A Linguagem $lang^2$

Neste documento é apresentada a especificação da linguagem de programação que será usada na implementação dos trabalhos da disciplina CSI506-Compiladores, denominada $lang^2$. A linguagem $lang^2$ tem propósito meramente educacional, contendo construções que se assemelham às de várias linguagens conhecidas. No entanto, a linguagem não é um subconjunto de nenhuma delas.

A descrição da linguagem é dividida entre seus diversos aspectos, a saber: sintático, descrito na Seção 1, e léxico, descrito na Seção 2. Este documento ainda não define formalmente a semântica de $lang^2$.

1 Estrutura Sintática

Em linhas gerais, um programa nesta linguagem é constituído por um conjunto de definições de tipos de dados, seguido por definições de funções. A estrutura sintática da linguagem é dividida em: *tipos de dados e declarações*, *funções*, *comandos* e *expressões*. Cada uma dessas estruturas é detalhada nas subseções subsequentes. A gramática completa da linguagem $lang^2$ pode ser encontrada na Seção 4.

1.1 Tipos de Dados e Declarações

Programas $lang^2$ podem conter definições de tipos de dados do tipo registro, os quais são definidos usando a palavra-chave **data**. Após a palavra-chave **data**, segue o nome do novo tipo, o qual deve começar com uma letra maiúscula, seguido por uma lista de declarações de variáveis delimitada por chaves. Por exemplo, um tipo para representar uma fração pode ser definido como:

```
1 data Racional {numerador :: Int;
2                denominador :: Int;
3               }
```

Esse tipo é denominado *Racional* e contém dois atributos do tipo inteiro, denominados *numerador* e *denominador*. A sintaxe para especificar os atributos de um tipo registro é a mesma usada para declarações de variáveis em funções, isto é, o nome do atributo ou variável seguido por dois pontos, o tipo do atributo (ou variável) e finalizado com ponto e vírgula.

1.2 Funções

A definição de funções e procedimentos é feita fornecendo-se o nome da função (ou procedimento) e sua lista de parâmetros. Após os parâmetros, segue o símbolo *::* e a anotação de tipo da função. Na anotação, representamos o tipo da função com a mesma notação usada na linguagem Haskell, isto é, *Tipo1 -> Tipo2 -> ... -> Tipo de retorno*. Note que uma função pode ter mais de um valor de retorno, os quais são separados pelo símbolo &. Para procedimentos, que não retornam valores, o tipo é anotado como *Void*. Por fim, segue o bloco de comandos.

```
main :: Void {
  print(str(fat(10)[0]));
}

fat num :: Int {
  if (num < 1)
    return 1;
  else
    return num * fat(num-1)[0];
}

divmod num div :: Int -> Int -> Int & Int {
  q = num / div;
  r = num % div;
  return q r;
}
```

Figura 1: Exemplos de funções e procedimentos na linguagem $lang^2$.

Um exemplo de programa na linguagem $lang^2$ é apresentado na Figura 1, contendo a definição de um procedimento denominado *main* e das funções *fat* e *divmod*. A função *fat* recebe um valor inteiro como parâmetro e retorna o fatorial desse valor. A função *divmod* é um exemplo de função com mais de um valor de retorno, recebendo dois parâmetros inteiros e retornando o quociente e o resto da divisão do primeiro pelo segundo.

1.3 Classes e Instâncias de Tipos

A linguagem $lang^2$ permite sobrecarga de nomes por meio de classes de tipos, que funcionam de forma análoga às da linguagem Haskell. Uma classe define apenas uma coleção de nomes e seus respectivos tipos, sendo análoga a uma interface na linguagem Java. Além disso, uma classe de tipo é sempre parametrizada por uma variável de tipo. Cada instância da classe deve substituir essa variável por um tipo concreto e prover todas as definições da interface com um corpo apropriado.

Na Figura 2, apresentamos um programa que define a classe **Chr** com o parâmetro de tipo a. Nessa classe, definimos apenas o símbolo **tychr**, cujo tipo representa uma função que mapeia o parâmetro de tipo a para um valor do tipo **Char**. Em seguida, definimos duas instâncias, uma para o tipo **Int** e outra para o tipo **Bool**. A declaração **instance Chr for Int** permite implementar a classe **Chr** para o tipo **Int**, substituindo o parâmetro de tipo a por **Int**. No corpo da função **main**, utilizamos o método **tyChr** com o mesmo nome para dois tipos distintos.

