UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
COENC - ENGENHARIA DE COMPUTAÇÃO
DISCIPLINA DE COMPILADORES

GABRIEL DUARTE SANTOS
GUSTAVO D'AVILA

DOCUMENTAÇÃO DE ESPECIFICAÇÃO DE REQUISITOS: COMPILADOR TRUCO++

TOLEDO - PR
2022

1 INTRODUÇÃO

A criação de um compilador é um processo complexo, que requer o entendimento e implementação de um software que traduza programas escritos no padrão correto seguindo as regras léxicas, sintáticas e semânticas da linguagem feita. Esse trabalho requer uma forte compreensão em estrutura de dados, lógica para computação, teoria da computação, algoritmos e também comprometimento para seu entendimento.

1.1 CONTEXTUALIZAÇÃO

O compilador Truco++ se oferece como uma opção temática, mais simples e muito intuitiva para entender como programar em outras linguagens, principalmente C e C++. É inspirado em um clássico jogo de cartas brasileiro, o truco, trazendo uma abordagem descontraída para a programação.

Programar é uma tarefa que é necessária lógica, resolução de problemas e criatividade, nada diferente do que é tratado neste compilador, porém é buscado que esses papéis sejam facilitados, com uma experiência diferenciada, substituindo nomes de funções e comandos por termos relacionados ao jogo truco.

1.2 FINALIDADE E APLICAÇÕES

Este compilador traz consigo como finalidade e aplicações para os programadores:

- Aprendizado divertido: proporcionando uma maneira diferente de aprender e praticar programação, principalmente para iniciantes;
- Aplicações variadas: podendo ser utilizado em uma variedade de projetos, como é possível na linguagem de C e C++, adicionando uma unicidade ao projeto;
- Estímulo de criatividade: por conta da temática de truco, o compilador causa um estímulo de descobrir como uma função nomeada como ‘tento’ faz, por exemplo, tornando o processo de programar mais interessante.

2 DESCRIÇÃO GERAL

Nesta seção será descrito uma visão geral do compilador Truco++, explicando como ele funciona, suas funções, operadores, e como foi organizado. Este compilador permite que os programadores escrevam e usem funções com baseadas no jogo de truco, usando algumas gírias existentes no jogo como funções. Ao longo deste artigo, será detalhado as principais etapas da construção deste compilador, dando exemplos de código, tanto com erros quanto corretos, demonstrando o uso prático do Truco++.

2.1 ESTRUTURA DO DOCUMENTO

A estrutura responsável pelo compilador é organizada de forma a usar quatro arquivos .h, sendo separados por Léxica, Sintática, Semântica e Biblioteca, e um arquivo .cpp que é utilizado como main.

- O primeiro arquivo mencionado **Lexica.h** é responsável por ler o arquivo com o código desejado (“programa.txt”) e analisa palavra por palavra até o final do código, separando as reconhecidas pelo código como Tokens e armazenando-as no formato tipo, valor, linha e coluna.
  O código também é responsável por mostrar um erro no caso de uma leitura onde não é possível definir um Token, sendo a única exceção para armazenar no formato Token o comentário, sendo o conteúdo dentro de duas aspas ignorado por completo (@@ Comentário @@).

- O arquivo **Sintatica.h** possui a função de ler Token a Token e analisar utilizando da EBNF para declarar caso o código fornecido está válido na sua forma sintática ou não.
  Devido a alguns imprevistos e descuidos acabamos por incluir funções que deveriam ser reconhecidas na fase semântica, deixando o código com uma complexidade muito maior e que impossibilitou nosso grupo de testar diversos casos e arrumar aqueles que retornem um erro indevido.

- A última função responsável pela análise do código está no arquivo **Semantica.h**, sendo assim, ele é responsável por averiguar as possíveis inconsistências do código, além do seu formato EBNF, como por exemplo um inteiro declarado duas vezes com o mesmo nome ou até uma soma com valores incongruentes.

- E por fim o arquivo responsável por unir todos os anteriores, **Biblioteca.h**, este arquivo é necessário para a declaração das funções responsáveis por cada .h anterior, além disso ele é utilizado para incluir todas bibliotecas utilizadas no código.

