Construção de Compiladores

Pedro Henrique Faria Teixeira

1ª, 2ª e 3ª Etapa do Projeto

Uberlândia - MG
2019

Execução dos Scripts e respectivas saídas:

Main.c # **Programa com somente o main estruturado:**

```c
#include <stdio.h>

int main(){

}
```

Main.txt # **Saída após a execução do código com o Clang:**

```
; ModuleID = 'main.c'
source_filename = "main.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 {
 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)"}
```

- Podemos observar que para este código somente o main e o return foram declarados. O return por padrão ja vem declarado no corpo da função main().

Int.c # **Programa que inclui uma variável do tipo int no corpo da main():**

```c
#include <stdio.h>

int main(){
   int i = 0;
}
```

Int.txt # **Saída após a execução do código com o Clang:**

```
; ModuleID = 'int.c'
source_filename = "int.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)"}
```

- Podemos observar que foi acrescentado 2 novas linhas na saída.
- 1ª alocando uma variável do tipo int .
- 2ª alocando o valor 0 na variável inteira.

If.c # **Programa que acrescenta um if ao corpo da main e com a várivel int declarada e faz operações de comparação e aritmética:**

```c
#include <stdio.h>

int main(){
   int i = 0;

   if (i > 0) {
       i = 1 + 2;
   }
}
```

If.txt **# Saída após a execução do código com o Clang:**

```
; ModuleID = 'if.c'
source_filename = "if.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
 store i32 0, i32* %1, align 4
 store i32 0, i32* %2, align 4
 %3 = load i32, i32* %2, align 4
 %4 = icmp sgt i32 %3, 0
 br i1 %4, label %5, label %6

; <label>:5:                                      ; preds = %0
 store i32 3, i32* %2, align 4
 br label %6

; <label>:6:                                      ; preds = %5, %0
 %7 = load i32, i32* %1, align 4
 ret i32 %7
```

```
}

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)"}
```

- Podemos observar que bastante coisa foi acrescentada.
- 1ª Foi alocado mais dois espaços na memória para o número 1 e 2 que se encontram dentro do if para ser adicionados a variável inteira.
- 2ª Ele faz o load da variável int para comparação no if.
- 3ª Chama o sinal de > para comparação.
- 4ª Chama os labels que foram declarados abaixo para fazer a operação de adição, e o load das variáveis 1 e 2.

While.c # **Programa que acrescenta um while e uma operação aritmética no código anterior If.c:**

```
#include <stdio.h>

int main(){
   int i = 0;
   if (i == 0) {
       i = 1 + 2;
   }
   while(i != 0) {
       i--;
   }
```

```
}
```

While.txt **# Saída após a execução do código com o Clang:**

```
; ModuleID = 'while.c'
source_filename = "while.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
 store i32 0, i32* %1, align 4
 store i32 0, i32* %2, align 4
 %3 = load i32, i32* %2, align 4
 %4 = icmp eq i32 %3, 0
 br i1 %4, label %5, label %6

; <label>:5:                                      ; preds = %0
 store i32 3, i32* %2, align 4
 br label %6

; <label>:6:                                      ; preds = %5, %0
 br label %7

; <label>:7:                                      ; preds = %10, %6
 %8 = load i32, i32* %2, align 4
 %9 = icmp ne i32 %8, 0
 br i1 %9, label %10, label %13

; <label>:10:                                     ; preds = %7
 %11 = load i32, i32* %2, align 4
 %12 = add nsw i32 %11, -1
 store i32 %12, i32* %2, align 4
 br label %7

; <label>:13:                                     ; preds = %7
 %14 = load i32, i32* %1, align 4
 ret i32 %14
```

```
}

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)"}
```

- Podemos observar que novamente muitas coisas foram acrescentadas.
- As mais importantes são:
- 1ª A operação de subtração que é o comando de adição com um -1 acrescentado que faz o inverso da adição.
- 2 ª Os novos labels criados para fazer a operação de desigualdade, load do i e o store para guardar na variável a cada iteração do while.

