Universidade Federal do Rio Grande do Norte
Departamento de Informática e Matemática Aplicada

Disciplina: DIM0437 — Linguagens de Programação: Conceitos e Paradigmas
Docente: Umberto Souza da Costa
Discentes: Dogival Ferreira da Silva Junior
Felipe Cortez de Sá
Gabriel Sebastian von Conta
Phellipe Albert Volkmer
Vinícius Araújo Petch

Relatório de uma linguagem de paradigma imperativo Lingmas

Sumário

1 Introdução **2**

2 Apresentação **2**

3 Descrição da linguagem **2**

4 Variáveis e tipos **2**

5 Vinculação de variáveis, informações de tipo **4**

6 Escopo **4**

7 Estruturas de controle, operadores e expressões **5**

8 Subprogramas **7**

9 Abstrações da linguagem **8**

10 BNF para a linguagem **8**

1 Introdução

Neste trabalho estruturamos o produto de resposta de cada subproblema provido durante o semestre referente a nossa linguagem, alcançado com discussões tanto fora quanto dentro da sala de aula, orientações do professor, livros fornecidos pelo professor e materiais encontrados online.

Cada tópico do sumário busca atender um dos subproblemas. Além do discutido na própria questão, adicionamos exemplos e pontos que consideramos precisar de maiores detalhes e/ou observações, junto com correções fornecidas após a entrega.

Cada tópico busca esclarecer o ponto tocante do subproblema, além de exemplos do tópico abordado na questão.

2 Apresentação

Nossa linguagem, denominada Lingmas, é uma linguagem de scripting destinada a aprendizado para estudantes entrando no universo da programação.

A Lingmas possui uma interpretação pura, facilitando a implementação de operações de verificação de erros, o aprendizado para iniciantes na área de programação e possuindo uma maior flexibilidade. Ela possui um caráter mais didático, tenta se aproximar das linguagens imperativas (Pascal, C), e foca em uma aproximação mais abstrata.

3 Descrição da linguagem

4 Variáveis e tipos

Os tipos primitivos da nossa linguagem serão os inteiros (**`int`**), ponto flutuante (**`float`**), booleano (**`bool`**), e cadeia de caracteres (**`string`**). O tipo primitivo char é "substituível"por uma string de 1 caractere, portanto desnecessário, já que otimização não é prioridade. Pelo mesmo motivo, não consideramos necessária a inclusão de tipos científicos ou de alta precisão, como números complexos ou decimais. De tipos definidos pelo usuário, adotaremos enumerações (**`enum`**), registros (**`structs`**) e arranjos ou vetores (**`arrays`**). Exemplos:

```
int integer = 7;
bool boolean = true;
float floatPoint = 4.3;
string str = "Hello World!";

struct person {
    num age;
    string name;
    string gender;
    bool alive = TRUE; // para valores default
};

john = person(age = 21, name = 'John Doe', gender = 'M');
assert(john.age == 21);
```

Os arrays terão índices inteiros e com sua faixa determinada pelo programador, onde o índice mínimo será 0. Caso seja informado apenas o tamanho do array, o índice mínimo padrão será 0, e o índice máximo será determinado pelo tamanho definido pelo usuário (no caso, será o tamanho definido menos um), e por fim, caso nada seja informado, ele terá o tamanho baseado no valor inicializado. Eles serão homogêneos e terão suporte à concatenação, comparação de igualdade e slices, e poderão ser definidos como um array de tipos ordenados de alguma forma sequencial. Na Lingmas, serão dos tipos mencionados anteriormente. Exemplos:

```
int a[2..6];
int b[5];
int c[] = {1, 2, 3, 4};
```

O tipo de alocação, não só dos arrays, mas também de enumerações e estruturas, será heap-dinâmico fixo, já que permite uma certa flexibilidade na escolha do tamanho, pois este só ocorre em tempo de execução, ao mesmo tempo que não permite ao usuário uma utilização sem restrições, como a do heap dinâmico implícito, que geraria maus costumes no uso de espaço de vetores e complicações de adequação ao utilizar outras linguagens.

