Construção de Compiladores

Matheus Cunha Reis
11521BCC030

3ª Etapa do Projeto

Uberlândia - MG
2019

Parte 1

Código em c:

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  return 0;
}
```

Código do IR:

```
; ModuleID = 'test.c'
source_filename = "test.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

; Function Attrs: noinline nounwind optnone sspstrong uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  ret i32 0
}

attributes #0 = { noinline nounwind optnone sspstrong uwtable
"correctly-rounded-divide-sqrt-fp-math"="false" "disable-tail-calls"="false"
"less-precise-fpmad"="false" "min-legal-vector-width"="0" "no-frame-pointer-elim"="true"
"no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-jump-tables"="false"
"no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false" "no-trapping-math"="false"
"stack-protector-buffer-size"="8" "target-cpu"="x86-64"
"target-features"="+fxsr,+mmx,+sse,+sse2,+x87" "unsafe-fp-math"="false"
"use-soft-float"="false" }

!llvm.module.flags = !{!0, !1, !2}
!llvm.ident = !{!3}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{i32 7, !"PIE Level", i32 2}
!3 = !{!"clang version 8.0.1 (tags/RELEASE_801/final)"}
```

Nesse codigo da pra perceber que ele mapeia a funcao main() alocando um inteiro de 32 bits, armazenando o com zero e depois retornando o vlor alocado

Código em c:

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  int x = 4;
  float y = 5;
  return 0;
}
```

Código do IR:

```
; ModuleID = 'test.c'
source_filename = "test.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

; Function Attrs: noinline nounwind optnone sspstrong uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca float, align 4
  store i32 0, i32* %1, align 4
  store i32 4, i32* %2, align 4
  store float 5.000000e+00, float* %3, align 4
  ret i32 0
}

attributes #0 = { noinline nounwind optnone sspstrong uwtable
"correctly-rounded-divide-sqrt-fp-math"="false" "disable-tail-calls"="false"
"less-precise-fpmad"="false" "min-legal-vector-width"="0" "no-frame-pointer-elim"="true"
"no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-jump-tables"="false"
"no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false" "no-trapping-math"="false"
"stack-protector-buffer-size"="8" "target-cpu"="x86-64"
"target-features"="+fxsr,+mmx,+sse,+sse2,+x87" "unsafe-fp-math"="false"
"use-soft-float"="false" }

!llvm.module.flags = !{!0, !1, !2}
!llvm.ident = !{!3}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{i32 7, !"PIE Level", i32 2}
!3 = !{!"clang version 8.0.1 (tags/RELEASE_801/final)"}
```

Nesse já da pra perceber que as variaveis criadas sao alocadas dentro do define da funcao main e sao dados numeros sequenciais como ids das variaveis

Código em c:

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  int x = 4;
  float y = 5;

  if (y == 5) {
    printf("Hello");
  }
  return 0;
}
```

Código do IR:

```
; ModuleID = 'test.c'
source_filename = "test.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@.str = private unnamed_addr constant [6 x i8] c"Hello\00", align 1

; Function Attrs: noinline nounwind optnone sspstrong uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca float, align 4
  store i32 0, i32* %1, align 4
  store i32 4, i32* %2, align 4
  store float 5.000000e+00, float* %3, align 4
  %4 = load float, float* %3, align 4
  %5 = fcmp oeq float %4, 5.000000e+00
  br i1 %5, label %6, label %8

; <label>:6:                                      ; preds = %0
  %7 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([6 x i8], [6 x i8]* @.str, i32 0,
i32 0))
  br label %8

; <label>:8:                                      ; preds = %6, %0
  ret i32 0
}
```

```
declare i32 @printf(i8*, ...) #1

attributes #0 = { noinline nounwind optnone sspstrong uwtable
"correctly-rounded-divide-sqrt-fp-math"="false" "disable-tail-calls"="false"
"less-precise-fpmad"="false" "min-legal-vector-width"="0" "no-frame-pointer-elim"="true"
"no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-jump-tables"="false"
"no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false" "no-trapping-math"="false"
"stack-protector-buffer-size"="8" "target-cpu"="x86-64"
"target-features"="+fxsr,+mmx,+sse,+sse2,+x87" "unsafe-fp-math"="false"
"use-soft-float"="false" }
attributes #1 = { "correctly-rounded-divide-sqrt-fp-math"="false" "disable-tail-calls"="false"
"less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf"
"no-infs-fp-math"="false" "no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false"
"no-trapping-math"="false" "stack-protector-buffer-size"="8" "target-cpu"="x86-64"
"target-features"="+fxsr,+mmx,+sse,+sse2,+x87" "unsafe-fp-math"="false"
"use-soft-float"="false" }