If_else.c # **Programa que acrescenta um if e um else com operações aritméticas básicas:**

```c
#include <stdio.h>

int main(){
   int i = 0;
   int j = 2;
   if (i > 0) {
       j = 1 - i;
   } else if(i == 0) {
```

```
        i = j + 2;
    }
}
```

If_else.txt # **Saída após a execução do código com o Clang:**

```
; ModuleID = 'if_else.c'
source_filename = "if_else.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 i32, align 4
 store i32 0, i32* %1, align 4
 store i32 0, i32* %2, align 4
 store i32 2, i32* %3, align 4
 %4 = load i32, i32* %2, align 4
 %5 = icmp sgt i32 %4, 0
 br i1 %5, label %6, label %9

; <label>:6:                                      ; preds = %0
 %7 = load i32, i32* %2, align 4
 %8 = sub nsw i32 1, %7
 store i32 %8, i32* %3, align 4
 br label %16

; <label>:9:                                      ; preds = %0
 %10 = load i32, i32* %2, align 4
 %11 = icmp eq i32 %10, 0
 br i1 %11, label %12, label %15

; <label>:12:                                     ; preds = %9
 %13 = load i32, i32* %3, align 4
 %14 = add nsw i32 %13, 2
 store i32 %14, i32* %2, align 4
 br label %15
```

```
; <label>:15:                                     ; preds = %12, %9
 br label %16

; <label>:16:                                     ; preds = %15, %6
 %17 = load i32, i32* %1, align 4
 ret i32 %17
}

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)"}
```

- Como anteriormente já havíamos visto como o if( ) funciona, agora a única diferença é que foi acrescentado o else if( ).
- As diferenças são:
- 1ª Foi alocado mais espaço a uma nova variável.
- 2 ª O comando de subtração na linha 20 do código, junto com o load dos inteiros.
- 3ª A chamada de mais uma operação de comparação que agora é a igualdade.
- 4 ª O comando de adição junto com o load da variável J + um inteiro 2 e o store na variável i.

Projeto da Linguagem

Linguagem:

Gramática:

G = (V, T, P, S)

V = {S, Bloco, Declarações, Comandos, Condição, Expressões, Tipo, Term, Relop}

T = {programa, início, fim, se, senão, então, faça, enquanto, para, id, >, <, >=, <=, ==, <>, = , +, -, *, /, char, int, real, not, and, or, [0-9], [a-zA-Z],ε}

P = { S -> programa Bloco,
Bloco -> Declarações Comandos | Declarações | Comandos | ε,
Declarações -> Tipo id,
Tipo -> int | char | real
Comandos -> se Condição Bloco | se Condição Bloco senão Bloco | enquanto Condição Bloco | Expressão, faça Bloco enquanto Condição, para id = [0...9] Condição id = [0...9] | ε,
Condição -> Term Relop Term |Term Relop Term Booleano Term Booleano Term | Term,
Expressão -> Term Relop Term | Term,
Term -> id | dígitos,
Relop -> <, >, >=, <=, ==, <>,
Booleano -> and, or
}

Tokens:

Identificadores (id)
Operadores (relop)
Dígitos
Enquanto
Programa
Se

Senão
Então
Início
Fim
Tipo
Faça
Booleanos
Símbolos

Padrões (expressões regulares):

- dígito -> [0...9]
- dígitos -> digito+
- letra -> [a-zA-Z]
- id -> letra(letra | dígito)*
- relop -> < | > | <= | >= | == | <> | or | not | and
- programa -> programa
- inicio -> inicio
- fim -> fim
- se -> se
- então -> então
- senão -> senão
- enquanto -> enquanto
- para -> para
- faça -> faça
- símbolos -> (,), [,], ‘, “, ; , .

Análise Léxica

O arquivo de input do projeto contém os seguintes lexemas:

```
<> <= ==
int aolf [ 2 ] ;
```

```
programa
    inicio
        se ( a >= b ) or ( a < b )  or ( a == b )
            skin + doi
        entao
            skol - o3as
    fim

se s
    faca
        ( troi / skoll ) + cha54a * pol
entao
    return troi + pol

faca
    bloco
enquanto ( condicao ) ;
```

Resultados:

Token <
Tipo: relop
Linha: 1 Coluna: [1, 4]

Palavra <> aceita
Tipo: relop
Linha: 1 Coluna: [2]

Token <
Tipo: relop
Linha: 1 Coluna: [1, 4]

