Universidade Federal de Alagoas
Instituto de Computação
Compiladores 2021.1

Especificação da linguagem neo-gorm

Eduardo Brasil Araujo, Lael de Lima Santa Rosa

Maceió - Alagoas
Junho de 2022

Sumário

Introdução

A *neo-gorm* é uma linguagem de paradigma procedural e imperativo, as instruções do programa devem ser passadas em sequência. Ela é baseada na linguagem de programação C e na linguagem de programação Rust.

Desenvolvimento

1. Estrutura geral do programa

1.1. Como é definido o escopo no programa

O escopo no programa é definido com abre-chaves '**{**', e é finalizado com fecha-chaves '**}**'. Ele não pode ser definido fora de uma função. Nele, as instruções e variáveis são encapsuladas limitadas ao contexto, isto é, as variáveis e instruções declaradas dentro de um escopo só são válidas para o escopo atual e os aninhados a ele, mas não aos anteriores.

1.2. Onde podem ser declaradas funções

As funções podem ser declaradas somente antes da função **main** e no escopo global, ou seja, o escopo que não detém nenhuma chave aberta (a chave raiz do objeto de tradução). É importante salientar que toda a função possui um escopo local, que é o escopo delimitado somente para a função, pois, toda função tem que ter declarado um escopo, que nele estão inseridas as instruções que esta função realizará.

1.3. Definição de declarações

Obrigatoriamente as declarações do programa devem ser finalizadas com um ponto e vírgula '**;**' e estarem dentro de um escopo.

1.4. Ponto inicial de execução

A execução do programa tem início na função **main**. Esta função é declarada escrevendo **fn i32 main()**, seguido do escopo da função (para mais detalhes de como declarar subprogramas ver seção 7). Percebe-se que o tipo de retorno da função **main** é um valor inteiro de 32 bits. Funções adicionais devem ser declaradas antes da função **main**.

1.5. Definição de variáveis

As variáveis são declaradas dentro de um escopo definido por chaves '**{}**', e devem ser declaradas somente no início do escopo. Elas são declaradas da seguinte forma: primeiro insere o tipo da variável, e depois o valor identificador da variável, e finaliza com o ponto e vírgula, podendo ter uma expressão na sua declaração ou não,.

Ex.: Exemplo em que é declarada uma variável:

```
i32 var;
```

Exemplo com inicialização:

```
i32 var = 100;
```

2. Esquema de palavras e comentários

2.1. Como os comentários são feitos

Os comentários na linguagem podem ser feitos uma linha de cada vez e começam com a cadeia de caracteres '*//*'. Quando uma linha do programa começa com a cadeia de caracteres que declaram um comentário, tudo que tiver na linha depois da cadeia de caracteres será ignorado.

2.2. Como a declaração de identificadores é feita

A declaração de identificadores é feita de modo que um identificador deve começar com uma letra, e conter letras, números ou sublinhado em sua composição. Ela atende à seguinte expressão regular:

[a-zA-Z][0-9a-zA-Z_]*

2.3. Palavras reservadas da linguagem

Existem palavras reservadas na linguagem, isto é, palavras que não podem ser utilizadas como identificadores, são elas as que especificam os tipos de dados, a palavra de declaração de função, a de conversão de tipo, as palavras de condicionais, as de estrutura de controle, entrada e saída e valores booleanos. Elas estão como segue: i32, u32, i64, u64, f32, f64, char, str, as, bool, array, fn, if, else, while, for, read, printf, true, false.

3. Tipos de dados

3.1 Tipos primitivos

3.1.1. Inteiro

Existem os tipos primitivos inteiros, mas eles serão divididos em dois tipos de inteiros, os tipos inteiros com 32 bits e os tipos inteiros de 64 bits.

3.1.1.1. Inteiro, com sinal, de 32 bits

O tipo inteiro de 32 bits é definido com a palavra reservada '**i32**', e ele é um inteiro que ocupa 32 bits de espaço. O literal é expresso com a presença do sinal negativo, ou não, e após isso, uma cadeia de dígitos decimais.

