd4};
I_mem[71] = {8'b01000011,8'd4,8'd0,8'd3};
I_mem[72] = {8'b01000011,8'd3,8'd0,8'd9};
I_mem[73] = {8'b01000011,8'd11,8'd0,8'd3};
I_mem[74] = {8'b01000011,8'd3,8'd0,8'd10};
I_mem[75] = {8'b01000010,8'd10,16'd3};
I_mem[76] = {8'b01000011,8'd9,8'd0,8'd3};
I_mem[77] = {8'b01000011,8'd3,8'd0,8'd11};
I_mem[78] = {8'b01000010,8'd11,16'd4};
I_mem[79] = {8'b01000010,8'd8,16'd1};
I_mem[80] = {8'b01000010,8'd9,16'd2};
I_mem[81] = {8'b01000001,16'd1,8'd8};
I_mem[82] = {8'b01000001,16'd2,8'd9};
I_mem[83] = {8'b01000011,8'd8,8'd0,8'd2};
I_mem[84] = {8'b01000000,16'd1,8'd3};
I_mem[85] = {8'b10000101,8'd2,8'd3,8'd4};
I_mem[86] = {8'b01000011,8'd4,8'd0,8'd3};
I_mem[87] = {8'b01000011,8'd3,8'd0,8'd8};
I_mem[88] = {8'b01000010,8'd8,16'd1};
I_mem[89] = {8'b01000010,8'd9,16'd2};
```

```
I_mem[90] = {8'b00000001,16'd0,8'd31};
I_mem[91] = {8'b01000001,16'd2,8'd8};
I_mem[92] = {8'b01000010,8'd8,16'd2};
I_mem[93] = {8'b01000001,16'd2,8'd8};
I_mem[94] = {8'b01000011,8'd8,8'd0,8'd0};
I_mem[95] = {8'b01000010,8'd8,16'd2};
I_mem[96] = {8'b11000100,24'd0};
I_mem[97] = {8'b11000100,24'd0};
I_mem[98] = {8'b00000000,24'd0};
limit_PC = 99;
```

5.2 Maximo Divisor Comum

MDC significa máximo divisor comum. O máximo divisor comum entre dois ou mais números naturais é o maior de seus divisores. Dois números naturais sempre tem divisores em comum.Como exemplo, o MDC de 16 e 24, MDC 16, 24 = 8, que é o maior número natural que divide ambos.

O código em C- que calcula o máximo divisor comum é representado abaixo:

```
int gcd(int u, int v)
{ if (v==0) return u;
else return gcd(v, u-u/v*v);
}
int input(void){
        int x;
        return x;
}
void output(int x){

}
void main(void)
{ int x; int y;
x = input(); y = input();
output(gcd(x,y));
}
```

O código assembly gerado é dado por:

```
STORE $8 0
STORE $9 1
PUSH_R 2
PUSH_PC 5
JMP 77
POP_R
LOAD 0 $8
LOAD 1 $9
```

```
REGCOPY $0 $3
REGCOPY $3 $8
STORE $8 0
STORE $9 1
PUSH_R 2
PUSH_PC 15
JMP 77
POP_R
LOAD 0 $8
LOAD 1 $9
REGCOPY $0 $3
REGCOPY $3 $9
REGCOPY $8 $3
STORE $8 0
STORE $3 2
REGCOPY $9 $3
STORE $9 1
STORE $3 3
PUSH_R 2
PUSH_PC 29
JMP 37
POP_R
REGCOPY $0 $3
STORE $3 2
PUSH_R 2
PUSH_PC 35
JMP 82
POP_R
JMP 83
LOAD 0 $8
LOAD 1 $9
REGCOPY $9 $2
SRVALUE 0 $3
STORE $8 0
STORE $9 1
JMPNE $2 $3 53
LOAD 0 $8
REGCOPY $8 $0
STORE $8 0
POP_PC
JMP 76
LOAD 0 $8
LOAD 1 $9
STORE $8 0
STORE $9 1
LOAD 0 $8
LOAD 1 $9
```

