Universidade Federal de Santa Catarina
Arthur Henrique Della Fraga 11200623
Nathan Molinari 13100769

Construção de Compiladores

Projeto 2

Informações básicas

Gerente de projeto:	Arthur Henrique D Fraga;
Projetista da linguagem:	Nathan Molinari;
Arquiteto do compilador:	Arthur Henrique D Fraga;
Testador:	Nathan Molinari.

Características do compilador

+2 : proposta de nova linguagem
+2 : interpretador

Especificação da Linguagem

1. Tipos

Tipos aceitos na linguagem:

a) int
   Aceita um valor inteiro. ex: 4, 12, -4; 10000000.
b) double
   Aceita um valor racional. ex: 2.4, -4.812.
c) bool
   Aceita um valor booleano, TRUE ou FALSE.
d) char
   Aceita um caracter. ex: ‘a’, ‘z’.
e) string
   Aceita um sequencia de caracteres. Ex: “Hello World”.

2. Definição de variáveis

2.1. Especificação

A definição de uma variável deve ser feita de acordo com a seguinte sintaxe: **def <tipo> identificador** ou **def <tipo> id1, id2, … , idn;**
<tipo>: deve ser um dos tipos especificados em no tópico 1.
identificador: deve obrigatoriamente iniciar com um carácter e pode ser seguido por caracteres ou dígitos.

2.2. Exemplos

```
def int var;
def bool a, b;
def string nome;
```

3. Atribuição

3.1. Especificação

A atribuição de um valor para uma variável deve ser feita de acordo com a seguinte sintaxe:
identificador = valor;
Não é possível atribuir um valor para uma variável que não foi definida.

3.2. Exemplos

```
def int a;
a = 2;
def bool isValid;
isValid = TRUE;
```

4. Atribuição composta

4.1. Especificação

Também é possível atribuir operar matematicamente sobre uma variavel, guardando nela o resultado da operação.

4.2. Exemplos

```
def double a;
a += 5.44;
a *= (double) ‘a’;
a /= TRUE;
a -= (4 - 8 / -9);
```

5. Coerção

5.1. Especificação

A coerção de tipos, quando possível, é realizada automaticamente durante uma atribuição. Assim ao atribuir um valor de um tipo distindo de uma variavel, este valor é convertido para o tipo adequado e então persistido.

5.2. Exemplos

```
def int a;
a = 2.66; // a recebe o valor 2
a = ‘a’; // a recebe o valor 97 (asc)
def bool b;
b = 5; // b recebe o valor TRUE;
```

6. Casting

6.1. Especificação

É possível fazer cast de um tipo para outro, desde que essa “conversao” seja suportada. O cast tem a seguinte sintaxe: **(tipo) instrução;**

6.2. Exemplos

```
def int a;
a = (int) TRUE + 5;
a = ((int) ‘a’ > (int) ‘b’); // Também ocorrerá coerção de bool para int no momento da atribuição.
```

7. Operações

7.1. Comparação

Operações de comparação são realizadas sempre sobre 2 instruções de mesmo tipo. Caso deseje-se operar sobre tipos distinto é necessário realizar o casting de um dos valores.
Para os exemplos a seguir serão considerados as declarações das variáveis int a,b e bool v.

7.1.1. Diferente (!=)

```
a = 4;
b = 1;
v = a != b;
```

7.1.2. Igual (==)

```
a = 4;
b = 4;
v = a == b;
```

7.1.3. Maior (>)

```
a = 5;
b = 4;
v = a > b;
```

7.1.4. Maior igual (>=)

```
a = 4;
b = 4;
v = a >= b;
```

7.1.5. Menor (<)

```
a = 3;
b = 4;
v = a < b;
```

7.1.6. Menor igual (<=)

```
a = 3;
b = 3;
v = a >= b;
```

7.2. Lógica

Operações lógicas aceitam apenas instruções booleanas como parâmetro.
Consideraremos as variáveis bool a, b, v já definidas.

7.2.1. And (&&)

```
a = 5 > 3;
b = TRUE;
v = a && b;
```

7.2.2. Or (||)

```
a = 5 == 3;
b = FALSE;
v = a || b;
```

7.2.3. Negação booleana (!)

```
a = 5 > 3;
b = FALSE;
v = !(a && b);
```

7.3. Matemática

Operações matemáticas são realizadas sobre valores dos tipos int e double.
Consideraremos as variáveis int a, b já definidas.

