Especificação da Linguagem

Elton Cardoso[1], Leonardo Reis[2], and Rodrigo Ribeiro[1]

[1]Universidade Federal de Ouro Preto
[2]Universidade Federal de Juiz de Fora

Neste documento é apresentada a especificação da linguagem de programação que será usada na implementação dos trabalhos da disciplina DCC045-Teoria dos Compiladores, denominada *lang*. A linguagem *lang* tem propósito meramente educacional, contendo construções que se assemelham a de várias linguagens conhecidas. No entanto, a linguagem não é um subconjunto de nenhuma delas. A descrição da linguagem é dividida entre os diversos aspectos da linguagem, a saber, léxico, sintático, sistema de tipos e semântico.

1 Estrutura Sintática

A Gramática 1 apresenta a gramática livre de contexto que descreve a sintaxe da linguagem *lang* usando a notação EBNF. Os meta-símbolos estão entre aspas simples, quando usados como tokens. As palavras reservadas estão em negrito e os demais tokens escritos com letras maiúsculas. A notação $[E]$ denota que E é **opcional**, i.e., ou a gramática derivará o conteúdo entre chaves ou λ. Por sua vez, a notação $\{E\}$ representa 0 ou mais ocorrências de E.

Em linhas gerais, um programa nesta linguagem é constituído por um conjunto de definições de tipos de dados e funções. A estrutura sintática da linguagem é dividida em: *tipos de dados e declarações*, *funções*, *comandos* e *expressões*. Cada uma dessas estruturas são detalhadas nas subseções subsequentes.

1.1 Tipos de Dados e Declarações

Programas *lang* podem conter definições de tipos de dados 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, e uma lista de declarações de variáveis delimitadas por chaves. Por exemplo, um tipo para representar uma fração pode ser definido como:

```
prog     →  {def}
def      →  data | fun
data     →  abstract data TYID '{' {decl  | fun} '}'
         |  data TYID '{' {decl} '}'
decl     →  ID '::' type ';'
fun      →  ID '(' [params] ')' [':' type (',' type)*] cmd
params   →  ID '::' type {',' ID '::' type}
type     →  type '[' ']'  | btype
btype    →  Int  | Char  | Bool  | Float  | TYID
block    →  '{' {cmd} '}'
cmd      →  block  | if '(' exp ')' cmd  | if '(' exp ')' cmd else cmd
         |  iterate '(' itcond ')' cmd  | read lvalue ';'  | print exp ';'
         |  return exp {',' exp} ';'  | lvalue '='  exp ';'
         |  ID '(' [exps] ')' ['<'  lvalue {',' lvalue} '>' ] ';'
itcond   →  ID ':' exp  | exp
exp      →  exp op exp  | '!'  exp  | '-'  exp  | lvalue  | '(' exp ')'
         |  new type [ '[' exp ']' ]
         |  ID '(' [exps] ')' '[' exp ']'
         |  true  | false  | null  | INT  | FLOAT  | CHAR
op       →  '&&'  | '<'  | '=='  | '!='  | '+'  | '-'  | '*'  | '/'  | '%'
lvalue   →  ID  | lvalue '[' exp ']'  | lvalue '.' ID
exps     →  exp {',' exp }
```

Gramática 1: Sintaxe da linguagem *lang*

```
data Racional {numerador :: Int;
               denominador :: Int;
              }
```

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

Um tipo de dados pode ser declarado como abstrato usando a palavra-chave **abstract** antes de **data**. Um tipo abstrato, além das declarações de variáveis, pode conter uma coleção de definições de funções, as quais são as únicas funções que podem manipular diretamente os campos do tipo. Funções definidas fora do escopo do tipo abstrato não têm visibilidade da definição interna do tipo. O código abaixo define o tipo Racional como abstrato juntamente com a operação de adição.

```
abstract data Racional {
  numerador :: Int;
```

```
3   denominador :: Int;
4
5   add(r1 :: Racional, r2 :: Racional) : Racional {
6     Racional r = new Racional;
7     r.denominador = r1.denominador * r2.denominador;
8     r.numerador =   r1.numerador * r2.denominador
9                   + r2.numerador * r1.denominador;
10    return r;
11  }
12 }
```

Ressalta-se que as funções definidas em um tipo de dados abstratos funciona como funções normais, porém essas podem acessar os campos do tipo, como no trecho *r1.denominador * r2.denominador*. Tentativas de acesso ao campo de tipos abstratos por funções definidas externamente são inválidas.