Ex.: i32 var;
var = -100002;

Como ele tem o tamanho de 32 bits e detém sinal, o número máximo que se pode atribuir é 2.147.483.647 e o mínimo é -2.147.483.647.

3.1.1.2. Inteiro, sem sinal, de 32 bits

O tipo inteiro de 32 bits sem sinal é definido com a palavra reservada '**u32**', ele é um inteiro que ocupa 32 bits de espaço. O literal é expresso com uma cadeia de dígitos decimais; não detém sinal.

Ex.: u32 var;
var = 323232;

Como ele tem o tamanho de 32 bits e não detém sinal, o número máximo que se pode atribuir é 4.294.967.295, e o mínimo é 0.

3.1.1.3. Inteiro, com sinal, de 64 bits

O tipo inteiro de 64 bits é definido com a palavra reservada '**i64**', e ele é um inteiro que ocupa 64 bits de espaço. O literal é expresso com a presença do sinal negativo, ou não, e após isso, uma cadeia de dígitos decimais.

Ex.: i64 var;
var = -109900000002;

Como ele tem o tamanho de 64 bits e detém sinal, o número máximo que se pode atribuir é 9.223.372.036.854.775.807 e o mínimo é -9.223.372.036.854.775.807.

3.1.1.4. Inteiro, sem sinal, de 64 bits

O tipo inteiro de 64 bits sem sinal é definido com a palavra reservada '**u64**', ele é um inteiro que ocupa 64 bits de espaço. O literal é expresso com uma cadeia de dígitos decimais; não detém sinal.

Ex.: u64 var;
var = 323111323232;

Como ele tem o tamanho de 64 bits e não detém sinal, o número máximo que se pode atribuir é 18.446.744.073.709.551.615, e o mínimo é 0.

3.1.2. Ponto Flutuante

3.1.2.1. Ponto flutuante de 32 bits

O tipo de ponto flutuante de 32 bits é definido com a palavra reservada '**f32**' e representa os números de ponto flutuante de 32 bits. O literal é expresso com o opcional de um sinal, e depois com dígitos decimais, seguidos de um ponto, e depois seguido de dígitos decimais.

Ex.: f32 var;
var = 10.023;
Ou, ainda,
var = -10.023;

3.1.2.2. Ponto flutuante de 64 bits

O tipo de ponto flutuante de 64 bits é definido com a palavra reservada '**f64**' e representa os números de ponto flutuante de 64 bits. O literal é expresso com o opcional de um sinal, e depois com dígitos decimais, seguidos de um ponto, e depois seguido de dígitos decimais.

Ex.: f64 var;
var = 10.023046;
Ou, ainda
var = -10.023046;

3.1.3. Caractere

O tipo de caractere, em suma, representa os caracteres, definido com a palavra reservada '**char**' e tem a capacidade de armazenamento de 8 bits. O literal é expresso com um apóstrofo '**'** seguido de um caractere símbolo da tabela ascii, seguido de outro apóstrofo '**'**.

Ex.: char var;
var = ' ';

3.1.4. Cadeia de caracteres

O tipo cadeia de caracteres é um agregado de vários caracteres, definido com a palavra reservada '**str**' e tem capacidade de armazenamento ilimitada. O literal é expresso como um agregado de caracteres, colocando os caracteres um por um, um atrás do outro, e nas extremidades da literal é posto aspas '**"**' para delimitar.

Ex.: str var;
var = "string";

3.1.5. Booleano

O tipo booleano é uma variável que corresponde à lógica booleana, ou seja, assume valores ou verdadeiros ou falsos; definida com a palavra reservada **'bool**', ela ocupa o espaço de 8 bits.

Ex.: bool var;
var = true;

3.2 Tipos agregados

3.2.1. Array

O tipo agregado array é um arranjo unidimensional. Para declarar uma variável do tipo arranjo unidimensional, é necessário declarar uma variável de algum tipo primitivo (Ver seção 3.1 Tipos Primitivos), e após o nome da variável (o identificador), colocar imediatamente depois um colchete-esquerdo '**[**', seguido de um comprimento, e depois um colchete-direito '**]**'.