7.3.1. Soma (+)

```
a = 4 + 6;
a = 2.6 + 3;
```

7.3.2. Subtração (-)

```
a = 4 - (-2);
a = 2.6 - 3;
```

7.3.3. Multiplicação (*)

```
a = 4 * 6;
a = -2.6 * 3;
```

7.3.4. Divisão (/)

```
a = 4 / -6;
a = 2.6 / 0.3;
```

8. Condicionais

8.1. If them else

```
def bool maior;
def double v1, v2;
v1 = 2;
v2 = 1.9999;
if(v1 > v2){
  maior = TRUE;
}
else{
  maior = FALSE;
};
```

8.2. While

```
def bool naoMandeiParar;
naoMandeiParar = TRUE;
i = 0;

while(naoMandeiParar){
  a = 10 * i;
  if(a == 50){
    naoMandeiParar = FALSE;
  };
  i += 1;
};
```

8.3. For

```
def int i;
for(i; i < 4; i += 8){
        def int a;
        a + =9;
};
```

8.4. Repeat

```
def double a;
repeat(5){
        a += 4.77;
}
```

9. Função

9.1. Definicao

```
def char c;
def int f(def bool b, def string s){
        if(b){
                print s;
        };
        c = ‘w’;
        return 5 + c;
};
```

9.2. Chamada

```
def int a, b;
def string k;
a = f(a+b, k);
```

Arquitetura do compilador

Para a construção de nosso compilador, uma série de classes foi criada, cada uma delas representando uma das estruturas, funções, operações, valores, tipos e tudo mais necessário para a execução do proscesso.

Como peça chave temos o modelo de classe **Nodo**, qual representa uma instrução e cuja grande maioria das demais classes do programa estendem. Um nodo, além de poder se *printar*, tem sempre um tipo (dentre os especificados no item 1). Este tipo representa o que se esperar do retorno da execução desta instrução.

Um tipo particular de Nodo são os **Primitivos**, estes representam valores *literais* inseridos no código; Como um inteiro, um booleano ou uma string, por exemplo. Outro Nodo importante são as **Variaveis**; Estas armazenam um valor, conforme seu tipo, devolvendo-o quando executam.

Operação é um outro Nodo. Além de seu tipo de retorno, este também tem os tipos de seus parâmetros como caraterista de seu modelo. Desde descendem cada uma das operações descritas anteriormente.

Os **Blocos**, por sua vez, são uma classe mais elaborada. Como um Nodo que nada retorna (isto é, do tipo *void*), os Blocos possuem uma lista de instruções. Sua execução é percorrer tal lista, executando-a por completo.

Utilizando-se de Blocos, temos os nodos **If, While, For** e inclusive as **Funções**. Nodos estes que controlam escopos limitando acesso e ‘tempo de vida’ de instruções que os descendam. Para tal gestão a classe **Contexto** se faz fundamental, visto que é ela a armazenar as variáveis e funções declaradas e seus valores.

Para suporte a flexibilidade das soluções empregadas o uso das classes **variant** e **static_visitor** da biblioteca boost foi essencial, inclusive para melhor organização e legibilidade do código.

Testes

Para a realização dos testes e validação do compilador foram planejados casos de teste para cada uma das situações descritas. Tendo como parâmetro os resultados esperados pudemos validar a saída da execução.

