UNIVERSIDADE FEDERAL DE RORAIMA

CENTRO DE CIÊNCIA E TECNOLOGIA

CURSO DE CIÊNCIA DA COMPUTAÇÃO

RELATÓRIO DE CONSTRUÇÃO DO COMPILEREXPRESSIONS

BOA VISTA – RR

2023.1

GUILHERME LUCAS PEREIRA BERNARDO

RELATÓRIO DE CONSTRUÇÃO DO COMPILEREXPRESSIONS

Relatório de construção do trabalho final entregue à docência do curso de Ciência da computação como requisito parcial para obtenção de notas da disciplina **CONSTRUÇÃO DE COMPILADORES** – DCC605 —, sob orientação do professor Dr. Luciano F. Silva.

BOA VISTA – RR

2023.1

SUMÁRIO

1. INTRODUÇÃO

Este trabalho tem como objetivo apresentar as conclusões obtidas pelo autor acerca do desenvolvimento de um compilador com base nas especificações pedidas pelo professor no início do prazo para confecção do mesmo.

2. DESENVOLVIMENTO DA APLICAÇÃO

a. Concepção

O projeto do CompilerExpressions apresentado neste relatório se dá por meio do esforço do aluno para se criar dito projeto. O trabalho se deu pela linguagem C++ e foi concebido através do editor de código VSCODE.

b. Características

O programa é case-sensitive em relação a suas palavras reservadas, além de ser baseado em uma linguagem similar à C, porém sem a declaração e utilização de funções

Suas capacidades envolvem o processamento de operações aritméticas básicas como adição(+), subtração(-), divisão(/) e multiplicação(*) em conjunto com a possibilidade de se trabalhar com números inteiros e floats. Suas gramáticas também são capazes de processar operadores de precedência como “(“ e “)”.

c. Limitações

i. Tamanho de variáveis limitado:

O CompilerExpressions é configurado propositalmente para rejeitar identificadores maiores que um caractere. Porém, o autômato presente no compilador é capaz de reconhecer sequências maiores.

ii. Não aceita funções:

O CompilerExpressions não foi pensado para suportar a declaração e chamada de funções.

iii. Limitação em números floats

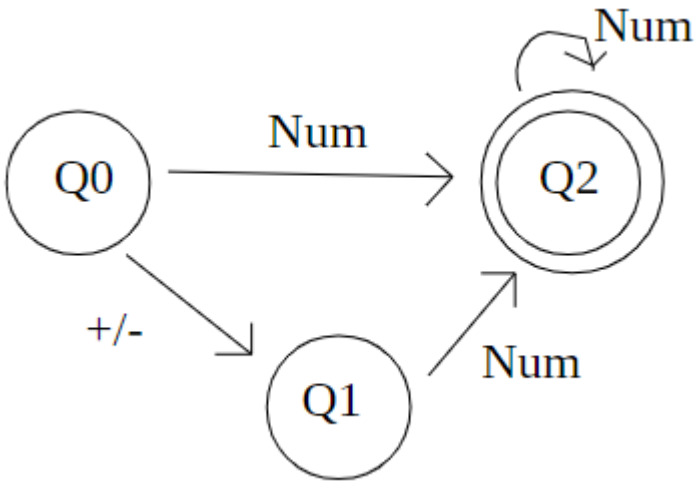
Apesar de números floats com precisão maior serem aceitos pelos seus autômatos, devido à uma limitação na análise sintática do compilador, números com mais que 1 valor após a vírgula serão truncados para apenas o primeiro valor no código fonte gerado.

d. Análise Léxica

A fase de Análise Léxica é a primeira fase de compilação de um código, por essa linha tem-se que definir suas gramáticas regulares para poder identificar seus tokens e separá-los para as próximas fases do compilador. O CompileExpressions possui 3 gramáticas regulares em sua fase de análise lexical, o que permite o reconhecimento dos mais variados tipos de tokens a serem introduzidos.

i. Gramática de números inteiros:

L = {w|w (+|-|num)(num)num}* é a gramática presente no autômato para reconhecer se um número é um inteiro.