1.2 Funções

A definição de funções e procedimentos é feita dando-se o nome da função (ou procedimento) e a sua lista de parâmetros, delimitados por parêntesis. Parâmetros são seguidos por dois-pontos e os tipos de retorno, em caso de função. Note que uma função pode ter mais de um retorno, os quais são separados por vírgula. Para procedimentos, não há informação sobre retorno, visto que procedimentos não retornam valores. Por fim, segue o comando da função. Um exemplo de programa na linguagem *lang* é apresentado a seguir, o qual contém a definição de um procedimento denominado *main* e as funções *fat* e *divmod*. A função *fat* recebe um valor inteiro como parâmetro e tem como retorno o fatorial desse valor. A função *divmod* é um exemplo de função com mais de um valor de retorno. Essa função recebe dois parâmetros inteiros e retorna o quociente e o resto da divisão do primeiro pelo segundo parâmetro.

```
1  main() {
2    print fat(10)[0];
3  }
4
5  fat(num :: Int) : Int {
6    if (num < 1)
7      return 1;
8    else
9      return num * fat(num-1)[0];
10 }
11
12 divmod(num :: Int, div :: Int) : Int, Int {
13   q = num / div;
14   r = num % div;
15   return q, r;
```

```
16 }
```

1.3 Comandos

A linguagem *lang* apresenta apenas 8 comandos básico, classificados em comandos de atribuição, seleção, entrada e saída, retorno, iteração e chamada de funções e procedimentos.

O comando de atribuição tem a mesma sintaxe das linguagens imperativas C/C++ e Java, na qual uma expressão do lado esquerdo especifica o endereço que será armazenado o valor resultante da avaliação da expressão do lado direito. A linguagem apresenta dois comados de seleção: um *if-then* e *if-then-else*. Leitura e escrita da entrada/saída padrão são realizadas usando os comandos **read** e **print**, respectivamente. O comando **read** é seguido por um endereço onde será armazenado o valor lido da entrada padrão e o comando **print** é seguido por uma expressão. Os valores de retorno de uma função são definidos por meio do comando **return**, o qual é seguido por uma lista de expressões, separadas por vírgula. A linguagem *lang* apresenta apenas um comando de iteração cuja estrutura pode ser:

```
1 iterate (expr) cmd
```

ou

```
1 iterate (ID : expr) cmd
```

No primeiro caso, o comando **iterate** especifica um trecho de código que será executado por uma quantidade de vezes determinada pela avaliação da expressão delimitada entre parêntesis. A expressão é avaliada uma única vez e o laço só será executado se o valor resultante da avaliação da expressão for maior que zero. Na segunda forma, o comportamento do **iterate** dependerá do resultado da avaliação da expressão *expr*: caso seja um valor inteiro, o comportamento é o mesmo do outra forma, porém o nome *ID* é iniciado com o contador da iteração corrente; caso a expressão seja um vetor, o **iterate** executará o comando associado pela quantidade de vezes do tamanho do vetor e, a cada iteração, o nome *ID* é associado com um elemento do vetor.

Chamadas de funções e procedimentos são comandos. A sintaxe para chamada de procedimento é o nome do procedimento seguido por uma lista de expressões separadas por vírgulas. Por exemplo, a chamada ao procedimento *main* seria:

```
1 main();
```

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

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

Esse comando define que os valores de retorno da função serão atribuídos as variáveis q e r. Por fim, um bloco de comandos é definido delimitando-se uma sequência de zero ou mais comandos por chaves.

1.4 Expressões

Expressões são abstrações sobre valores e, em *lang*, são muito semelhantes às expressões 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 subexpressão.

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 $divmod(5, 2)[0]$ se refere ao primeiro retorno, enquanto a expressão $divmod(5, 2)[1]$ 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, primeiro 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 têm 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. Nesse 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*. 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

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.

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.

1.5 Escopo

Um nome é visível desde o local de sua definição até o fim do bloco que o definiu, sendo visível, inclusive, nos blocos mais internos. Uma vez definido, um novo nome não pode ser redefinido em blocos mais internos, sendo assim, o seguinte trecho de código que contém uma redefinição do nome x no **iterate** é inválido:

```
1 read num;
2 x = false;
3 if (num % 2 == 0) {
4   x = true;
5 }
6
7 iterate (x : 10) {
8   print x;
9 }
```

Nomes definidos em blocos internos não são visíveis nos blocos mais externos.