Para declarar um tipo agregado de comprimento 2, com primitiva inteiro de 32 bits, deve-se fazer como segue:

i32 var[2];

E assim por diante para os diversos tipos de primitivas:

f32 var[2];

char var[2];

Quando na declaração de um tipo agregado tiver uma inicialização, o que ocorre é a inicialização de todos os índices do tipo agregado para o valor dado. Por exemplo, o seguinte, inicializa um tipo agregado com valor 1:

i32 var[2] = 1;

O tamanho em memória que o array assumirá será o produto do comprimento pelo tamanho da primitiva, no primeiro exemplo, com o inteiro de 32 bits, temos um comprimento de 2 para o tipo agregado e 32 bits para o tipo primitivo, portanto, temos um tamanho de 64 bits, ou 4 bytes, de espaço ocupado.

Sua indexação ocorre como na linguagem de programação C, o primeiro índice, valor zero, é o valor do ponteiro da variável base, e o próximo é a posição do ponteiro somados a um, e assim por diante.

3.2.1.1. Operador para acessar o tamanho do array

Para acessar o tamanho de uma variável do tipo array, basta adicionar o símbolo '**^**' após o identificador, com isto ele retorna um inteiro de 64 bits sem sinal com o valor do comprimento do tipo agregado. Nesse sentido, utilizando o exemplo anterior, quando fazemos:

var^;

Estamos adquirindo o valor 2, já que este foi o comprimento que foi declarado na declaração da variável.

3.3. Coerção e conversões de tipo

Na linguagem proposta não há coerção, isto é, conversões implícitas, todas as variáveis precisam ser convertidas explicitamente pelo programador.

As conversões explícitas podem ser feitas utilizando a palavra reservada '**as**', que deve ser posto após um identificador. A seguir um exemplo que transforma a variável var1 em um inteiro de 64 bits para poder somar dois números com var2.

i32 var1 = 10;

i64 var2 = 123;

i64 var3 = var1 as i64 + var2;

4. Operadores

4.1. Operações de cada tipo

Os tipos possuem operações que podem ser feitas entre aqueles do mesmo tipo. Por exemplo, podemos somar dois números, mas só podemos fazer isso se ambos forem inteiros, ou ambos de ponto flutuante, e assim por diante. Outro ponto importante é que caso um inteiro tenha 32 bits de tamanho e outro de 64 bits de tamanho, não pode haver uma operação entre eles sem antes transformar um deles no tipo do outro. Também não há a possibilidade de realizar uma operação com inteiros com sinal e inteiros sem sinal.

4.1.1. Operadores dos tipos: i32, u32, i64, u64, f32, f64

Os operadores suportados para estes são os aditivos, multiplicativos, e unário negativo e positivo. A seguir exemplos de código com as operações:

```
i32 var = 1;
i32 var2 = var + 2 * 3;
i32 result = -var / 2 - var % 2;
```

Lista dos operadores aritméticos suportados, separados por vírgula: **+** (soma e unário positivo)**, -** (subtração e unário negativo)**, *** (multiplicação)**, /** (divisão) e **%** (módulo).

Além disso, estes tipos suportam os operadores comparativos, sendo eles: '**<**', '**<=**', '**>**', '**>=**', '**==**', '**!=**', '**=**'.

Exemplo com código:

```
i32 var = 12;
bool result = var <= 0 || var > 10 || var != 9;
```

4.1.2. Operadores do tipo booleano

Os operadores suportados para esse tipo são os lógicos com negação, conjunção e disjunção. A seguir exemplo de código com as operações:

```
bool var = true;
bool var2 = var & false;
bool var3 = !var2 || var;
```

Lista dos operadores suportados, separados por vírgula: **!** (negação), **&&** (conjunção), **||** (disjunção).