1.4 Comandos

A linguagem $lang^2$ apresenta apenas seis comandos básicos, classificados em comandos de atribuição, seleção, retorno, iteração e chamada de funções e procedimentos.

O comando de atribuição possui sintaxe semelhante à das linguagens imperativas C/C++ e Java, na qual uma expressão do lado esquerdo especifica o endereço onde será armazenado o valor resultante da avaliação da expressão do lado direito. A linguagem apresenta dois comandos

```
class Chr a {
    tychr :: a -> Char;
}
instance Chr for Int {
     tychr v :: Int -> Char {
        return 'I';
     }
}
instance Chr for Bool {
     tychr v :: Bool -> Char {
        return 'B';
     }
}
main :: Void {
  print(tyChr(0)[0]);
  print(tyChr(true)[0]);
}
```

Figura 2: Exemplos de classes de tipos e instâncias na linguagem $lang^2$.

de seleção: *if-then* e *if-then-else*. Os valores de retorno de uma função são definidos por meio do comando **return**, seguido por uma lista de expressões separadas por espaço.

A linguagem $lang^2$ apresenta apenas um comando de iteração, com as seguintes estruturas:

```
iterate (expr) cmd
iterate (i : expr) cmd
```

O comando **iterate** especifica um trecho de código que será executado uma quantidade de vezes determinada pela avaliação da expressão delimitada por parênteses. Ressalta-se que a expressão é avaliada **uma única vez**, e o laço só será executado se o valor resultante for maior que zero. A segunda forma do comando **iterate** permite o uso de uma variável contadora i, que pode ser utilizada dentro do corpo do laço. Caso a expressão seja um vetor, o comando itera sobre o número de elementos do vetor na primeira forma, enquanto, na segunda forma, a variável contadora assume cada elemento do vetor.

Chamadas de funções e procedimentos são consideradas comandos. A sintaxe para chamada de procedimento consiste no nome do procedimento seguido por uma lista de expressões separadas por espaço e delimitadas por parênteses. Em $lang^2$, os parênteses não devem ser separados do nome da função por espaço. Por exemplo, a chamada ao procedimento *main* da Figura 1 é:

```
main();
```

A chamada de função é semelhante; no entanto, deve-se especificar uma lista de endereços para armazenar os valores de retorno, como na seguinte chamada da função *divmod*:

```
divmod(5 2)<q r>;
```

Esse comando define que os valores de retorno da função serão atribuídos às variáveis q e r.

Por fim, um bloco de comandos é definido como uma sequência de zero ou mais comandos delimitada por chaves.

1.5 Expressões

Expressões são abstrações sobre valores e, em $lang^2$, são muito semelhantes as expressões aritméticas de outras linguagens, i.e., possuem os operadores aritméticos usuais(+,-,*,/,%) além de operadores lógicos (&&, !), de comparação (<,==,!=) e valores (inteiros, caracteres, booleanos, floats, registros, vetores e chamadas de métodos). Adicionalmente, parêntesis podem ser usados para determinar a **prioridade** de uma sub-expressão.

Observe, no entanto, que o conjunto de operadores é reduzido. Por exemplo, operações com valores lógicos (tipo booleano) são realizadas com os operadores de conjunção (&&) e negação (!). A linguagem não prover operadores para as demais operações lógicas. Como consequência, se queremos realizar uma seleção quando o valor de ao menos uma de duas expressões, p e q, resulta em verdadeira, escrevemos:

```
1 if(!(!p && !q)) { ... }
```

As chamadas de funções são expressões. Porém, diferentemente das linguagens convencionais, usa-se um índice para determinar qual dos valores de retorno da função será usado. Assim, a expressão $divmod(52)[0]$ se refere ao primeiro retorno, enquanto a expressão $divmod(52)[1]$ ao segundo. Note que a indexação dos retornos da função é feita de maneira análoga ao acesso de vetores, no qual o primeiro retorno é indexado por 0, o segundo por 1 e assim sucessivamente.