!llvm.module.flags = !{!0, !1, !2}
!llvm.ident = !{!3}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{i32 7, !"PIE Level", i32 2}
!3 = !{!"clang version 8.0.1 (tags/RELEASE_801/final)"}
```

Aqui dá pra perceber que ao comparar dois numeros, ele aloca um numero para guardar o y, e outro para alocar o 5. Depois usa um comando de comparacao com algo parecido com um goto e redireciona dependendo do resultado

Código em c:

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  int x = 4;
  float y = 5;

  if (y == 5) {
    printf("Hello");
  }

  while (x < 10) {
    x++;
  }
```

```
  return 0;
}
```

Código do IR:

```
; ModuleID = 'test.c'
source_filename = "test.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@.str = private unnamed_addr constant [6 x i8] c"Hello\00", align 1

; Function Attrs: noinline nounwind optnone sspstrong uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca float, align 4
  store i32 0, i32* %1, align 4
  store i32 4, i32* %2, align 4
  store float 5.000000e+00, float* %3, align 4
  %4 = load float, float* %3, align 4
  %5 = fcmp oeq float %4, 5.000000e+00
  br i1 %5, label %6, label %8

; <label>:6:                                      ; preds = %0
  %7 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([6 x i8], [6 x i8]* @.str, i32 0,
i32 0))
  br label %8

; <label>:8:                                      ; preds = %6, %0
  br label %9

; <label>:9:                                      ; preds = %12, %8
  %10 = load i32, i32* %2, align 4
  %11 = icmp slt i32 %10, 10
  br i1 %11, label %12, label %15

; <label>:12:                                     ; preds = %9
  %13 = load i32, i32* %2, align 4
  %14 = add nsw i32 %13, 1
  store i32 %14, i32* %2, align 4
  br label %9

; <label>:15:                                     ; preds = %9
```

```
  ret i32 0
}

declare i32 @printf(i8*, ...) #1

attributes #0 = { noinline nounwind optnone sspstrong uwtable
"correctly-rounded-divide-sqrt-fp-math"="false" "disable-tail-calls"="false"
"less-precise-fpmad"="false" "min-legal-vector-width"="0" "no-frame-pointer-elim"="true"
"no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-jump-tables"="false"
"no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false" "no-trapping-math"="false"
"stack-protector-buffer-size"="8" "target-cpu"="x86-64"
"target-features"="+fxsr,+mmx,+sse,+sse2,+x87" "unsafe-fp-math"="false"
"use-soft-float"="false" }
attributes #1 = { "correctly-rounded-divide-sqrt-fp-math"="false" "disable-tail-calls"="false"
"less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf"
"no-infs-fp-math"="false" "no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false"
"no-trapping-math"="false" "stack-protector-buffer-size"="8" "target-cpu"="x86-64"
"target-features"="+fxsr,+mmx,+sse,+sse2,+x87" "unsafe-fp-math"="false"
"use-soft-float"="false" }

!llvm.module.flags = !{!0, !1, !2}
!llvm.ident = !{!3}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{i32 7, !"PIE Level", i32 2}
!3 = !{!"clang version 8.0.1 (tags/RELEASE_801/final)"}
```

Com o while, ele tambem faz igual a um if, faz a comparacao que direciona a blocos (<label>) diferentes dependendo do resultado. A diferenca é que sempre tem uma opcao que volta na primeira <label> do while. E tambem quando o x é adicionado de 1 unidade, é usado uma variavel a mais para calcular a soma e depois salvar na variavel original

Código em c:

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  int i = 0;
  for (i = 0; i < 5; i++) {
    printf("%d\n", i);
  }

  return 0;
```

```
}
```

Código do IR:

```
; ModuleID = 'test.c'
source_filename = "test.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@.str = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1

; Function Attrs: noinline nounwind optnone sspstrong uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 0, i32* %2, align 4
  store i32 0, i32* %2, align 4
  br label %3

; <label>:3:                                      ; preds = %9, %0
  %4 = load i32, i32* %2, align 4
  %5 = icmp slt i32 %4, 5
  br i1 %5, label %6, label %12

; <label>:6:                                      ; preds = %3
  %7 = load i32, i32* %2, align 4
  %8 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @.str, i32 0,
i32 0), i32 %7)
  br label %9

; <label>:9:                                      ; preds = %6
  %10 = load i32, i32* %2, align 4
  %11 = add nsw i32 %10, 1
  store i32 %11, i32* %2, align 4
  br label %3

; <label>:12:                                     ; preds = %3
  ret i32 0
}

declare i32 @printf(i8*, ...) #1

attributes #0 = { noinline nounwind optnone sspstrong uwtable
"correctly-rounded-divide-sqrt-fp-math"="false" "disable-tail-calls"="false"
"less-precise-fpmad"="false" "min-legal-vector-width"="0" "no-frame-pointer-elim"="true"
```