2.2 ESTRUTURA BÁSICA DO CÓDIGO

Aqui será explicada a documentação, explicando e demonstrando como funcionam os comandos criados seguindo o tema determinado no início do projeto. Cada comando será seguido por uma breve descrição, explicando o que cada um fará.

No processo de declaração de variáveis temos os seguintes tipos:

```
tento var1 = 0;          @@ Inteiro @@
casal var2 = 1.4;        @@ Flutuante @@
pe var3 = “truco”;       @@ String @@
mao var4 = ‘x’;          @@ Caracter @@
blefe var5 = 1;          @@ Booleano @@
```

Como descrito, temos que declarar o tipo da variável primeiro, seu nome, sua atribuição e após isso o ‘;’. É possível também mudar o valor atribuído à variável a qualquer momento, caso ela já tenha sido declarada no código, e no caso de estar em seu escopo correto, não foi possível de implementar em uma forma agradável.

Para o caso de funções, elas devem ser iniciadas da seguinte forma:

```
truco (var1 == 0) {                    @@ Se @@
        var3 = “Correto”;
};
meio-pau (var1 == 1){                  @@ Senão se @@
        var3 = “Errado”;
};
cai {                                  @@ Senão @@
        var3 = “Errado”
};
```

Apenas como um exemplo, porém, exemplificando bem como funcionam as condições para algo ser feito no código, sendo possível usar de diversas outras formas. Deixando claro também que é necessário um ‘;’ após o fechamento de colchetes de cada uma das condições.

Para declarar um bloco de repetição:

```
rodada (var1; var1 <= 3; var++){
        @@ Código @@
        reembaralha var1;
};
```

Sendo usado como um for contido nas linguagens C e C++, identificando a variável com seu valor inicial, adicionando um até chegar ao número definido. Fazendo assim um loop, que neste caso está retornando a variável que foi antes usada com a função ‘reembaralha’.

Operadores matemáticos:

```
var1 = var1 + var2;        @@ Soma @@
var1 = var1 - var2;        @@ Subtração @@
var1 = var1 * var2;        @@ Multiplicação @@
var1 = var1 / var2;        @@ Divisão @@
var1 = var1 ^ var2;        @@ Elevar @@
```

É visto que os operadores matemáticos seguem exatamente a mesma lógica das linguagens C e C++. Mas também é aqui que fizemos nosso novo operador, sendo o operador de elevar números.

Operadores lógicos:

```
var1 || var2        @@ Ou @@
var1 && var2        @@ E @@
var1 != var2        @@ Diferente @@
var1 > var2         @@ Maior @@
var1 >= var2        @@ Maior ou igual @@
var1 < var2         @@ Menor @@
var1 <= var2        @@ Menor ou igual @@
var1 == var2        @@ Igualdade @@
!var1               @@ Negação @@
```

3 FASES DO COMPILADOR

O compilador Truco++ realiza diversas etapas para verificar se o arquivo .txt está escrito dentro dos padrões do compilador. Cada fase desempenha uma função específica, acarretando em uma correta tradução e verificação do código. A seguir, são apresentadas fases do compilador Truco++:

3.1 ANÁLISE LÉXICA

Como citado anteriormente, a análise léxica tem como principal objetivo “ler” e “traduzir” o código criado para facilitar o manuseio futuro do mesmo.

Esse processo é efetuado de forma a percorrer o código fonte completo e ao analisar um Token ele separa o valor analisado em quatro variáveis: type, value, line, column.

Abaixo segue um exemplo da sua aplicação:

```
tento temporario = 10;
```

Ele será analisado e convertido ao código da seguinte forma:

Figura 1 - Tabela gerada

TokenType type;	**string value;**	**int line;**	**int column;**
INT	tento	1	4
IDENTIFIER	temporario	1	15
ASSIGNMENT	=	1	18
INTEGER	10	1	19
SEMICOLON	;	1	21

3.2 ANÁLISE SINTÁTICA

O analisador sintático possui a função de percorrer a lista de Tokens separados anteriormente e verificando caso o formato esteja correto ou não, para essa verificação, o programa utiliza uma EBNF para garantir essa formação.