A expressão $x * x + 1 < fat(2 * x)[0]$ contém operadores lógicos aritméticos e chamadas de funções. Porém, em qual ordem as operações devem ser realizadas? Se seguirmos a convenção adotada pela aritmética, primeiramente deve ser resolvidas as **operações mais fortes** ou de **maior precedência**, i.e. a multiplicação e a divisão, seguida das **operações mais fracas** ou de **menor precedência**, i.e. a soma e subtração. Assim certos operadores tem prioridade em relação a outros operadores, i.e., devem ser resolvidos antes de outros. Para a expressão $x*x+1$ é fácil ver que a expressão $x * x$ deve ser resolvida primeiro e em seguida deve-se somar 1 ao resultado. E quando há operadores de tipos diferentes, como na expressão $x*x+1 < fat(2*x)[0]$? A situação é semelhante, resolve-se o que tem maior precedência, a qual é determinada pela linguagem. Neste exemplo, a última operação a ser realizada é a operação de comparação, cuja precedência é a menor dentre todos os operadores da expressão. A Tabela 1 apresenta a precedência dos operadores da linguagem $lang^2$. O operador que tiver o maior valor da coluna *nível* tem maior precedência.

Sabendo a precedência dos operadores podemos determinar em qual ordem as operações devem ser executadas quando há operadores com diferentes níveis de precedência. Entretanto, como determinar a ordem das operações se uma determinada expressão contém diferentes operadores com a mesma precedência, como nas expressões $v[3].y[0]$ e $x/3 * y$?

Em situações como essas, determinamos a ordem de avaliação das operações a partir da associatividade dos operadores, que pode ser à esquerda ou à direita. Quando os operadores são associativos à esquerda, resolvemos a ordem das operações da esquerda para a direita. Caso os operadores sejam associativos à direita, fazemos o inverso. Em ambas as expressões $v[3].y[0]$ e $x/3 * y$, os operadores são associativos à esquerda. Portanto, na primeira expressão, primeiro é realizado o acesso ao vetor v, depois acesso ao membro y e, por fim, acesso ao vetor de y. Na segunda expressão, realiza-se primeiro a divisão de x por 3 e, em seguida, a multiplicação do resultado por y.

Nível	Operador	Descrição	Associatividade
7	[ ] . ( )	acesso a vetores acesso aos registros parêntesis	esquerda
6	! -	negação lógica menos unário	direita
5	* / %	multiplicação divisão resto	esquerda
4	+ -	adição subtração	
3	<	relacional	não é associativo
2	== !=	igualdade diferença	esquerda
1	&&	conjunção	esquerda

Tabela 1: Tabela de associatividade e precedência dos operadores. Tem a maior precedência o operador de maior nível.

2 Estrutura Léxica

A linguagem usa o conjunto de caracteres da tabela ASCII[1]. Cada uma das possíveis categorias léxicas da linguagem são descritas a seguir:

- Um **identificador (ID)** é uma sequência de letras, dígitos e sobrescritos (*underscore*) que, obrigatoriamente, começa com uma letra minúscula. Exemplos de identificadores: *var*, *var_1* e *fun10*;
- Um **nome de tipo (TYID)** é semelhante a regra de identificadores, porém a primeira letra é maiúscula; Exemplos de nomes de tipos: *Racional* e *Point*;
- Um **literal inteiro (INT)** é uma sequência de um ou mais dígitos;
- Um **literal ponto flutuante (FLOAT)** é uma sequência de zero ou mais digitos, seguido por um ponto e uma sequência de um ou mais digitos. Exemplos de literais ponto flutuante: 3.141526535, 1.0 e .12345;
- Um **literal caractere (CHAR)** é um único caractere delimitado por aspas simples. Os caracteres especiais quebra-de-linha, tabulação, *backspace* e *carriage return* são definidos usando os caracteres de escape \n, \t, \b e \r, respectivamente. Para especificar um caractere \, é usado \\e para a aspas simples o \' e aspas duplas \". Também é permitido especificar uma caráctere por meio de seu código ASCII, usado \seguido por exatamente três dígitos. Exemplos de literais caractere: 'a', '\n', '\t', '\\', '\065';
- Um **literal lógico** é um dos valores **true** que denota o valor booleano verdadeiro ou **false** que denota o valor booleano falso;