Também suporta os operadores de comparação, sendo eles: '**==**', '**!=**'. Exemplo de código:

```
bool var = true;
bool result = var == true || var != true;
```

4.1.3. Operadores do tipo caractere e cadeia de caracteres

Os operadores suportados para esse tipo são de atribuição e comparação, sendo eles: '**<**', '**<=**', '**>**', '**>=**', '**==**', '**!=**', '**=**'. Exemplo de código:

```
char ch = 'a';
bool ch1 = ch == 'b' || ch >= 'f';
str simple_string = "this simple";
bool result = simple_string == "another simple";
```

O tipo de cadeia de caracteres tem um operador a mais do que o tipo caractere, pois ele tem o operador de concatenar uma strings ao final de uma string, como na linguagem de programação lua, com o operador '**..**', do seguinte modo:

```
str string = "simple";
str another = string .. " text"; // another == "simple text"
```

4.1.4. Operadores do array

Os operadores suportados para esse tipo são de retornar o tamanho do array. Exemplo de código:

```
i32 arr[3];
arr^; // retorna como valor o comprimento, 3
```

4.2. Precedência e associatividade dos operadores

A seguir uma tabela que diz a associatividade e precedência dos operadores. Aqueles operadores que aparecem mais ao topo da lista apresentam precedência maior que aqueles mais à baixo da lista. Lembrando que o **as type** é um único operador.

Categoria	Operador	Associatividade
Pós fixado	^	Esquerda para direita
Coerção	as type	Esquerda para direita
Unário	+ - !	Direita para esquerda
Multiplicativo	* / %	Esquerda para direita
Aditivo	+ -	Esquerda para direita
Relacional	< <= > >=	Esquerda para direita
Igualdade	== !=	Esquerda para direita
Conjunção	&	Esquerda para direita
Disjunção	|	Esquerda para direita
Concatenação	..	Esquerda para direita
Atribuição	=	Direita para esquerda

5. Instruções

5.1. Estrutura condicional

A estrutura condicional da linguagem proposta é adotar o modo com o qual a linguagem de programação C realiza a estrutura condicional. Desse modo, o que será feito é a adoção de um **if** para condicional com uma única via, e **if** e **else**, para condicionais de duas vias. Precisa-se de um abre-parênteses '**(**', uma expressão ou variável booleana (denotado como **expr**), depois o fecha-parênteses '**)**'; entrará na primeira via caso seja verdadeiro o **expr** e na segunda caso seja falso.

É necessário declarar um escopo para cada uma das vias, e este escopo pode conter várias sentenças (denotado como **stmt**), que podem conter declarações de variáveis, outras estruturas iterativas ou condicionais e chamadas de procedimentos. Segue o seguinte exemplo para condicional de uma via:

```
if (expr) {
        stmt
}
```

Agora para condicional de duas vias:

```
if (expr) {
        stmt
}
else {
        stmt
}
```

5.2. Estrutura iterativa com controle lógico

A estrutura iterativa com controle lógico será feita como na linguagem de programação C, em suma, terá um **while**, seguido de abre-parêntesis '**(**' e uma expressão (denotada por **expr)** ou variável booleana que será avaliada em pré-teste, e depois o fecha-parênteses '**)**'. Será declarado um escopo para a estrutura iterativa, que nela terão sentenças (denotada por **stmt**), que podem seguir assim como no tópico anterior (5.1. Estrutura condicional). A seguir um exemplo de código:

```
while (expr) {
        stmt
}
```

5.3. Estrutura iterativa controlada por contador

A estrutura iterativa controlada por contador tem uma semântica similar à linguagem de programação Rust, mas que não é sintaticamente similar à mesma. O que ocorre é uma estrutura de iteração baseada em alcance.