Não haverão tipos recursivos, porém as estruturas recursivas, como listas encadeadas e árvores, poderão ser criadas de maneira semelhante à maioria das linguagens de alto nível. Por exemplo:

```
int recursion(int value) {
    if (value % 2 == 0) {
        return recursion(value + 1);
    } else if (value % 3 == 0) {
        return recursion(value / 2);
    } else {
    return value;
    }
}
```

Nomes de variáveis não serão case sensitive, e serão descritas começando sempre por uma letra do alfabeto, e com letras do alfabeto, números e underline (_) aceitos no corpo do nome. Haverá palavras reservadas, entre elas identificadores de tipos e estruturas de controle (**`enum`**, **`string`**, **`while`**, **`for`**, etc). O comprimento da palavra que define o nome da variável é de 31 caracteres. Os descritores dos *arrays* e *strings* terão a seguinte forma:

Tipo de elemento
Índice
Menor Índice
Maior Índice
Endereço

A ligação será de tipo estática, com declaração explícita, onde há a listagem do nome das variáveis e especificação de tipo. Em relação ao tempo de vida da variável, elas serão stack-dinâmico (seu espaço de memória é alocado durante execução, mas seu tipo ainda é alocado estaticamente). Essa escolha foi feita pois, graças a ligação estática explícita, direcionamos os programadores a se atentarem e direcionarem sua atenção para as variáveis presentes e os valores associados a elas.

A linguagem será estaticamente e fortemente tipada, de modo que sempre seja feita verificação e evite erros de tipos, e , por estática, terá checagem de tipo estática. Definimos

nossa linguagem dessa forma, pois desejamos sacrificar liberdade de programação dos usuários em favor de conscientizá-los e formar o funcionamento e a estruturação dos tipos. Seguindo esse mesmo intuito, não possuiremos coerção em nossa linguagem. Há presença de tipos derivados (enum, struct, entre outros).

Apelidos são gerados quando se têm variáveis de referência, já que diversas variáveis podem ter acesso ao mesmo espaço de memória. Como a nossa linguagem não possui ponteiros ou *unions*, então não teremos problemas com apelidos.

5 Vinculação de variáveis, informações de tipo

A vinculação de tipo será estática com declaração explícita, onde há a listagem do nome das variáveis e especificação de tipo, acostumando os usuários da linguagem a pensarem nos tipos específicos que estão usando. Em relação à alocação de espaço, a vinculação será stack-dinâmica na maioria dos casos, onde seu espaço de memória será alocado quando for feita a sua declaração, e desalocado somente no final do método ou função, sendo útil na criação de subprogramas recursivos.

A linguagem será estaticamente e o mais próximo de fortemente tipada possível, de modo que sempre sejam feitas verificações para evitar erros de tipos. Definimos nossa linguagem dessa forma, pois desejamos sacrificar liberdade de programação dos usuários em favor de conscientizá-los das restrições de tipo. As construções e elementos que serão verificados em relação aos tipos serão operações e atribuição de variáveis, parâmetros e retorno de função.

Dado o propósito da linguagem, não usaremos a conversão implícita, com isso perdendo a coerção. Usaremos apenas a conversão explícita, para aumentar o controle do programador sobre o código. Todos os tipos da nossa linguagem serão verificados estaticamente devido a utilização do binding estático de tipo das variáveis.

Todas as conversões da linguagem serão explícitas. Operadores de strings aceitam como operandos somente strings, e não há casos em que se poderia ter coerção. O mesmo acontece com arrays. Todos os tipos da linguagem terão equivalência por nome, com exceção de structs, que terão equivalência por estrutura, pois estes terão mais problemas com a restrição da equivalência por nome.

6 Escopo

Para a escolha do escopo, decidimos utilizar o escopo estático, apesar das suas desvantagens, como em casos em que permite mais acesso a variáveis e subprogramas do que o desejado, achamos seu uso adequado em nossa implementação devido à melhor legibilidade, à execução mais rápida e à confiabilidade.