```
REGCOPY $9 $3
STORE $3 2
REGCOPY $8 $2
REGCOPY $9 $3
DIV $2 $3 $4
REGCOPY $4 $2
REGCOPY $9 $3
STORE $9 1
MUL $2 $3 $4
REGCOPY $8 $2
STORE $8 0
REGCOPY $4 $3
SUB $2 $3 $4
REGCOPY $4 $3
STORE $3 3
PUSH_R 2
PUSH_PC 73
JMP 37
POP_R
REGCOPY $0 $0
POP_PC
POP_PC
LOAD 0 $8
REGCOPY $8 $0
STORE $8 0
POP_PC
POP_PC
POP_PC
NOP
```

E o código de máquina já convertido para o formato do FPGA é:

```
I_mem[0] = {8'b01000010,8'd8,16'd0};
I_mem[1] = {8'b01000010,8'd9,16'd1};
I_mem[2] = {8'b11000001,16'd2,8'd0};
I_mem[3] = {8'b11000011,8'd5,16'd0};
I_mem[4] = {8'b00000001,16'd0,8'd77};
I_mem[5] = {8'b11000010,24'd0};
I_mem[6] = {8'b01000001,16'd0,8'd8};
I_mem[7] = {8'b01000001,16'd1,8'd9};
I_mem[8] = {8'b01000011,8'd0,8'd0,8'd3};
I_mem[9] = {8'b01000011,8'd3,8'd0,8'd8};
I_mem[10] = {8'b01000010,8'd8,16'd0};
I_mem[11] = {8'b01000010,8'd9,16'd1};
I_mem[12] = {8'b11000001,16'd2,8'd0};
I_mem[13] = {8'b11000011,8'd15,16'd0};
I_mem[14] = {8'b00000001,16'd0,8'd77};
I_mem[15] = {8'b11000010,24'd0};
```

```
I_mem[16] = {8'b01000001,16'd0,8'd8};
I_mem[17] = {8'b01000001,16'd1,8'd9};
I_mem[18] = {8'b01000011,8'd0,8'd0,8'd3};
I_mem[19] = {8'b01000011,8'd3,8'd0,8'd9};
I_mem[20] = {8'b01000011,8'd8,8'd0,8'd3};
I_mem[21] = {8'b01000010,8'd8,16'd0};
I_mem[22] = {8'b01000010,8'd3,16'd2};
I_mem[23] = {8'b01000011,8'd9,8'd0,8'd3};
I_mem[24] = {8'b01000010,8'd9,16'd1};
I_mem[25] = {8'b01000010,8'd3,16'd3};
I_mem[26] = {8'b11000001,16'd2,8'd0};
I_mem[27] = {8'b11000011,8'd29,16'd0};
I_mem[28] = {8'b00000001,16'd0,8'd37};
I_mem[29] = {8'b11000010,24'd0};
I_mem[30] = {8'b01000011,8'd0,8'd0,8'd3};
I_mem[31] = {8'b01000010,8'd3,16'd2};
I_mem[32] = {8'b11000001,16'd2,8'd0};
I_mem[33] = {8'b11000011,8'd35,16'd0};
I_mem[34] = {8'b00000001,16'd0,8'd82};
I_mem[35] = {8'b11000010,24'd0};
I_mem[36] = {8'b00000001,16'd0,8'd83};
I_mem[37] = {8'b01000001,16'd0,8'd8};
I_mem[38] = {8'b01000001,16'd1,8'd9};
I_mem[39] = {8'b01000011,8'd9,8'd0,8'd2};
I_mem[40] = {8'b01000000,16'd0,8'd3};
I_mem[41] = {8'b01000010,8'd8,16'd0};
I_mem[42] = {8'b01000010,8'd9,16'd1};
I_mem[43] = {8'b00000011,8'd2,8'd3,8'd53};
I_mem[44] = {8'b01000001,16'd0,8'd8};
I_mem[45] = {8'b01000011,8'd8,8'd0,8'd0};
I_mem[46] = {8'b01000010,8'd8,16'd0};
I_mem[47] = {8'b11000100,24'd0};
I_mem[48] = {8'b00000001,16'd0,8'd76};
I_mem[49] = {8'b01000001,16'd0,8'd8};
I_mem[50] = {8'b01000001,16'd1,8'd9};
I_mem[51] = {8'b01000010,8'd8,16'd0};
I_mem[52] = {8'b01000010,8'd9,16'd1};
I_mem[53] = {8'b01000001,16'd0,8'd8};
I_mem[54] = {8'b01000001,16'd1,8'd9};
I_mem[55] = {8'b01000011,8'd9,8'd0,8'd3};
I_mem[56] = {8'b01000010,8'd3,16'd2};
I_mem[57] = {8'b01000011,8'd8,8'd0,8'd2};
I_mem[58] = {8'b01000011,8'd9,8'd0,8'd3};
I_mem[59] = {8'b10001000,8'd2,8'd3,8'd4};
I_mem[60] = {8'b01000011,8'd4,8'd0,8'd2};
I_mem[61] = {8'b01000011,8'd9,8'd0,8'd3};
I_mem[62] = {8'b01000010,8'd9,16'd1};
```