Para utilizá-la deve-se utilizar a palavra reservada **for** em seguida de um abre-parênteses '**(**', o tipo da variável do laço de repetição que será utilizada, depois colocará o símbolo dois-pontos '**:**', em seguida será especificado, separados por vírgula, o

valor inicial do alcance, o valor final e em seguida o valor do incremento, este poderá ser omitido, admitindo valor 1 para o incremento, e por último o fecha-parêntesis '**)**'. Depois será declarado um escopo, e nele terá sentenças (denotadas por **stmt**). A seguir um exemplo de um laço que começa do -3 e vai até 30, com incremento de 1:

```
for (i32 index : -3, 30) {
        stmt
}
```

5.4. Atribuição

A atribuição é feita utilizando o operador '**=**', e é feita do lado esquerdo para o direito em passo único, em que será atribuído o conteúdo contido do lado direito do símbolo, com ele sendo um valor ou expressão, e então, irá atribuir esse valor para um identificador.

6. Entrada e saída

6.1. Entrada

A função utilizada para a entrada da linguagem proposta é a função **read**. Nela você passará os identificadores, sendo possível passar mais de um identificador, podendo receber várias entradas de uma só vez. Segue um exemplo de uso e qual o valor que ela retornou:

Valor da entrada: 1.2 32
Programa:
f32 val1;
i32 val2;
read(val1, val2);
// nessa linha o identificador val1 admite valor 1.2
// e o identificador val2 admite valor 32

Importante frisar que caso a entrada seja de ponto flutuante mas a variável na função não corresponda a isso, o valor ficará como segue no exemplo:

Valor da entrada: 1.2 32
Programa:
i32 val1;
i32 val2;
read(val1, val2);
// nessa linha o identificador val1 admite valor 1
// e o identificador val2 admite valor 32

6.2. Saída

A função de saída da linguagem proposta é utilizar a função **printf**, que terá como primeiro parâmetro uma cadeia de caracteres, sendo que esta poderá ser formatada, de modo que consigamos pôr o valor de variáveis na saída do programa. A formatação será feita como na linguagem de programação C, que coloca-se '**%d**' para números decimais, '**%f**' para pontos flutuantes, '**%c**' para caracteres e '**%s**' para cadeia de caracteres. A seguir um exemplo:

i32 var1 = 12;
f32 var2 = 3.2;
char ch = 'b';
str string = "Hello world!";
printf("%d %f %c %s\n", var1, var2, ch, string);
// a saída será: 12 3.2 b Hello World! . Seguido de uma quebra de linha

Admite-se também a saída padrão sem formatação, como segue:

printf("Hello world!\n");
// a saída será: Hello World . Seguido de uma quebra de linha

7. Subprograma

7.1. Definição de funções

Toda função, obrigatoriamente, inicia com '**fn**', depois a declaração do tipo de retorno e o valor identificador dela. O tipo de retorno pode ser omitido, desse modo estará declarando uma função que não retorna. Após isso, deve-se colocar o abre-parêntesis '**(**', pode ou não receber parâmetros entre os parênteses, e depois é finalizado com o fecha-parêntesis '**)**'.

Os parâmetros da função são declarados semelhantes às variáveis, mas sem atribuição de valor, e são separados por vírgula. Depois é declarado o escopo da função, do mesmo modo que o escopo é declarado, mas sendo que este é imediatamente após o fecha-parêntesis que finaliza a declaração de parâmetros da função. Dentro do escopo contém as sentenças e instruções da função.

A função finaliza quando chega na palavra especificadora de retorno, o '**return**', e junto a isto um valor a ser retornado; ou caso não haja tipo especificado na declaração da função, ou seja, uma função sem retorno, não precisa de valor algum associado à palavra reservada de retorno e a função termina automaticamente ao chegar no final do escopo desta. Importante notar que o tipo do valor a ser retornado da função deve ser igual ao tipo declarado da função. Segue exemplo de função com parâmetros:

```
fn i32 add_one(i32 var1) {
        return var1 + 1;
}
```

A seguir a declaração de uma função em que não tenha parâmetros e possui um retorno de função:

```
fn i32 procedure() {
        return 5; //  nesse caso a função só retorna o valor 5
}
```

E por último uma função que não tem retorno:

```
fn say_hello() {
        printf("Hello\n");
}
```