PONTIFÍCIA UNIVERSIDADE CATÓLICA DO PARANÁ
ESCOLA POLITÉCNICA
CURSO DE GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

ERICK LEMMY DOS SANTOS OLIVEIRA

GABRIELLE BATISTA GARCIA

MATHEUS HERMAN BERNARDIM ANDRADE

LEANDRO RICARDO GUIMARÃES

PROJETO COMPILADOR:
FASE 2

CURITIBA

2023

ERICK LEMMY DOS SANTOS OLIVEIRA
GABRIELLE BATISTA GARCIA
MATHEUS HERMAN BERNARDIM ANDRADE
LEANDRO RICARDO GUIMARÃES

PROJETO COMPILADOR:
FASE 2

Projeto entregue como requisito parcial para avaliação final do projeto desenvolvido durante o semestre, para a disciplina de Linguagens Formais e Compiladores, do Curso de Engenharia de Computação, da Pontifícia Universidade Católica do Paraná – PUCPR.

Orientador: Prof. Frank Coelho de Alcantara

CURITIBA
2023

SUMÁRIO

1 PROJETO COMPILADOR

1.1 CONTEXTO

O objetivo do presente projeto é desenvolver um compilador para aplicação em um sistema embarcado, através da estruturação de uma nova linguagem de programação. Para o seu desenvolvimento serão utilizados alguns recursos de software, tais como a plataforma *Github* para controle de versionamento dos códigos fonte, o *Visual Studio Code* para a geração do *Assembly*. Em relação ao hardware, será utilizado o *Rapsberry* Pi 3, baseado em arquitetura ARM de 64bits, onde todos os exemplos gerados pela linguagem serão executados.

1.2 DEFINIÇÃO DA LINGUAGEM

A linguagem foi desenvolvida com base na sintaxe do Python, incorporando elementos da linguagem C, como a declaração de tipos de dados, uso de ponto e vírgula e chaves para definir blocos de código. A seguir, apresentamos as principais características da linguagem, com ênfase nas regras de produção expandidas para interação com hardware.

1.2.1 Regras de Produção

As regras de produção da linguagem foram definidas como:

```
1    program -> block
2    block -> { decls stmts }
3     decls -> decls decl | null
4     decl -> type id;
5 params -> param, params | params | null
6 param -> type id
7 func_decl -> def id ( params ) -> type: block | def id ( params ): block
8 args -> arglist | expr | null
9     arglist -> arglist, bool | bool
10| arglist, num | num
11 | arglist, id | id
12 func_call -> id ( args );
13    type -> int | float16 | bool | void | null
14 stmts -> stmts stmt | null
```

As mudanças básicas em relação ao bloco de declaração fornecido estão relacionadas à expansão da sintaxe para acomodar a criação de funções, a introdução de parâmetros de função e alterações na forma de uso, a seguir são apresentadas as principais alterações.

- **Adição de funções e Parâmetros**
  - A produção ***func_decl*** foi introduzida para permitir a declaração de funções com parâmetros e tipo de retorno.
  - A produção ***params*** foi adicionada para definir a lista de parâmetros em uma função.
  - A produção ***param*** permite a definição de parâmetros de função com seu tipo e nome.
  - A produção ***func_call*** foi introduzida para representar a possibilidade da chamada de uma função dentro de um bloco de instruções.

- **Tipos Básicos**
  - A produção ***type*** continua a permitir tipos básicos, como ***int***, ***float16***, ***bool*** e pode ser ***void***, ou seja, nulo.

- **Identificadores**
  - A produção ***id*** permanece semelhante, permitindo identificadores compostos por letras maiúsculas e minúsculas, bem como dígitos numéricos, e é usada para representar entidades no código, como funções.

- **Outras Instruções**
  - A produção ***stmt*** foi expandida para acomodar instruções específicas relacionadas ao hardware e funções.

- **Blocos de Função**
  - As produções *func_decl* e block são usadas para definir o escopo de uma função, incluindo a lista de parâmetros e o corpo da função.

1.2.2 Lexema Básico

O lexema básico da linguagem desenvolvida se deu da seguinte forma:

```
1     stmt → assign
2     | func_decl
3| func_call
4| return_stmt
5| if ( bool ): { stmt }
```

```
6| if ( bool ): { stmt } else: { stmt }
7    | if ( bool ): { stmt } else: stmt
8    | if ( bool ): { stmt } elif (bool): { stmt }
9    | for (var → num; bool; var → var + num): { stmt }
10| while ( bool ): { stmt }
11| do { stmt } while ( bool );
12   | break;
13   | id (args) { stmt }
14   | return bool;
15   | block
16   | pin(id, num, id)
17   | write(id, bool)
18   | read(id)
19   | delay(num)
20   var → var | id
21   | var [ bool ] | id
22   return_stmt → return expr;
23   assign → id: type = expr | var [ bool ] = expr;
```

As mudanças básicas em relação ao lexema básico fornecido estão relacionadas à expansão da sintaxe e adição de instruções específicas para interação com hardware. A seguir são apresentadas as principais adições:

- **Adição de funções e Parâmetros**
  - A produção ***return_stmt*** foi introduzida para representar a possibilidade de retorno dentro de um bloco de estado.