Execução dos testes

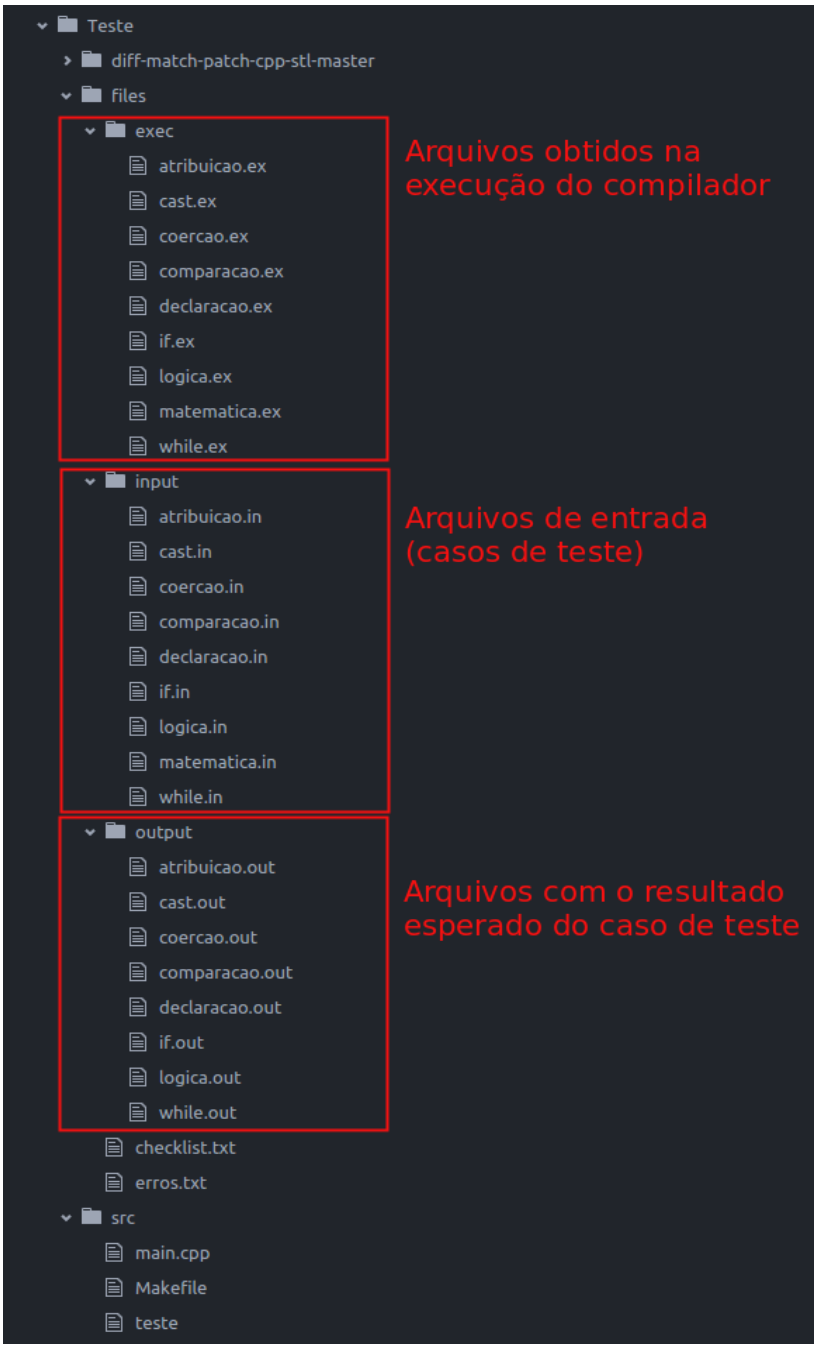
Para a execução dos testes foi criado um programa que roda cada arquivo de entrada e salva o resultado obtido um segundo arquivo. Após isso o programa faz uma comparação simples (byte) entre o resultado obtido e esperado.

Verificação dos resultados

Para verificar os resultados dos testes foi utilizado uma biblioteca *google-diff-match-patch* que faz o diff entre o arquivo gerado pelo compilador e a saída planejada. Porém essa biblioteca gera um html com o diff, o que dificultou a visualização do resultado. Como alternativa foi utilizado a ferramenta *Beyond Compare* que exibe o diff entre arquivos em uma gui.

Arquitetura dos testes

Para os testes, organizamos os arquivos em três grupos de pastas. Uma com os códigos de entrada, outro com as saídas esperadas e o último com o resultado da comparação entre a espectativa e a efetiva execução. A estrutura está apresentada a seguir:


Teste
diff-match-patch-cpp-stl-master
files
exec
atribuicao.ex
cast.ex
coercao.ex
comparacao.ex
declaracao.ex
if.ex
logica.ex
matematica.ex
while.ex
Arquivos obtidos na execução do compilador
input
atribuicao.in
cast.in
coercao.in
comparacao.in
declaracao.in
if.in
logica.in
matematica.in
while.in
Arquivos de entrada (casos de teste)
output
atribuicao.out
cast.out
coercao.out
comparacao.out
declaracao.out
if.out
logica.out
while.out
Arquivos com o resultado esperado do caso de teste
checklist.txt
erros.txt
src
main.cpp
Makefile
teste