Palavra <= aceita
Tipo: relop
Linha: 1 Coluna: [5, 7, 8]

Token =
Tipo: relop

Linha: 1 Coluna: [5, 7, 8]

Palavra == aceita
Tipo: relop
Linha: 1 Coluna: [5, 7, 8]

Token i
Tipo: letra
Linha: 2 Coluna: [1]

Token n
Tipo: letra
Linha: 2 Coluna: [2]

Palavra int aceita
Tipo: letra
Linha: 2 Coluna: [3]

Token a
Tipo: letra
Linha: 2 Coluna: [5]

Token o
Tipo: letra
Linha: 2 Coluna: [6]

Token l
Tipo: letra
Linha: 2 Coluna: [7]

Palavra aolf aceita
Tipo: letra
Linha: 2 Coluna: [8]

Palavra [aceita
Tipo: simbolo
Linha: 2 Coluna: [10]

Palavra 2 aceita
Tipo: digito
Linha: 2 Coluna: [12]

Palavra] aceita
Tipo: simbolo
Linha: 2 Coluna: [14]

Palavra ; aceita
Tipo: simbolo
Linha: 2 Coluna: [16]

Token p
Tipo: letra
Linha: 4 Coluna: [1]

Token r
Tipo: letra
Linha: 4 Coluna: [2, 5]

Token o
Tipo: letra
Linha: 4 Coluna: [3]

Token g
Tipo: letra
Linha: 4 Coluna: [4]

Token r
Tipo: letra
Linha: 4 Coluna: [2, 5]

Token a
Tipo: letra
Linha: 4 Coluna: [6, 8]

Token m
Tipo: letra
Linha: 4 Coluna: [7]

Palavra programa aceita
Tipo: letra
Linha: 4 Coluna: [6, 8]

Token i
Tipo: letra

Linha: 5 Coluna: [5, 7, 9]

Token n
Tipo: letra
Linha: 5 Coluna: [6]

Token i
Tipo: letra
Linha: 5 Coluna: [5, 7, 9]

Token c
Tipo: letra
Linha: 5 Coluna: [8]

Token i
Tipo: letra
Linha: 5 Coluna: [5, 7, 9]

Palavra inicio aceita
Tipo: letra
Linha: 5 Coluna: [10]

Token s
Tipo: letra
Linha: 6 Coluna: [9]

Palavra se aceita
Tipo: letra
Linha: 6 Coluna: [10]

Palavra (aceita
Tipo: simbolo
Linha: 6 Coluna: [12, 26, 40]

Palavra a aceita
Tipo: letra
Linha: 2 Coluna: [5]

Token >
Tipo: relop
Linha: 6 Coluna: [16]

Palavra >= aceita
Tipo: relop
Linha: 6 Coluna: [17, 44, 45]

Palavra b aceita
Tipo: letra
Linha: 6 Coluna: [19, 32, 47]

Palavra) aceita
Tipo: simbolo
Linha: 6 Coluna: [21, 34, 49]

Token o
Tipo: letra
Linha: 6 Coluna: [23, 37]

Palavra or aceita
Tipo: letra
Linha: 6 Coluna: [24, 38]

Palavra (aceita
Tipo: simbolo
Linha: 6 Coluna: [12, 26, 40]

Palavra a aceita
Tipo: letra
Linha: 2 Coluna: [5]

Palavra < aceita
Tipo: relop
Linha: 1 Coluna: [1, 4]

Palavra b aceita
Tipo: letra
Linha: 6 Coluna: [19, 32, 47]

Palavra) aceita
Tipo: simbolo
Linha: 6 Coluna: [21, 34, 49]

Token o
Tipo: letra

Linha: 6 Coluna: [23, 37]

Palavra or aceita
Tipo: letra
Linha: 6 Coluna: [24, 38]

Palavra (aceita
Tipo: simbolo
Linha: 6 Coluna: [12, 26, 40]

Palavra a aceita
Tipo: letra
Linha: 2 Coluna: [5]

Token =
Tipo: relop
Linha: 1 Coluna: [5, 7, 8]

Palavra == aceita
Tipo: relop
Linha: 1 Coluna: [5, 7, 8]

Palavra b aceita
Tipo: letra
Linha: 6 Coluna: [19, 32, 47]