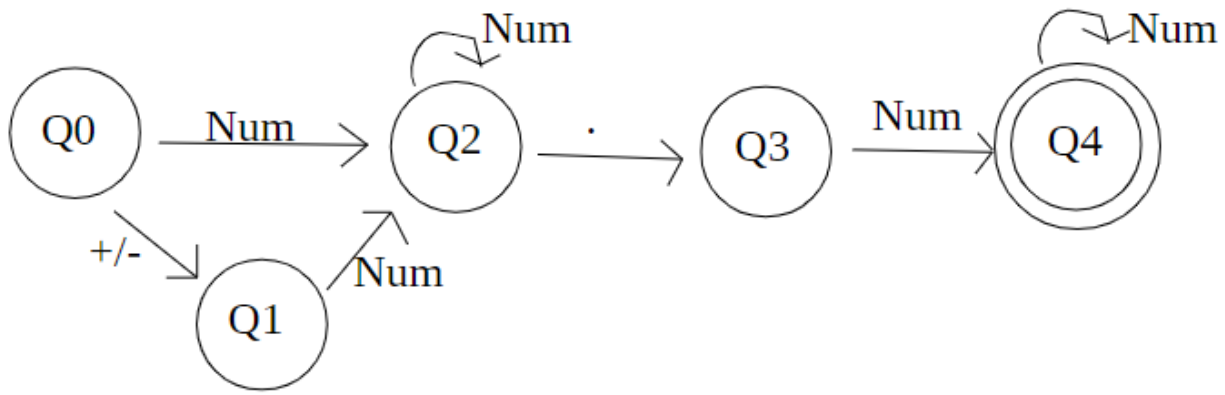


	Q0	Q1	Q2
+	Q1	-	-
-	Q1	-	-
NUM	Q2	Q2	Q2

Este autômato reconhece como inteiros sequências que possuam ou não os sinais de ‘+’ ou ‘-’ na frente e que possuam apenas números.

ii. Gramática de números floats:

L = {w|w (+|-|num)(num)num(.)(num)num*}* é a gramática presente no autômato utilizado para reconhecer números float.

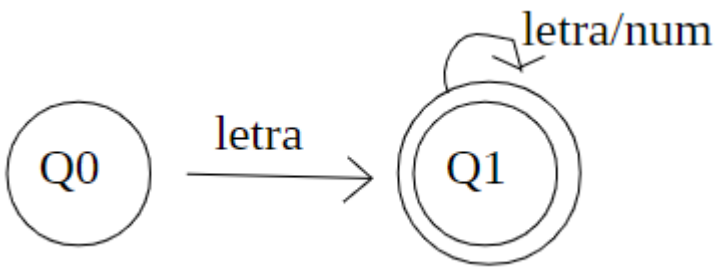


	Q0	Q1	Q2	Q3	Q4
+	Q1	-	-	-	-
-	Q1	-	-	-	-
NUM	Q2	Q2	Q2	Q4	Q4
.	-	-	Q3	-	-

Este autômato é o responsável por reconhecer sequências que possuam ou não os sinais de “+” ou “-” na frente e que possuam “.”. Observe que ele é capaz de reconhecer sentenças mais complexas também.

iii. Gramática de identificadores:

L = {w|w letra(letra|número)}* é a gramática utilizada no autômato para reconhecer um identificador.



	Q0	Q1
NUM	-	Q1
LETRA	Q1	Q1

Este autômato reconhece como identificador sequências que começam com letras maiúsculas ou minúsculas e que possam ser seguidas por números ou letras.

e. Análise Sintática

Como o CompilerExpressions é um compilador pensado para lidar com linguagem matemática, foi escolhido um analisador sintático de precedência fraca, por proporcionar uma gama de adaptabilidade essencial para esses tipos de operações.

i. Gramática

A gramática desenvolvida para o analisador sintático foi a seguinte:

- Símbolos não-terminais = {E, S, M, D, P}

Vn =	Símbolos não-terminais				
	E	S	M	D	P

- Símbolos terminais Vt = {+, -, *, /, (,), v}

Vt =	Símbolos terminais						
	+	-	*	/	(	)	v

- Produções P={E→ E+S, E→ S, S→ S-M, S→ M, M→ M*D, M→ D, D→ D/P, D→ P, P→ (E), P

P=	Produções								
	E→ S	S→ S-M	S→ M	M→ M*D	M→ D	D→ D/P	D→ P	P→ (E)	P→ v

ii. Tabela de deslocamento e redução

	+	-	*	/	(	)	v	$
E	D					D		
S	R	D				R		R
M	R	R	D			R		R
D	R	R	R	D		R		R
P	R	R	R	R		R		R
+					D		D	
-					D		D	
*					D		D	
/					D		D	
(	R	R	R	R		R		R
)	R	R	R	R		R		R
v	R	R	R	R		R		R
$					D		D	

f. Respostas à erros

O CompilerExpressions é capaz de detectar e diagnosticar erros léxicos, sintáticos e de semântica em suas entradas a serem compiladas.

i. Erro Léxico

O Compilador detecta se o token de uma keyword é grande demais para ser atribuído à ele:

1. Exemplo de entrada

```
float qudruplolocuratotal;
int qudruplolocuratotal;
```

2. Código de erro

```
**********inicializando fase de analise lexica dos dados providos**********

float IS A KEYWORD
erro lexico
Linha: 1 qudruplolocuratotal eh muito grande este compilador nao esta apto a lidar com identificadores maiores do que 1 caractere
PS C:\Users\Guilherme\Documents\GitHub\DCC605-Compilers>
```

ii. Erro Sintático 1

Durante a análise sintática de um token de declaração de variável, o compilador detecta se o próximo do próximo token é “;”, se não for, ele atribui como erro e joga a mensagem corretiva:

1. Exemplo de entrada

```
int a = 3 4;
```

2. Código de erro

```
          **********inicializando fase de analise lexica dos dados providos**********

int IS A KEYWORD
a IS A VALID IDENTIFIER
= IS AN OPERATOR
3 IS AN INTEGER
4 IS AN INTEGER

          **********Inicio da fase de analise sintatica dos dados providos**********
se nenhum erro for detectado pela analise lexica entao o compilador executa a analise sintatica



          ********erro sintatico detectado**********
       Linha: 2        falta ; antes de 4
PS C:\Users\Guilherme\Documents\GitHub\DCC605-Compilers>
```

iii. Erro Sintático 2

Durante a análise sintática de um token, o analisador verifica se o existe um token de declaração, se não for, ele aponta erro de sintaxe:

1. Exemplo de entrada

```
float y;
;x
```

2. Mensagem de erro

```
          **********inicializando fase de analise lexica dos dados providos**********

float IS A KEYWORD
y IS A VALID IDENTIFIER

          **********Inicio da fase de analise sintatica dos dados providos**********
se nenhum erro for detectado pela analise lexica entao o compilador executa a analise sintatica

$
erro Sintatico
Linha: 2 erro de syntax
passou da analise sintatica
```

iv. Erro Sintático 3

O erro sintático número 3 ocorre quando em uma entrada, o analisador sintático detecta que há parênteses sem serem fechados ou abertos:

1. Exemplo de entrada

```
float y;
y = 5);
```

2. Mensagem de erro

```
          **********inicializando fase de analise lexica dos dados providos**********

float IS A KEYWORD
y IS A VALID IDENTIFIER
y IS A VALID IDENTIFIER
= IS AN OPERATOR
5 IS AN INTEGER

          **********Inicio da fase de analise sintatica dos dados providos**********
se nenhum erro for detectado pela analise lexica entao o compilador executa a analise sintatica

erro Sintatico
Linha: 2 esperava (
PS C:\Users\Guilherme\Documents\GitHub\DCC605-Compilers>
```

g. Análise Semântica

A fase de análise semântica do compilador é onde é feita a verificação das declarações nas árvores semânticas, porém, devido à um erro envolvendo overflow de caracteres, acaba que essa verificação fica com baixa precisão.

Implementação das chamadas das funções que verificam as variáveis declaradas(não funciona corretamente):

```
     std::cout << "\t\t**********Inicio da analise semantica**********" <<
std::endl;
    espaco();
    if(error==0){
        var_declara=lst_cria();
        var_declara=lst_cpy_var(var_declara,l2);
        checa_var_declara(l2);
        //visualiza_lista(var_declara);
        lst_libera(var_declara);
        if(error==1) quadrupla_libera(q2);
    }
```

h. Geração de Código Intermediário

A geração de código intermediário é criada a partir das quadruplas geradas em cada 1 das iterações da lista de tokens gerados pela análise léxica. É importante ressaltar

que a geração de código intermediário é gerada somente a partir da análise sintática e após a validação dos tokens.