```
I_mem[63] = {8'b10000111,8'd2,8'd3,8'd4};
I_mem[64] = {8'b01000011,8'd8,8'd0,8'd2};
I_mem[65] = {8'b01000010,8'd8,16'd0};
I_mem[66] = {8'b01000011,8'd4,8'd0,8'd3};
I_mem[67] = {8'b10000110,8'd2,8'd3,8'd4};
I_mem[68] = {8'b01000011,8'd4,8'd0,8'd3};
I_mem[69] = {8'b01000010,8'd3,16'd3};
I_mem[70] = {8'b11000001,16'd2,8'd0};
I_mem[71] = {8'b11000011,8'd73,16'd0};
I_mem[72] = {8'b00000001,16'd0,8'd37};
I_mem[73] = {8'b11000010,24'd0};
I_mem[74] = {8'b01000011,8'd0,8'd0,8'd0};
I_mem[75] = {8'b11000100,24'd0};
I_mem[76] = {8'b11000100,24'd0};
I_mem[77] = {8'b01000001,16'd0,8'd8};
I_mem[78] = {8'b01000011,8'd8,8'd0,8'd0};
I_mem[79] = {8'b01000010,8'd8,16'd0};
I_mem[80] = {8'b11000100,24'd0};
I_mem[81] = {8'b11000100,24'd0};
I_mem[82] = {8'b11000100,24'd0};
I_mem[83] = {8'b00000000,24'd0};
limit_PC = 84;
```

6 Conclusões

Depois de realizado o projeto, observou-se que o compilador funcionou perfeitamente, passando por todos os chamados "teste de mesa"no qual foram analisados vários códigos e obteve-se o resultado esperado.

No dia da apresentação do compilador, um dos maiores problemas foi sem dúvida alinhar com o comportamento da arquitetura. Foi descoberto no último minuto que o problema era a velocidade que apertava-se o botão do "clock". Esse problema pode ser classificado como de extrema importância, pois para o laboratório de sistemas operacionais é necessário que ele esteja conseguindo executar instruções de forma rápida e segura. Um das possíveis soluções, que foi pensada logo após a apresentação é de verificar o caso *default* da unidade de controle, pois isso pode ser o determinante.

Outro problema encontrado foi a falta de interface de comunicação com o FPGA, tendo que analisar o comportamento das instruções diretamente pelos bits usando os LEDs.

Durante a implementação a parte mais trabalhosa e sem dúvida a que mais é "sensível"a erros é a análise semântica. Essa etapa custou um pouco mais de tempo que eu esperava, atrasando a entrega da análise semântica. Enquanto as últimas duas etapas, mesmo com o tempo reduzido, foi relativamente tranquilo de se implementar.

Mesmo com bons resultados é inegável a insatisfação com o resultado do FPGA, que rodou no último momento. E fez perder tempo "escovando bit". Portanto, agora precisa-se otimizar e limpar tanto a arquitetura quanto o compilador o mais rápido possível, pois o próximo laboratório irá demandar um sistema estável.

Referências

1 LOUDEN, K. C. *Compiler Construction: Principles and Practice*. Boston, MA, USA: PWS Publishing Co., 1997. ISBN 0534939724. Citado 3 vezes nas páginas 3, 7 e 8.