```
"no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-jump-tables"="false"
"no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false" "no-trapping-math"="false"
"stack-protector-buffer-size"="8" "target-cpu"="x86-64"
"target-features"="+fxsr,+mmx,+sse,+sse2,+x87" "unsafe-fp-math"="false"
"use-soft-float"="false" }
attributes #1 = { "correctly-rounded-divide-sqrt-fp-math"="false" "disable-tail-calls"="false"
"less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf"
"no-infs-fp-math"="false" "no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false"
"no-trapping-math"="false" "stack-protector-buffer-size"="8" "target-cpu"="x86-64"
"target-features"="+fxsr,+mmx,+sse,+sse2,+x87" "unsafe-fp-math"="false"
"use-soft-float"="false" }

!llvm.module.flags = !{!0, !1, !2}
!llvm.ident = !{!3}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{i32 7, !"PIE Level", i32 2}
!3 = !{!"clang version 8.0.1 (tags/RELEASE_801/final)"}
```

Pelo que foi percebido os comandos de iteracao, trabalham da mesma forma, usando goto para ir a blocos <label> diferentes de acordo com a comparacao. O for nao foge da regra e trabalha da mesma forma, porem com 1 operacao a mais de adicao. Já o printf é um comando diferente que recebe as variaveis alocadas como argumento.