- **Tipos Básicos**
  - A produção ***return_stmt*** foi introduzida para representar a possibilidade de retorno dentro de um bloco de instruções.

- **Identificadores**
  - A produção ***var*** foi criada para representar uma variável, que é um tipo específico de entidade que pode armazenar e conter valores. Variáveis têm um nome (representado por um ***id***) e um tipo de dados (representado por *type* na gramática). ***var*** é usado para definir e trabalhar com variáveis dentro das regras gramaticais.

- **Outras Instruções**
  - A produção ***stmt*** foi expandida para acomodar instruções específicas relacionadas ao hardware e funções.
  - A produção ***assign*** foi introduzida para representar a atribuição de valor a uma variável ou elemento de um array, seguido pelo tipo (***type***) e a expressão (***expr***) ou valor a ser atribuído.

1.2.3 Regras de Produção para Expressões

```
1     bool → join { || Join }
2     join → equality { && equality }
3    equality → rel { ( == | != ) rel }
4    rel → expr { (< | <= | >= | >) expr }
5     expr → term { ( + | - ) term }
6    term → unary { ( * | / ) unary }
7    unary → ( ! | - ) unary | factor
8    factor → ( bool ) | var | num | real | true | false
9     bin_op → expr + expr
10| expr - expr
11| expr * expr
12| expr / expr
13| expr == expr
14| expr != expr
15| expr < expr
16| expr <= expr
17| expr > expr
18| expr >= expr
```

De modo geral, as modificações realizadas nas regras de expressões tornaram as regras mais estruturadas e legíveis, enfatizando a hierarquia das operações e a relação entre os diferentes níveis da gramática.

1. ***bool***: Foi adicionado chaves { } para indicar que o operador lógico || se aplica a expressões *join*. Isso torna mais claro que *join* é a base da expressão *bool*.
2. ***join***: Foi adicionado chaves { } para indicar que o operador lógico && se aplica a expressões *equality*. Isso enfatiza a estrutura hierárquica da gramática.

3. ***equality***: Foi adicionado chaves { } para indicar que os operadores de igualdade (== e !=) se aplicam a expressões *rel*. Isso destaca a relação entre *rel* e operações de igualdade.

4. ***rel***: Foi adicionado chaves { } para indicar que os operadores de comparação se aplicam a expressões *expr*. Isso simplifica a regra e torna a relação mais clara.

5. ***expr***: Foi adicionado chaves { } para indicar que os operadores de adição e subtração se aplicam a expressões *term*. Isso simplifica a regra e destaca as operações aritméticas.

6. ***term***: Foi adicionado chaves { } para indicar que os operadores de multiplicação e divisão se aplicam a expressões *unary*. Isso torna a regra mais clara.

7. ***unary***: A regra foi mantida a mesma.

8. ***fator***: A regra foi mantida a mesma.

1.2.4 Tratamento de Letras e Números

```
1     id → [a-zA-Z_][a-zA-Z_0-9]*
2 var → [a-zA-Z-Z0-9]+
3 num → [0-9]+
4 real → [0-9].[0-9]
5 int → num
6    float16 → real
7    bool → true | false
```

1.2.5 Comunicação com o Hardware

Para comunicação com o hardware foram criadas e adicionadas as seguintes regras ao lexema básico:

```
pin(nome, pino, direcao)

write(nome, valor)

read(nome)

delay(ms)

stmt → id (args) { stmt }

    |   pin(id, num, id)
    |   read(id)
    |   write(id, bool)
    |   delay(num)
```

2 EXEMPLOS DE APLICAÇÕES

A seguir, estão alguns exemplos de código que demonstram as funcionalidades da linguagem criada, abrangendo diferentes aspectos, como declaração de variáveis, expressões matemáticas e interação com hardware.