Palavra) aceita
Tipo: simbolo
Linha: 6 Coluna: [21, 34, 49]

Token s
Tipo: letra
Linha: 7 Coluna: [13]

Token k
Tipo: letra
Linha: 7 Coluna: [14]

Token i
Tipo: letra
Linha: 7 Coluna: [15, 22]

Palavra skin aceita
Tipo: letra
Linha: 7 Coluna: [16]

Palavra + aceita
Tipo: operador
Linha: 7 Coluna: [18]

Token d
Tipo: letra
Linha: 7 Coluna: [20]

Token o
Tipo: letra
Linha: 7 Coluna: [21]

Palavra doi aceita
Tipo: letra
Linha: 7 Coluna: [15, 22]

Token e
Tipo: letra
Linha: 8 Coluna: [9]

Token n
Tipo: letra
Linha: 8 Coluna: [10]

Token t
Tipo: letra
Linha: 8 Coluna: [11]

Token a
Tipo: letra
Linha: 8 Coluna: [12]

Palavra entao aceita
Tipo: letra
Linha: 8 Coluna: [13]

Token s
Tipo: letra

Linha: 9 Coluna: [13, 23]

Token k
Tipo: letra
Linha: 9 Coluna: [14]

Token o
Tipo: letra
Linha: 9 Coluna: [15, 20]

Palavra skol aceita
Tipo: letra
Linha: 9 Coluna: [16]

Palavra - aceita
Tipo: operador
Linha: 9 Coluna: [18]

Token o
Tipo: letra
Linha: 9 Coluna: [15, 20]

Token 3
Tipo: digito
Linha: 9 Coluna: [21]

Token a
Tipo: letra
Linha: 9 Coluna: [22]

Palavra o3as aceita
Tipo: letra
Linha: 9 Coluna: [13, 23]

Token f
Tipo: letra
Linha: 10 Coluna: [5]

Token i
Tipo: letra
Linha: 10 Coluna: [6]

Palavra fim aceita
Tipo: letra
Linha: 10 Coluna: [7]

Token s
Tipo: letra
Linha: 6 Coluna: [9]

Palavra se aceita
Tipo: letra
Linha: 6 Coluna: [10]

Palavra s aceita
Tipo: letra
Linha: 6 Coluna: [9]

Token f
Tipo: letra
Linha: 13 Coluna: [5]

Token a
Tipo: letra
Linha: 13 Coluna: [6, 8]

Token c
Tipo: letra
Linha: 13 Coluna: [7]

Palavra faca aceita
Tipo: letra
Linha: 13 Coluna: [6, 8]

Palavra (aceita
Tipo: simbolo
Linha: 6 Coluna: [12, 26, 40]

Token t
Tipo: letra
Linha: 14 Coluna: [11]

Token r
Tipo: letra

Linha: 14 Coluna: [12]

Token o
Tipo: letra
Linha: 14 Coluna: [13, 20, 38]

Palavra troi aceita
Tipo: letra
Linha: 14 Coluna: [14]

Palavra / aceita
Tipo: operador
Linha: 14 Coluna: [16]

Token s
Tipo: letra
Linha: 14 Coluna: [18]

Token k
Tipo: letra
Linha: 14 Coluna: [19]

Token o
Tipo: letra
Linha: 14 Coluna: [13, 20, 38]

Token l
Tipo: letra
Linha: 14 Coluna: [21, 22, 39]

Palavra skoll aceita
Tipo: letra
Linha: 14 Coluna: [21, 22, 39]

Palavra) aceita
Tipo: simbolo
Linha: 6 Coluna: [21, 34, 49]

Palavra + aceita
Tipo: operador
Linha: 7 Coluna: [18]

Token c
Tipo: letra
Linha: 14 Coluna: [28]

Token h
Tipo: letra
Linha: 14 Coluna: [29]

Token a
Tipo: letra
Linha: 14 Coluna: [30, 33]

Token 5
Tipo: digito
Linha: 14 Coluna: [31]

Token 4
Tipo: digito
Linha: 14 Coluna: [32]

Palavra cha54a aceita
Tipo: letra
Linha: 14 Coluna: [30, 33]

Palavra * aceita
Tipo: operador
Linha: 14 Coluna: [35]

Token p
Tipo: letra
Linha: 14 Coluna: [37]