Parte 2

Gramática que especifica a Linguagem:

G = (V, T, P, S)
V = { S, bloco, declaracao, comandos, tipo, condicao, expressao, termo, relop, artop, atrop }
T = { int, char, real, se, enquanto, ==, <>, <, >, <=, >=, +, -, *, /, =, id, ε, numero, ;, . }
P = { S -> programa bloco, bloco -> inicio declaracao comandos fim, declaracao -> tipo id ; declaracao | ε , tipo -> int | char | real, comandos -> se (condicao) bloco comandos | enquanto (condicao) bloco comandos | id atrop expressao ; comandos | ε, condicao -> termo relop termo , expressao -> expressao artop termo | expressao atrop termo | termo, termo -> id | numero | (expressao),
relop -> == | <> | < | > | <= | >=,
artop -> + | - | * | /,
atrop -> =}

Tokens:

ID (identificadores -> contem letras, digitos e _)
numero (constantes numericas
programa
inicio
fim
enquanto
relop (operadores relacionais -> ==, <>, <, >, <=, >=)
artop (operadores aritmeticos -> +, -, *, /)
atrop (operadores de atribuicao -> =)
se
tipo (tipos de variaveis -> int, char, real)
simbolos (simbolos usados no codigo -> (,))

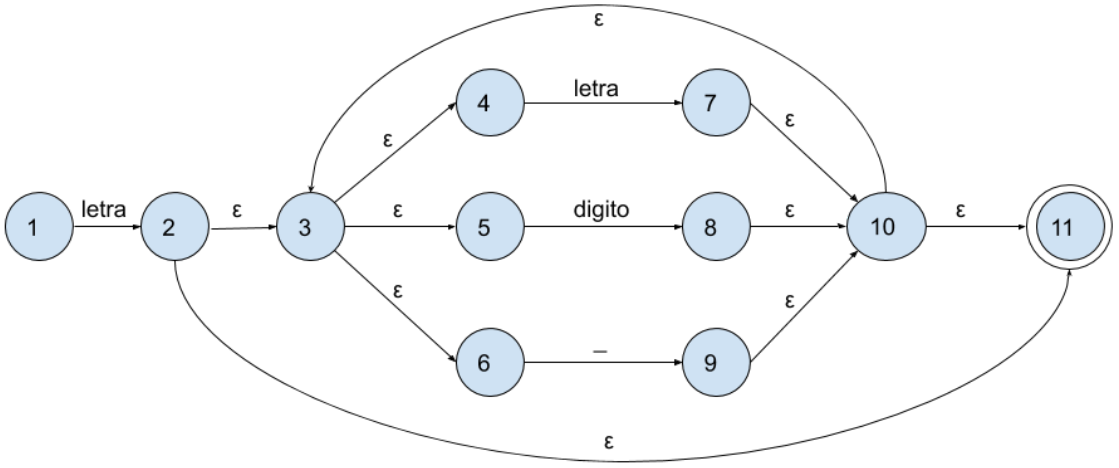
Expressões Regulares dos Lexemas:

letra -> [a-zA-Z]