[1] http://www.asciitable.com

- O **literal nulo** é **null**;
- Os símbolos usados para **operadores** e **separadores** são *(*, **)**, **[**, **]**, **{**, **}**, >, **;**, **:**, **::**, **.**, **,**, =, <, ==, **!**=, +, **-**, ***, /, %, **&&**, **&**, e **!**.
- **Palavras reservadas** são identificadores especiais usados pela linguagem como **if**, **else**, **iterate**, **data**, **class**, **instance**, **for**, **return**, **new**, **Int**, **Char**, **Float**, **Bool** e **Void**, etc...

Todos os nomes de tipos, comandos e literais são palavras reservadas pela linguagem. Há dois tipos de comentários: comentário de uma linha e de múltiplas linhas. O comentário de uma linha começa com -- e se estende até a quebra de linha. O comentário de múltiplas linhas começa com {- e se estende até os caracteres de fechamento do comentário, -}. A linguagem não suporta comentários aninhados.

3 Semântica de Lang

A semântica de um programa $lang^2$ é definida em termos de um estado formado por uma memória local M, uma memória global H (ambas associam identificadores a valores) e uma pilha P de valores, que será usada para computações temporárias. Escreveremos $v : P$ para indicar que v foi empilhado no topo da pilha. A pilha vazia será denotada por $\square$. Denotaremos a associação de um identificador I a um valor V na memória como $M[I = V]$. Caso o identificador já esteja associado a algum valor na memória, este será substituído pelo novo valor. A notação $M(I)$ será usada para denotar o valor V associado a um identificador I. Se não houver valor associado ao identificador, a operação resultará em erro. A notação $M/\{I\}$ denota a memória obtida a partir da subtração da associação de $I = V$ de M. Remoções de múltiplas associações simultâneas serão denotadas por $M/\{I_1, \ldots, I_n\}$. Adicionalmente, vamos supor a existência de um contexto de funções $\mathbb{F}$ no qual estão armazenadas todas as funções definidas pelo programa. Também será assumido que $\mathbb{F}$ é um contexto imutável durante a execução do programa.

Para simplificar a escrita, usaremos $M[I_1 = V_1, I_2 = V_2, \ldots, I_n = V_n]$ para especificar várias atribuições simultaneamente na memória. A memória vazia será denotada por []. A mesma notação será usada para a memória global, compartilhada entre funções. Por fim, usaremos a notação α para denotar identificadores únicos utilizados na memória global.

Escreveremos $\langle H, M, P\rangle$ para denotar o estado. Os identificadores serão representados por strings de caracteres. Os valores podem ser inteiros, valores de ponto flutuante, booleanos, a constante *null*, vetores ou registros.

O conjunto de valores $\mathcal{L}$ da linguagem $lang^2$ pode ser descrito recursivamente como:

1. Se $i \in \mathbb{Z}$, então $i \in \mathcal{L}$.
2. Se $f \in \mathbb{R}$, então $f \in \mathcal{L}$.
3. Se $b \in \{True, False\}$, então $b \in \mathcal{L}$.
4. Se c é um caractere ASCII, então $c \in \mathcal{L}$.
5. O valor $null \in \mathcal{L}$.

6. Sejam n valores $v_0, \ldots, v_{n-1}$ e seja A um arranjo de n posições tal que, para $0 \leq i < n$, $A[i] = v_i$. Então, $A \in \mathcal{L}$.

7. Seja uma sequência I de n identificadores e uma sequência V de n valores. O mapeamento $\{(I_i \rightarrow V_i)\}$, $0 \leq i < n$, pertence a $\mathcal{L}$. Valores do tipo mapeamento podem ter seus campos acessados com o operador ".". Por exemplo, se $V = \{x = 0, y = 1\}$, então $V.x$ denota o valor 0. Além disso, os campos podem ter seus valores sobrescritos.

8. α: o identificador (ou endereço) em uma memória global será considerado como um valor.

A regra número 7 descreve valores de tipo registro. Note que registros são associações entre identificadores e valores. Essa associação é única no escopo do registro. Contudo, um campo de um registro pode ser outro registro que possua os mesmos nomes de campo. Um exemplo dessa situação é um tipo de dados recursivo, como uma lista.