Token o
Tipo: letra
Linha: 14 Coluna: [13, 20, 38]

Palavra pol aceita
Tipo: letra
Linha: 14 Coluna: [21, 22, 39]

Token e
Tipo: letra

Linha: 8 Coluna: [9]

Token n
Tipo: letra
Linha: 8 Coluna: [10]

Token t
Tipo: letra
Linha: 8 Coluna: [11]

Token a
Tipo: letra
Linha: 8 Coluna: [12]

Palavra entao aceita
Tipo: letra
Linha: 8 Coluna: [13]

Token r
Tipo: letra
Linha: 16 Coluna: [5, 9, 13]

Token e
Tipo: letra
Linha: 16 Coluna: [6]

Token t
Tipo: letra
Linha: 16 Coluna: [7, 12]

Token u
Tipo: letra
Linha: 16 Coluna: [8]

Token r
Tipo: letra
Linha: 16 Coluna: [5, 9, 13]

Palavra return aceita
Tipo: letra
Linha: 16 Coluna: [10]

Token t
Tipo: letra
Linha: 14 Coluna: [11]

Token r
Tipo: letra
Linha: 14 Coluna: [12]

Token o
Tipo: letra
Linha: 14 Coluna: [13, 20, 38]

Palavra troi aceita
Tipo: letra
Linha: 14 Coluna: [14]

Palavra + aceita
Tipo: operador
Linha: 7 Coluna: [18]

Token p
Tipo: letra
Linha: 14 Coluna: [37]

Token o
Tipo: letra
Linha: 14 Coluna: [13, 20, 38]

Palavra pol aceita
Tipo: letra
Linha: 14 Coluna: [21, 22, 39]

Token f
Tipo: letra
Linha: 13 Coluna: [5]

Token a
Tipo: letra
Linha: 13 Coluna: [6, 8]

Token c
Tipo: letra

Linha: 13 Coluna: [7]

Palavra faca aceita
Tipo: letra
Linha: 13 Coluna: [6, 8]

Token b
Tipo: letra
Linha: 19 Coluna: [5]

Token l
Tipo: letra
Linha: 19 Coluna: [6]

Token o
Tipo: letra
Linha: 19 Coluna: [7, 9]

Token c
Tipo: letra
Linha: 19 Coluna: [8]

Palavra bloco aceita
Tipo: letra
Linha: 19 Coluna: [7, 9]

Token e
Tipo: letra
Linha: 20 Coluna: [1]

Token n
Tipo: letra
Linha: 20 Coluna: [2, 6, 14]

Token q
Tipo: letra
Linha: 20 Coluna: [3]

Token u
Tipo: letra
Linha: 20 Coluna: [4]

Token a
Tipo: letra
Linha: 20 Coluna: [5, 18]

Token n
Tipo: letra
Linha: 20 Coluna: [2, 6, 14]

Token t
Tipo: letra
Linha: 20 Coluna: [7]

Palavra enquanto aceita
Tipo: letra
Linha: 20 Coluna: [8, 13, 19]

Palavra (aceita
Tipo: simbolo
Linha: 6 Coluna: [12, 26, 40]

Token c
Tipo: letra
Linha: 20 Coluna: [12, 17]

Token o
Tipo: letra
Linha: 20 Coluna: [8, 13, 19]

Token n
Tipo: letra
Linha: 20 Coluna: [2, 6, 14]

Token d
Tipo: letra
Linha: 20 Coluna: [15]

Token i
Tipo: letra
Linha: 20 Coluna: [16]

Token c
Tipo: letra

Linha: 20 Coluna: [12, 17]

Token a
Tipo: letra
Linha: 20 Coluna: [5, 18]

Palavra condicao aceita
Tipo: letra
Linha: 20 Coluna: [8, 13, 19]

Palavra) aceita
Tipo: simbolo
Linha: 6 Coluna: [21, 34, 49]

Palavra ; aceita
Tipo: simbolo
Linha: 2 Coluna: [16]

Cada token é apresentado com o seu tipo, linha e coluna. A coluna é apresentada por um array, pois pode haver mais de um token em um lexema, então mostra-se todas as linha que contém aquele token naquele lexema.

A tabela de símbolos está em anexo no zip.