Exemplo de uma quádrupla onde v simboliza número ou identificador:

-	v	v	_t1
/	_t1	v	_t2
*	v	_t2	_t3
+	v	_t3	_t4
=	_t4	null	var

A geração de código escolhida para esse compilador se baseia no código de três endereços. Onde a instrução que não é uma declaração ou atribuição é criada temporáriamente para que as operações possam ser executadas wempre utilizando no máximo três endereços.

Exemplo: h = 2+y*(5-1)/2;

```
_t1 := 5 - 1
_t2 := _t1 / 2
_t3 := y * _t2
_t4 := 2 + _t3
h := _t4
```

i. Otimização de código intermediário

A otimização de código intermediário é a implementação de um código mais legível para humanos, baseado na implementação de código de três endereços feita na fase anterior. A otimização foi baseada nas seguintes regras, aplicadas durante um novo percorrimento das quadruplas, onde se for encontrado uma operação baseado nas regras abaixo, será substituído:

Substituir	por
x+0	x
0+x	x
x-0	x
x*1	x
1*x	x
x/1	x

Exemplo de otimização de código:

Sem otimização	Com otimização
_t1 := t / 1	K := 1
k := _t1	

j. Geração de código em linguagem simbólica

A geração de código em linguagem simbólica é baseada na arquitetura MIPS 32 bits, onde primeiro é escrito as variáveis no arquivo de saída a partir do percorrimento da quádrupla já otimizada. Após a inscrição das variáveis, as operações existentes são detectadas a partir de mais um percorrimento das quádruplas, que atribui os valores padronizados do MIPS no arquivo a ser gerado.

Exemplo de compilador funcionando:

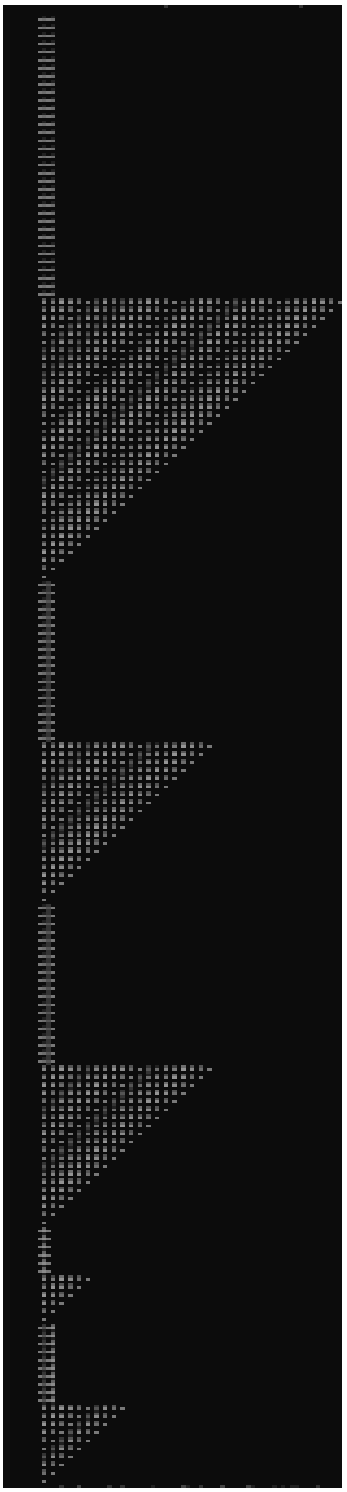
arquivo de entrada:

```
float t;
int y;
int h = 2+y*(5-1)/2;
int w=w+(466.1);
y+(466);
float t;
float V=t;
float a = t/1;
```