digito -> [0-9]
digitos -> digito+
numero -> digitos (ε | (.digitos)) (ε | (E (+ | - | ε) digitos))
id -> letra (letra | _ | digito) *
programa -> programa
inicio -> inicio
fim -> fim
enquanto -> enquanto
relop -> == | <> | > | < | >= | <=
atrop -> =
artop -> + | - | * | /

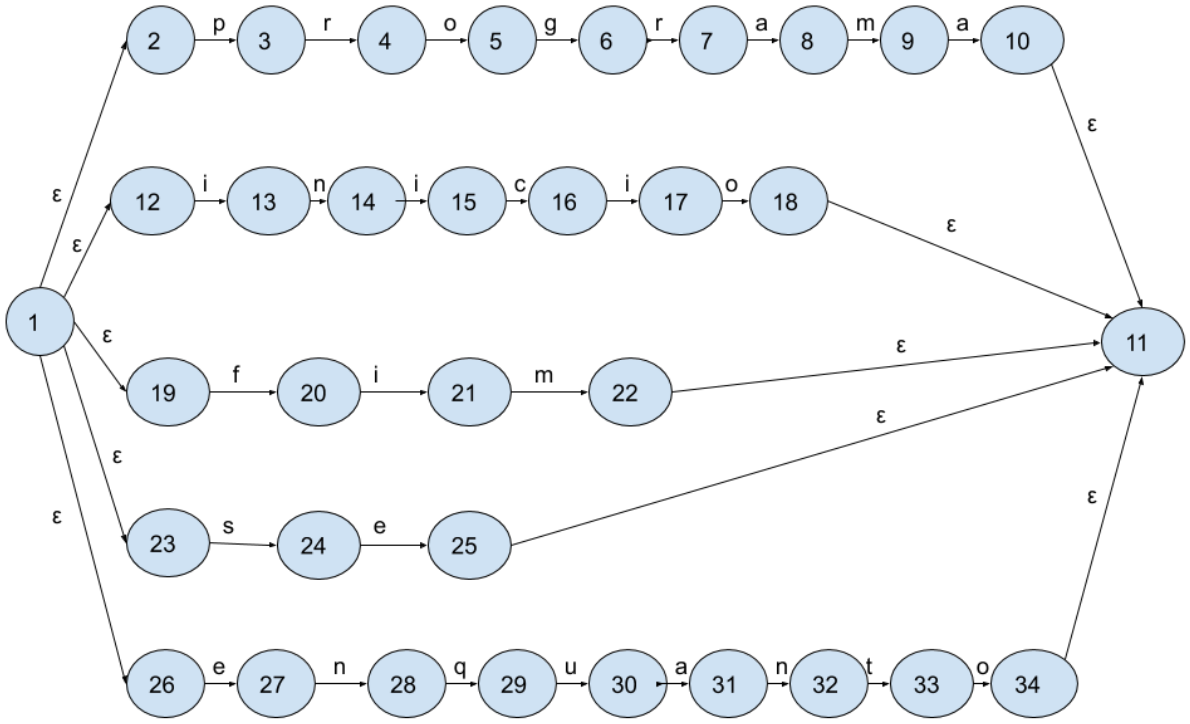
tipo -> int | char | real
se -> se
simbolos -> (|)

Parte 3
Diagramas

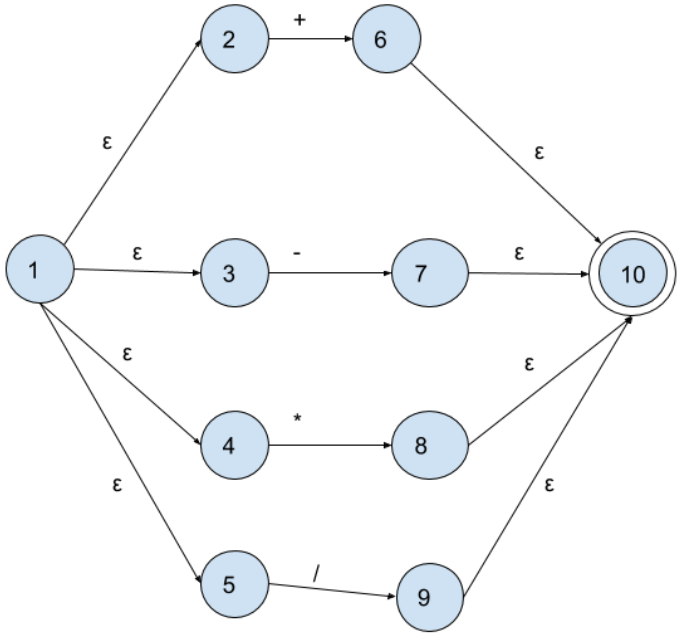
ID



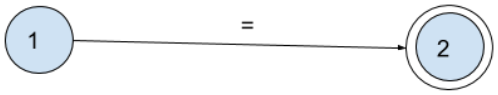
Palavras Reservadas



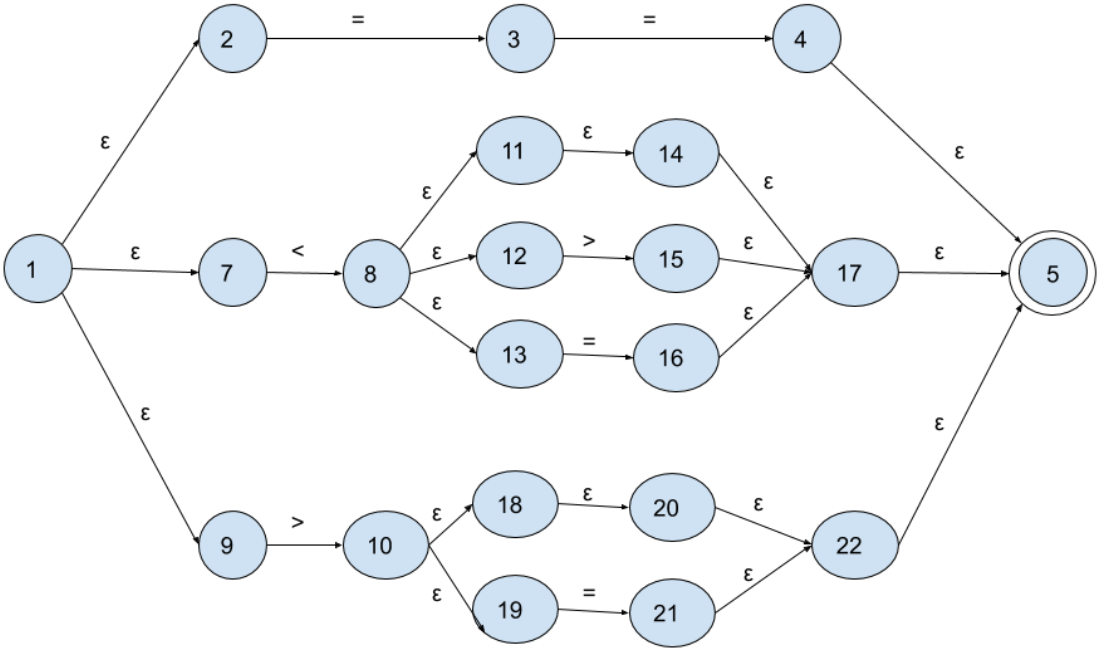
Artop (Operadores Aritmeticos)



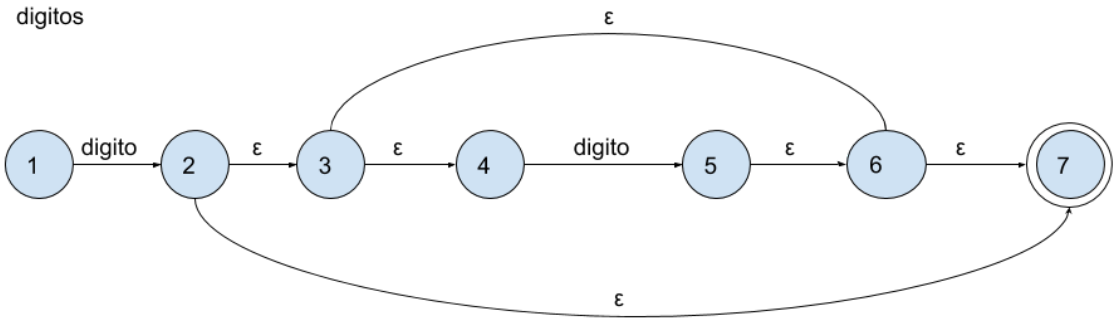
Atrop (Operador de Atribuicao)

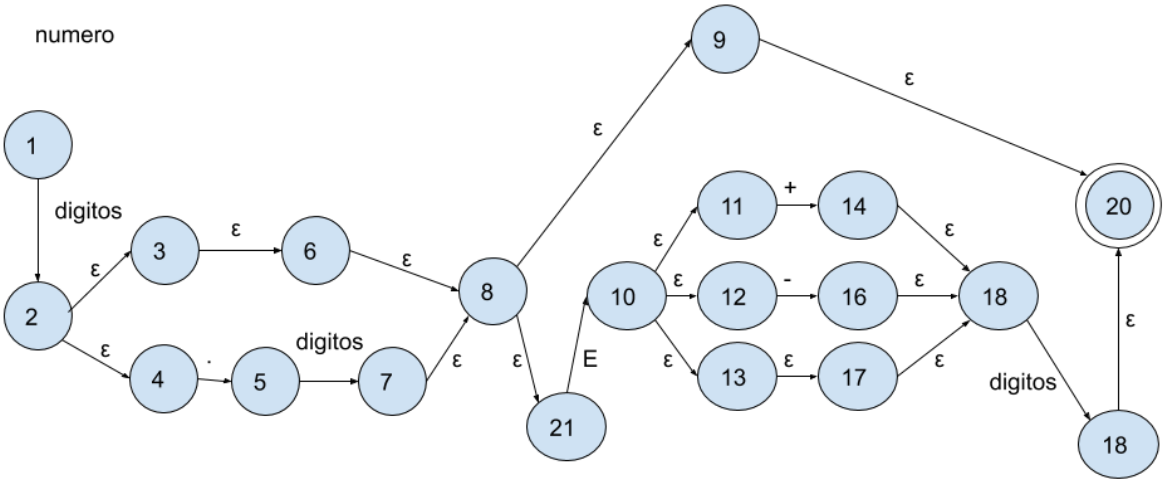


Relop (Operadores relacionais)

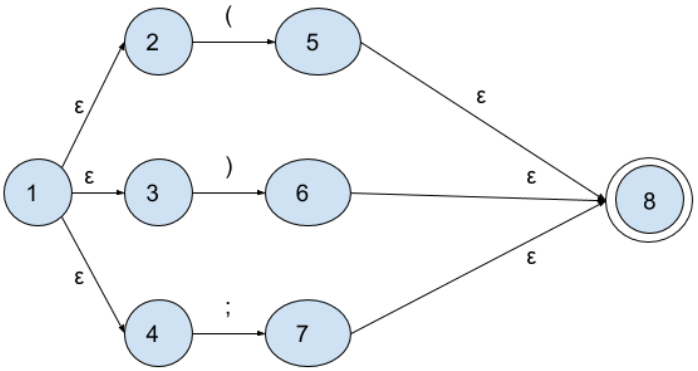


Numero

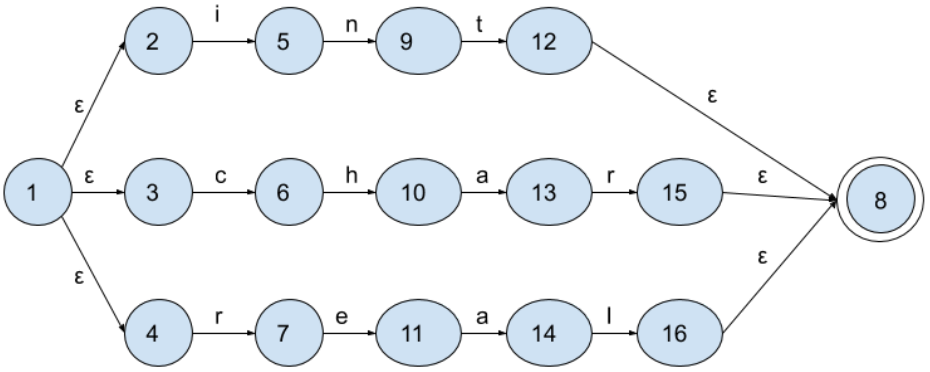




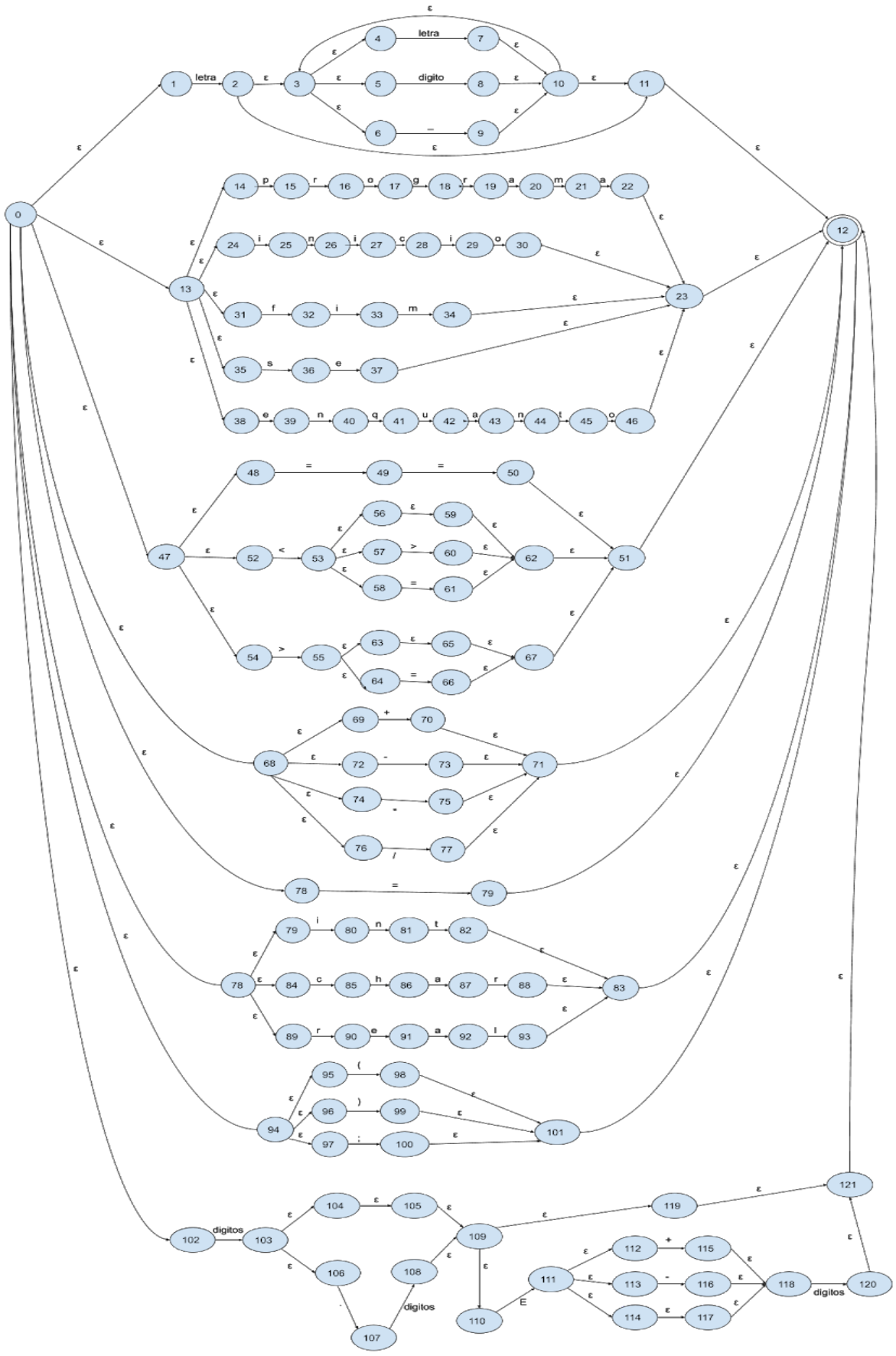
Simbolos



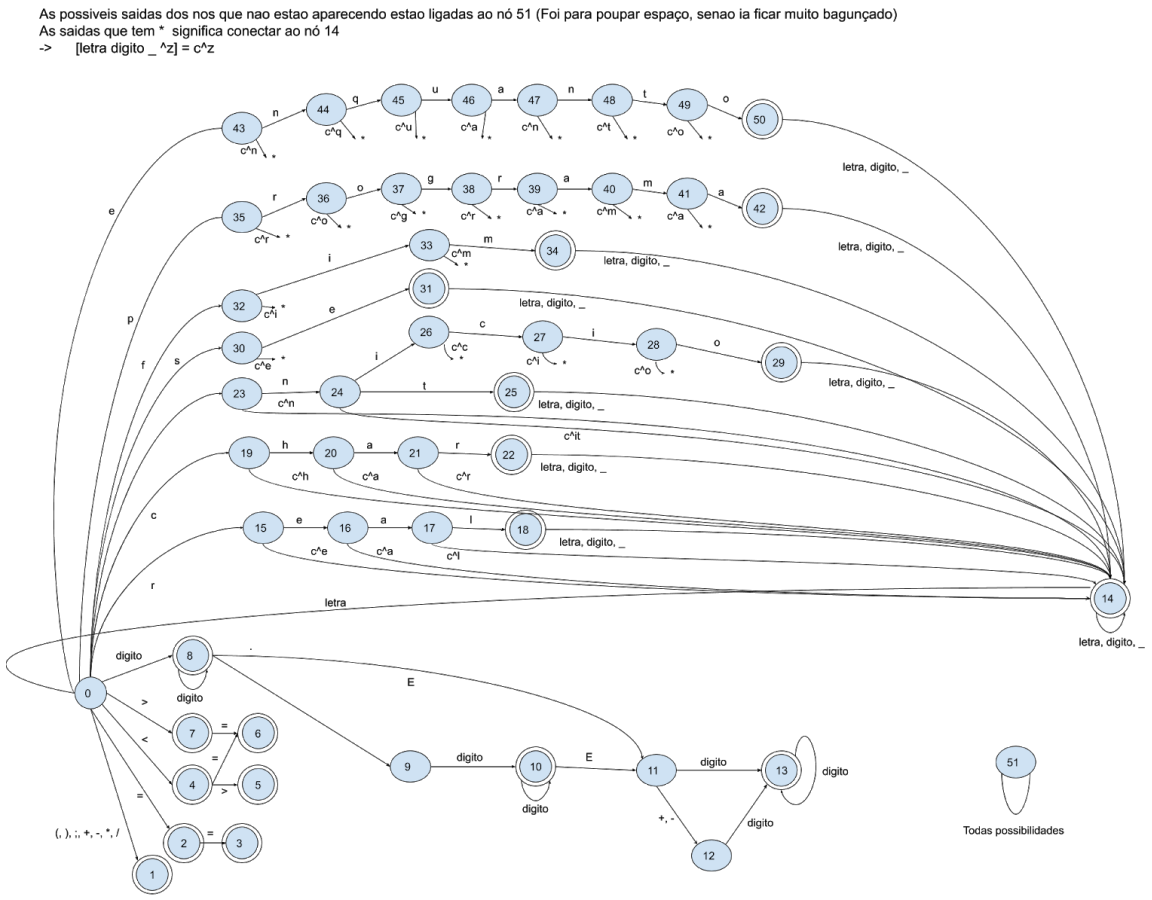
Tipo



Afnd Uniao



Afd da Uniao



Parte 4

Gramática sem ambiguidade e recursão à esquerda:

G = (V, T, P, S)

V = {S, bloco, declaracao, comandos, tipo, condicao, expressao, termo, relop, artop, atrop }

T = { int, char, real, se, enquanto, ==, <>, <, >, <=, >=, +, -, *, /, =, id, ε, numero }