A regra 6 descreve vetores como uma sequência indexada de valores arbitrários.

Dadas as definições de valores e de estado, podemos descrever a semântica de cada comando da linguagem $lang^2$. A semântica apresentada a seguir é informal, ou seja, não descreve de forma precisa o comportamento de um programa. As lacunas nas definições devem ser preenchidas levando-se em conta parcimônia e clareza do comportamento do programa.

Expressões A semântica de expressões é dada pela função $S : e \rightarrow \langle H, M, P\rangle \rightarrow \langle H', M', P'\rangle$, definida pelos casos a seguir.

- $S(\text{INT}, \langle H, M, P\rangle) = \langle H, M, i : P\rangle$, onde i é o inteiro denotado por INT.
- $S(\text{FLOAT}, \langle H, M, P\rangle) = \langle H, M, f : P\rangle$, onde f é o valor de ponto flutuante denotado por $FLOAT$.
- $S(\textbf{true}, \langle H, M, P\rangle) = \langle H, M, true : P\rangle$.
- $S(\textbf{false}, \langle H, M, P\rangle) = \langle H, M, false : P\rangle$.
- $S(\textbf{null}, \langle H, M, P\rangle) = \langle H, M, null : P\rangle$.
- $S(e_1 \oplus e_2, \langle H, M, P\rangle) = \langle H, M, f_{op}(\oplus, v_1, v_2) : P\rangle$, onde $\oplus$ é um operador binário qualquer, $\langle H, M, v_1 : P\rangle = S(e_1, \langle H, M, P\rangle)$, $\langle H, M, v_2 : v_1 : P\rangle = S(e_2, \langle H, M, v_1 : P\rangle)$ e a função f_{op} mapeia cada operador para sua semântica correspondente na álgebra. Por exemplo, $f_{op}(+, 1, 2) = 3$. O operador f_{op} está definido para operadores aritméticos, booleanos e relacionais, bem como para valores inteiros, booleanos e de ponto flutuante. No entanto, ele não é definido para vetores e registros.
- $S(lvalue[e], \langle H, M, P\rangle) = \langle H, M, A[i] : P\rangle$, onde $\langle H, M, A : P\rangle = S(lvalue, \langle H, M, P\rangle)$, A é um arranjo e $\langle H, M, i : A : P\rangle = S(e, \langle H, M, A : P\rangle)$.
- $S(lvalue.ID, \langle H, M, P\rangle) = \langle H, M, V : P\rangle$, onde $\langle H, M, R : P\rangle = S(lvalue, \langle H, M, P\rangle)$, R é um mapeamento entre identificadores e valores, e $(ID, V) \in R$.
- $S(ID, \langle H, M, P\rangle) = \langle H, M, M(ID) : P\rangle$.
- $S(ID(e_1 \ldots e_n)[e], \langle H, M, P\rangle) = \langle H, M, V_k : P\rangle$, onde:

  - Verifica-se que há apenas um ID tal que $ID\ ID_1 \ldots ID_n :: \tau\ C \in \mathbb{F}$.
  - Avaliam-se as expressões dos argumentos: $\langle H, M, V_1 : \ldots : V_n : P\rangle = S(e_1, S(e_2, \ldots, S(e_n, \langle H, M, P\rangle)$
  - Constrói-se uma nova memória $M'[ID_1 = V_1, \ldots, ID_n = V_n]$.
  - Executa-se $S(C, \langle H, M', P\rangle) = \langle H, M'', V_1 : \ldots : V_m\rangle$, sendo C o corpo da função executada.
  - Avalia-se a expressão que denota o valor de retorno a ser usado: $S(e, \langle H, M, V_1 : \ldots : V_m\rangle) = \langle H, M, k : V_1 : \ldots : V_m\rangle$, em que $0 \leq k \leq m$.
  - Finalmente, salva-se apenas o k-ésimo valor de retorno na pilha, removendo-se os demais valores: $\langle H, M, V_k : P\rangle$.