Saída da análise léxica:

```
float IS A KEYWORD
t IS A VALID IDENTIFIER
int IS A KEYWORD
y IS A VALID IDENTIFIER
int IS A KEYWORD
h IS A VALID IDENTIFIER
= IS AN OPERATOR
2 IS AN INTEGER
+ IS AN OPERATOR
y IS A VALID IDENTIFIER
* IS AN OPERATOR
5 IS AN INTEGER
- IS AN OPERATOR
1 IS AN INTEGER
/ IS AN OPERATOR
2 IS AN INTEGER
int IS A KEYWORD
w IS A VALID IDENTIFIER
= IS AN OPERATOR
w IS A VALID IDENTIFIER
+ IS AN OPERATOR
466.1 IS A FLOAT NUMBER
y IS A VALID IDENTIFIER
+ IS AN OPERATOR
466 IS AN INTEGER
float IS A KEYWORD
t IS A VALID IDENTIFIER
float IS A KEYWORD
V IS A VALID IDENTIFIER
= IS AN OPERATOR
t IS A VALID IDENTIFIER
float IS A KEYWORD
a IS A VALID IDENTIFIER
= IS AN OPERATOR
t IS A VALID IDENTIFIER
/ IS AN OPERATOR
1 IS AN INTEGER
```

Saída da análise sintática:



Geração das quádruplas:

op	arg1	arg2	result
-	5	1	_t1
/	_t1	2	_t2
*	y	_t2	_t3
+	2	_t3	_t4
=	_t4	&	h
+	w	4	_t1
=	_t1	&	w
+	y	4	_t1
=	t	&	V
/	t	1	_t1
=	_t1	&	a

Geração do código intermediário

```
_t1 := 5 - 1
_t2 := _t1 / 2
_t3 := y * _t2
_t4 := 2 + _t3
h := _t4
_t1 := w + 4
w := _t1
_t1 := y + 4
V := t
_t1 := t / 1
a := _t1
```

Geração de Código MIPS:

```
op: -5
arg1: 5
arg2: 1
result: $t1

op: /$t1
arg1: $t1
arg2: 2
result: $t2

op: *y
arg1: y
arg2: $t2
result: $t3

op: +2
arg1: 2
arg2: $t3
result: $t4

op: =$t4
arg1: $t4
arg2: &
result: h

op: +w
arg1: w
arg2: 4
result: $t1

op: =$t1
arg1: $t1
arg2: &
result: w

op: +y
arg1: y
arg2: 4
result: $t1

op: =t
arg1: t
arg2: &
result: V

op: /t
arg1: t
arg2: 1
result: &

op: =t
arg1: t
arg2: &
result: a
```

3. PROBLEMAS DE IMPLEMENTAÇÃO

Apesar do projeto do CompilerExpressions envolver muito mais do que apenas análise léxica, sintática e semântica. Durante a implementação do compilador, foram detectados erros de implementação que impossibilitaram uma geração de código efetiva.

O problema concentra-se especificamente na utilização das quadruplas e na inserção das expressões matemáticas nas árvores de nós no final da análise sintática.

Outros problemas de implementação foram analisados, como o uso de variáveis globais e a aglomeração de todas as funções e partes do compilador em um arquivo só, o que dificulta a análise e destrinchamento do funcionamento de suas funções.

4. CONSIDERAÇÕES FINAIS

O projeto do Compilerexpressions é um projeto complexo, capaz de identificar sentenças e gerar código compreensível para máquinas, semelhante ao MIPS, porém, erros em sua implementação foram detectados.

Dos erros observados, em algum momento entre a análise semântica e a geração de código intermediário, mais especificamente, os ponteiros das arrays de caracteres acabam recebendo overflow de algum lugar, existe um problema de lógica nas funções das árvores também. Tudo isso aliado a uma má estruturação do compilador onde todas as funções e variáveis foram colocadas em um só arquivo. Estes pontos foram cruciais em inutilizar toda a compilação a partir da análise semântica e de geração de código. Deixando com que o CompilerExpressions fique inutilizável para utilização prática no dia-a-dia.

Para fins de demonstração, foram feitas duas versões do compilador, uma que para a execução assim que encontra um erro e a outra que continua a execução a fim de apresentar todos os erros possíveis no final da execução do compilador. Para adquirir mais informações acerca do projeto, pode-se entrar no website GuilhermeBn198/DCC605-Compilers para poder visualizar o projeto e o código.