P = {S -> programa bloco, bloco -> inicio declaracao comandos fim, declaracao -> tipo id ; declaracao | ε, tipo -> int | char | real, comandos -> se (condicao) bloco comandos | enquanto (condicao) bloco comandos | id atrop expressao comandos | ε, condicao -> termo relop termo, expressao -> termo expressao', expressao' -> artop termo expressao' | atrop termo expressao' | ε, termo -> id | numero, relop -> == | <> | < | > | <= | >=, artop -> + | - | * | /, atrop -> =}

FIRST e FOLLOW

FIRST(S) = {programa}

FIRST(bloco) = {inicio}

FIRST(declaracao) = {tipo, ε}

FIRST(tipo) = {int, char, real}

FIRST(comandos) = {se, enquanto, id, ε}

FIRST(condicao) = {id, numero}

FIRST(expressao) = {id, numero}

FIRST(expressao') = {+, -, *, /, =, ε}
FIRST(termo) = {id, numero, (}
FIRST(relop) = {== , <>, <, <=, >, >=}
FIRST(artop) = {+, -, *, /}
FIRST(atrop) = {=}

FOLLOW(S) = {$}
FOLLOW(bloco) = FOLLOW(S) + FIRST(comandos) + FOLLOW(comandos) = {$, se, enquanto, id, ε, fim}
FOLLOW(declaracao) = {se, enquanto, id, fim}
FOLLOW(tipo) = {id}
FOLLOW(comandos) = {fim}
FOLLOW(condicao) = {)}
FOLLOW(expressao) = {;,)}
FOLLOW(expressao') = FOLLOW(expressao) = {;,)}