- $S(new\ \ TYPE\ \ n, \langle H, M, P\rangle) = \langle H[\alpha = V], M, \alpha : P\rangle$, onde o valor V é obtido por uma instanciação do tipo $TYPE$, e o valor n pode ser omitido caso $TYPE$ seja o nome de um registro. Note que a operação *new* insere, na memória global, o valor associado a um identificador α e insere na pilha o nome, ou endereço, do valor na memória global.

Instanciação de valores Uma instanciação de valores $new\ \ TYPE\ \ n$ ocorre da seguinte forma:

- Se o tipo a ser instanciado for primitivo (*Int*, *Float*, *Char* ou *Bool*), deve-se criar um vetor de valores com tamanho $n \in \mathbb{Z}$, $n \geq 0$.
- Se o tipo a ser instanciado for um tipo definido pelo usuário e o tamanho for omitido, deve-se utilizar a definição do tipo de dados para criar um mapeamento dos identificadores de cada campo para seus valores padrão correspondentes. O valor padrão de tipos numéricos é 0, o valor padrão de um tipo *Char* é '\000' e o valor padrão de um tipo *Bool* é $false$. O valor padrão de um tipo definido pelo usuário é *null*. Caso o valor n seja especificado, deve-se criar um vetor com n elementos preenchidos com o valor *null*.

Semântica de comandos A semântica de comandos é dada pela função $C : c \to \langle H, M, P\rangle \to \langle H', M', P'\rangle$, definida pelos casos a seguir:

1. $C(lvalue = exp, \langle H, M, P\rangle) =$ Seja $\langle H', M, V : P\rangle = S(exp, \langle H, M, P\rangle)$. Caso V não seja um identificador de memória global, o resultado é $update(\langle H', M, P\rangle, lvalue, V)$, onde *update* é uma operação que escreve o valor no local apropriado da memória, retornando um novo estado. Note que essa função pode precisar acessar a memória global para atualizar um campo de registro ou uma posição de um vetor. Caso $V = \alpha$, então $update(\langle H', M, P\rangle, lvalue, H'[V])$.
2. $C(if\ exp\ stmtBlock, \langle H, M, P\rangle) =$ Seja $\langle H', M, V : P\rangle = S(exp, \langle H, M, P\rangle)$. Caso $V = true$, então $C(stmtBlock, \langle H', M, P\rangle)$; caso contrário, $\langle H', M, P\rangle$.
3. $C(if\ exp\ stmtBlock_1\ stmtBlock_2, \langle H, M, P\rangle) =$ Seja $\langle H', M, V : P\rangle = S(exp, \langle H, M, P\rangle)$. Caso $V = true$, então $C(stmtBlock_1, \langle H', M, P\rangle)$; caso contrário, $C(stmtBlock_2, \langle H', M, P\rangle)$.
4. $C(iterate\ exp\ stmtBlock, \langle H, M, P\rangle)$: Seja $\langle H', M', V : P\rangle = S(exp, \langle H, M, P\rangle)$, onde $M' = M[I_c = V]$ é uma nova memória que mapeia um identificador I_c, que não ocorre em lugar algum do programa[2], para o valor V.

[2]Tais identificadores são referidos como variáveis *fresh*.

(a) Caso $M[I_c] = 0$, então o resultado é $\langle H', M'/\{I_c\}, P\rangle$, concluindo a execução do *iterate*.

(b) Caso $M[I_c] = i$, com $i > 0$, então $\langle H'', M'', P'\rangle = C(stmtBlock, \langle H', M', P\rangle)$ e, em seguida, $M''' = M''[I_c = M''[I_c] - 1]$, e o estado resultante é $\langle H'', M''', P'\rangle$. Considerando a memória desse novo estado, retomamos a execução a partir da regra 4a.

(c) Caso $V = A$, sendo A um arranjo de tamanho n, considere n como um valor inteiro e prossiga de acordo com as regras 4a e 4b.

5. $C(iterate\ id : exp\ stmtBlock, \langle H, M, P\rangle)$: Nesse caso, considera-se id como uma variável de acesso ao contador. Além disso, a variável id, se declarada no comando *iterate*, é local ao bloco do comando, isto é, não deve ser acessível fora dele. Caso a variável já tenha sido previamente declarada, ela permanecerá acessível fora do bloco do comando. Seja $\langle H', M', V : P\rangle = S(exp, \langle H, M, P\rangle)$, onde $M' = M[I_c = V, id = V]$ é uma nova memória que mapeia um identificador *fresh* I_c.