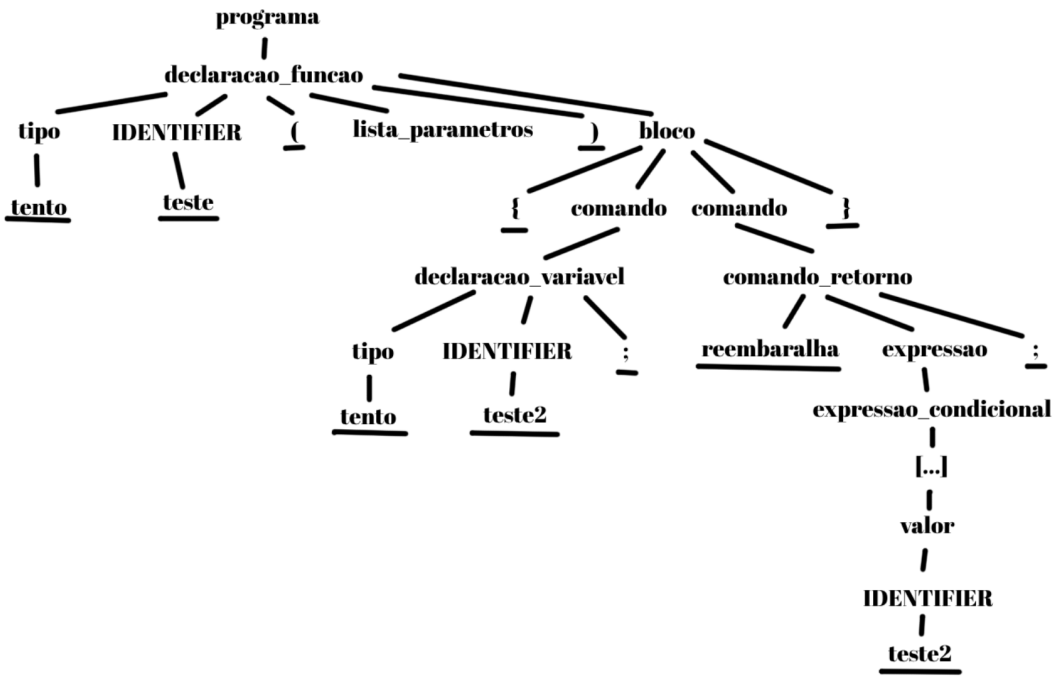
Por fim, ele transforma a sequência de Tokens em uma árvore sintática que representa a hierarquia do programa.

Exemplo do código:

```
tento teste(){
        tento teste2 = 0;
        reembaralha teste2;
};
```

E sua equivalente na EBNF:

Figura 2 - Árvore Sintática do código



3.3 ANÁLISE SEMÂNTICA

O analisador semântico é uma etapa crucial no processo de compilação de um programa. Sua principal função é verificar se as construções do código-fonte têm um significado válido e estão de acordo com as regras semânticas da linguagem utilizada. Um exemplo desta fase é a verificação de identificadores, para garantir que ele não esteja sendo declarado de forma errada ou repetida.

O analisador semântico realiza várias tarefas para garantir a consistência semântica do código-fonte. Algumas dessas tarefas são:

- Verificação de tipos: O analisador verifica se os tipos de dados utilizados nas operações são compatíveis, isso significa que ele garante que não possua uma inconsistência quando utilizado um inteiro, flutuante, entre outros. Também é utilizado para a verificação em caso de uma operação matemática, retornando erro caso seja utilizado tipos inválidos.
- Declaração de variáveis: Nesta tarefa o analisador semântico analisa o código inteiro para verificar caso o identificador tenha sido declarado mais de uma vez com o mesmo nome ou até tente ser utilizado sem ter sido declarado anteriormente.
- Escopo: A verificação do escopo no analisador semântico tem a função de garantir que um identificador esteja sendo utilizado de forma correta, sendo uma variável local impossível de utilizar em termos globais por exemplo.