2.1 EXEMPLO BÁSICO

```
1     // Este eh um comentário
1     // Declara uma variável inteira
2     numero: int = 10;
3    led_on: int = 1;
4    led_off: int = 0;
5
6    // Declara uma variável real
7     pi: float16 = 3.14159;
8
9     // Declara uma variável booleana
10    ligado: bool = true;
11
12    // Declara uma função para somar dois números
13    def somar(a: int, b: int) -> int: {
14            return a + b;
15    }
16
17    // Declara uma função para calcular a área de um círculo
18    def area_circulo(raio: float16) -> float16: {
19            return PI * raio * raio;
20    }
21
22    def main() -> int: {
23            soma: int = somar(numero, numero);
24            return 0;
25    }
```

2.2 INTERAÇÃO COM O HARDWARE

2.2.1 EXEMPLO 1: Piscar LED com Intervalo de 1 segundo

```
//    Declara um pino como saída
pin(led, GPIO_2, GPIO_OUT);

def main() -> int: {
      while (true): {
            //     Lê o estado atual do LED
            estado_led: int = read(led);

            //     Inverte o estado do LED
            if (estado_led == 0):
                  write(led, 1);
            else:
                  write(led, 0);

            //     Aguarda 1 segundo
            delay(1000);
    }
    return 0;
}
```

2.2.2 EXEMPLO 2: Controle do LED com Botão

```
//    Declara um pino como saída para o LED
pin(led, GPIO_2, GPIO_OUT);

//    Declara um pino como entrada para o botão
pin(botao, GPIO_3, GPIO_IN);

def main() -> int: {
      while (true): {
            //     Lê o estado atual do botão
            estado_botao: int = read(botao);
            //     Se o botão estiver pressionado, alterna o estado do LED
            if (estado_botao == 1): {
                  estado_led = read(led);
                  if (estado_led == 0):
                        write(led, 1);
                  else:
                        write(led, 0);
```

```
                    }

                    //      Aguarda um curto período de tempo para evitar leituras múltiplas
            do botão
                    delay(100);
                }
                return 0;
        }
```

2.2.3 EXEMPLO 3: Controle de Intensidade do LED com Potenciômetro

```
//      Declara um pino como saída para o LED
pin(led, GPIO_2, GPIO_OUT);

//      Declara um pino como entrada analógica para o potenciômetro (ADC)
pin(potenciometro, ADC_0, GPIO_IN);

def ler_valor_potenciometro() -> int: {
        //      Lê o valor analógico do potenciômetro (0-1023)
        valor: int = read(potenciometro);
        return valor;
}

def ajustar_brilho_led(valor_potenciometro: int) -> void: {
        //      Converte o valor do potenciômetro para um valor de brilho (0-1)
        brilho: float16 = valor_potenciometro / 1023.0;

        //      Define o brilho do LED com base no valor do potenciômetro
        write(led, brilho);
}

def main() -> int: {
        while (true): {
                //      Lê o valor atual do potenciômetro
                valor_potenciometro: int = ler_valor_potenciometro();
                //      Ajusta o brilho do LED com base no valor do potenciômetro
                ajustar_brilho_led(valor_potenciometro);
                //      Aguarda um curto período de tempo antes da próxima leitura
                delay(100);
        }
        return 0;
    }
```

3 VERIFICAÇÃO DE CÓDIGO – ANÁLISE SINTÁTICA E SEMÂNTICA

3.1 DESENVOLVIMENTO DO CÓDIGO

O analisador sintático implementado é uma parte essencial do código desenvolvido, responsável por analisar a estrutura gramatical de um programa escrito na linguagem .Cpy. Esse analisador foi construído com base na documentação do ANTLR, uma poderosa ferramenta de geração de analisadores sintáticos. O ANTLR permite definir as regras gramaticais da linguagem de forma clara e eficiente. Com base nesses padrões gramaticais definidos, o ANTLR gera automaticamente o código-fonte do analisador sintático.

Além disso, o código desenvolvido inclui a execução do analisador léxico, que é gerado a partir do ANTLR. O analisador léxico é responsável por analisar o código-fonte de entrada e transformá-lo em uma sequência de tokens, que são unidades léxicas básicas, como palavras-chave, identificadores, números e símbolos. Essa etapa é crucial para a análise sintática, pois fornece ao analisador sintático os tokens necessários para a identificação e compreensão da estrutura do código.

3.2 ANALISADOR LÉXICO

O analisador léxico desempenha um papel crucial na fase inicial do processo de compilação, transformando o código-fonte em uma sequência estruturada de tokens que são então utilizados pelo analisador sintático para compreender a estrutura e a semântica do programa.

A principal função do analisador léxico desenvolvido é transformar o código-fonte em uma sequência de unidades atômicas chamadas tokens. Cada token representa uma categoria específica, como palavras-chave, identificadores, números, operadores e símbolos.

Este analisador léxico específico é projetado para uma linguagem de programação que suporta tipos de dados como ***'int'***, ***'float16'***, e ***'bool'***, além de estruturas de controle de fluxo como ***'def'***, ***'while'***, ***'if'***, ***'elif'***, ‘***else’*** e ‘***return’***. Ele também reconhece operadores aritméticos, de atribuição e de comparação, bem como símbolos como ponto e vírgula, parênteses, chaves, colchetes e setas.