   (a) Caso $M[I_c] = 0$, então o resultado é $\langle H', M'/\{I_c\}, P\rangle$, se id foi declarada previamente ao comando *iterate*, ou $\langle H', M'/\{I_c, id\}, P\rangle$, se id foi declarada no comando *iterate*. Esse passo conclui a execução do *iterate*.

   (b) Caso $M[I_c] = i$, com $i > 0$, então $\langle H'', M'', P'\rangle = C(stmtBlock, \langle H', M', P\rangle)$ e, em seguida, $M''' = M''[I_c = M''[I_c] - 1, id = M''[I_c] - 1]$, e o estado resultante é $\langle H'', M''', P'\rangle$. Considerando a memória desse novo estado, retomamos a execução a partir da regra 5a.

   (c) Caso $V = A$, sendo A um arranjo de tamanho n, considere $M' = M[I_c = n, id = A[n - M[I_c]]]$ e:

      i. Caso $M[I_c] = 0$, então o resultado é $\langle H', M'/\{I_c\}, P\rangle$, se id foi declarada previamente ao comando *iterate*, ou $\langle H', M'/\{I_c, id\}, P\rangle$, se id foi declarada no comando *iterate*. Esse passo conclui a execução do *iterate*.

      ii. Caso $M[I_c] = i$, com $i > 0$, então $\langle H'', M'', P'\rangle = C(stmtBlock, \langle H', M', P\rangle)$ e, em seguida, $M''' = M''[I_c = M''[I_c] - 1, id = A[M''[I_c] - 1]]$, e o estado resultante é $\langle H'', M''', P'\rangle$. Considerando a memória desse novo estado, retomamos a execução a partir da regra 5(c)i.

6. $C(return\ exp_1 \dots exp_n, \langle H, M, P\rangle)$:

   - $\langle H_1, M, V_1 : P\rangle = S(exp_1, \langle H, M, P\rangle)$
   - $\dots$
   - $\langle H_n, M, V_n \dots V_1 : P\rangle = S(exp_n, \langle H_{n-1}, M, V_{n-1} \dots V_1 : P\rangle)$

   O estado resultante é $\langle H_n, M, V_n \dots V_1 : P\rangle$. Adicionalmente, o comando *return* deve causar o término da execução da função corrente. Após a chamada de *return*, nenhum comando subsequente na função deve ser executado, e o fluxo de controle deve retornar para quem chamou a função.

7. A semântica da chamada de função no nível de comando é análoga à chamada de função em expressões, exceto pelo fato de que os valores de retorno sempre devem ser removidos da pilha e associados aos *lvalues* anotados entre $<$ e $>$, respectivamente.

8. A semântica de uma sequência $C(\{c_1\ c_2 \dots\ c_n\}, \langle H, M, P\rangle) = \langle H'', M'', P''\rangle$, onde $\langle H', M', P'\rangle = C(c_1, \langle H, M, P\rangle)$ e $\langle H'', M'', P''\rangle = C(\{c_2 \dots\ c_n\}, \langle H', M', P'\rangle)$.

Além da parte formal apresentada neste texto, considere também que programas corretos em $lang^2$ sempre satisfazem as seguintes afirmações:

- Os operadores aritméticos são homogêneos em relação aos tipos de seus operandos. Ou seja, um operador aritmético $a \oplus b$ pode operar apenas sobre argumentos inteiros (ambos a e b) ou sobre argumentos de ponto flutuante, mas nunca sobre um argumento inteiro e um argumento de ponto flutuante.
- Todas as funções devem ser declaradas antes de seu uso, e funções recursivas e mutuamente recursivas são suportadas pela linguagem.
- Não deve existir código morto (sequência de comandos que nunca é alcançada pelo fluxo de controle).
- As chamadas de funções são sempre realizadas observando-se o **número correto de argumentos** esperados na definição.
- A quantidade de expressões passadas ao comando Códigoreturn sempre corresponde à quantidade de termos na anotação de tipo da função.
- Toda variável deve ser declarada antes de ser usada.
- Todo tipo de dado definido pelo usuário deve ser declarado antes de ser usado, sendo permitido o uso de tipos de dados recursivos e mutuamente recursivos.