Também teremos blocos, variáveis globais e escopo aninhado. Decidimos dessa forma, mesmo que abdicando de um certo grau de flexibilidade, para atentarmos à noção de alcance da declaração das variáveis para nossos usuários, que poderão dessa forma compreender e criar noções de estruturação do código. Além disso, como a definição do escopo ocorrerá antes do tempo de execução, os erros de declaração serão indicados antes da

execução, e como as falhas do escopo estático ocorrem normalmente ao aplicar alterações no código e nos seus requisitos, nossos códigos serão executados um número pequeno de vezes, sem necessidade de um grande número de alterações ou uma estruturação bem definida.

Utilizamos também o armazenamento de variavéis do tipo stack-dinâmico, onde a amarração de espaço são criadas quando a variável é declarada, mas o tipo é estaticamente amarrado.

7 Estruturas de controle, operadores e expressões

Subprogramas e estruturas de controle (repetições e condicionais) possuirão blocos, que serão delimitados por chaves, estando aninhados dentro do bloco principal e outros blocos do programa. Uma exceção da regra das chaves são enums e structs.

Temos assim aumento de legibilidade e facilidade de delimitação de escopo. Blocos aninhados serão permitidos, pois os consideramos essenciais para a melhor estruturação do código, permitindo que que cada bloco tenha suas próprias variáveis locais. Tal qual em C e C++, permitiremos a reutilização de nomes de variáveis dentro de blocos aninhados, pois o escopo estático já permite o tratamento desse tipo de situação, utilizando a variável mais interna do bloco.

A visibilidade das variáveis será no mesmo modelo do C, onde um bloco ou subprograma só enxergará variáveis no seu nível, ou em níveis acima (parentes estáticos). No caso do ambiente de referenciamento, como utilizamos escopo estático, o ambiente de referenciamento de uma instrução são as variáveis declaradas em seu escopo local mais o conjunto de todas as variáveis de seus escopos ancestrais visíveis.

A alocação de memória para uma variável composta ocorrerá da seguinte maneira: será vista apenas pelo bloco que a alocou e, por causa do escopo estático, seus blocos internos. Com isso, o espaço da variável composta alocado por um bloco não poderá ser utilizado por blocos externos ou blocos que chamarem este.

Nós seguiremos a associação da esquerda para a direita, com exceção do operador de exponenciação, que seguirá a associação da direita para a esquerda. Nós utilizaremos os parênteses para atribuir precedência máxima à expressão entre eles.

Operadores aritméticos, da maior à menor prioridade

1. `^` (exponencial), `++` pósfixo, `--` pósfixo
2. `+` unário, `-` unário
3. `*`, `/`, `%`
4. `+` binário, `-` binário

Operadores lógicos, da maior à menor prioridade

1. `!` (not)
2. `&&` (and)
3. `||` (or)

Operadores relacionais, da maior à menor prioridade

1. `>`, `>=`, `<`, `<=`
2. `==` (igualdade), `!=` (desigualdade)

Com exceção do operador módulo (`%`), que só opera sobre inteiros, todos os operadores aritméticos possuem sobrecarga sobre os tipos **`int`** e **`float`**. Nós também teremos avaliação de curto-circuito.

```
int a, b, c;
float x, y, z;
a = b + c;
x = y + z;
a = b * c;
x = y * z;
```

No caso acima todas as atribuições serão válidas, graças a sobrecarga.

```
int a, b, c, d;
if(a < b && c < d) {
    ...
}
```

No caso acima, caso a afirmativa **`a < b`** seja falsa, o compilador sequer analisará a segunda ocorrência (**`c < d`**) saindo da condição.

Nossa linguagem também terá expressões com efeito colateral. Isso é fundamental, já que o paradigma com que trabalhamos é imperativo.

Não há operadores de tipos distintos e coerção na nossa linguagem, portanto não teremos expressões de tipo misto. Descartamos a liberdade do usuário em prol de conscientizá-lo das restrições de tipo.

As estruturas de controle que serão suportadas serão **`if`**, **`if-else`**, **`switch`**, **`for`**, **`while`** e **`do-while`**, com sua semântica idêntica à semântica em C.