FOLLOW(termo) = FIRST(relop) + FOLLOW(condicao) + FIRST(expressao') + FOLLOW(expressao) + FOLLOW(expressao') = {==, <>, <, <=, >, >=,), +, -, *, /, =, ε, ;}
FOLLOW(relop) = FIRST(termo) = {id, numero}
FOLLOW(artop) = FIRST(termo) = {id, numero}
FOLLOW(atrop) = FIRST(termo) = {id, numero}

Verificação de Gramática LL
Regra 1:
FIRST(tipo id ; declaracao) ∩ FIRST(ε) = ∅
FIRST(se (condicao) bloco comandos) ∩ FIRST(enquanto (condicao) bloco comandos) ∩ FIRST(id atrop expressao ; comandos) ∩ FIRST(ε) = ∅
FIRST(artop termo expressao') ∩ FIRST(atrop termo expressao') ∩ FIRST(ε) = ∅
FIRST(id) ∩ FIRST(numero) ∩ FIRST((expressao)) = ∅
FIRST(==) ∩ FIRST(<>) ∩ FIRST(<) ∩ FIRST(>) ∩ FIRST(<=) ∩ FIRST(>=) = ∅
FIRST(+) ∩ FIRST(-) ∩ FIRST(*) ∩ FIRST(/) = ∅

Regra 2:
FIRST(tipo id ; declaracao) ∩ FOLLOW(declaracao) = ∅
FIRST(se (condicao) bloco comandos) ∩ FOLLOW(comandos) = ∅
FIRST(enquanto (condicao) bloco comandos) ∩ FOLLOW(comandos) = ∅
FIRST(id atrop expressao ; comandos) ∩ FOLLOW(comandos) = ∅

FIRST(artop termo expressao') ∩ FOLLOW(expressao') = ∅
FIRST(atrop termo expressao') ∩ FOLLOW(expressao') = ∅

Como a Gramática obedece as 2 regras então ela é LL

Ordem das Produções:
1 - S -> programa bloco
2 - bloco -> inicio declaracao comandos fim
3 - declaracao -> tipo id ; declaracao