3.1 Funções Primitivas da Linguagem $lang^2$

A linguagem $lang^2$ não define construções sintáticas para entrada e saída de dados, nem para algumas outras funcionalidades. Em vez disso, define algumas funções como primitivas.

Uma função primitiva é uma função que existe na linguagem, mas não é necessariamente definida na própria linguagem. Tal mecanismo permite expressar funcionalidades como entrada e saída (I/O), que não são triviais de serem definidas na linguagem. Outro uso comum é a integração de recursos de baixo nível que podem não ser possíveis de se expressar em alto nível, mas que podem aparecer na linguagem como funções primitivas.

Neste momento, as seguintes funções primitivas devem ser consideradas como parte da linguagem $lang^2$:

- `print :: Char -> Void`. Imprime um único caractere na saída padrão.
- `printb :: Char[] -> Int -> Int -> Void`. Imprime um intervalo de um vetor de caracteres na saída padrão. Por exemplo, `printb(buff, i, l)` imprime `l` caracteres, começando da posição `i` do vetor.
- `read :: Char`. Lê um único caractere da entrada padrão.
- `readb :: Char[] -> Int -> Int -> Int`. Lê uma sequência de caracteres da entrada, transferindo-os para um vetor de caracteres. Retorna o número de caracteres efetivamente lidos. Por exemplo, `readb(buff, i, l)` lê até `l` caracteres da entrada e os armazena em `buff` a partir da posição `i`. Caso a quantidade de dados a serem lidos exceda o limite do vetor (a partir da posição `i`), a leitura é encerrada.

4 Definição Formal da Sintaxe de $lang^2$

A Gramática 2 apresenta a gramática livre-de-contexto que descreve a sintaxe da linguagem $lang^2$ usando a notação EBNF. Os símbolos terminais serão grafados como **'sym'** e as palavras reservadas como **palavra**. Um símbolo com a grafia de *TOKEN* significa que esse símbolo denota um conjunto de lexemas que denotam o mesmo token, como o caso de identificadores e números inteiros, por exemplo. A forma { x } denota a repetição de 0 ou mais vezes de x e a forma $[x]$ denota que x é opcional.

prog	→	{decl}
decl	→	data | func | classDec | instDec
data	→	**data** *TYID* **'{'** {bind} **'}'**
bind	→	*ID* **'::'** typeAnnot **';'**
func	→	*ID* {*ID* } **'::'** typeAnnot block
classDec	→	**class** *TYID* *ID* **'{'** {bind} **'}'**
instDec	→	**instance** *TYID* **for** *ID* **'{'** {func} **'}'**
typeAnnot	→	tyJoin
	|	type **'->'** typeAnnot
tyJoin	→	type
	|	type { **'&'** type}
type	→	type **'['** **']'** | btype
btype	→	**Int** | **Char** | **Bool** | **Float** | *TYID* | **Void**
block	→	**'{'** {cmd} **'}'**
stmtBlock	→	block | cmd
cmd	→	**if** '(' exp ')' stmtBlock
	|	**if** '(' exp ')' stmtBlock **else** stmtBlock
	|	**iterate** '(' loopCond ')' stmtBlock
	|	**return** {exp} **';'**
	|	lvalue '=' exp **';'**
	|	*ID* '(' [exps] ')' ['<' lvalue {',' lvalue} '>'] **';'**
loopCond	→	*ID* ':' exp
	|	exp
exp	→	exp operator exp
	|	'!' exp | '-' exp
	|	**true** | **false** | **null**
	|	*INT* | *FLOAT* | *CHAR*
	|	**new** type ['[' exp **']'**]
	|	*ID* '(' [exps] ')' '[' exp **']'**
	|	lvalue
	|	'(' exp ')'
operator	→	'&&' | '==' | '!=' | '&&' | '+' | '-' | '*'
	|	'/' | '%' | '<'
lvalue	→	lvalue '.' lvalue
	|	**'['** exp **']'**
	|	*ID*
exps	→	exp {',' exp }

Gramática 2: Sintaxe da linguagem $lang^2$