Desvios (*jumps*) não serão suportados pela nossa linguagem, uma vez que, após debater o tema diversas vezes, entramos em acordo que a presença de desvios e labels em excesso criaria um código desorganizado e com péssima legibilidade. Além disso, sua funcionalidade torna-se desnecessária com as estruturas de controle que temos . A presença de escapes foi considerada essencial para o programa. Ela se encontra presente na forma dos elementos **`return`**, usado em subprogramas, e o **`break`**, usado no condicional **`switch`**.

O tratamento de exceções em nossa linguagem será feito através dos comandos try e catch, no qual operações inválidas potenciais em um bloco (**`try`**) são redirecionadas para fluxos alternativos em outros blocos preventivos (**`catch`**), onde poderão ser tratados. Um bloco **`try`** é chamado de bloco "protegido" porque, caso ocorra algum problema com os comandos dentro do bloco, a execução desviará para os blocos **`catch`** correspondentes.

```
try {
    int a = 0;
    int b = 7;
    int c = b / a;
} catch (DivError e) {
    print("Erro na divisao!");
}
```

Em relação às categorias de declaração, nós teremos declaração de variável, declaração de tipo, declaração de constante e declaração de procedimento (função). Também teremos declaração recursiva para permitir a criação de funções e tipos recursivos. A linguagem permitirá inicialização de variáveis. Em uma declaração simples (apenas uma variável), a inicialização será feita com um valor de mesmo tipo que a variável.

Em **`int a`**, **`a`** já está setado com um valor de inteiro, pois a única forma de haver um espaço alocado na memória para essa variável é caso ela possua um valor. Esse valor é setado como nulo (**`null`**) até ser redefinido. Para declarações múltiplas, o valor atribuído será usado para inicializar todas as variáveis na declaração.

```
int a, b, c = 7;
```

No caso acima, o valor das variáveis **`a`**, **`b`** e **`c`** será setado como 7. Declarações de variáveis compostas (arrays, structs) só poderão ser inicializadas com um conjunto de elementos que seja compatível e tenha o mesmo tamanho que a variável em questão. Em **`int vetor[3]`**, **`vetor`** tem valor **`null`**.

8 Subprogramas

Em Lingmas teremos suporte tanto a funções quanto a procedimentos, explicitando sua diferença através da sintaxe para deixar claro aos usuários.

O tipo dos parâmetros de um subprograma pode ser especificado como qualquer tipo primitivo ou um tipo criado pelo usuário, mas subprogramas não poderão ser utilizados como parâmetros. A utilização dos parâmetros será de forma posicional, que em comparação ao uso de palavras-chave, melhora a capacidade de escrita do código. Os tipos de passagem utilizados serão por referência e por cópia. A passagem por referência será representada por um asterisco (`*`) junto ao nome da variável que será passada por referência. Já que não temos ponteiros na nossa linguagem, não há problema de conflito..

Subprogramas terão declaração de protótipos e também teremos o recurso de evasão de checagem de tipo, onde utilizaremos reticências (`...`) para demarcar que a partir dali não importa quais são as variáveis. Como exemplo, temos:

```
int printf(string format, ...);
int scanf(string format, ...);
```

Nós teremos suporte apenas a subprogramas por sobrecarga. A escolha da sobrecarga se deu por causa da legibilidade e da utilidade dessa funcionalidade.

Será utilizado compilação separada, por ser uma compilação prática, evitando que todo o código necessite ser compilado caso haja uma edição em um módulo, e pelo fato que a compilação independente não tem verificação de coerência.

Como Lingmas possui um escopo educativo, não implementaremos co-rotinas, por ser uma funcionalidade mais avançada.

9 Abstrações da linguagem

Abstrações são essenciais para a simplificação dos programas presentes na nossa linguagem, filtrando informações e operações, e apresentando ao usuário apenas o que é relevante. Exemplos de abstrações são o tipo float, uma vez que na realidade, não é possível ao computador expressar decimais muitos pequenos ou números muito grandes, apenas aproximações, ou então o uso de classes em linguagens como c++ ou java, que escondem dados do tipo private do restante do programa e do programador.