4 Geração de Código

Durante o processo de desenvolvimento das três principais etapas de análise, inúmeras dificuldades e obstáculos foram enfrentados, o que acabou impedindo a realização da fase de geração de código. Neste momento, o sistema está limitado a realizar uma verificação básica do código-fonte, extraído de um arquivo de extensão .txt, com o objetivo de determinar sua correção ou incorreção, oferecendo um retorno indicando sua validade sem a capacidade de gerar um código executável.

Exemplos de Código

```
tento teste(){
        tento hmm = 0;
        hmm = hmm + 1;
        reembaralha 1;
};

tento testeNegativo(){
        reembaralha 0;
};

pato temporario(){};
```

```
tento main(){
	tento b = 2;
	casal c = 2.1;
	mao l = 'k';
	blefe boo = 1;
	pe st = "popo";

	partida(tento i = 0; i <= 10; i++){
		truco(!(i > 0)){
			b = i;
		};
		cai{
			i = b;
		};
	};
	reembaralha 0;
};
```

```
pato declaraInutil(){
	tento c = 0;
};

tento retornaSoma(tento i, tento ii){
	tento temp = 0;
	temp = i + ii;
	reembaralha temp;
};

tento main(){
	tento g = 50;
	@@ não foi possível implementar a chamada de funções @@
	partida(g > 0){
		g = 100;
	};
	cai{
	};

	reembaralha 0;
};
```

5 Tabela de Símbolos

A tabela de símbolos utilizada no Truco++ foi criada com o propósito de estabelecer uma estrutura organizada e aprimorar a compreensão das etapas de produção do programa. Essa tabela é composta por informações como tipo, valor, linha e coluna, fornecendo um contexto completo para a análise do código-fonte. No entanto, até o momento, não foi implementada uma análise de erro por linha na análise sintática e semântica.

A análise léxica desempenha o papel de converter o código-fonte, percorrendo-o de início a fim e armazenando os tokens extraídos no formato mencionado anteriormente. Essa etapa estabelece a base para as análises subsequentes.

Já a análise sintática utiliza esses tokens obtidos para organizar o código-fonte de acordo com a EBNF (Extended Backus-Naur Form), garantindo uma estrutura consistente e coerente. Essa análise sintática visa verificar a conformidade gramatical do programa.

Por sua vez, a análise semântica percorre o código-fonte com o intuito de verificar a correta utilização dos elementos semânticos. Ela assegura que as expressões, atribuições e demais construções estejam declaradas de forma congruente e em conformidade com as regras semânticas da linguagem de programação.

6 Conclusão

Neste projeto, desenvolvemos um compilador denominado Truco++, com inspiração no famoso jogo brasileiro truco e também baseado na linguagem C++, seu objetivo é disponibilizar uma abordagem mais divertida e prática na programação, oferecendo uma experiência única para o programador.

A criação deste compilador foi uma grande tarefa, onde foi enfrentado diversos desafios, o que limitou nossa capacidade da realização testes mais abrangentes, não sendo possível corrigir todos os erros identificados.

Mesmo com as limitações encontradas, o Truco++ possui as principais funcionalidades básicas para o funcionamento correto de um compilador, dentre

elas a análise léxica, análise sintática e análise semântica, sendo usado a tabela de símbolos e também seguido corretamente nossa EBNF.

Ao longo deste documento foi descrito a estrutura e descrição geral do compilador Truco++, exemplificando as dificuldades que foram encontradas, detalhando as principais etapas de análises do código-fonte e suas funcionalidades.

Embora o compilador ainda tenha espaço para melhorias e aprimoramentos, resolvendo alguns bugs existentes, consideramos que o desenvolvimento deste compilador foi de suma importância para o entendimento de como funciona cada parte de um compilador, que causou um interesse de concluí-lo da melhor forma possível. Uma das principais coisas que gostaríamos de termos explorado melhor era a realização de testes mais abrangentes, aprimorando a robustez do compilador, corrigindo os erros e melhorando a capacidade de lidar com toda a variedade possível de casos.