Além disso, o analisador léxico lida com elementos específicos da linguagem, como declarações de hardware (***'pin'***, ***'write'***, 'delay', ***'read'***) e valores booleanos (***'true'*** e ***'false'***). Ele também incorpora regras para ignorar espaços em branco e

comentários de linha única, garantindo que esses elementos não afetem o processo de análise.

https://replit.com/@matheusherman/Compiladores#Fase_2
https://github.com/matheusherman/Projeto_Compilador/

3.3 ANÁLISE SINTÁTICA E SEMÂNTICA

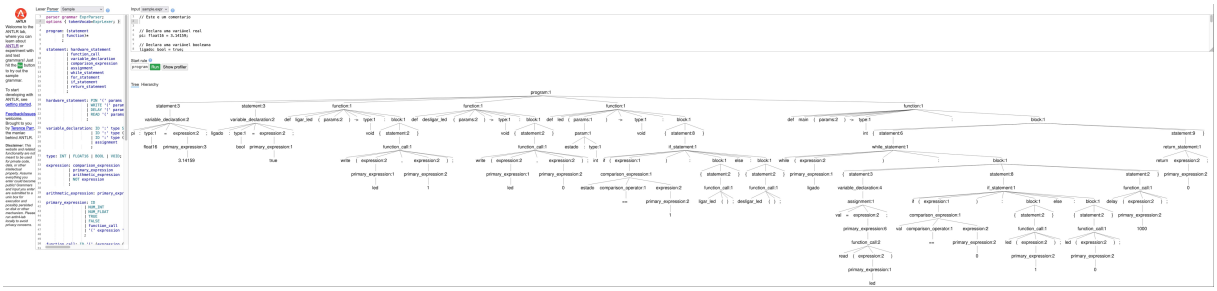
O analisador sintático foi desenvolvido utilizando um *parser* recursivo para interpretar a programas desenvolvidos na linguagem .Cpy. Utilizando regras gramaticais, ele percorre a entrada de cima para baixo, reconhecendo e interpretando declarações de hardware, chamadas de funções, operações aritméticas e estruturas de controle de fluxo. A implementação utiliza expressões recursivas para lidar com aninhamentos, como expressões dentro de expressões e blocos dentro de blocos. Em resumo, é um parser para a análise inicial de código-fonte, fornecendo uma base para análises semânticas e a geração subsequente de código.

O analisador sintático foi criado com o uso do ANTLR para interpretar a linguagem .Cpy. Ele reconhece a estrutura gramatical do código, como declarações de hardware, chamadas de funções e operações aritméticas, utilizando regras definidas. O ANTLR automatiza a análise sintática, mas a análise semântica, que envolve verificação de significado e contexto, requer implementação personalizada. O analisador serve como base para análises semânticas posteriores e geração de código.

A análise semântica verifica o significado e o contexto do código-fonte, garantindo que as operações sejam realizadas com tipos apropriados, variáveis sejam usadas corretamente, funções sejam chamadas adequadamente e outras regras da linguagem sejam seguidas. É essencial para evitar erros e garantir que o programa funcione corretamente e de acordo com as regras da linguagem.

3.3.1 Árvore sintática

Árvore sintática para o exemplo teste.Cpy, um exemplo simples para a realização do teste.



Árvore sintática para o exemplo teste4.Cpy, um exemplo mais completo contendo todas as funcionalidades para teste.

3.4 TESTES E VALIDAÇÃO

Para a realização dos testes e das validações do código desenvolvido, foram utilizados três códigos escritos na linguagem .Cpy. Os códigos de teste devem testar todas as funcionalidades da linguagem, permitindo uma avaliação abrangente do analisador, além de verificar os erros do analisador.

Após a geração do parser no ANTLR, foram testados os arquivos teste.cpy, teste2.cpy, teste3.cpy e teste4.cpy, a fim de verificar se os erros foram gerados corretamente. A seguir podem ser visto alguns exemplos dos erros apresentados.

```
Arquivo sendo testado: teste.cpy

line 39:8 extraneous input 'else' expecting {'for', 'while', 'if', 'return', ID, 'pin', 'wri
te', 'delay', 'read', '}'}
```

```
~/Compiladores/Fase_2$ python parser.py teste.cpy
Arquivo sendo testado: teste.cpy

line 33:14 missing {'int', 'float16', 'bool', 'void'} at ':'
```

```
Arquivo sendo testado: teste.cpy

line 4:9 no viable alternative at input 'numero:='
```

```
~/Compiladores/Fase_2$ python parser.py teste.cpy
Arquivo sendo testado: teste.cpy

line 3:17 token recognition error at: 'á'
line 3:9 no viable alternative at input 'Declarauma'
```