Na nossa linguagem, utilizaremos tipos abstratos de dados, presentes tanto no tipo float, quanto em tipos criados pelo usuário. Dessa forma poderemos facilitar o manuseio e definição de tipos para nossos usuário.

Depois de discussões, optamos por não aceitar classes em nossa linguagem, pois mesmo com o ganho de abstração e declaração, consideramos que a adição de classes não seria necessário, uma vez que já temos structs.

Nós também utilizaremos pacotes (packages). A ideia de pacotes será implementada semelhante à utilização de módulos em Ada, onde cabeçalho e corpo podem ser separados. Porém, na nossa implementação, cabeçalho e corpo terão que estar no mesmo arquivo.

Para chamar um pacote em um arquivo, será utilizado o comando **`import <nome_do_arquivo>`**. Por fim, não utilizaremos módulos, pois como ele se destina a um público iniciante, o caráter dos programas é simples, não necessitando das abstrações de módulo.

O estilo escolhido para nossa linguagem foi a implementação de parametrização em C++. Nele, é possível parametrizar uma TAD (ou classe) apenas alterando o construtor, ou adicionando um construtor genérico (que pode ser feito caso a linguagem possua sobrecarga, que é o caso do Lingmas). Porém, como não possuímos classes, então tal parametrização se aplica apenas a TADs.

Além disso, na nossa linguagem teremos encapsulamentos,que serão da seguinte forma: uma coleção de funções e dados relacionados podem ser colocadas em arquivos compilados separadamente, agindo como uma biblioteca. A interface para tal arquivo é colocada em um arquivo separado, chamado de header file (arquivo de cabeçalho). Quando o pré-processador lê uma declaração de **`#include`**, como **`#include "biblioteca.h"`**, o conteúdo do arquivo de cabeçalho (neste caso, **`biblioteca.h`**) é inserido. Apesar de possuir certas inseguranças, optamos por usar dessa forma pois o conceito é simples e útil, e não temos como objetivo dar suporte ao desenvolvimento de aplicações muito grandes. Pelo mesmo motivo, não teremos encapsulamento de nomes, pois os programas não devem ser grandes o suficiente para poder causar conflito de nomes.

10 BNF para a linguagem

```
<program> ::= <stmts>
<stmts> ::= { <stmt>";" }
<stmt> ::= <ctrl> | <func> | <attr> | <exp> | <condt>
<func> ::= <func_type> <id> "(" <prmts> ")"
                  "{" <stmts> <return> {<stmts> <return>}"}"
<ctrl> ::= <condt> | <loop>
<condt> ::= <if> | <switch>
```

```
<if> ::= "if" "(" <exp> ")" "{" <stmts> "}" [ "else" "{" <stmts> "}" ]
<switch> ::= "switch" "(" <id> ")" "{" <case_list> "}"
<case_list> ::= "case" <number> ":" <stmts> [ "break" ] | <case_list>
<loop> ::= <while> | <for> | <do_while>
<while> ::= "while" "("<boolean_exp>")" "{" <stmts> "}"
<for> ::= "for" "(" <attr> ";" <boolean_exp> ";" <exp> ")" "{" <stmts> "}"
<do_while> ::= "do" "{" <stmts> "}" "while" "(" <exp> ")"
<trmt> ::= <return> | "break"
<return> ::= "return" <id> ";"
<attr> ::= <id> "=" <exp> ";"
<exp> ::= <boolean_exp> | <arit_exp>
<id> ::= <char> { <id> }
<arit_exp> ::= <arit_exp> <binary_op> <arit_exp> | <id><unary_op> | <id>

<binary_op> ::= + | - | * | /
<unary_op> ::= ++ | --
<boolean_exp> ::= <func> | <exp> <rel_op> <exp>|<id>
<rel_op> ::= < | <= | >= | > | != | ==
<var_dec> ::= <type> <id>
<type> ::= "int" | "float" | "string" | "bool"
<letter> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s
    | t | u | v | x | w | y | z
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<number> ::= <digit> | { <digit> }
```