4 - declaracao -> ε
5 - tipo -> int
6 - tipo -> char
7 - tipo -> real
8 - comandos -> se (condicao) bloco comandos
9 - comandos -> enquanto (condicao) bloco comandos
10 - comandos -> id atrop expressao ; comandos
11 - comandos -> ε
12 - condicao -> termo relop termo
13 - expressao -> termo expressao'
14 - expressao' -> artop termo expressao'
15 - expressao' -> atrop termo expressao'
16 - expressao' -> ε
17 - termo -> id
18 - termo -> numero
19 - termo -> (expressao)
20 - relop -> ==
21 - relop -> <>
22 - relop -> <
23 - relop -> >
24 - relop -> <=
25 - relop -> >=
26 - artop -> +
27 - artop -> -
28 - artop -> *
29 - artop -> /
30 - atrop -> =

Tabela de Análise Preditiva

	programa	inicio	fim	id	;	int	char	real	se	(	)	enquanto	==	<>	<	<=	>	>=	+	-	*	/	=	$	numero
S	1																								
bloco		2																							
declaracao			4	4		3	3	3	4			4													
tipo						5	6	7																	
comandos			11	10					8			9													
condicao				12						12															12
expressao				13						13															13
expressao'					16						16								14	14	14	14	15		
termo				17						19															18
relop													20	21	22	24	23	25							
artop																			26	27	28	29			
atrop																							30		