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;

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

- 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 \'. 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', '\\' e '\065';
- Um **literal lógico** é um dos valores **true** que denota o valor booleano verdadeiro ou **false** que denota o valor booleano falso;
- O **literal nulo** é **null**;
- Os símbolos usados para **operadores** e **separadores** são *(*, **)**, **[**, **]**, **{**, **}**, >, **;**, **:**, **::**, **.**, **,**, =, <, ==, **!**=, +, **-**, *, /, %, **&&** e **!**.

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 Estrutura Semântica

Nesta seção descreveremos a estrutura semântica de *lang*. Primeiramente, descreveremos a semântica estática da linguagem e em seguida sua semântica operacional.

3.1 Semântica Estática de Lang

A semântica estática de *lang* é descrita como um conjunto de julgamentos que determinam se um certo programa *lang* é considerado válido ou não. Para isso, descreveremos a estrutura da semântica estática de acordo com os níveis de sintaxe de *lang*. A seção 4.1.1 descreve a sintaxe abstrata e os contextos utilizados para definir todas as regras semânticas. A seções 4.1.2, 4.1.3 e 4.1.4 descrevem regras para expressões, comandos e declarações, respectivamente. A Seção 4.1.5 descreve as regras da semântica estático de um programa completo em *lang*.

3.1.1 Sintaxe Abstrata e Contextos

A sintaxe abstrata de expressões é definida pela seguinte gramática, em que $\bullet$ denota uma lista vazia, n denota constantes inteiras, c constantes de caracteres, f constantes de ponto flutuante, $\circ$ denota operadores binários, $\surd$ denota operadores unários e v denota um identificador.

$$
\begin{array}{lcl}
e & \rightarrow & n \mid f \mid c \mid \textbf{true} \mid \textbf{false} \mid \textbf{null} \mid e \circ e \mid \surd e \mid e[e] \mid v(es)[e] \mid \textbf{new}\ \tau\ e \mid lv \\
lv & \rightarrow & v \mid lv\ .\ v \mid lv[e] \\
\circ & \rightarrow & + \mid - \mid * \mid / \mid \% \mid \&\& \mid == \mid != \mid < \\
\surd & \rightarrow & ! \mid - \\
es & \rightarrow & e\ es \mid \bullet
\end{array}
$$

Expressões são formadas por literais, constantes booleanas, o valor **null**, operadores binários, unários, acesso a arranjos, chamadas de funções, alocação de memória, variáveis e acesso a campos de registros.

A seguir, apresentamos a sintaxe abstrata de comandos.

$$
\begin{array}{lcl}
c & \rightarrow & lv = e \\
 & \mid & \tau\ v = e \\
 & \mid & \textbf{if}\ e\ \textbf{then}\ c\ \textbf{else}\ c \\
 & \mid & \textbf{if}\ e\ \textbf{then}\ c \\
 & \mid & \textbf{iterate}\ e\ c \\
 & \mid & \textbf{iterate}\ id\ e\ c \\
 & \mid & \textbf{read}\ lv \\
 & \mid & \textbf{print}\ e \\
 & \mid & \textbf{return}\ es \\
 & \mid & v(es)\ lvs \\
 & \mid & cs \\
cs & \rightarrow & c\ cs \mid \bullet \\
lvs & \rightarrow & lv\ lvs \mid \bullet
\end{array}
$$

Comandos podem ser atribuições, declaração de variáveis, condicionais, repetição, leitura de valores, impressão no console, retorno de função, chamada de função ou uma sequência de comandos.

Tipos podem ser tipos básicos, tipos definidos pelo usuário ou arranjos.

$$
\begin{array}{lcl}
\tau & \rightarrow & \rho \mid \tau\,[\,] \\
\rho & \rightarrow & \textbf{Int} \mid \textbf{Float} \mid \textbf{Char} \mid \textbf{Bool} \mid \textbf{Void} \mid v
\end{array}
$$

Programas *lang* são formados por declarações de tipos ou de funções, cuja sintaxe abstrata é apresentada a seguir.

$$
\begin{array}{lcl}
prog & \rightarrow & def\ prog \mid \bullet \\
def & \rightarrow & nd \mid fun \\
nd & \rightarrow & v\ fds \mid v\ fds\ funs \\
fun & \rightarrow & v\ (fds)\ ts\ c \\
funs & \rightarrow & fun\ funs \mid \bullet \\
fd & \rightarrow & v{::}\tau \\
fds & \rightarrow & fd\ fds \mid \bullet \\
ts & \rightarrow & \tau\ ts \mid \bullet
\end{array}
$$

Para verificação da semântica estática de programas *lang* utilizaremos três contextos:

- Θ: contexto contendo os tipos de funções definidas pelo usuário. Esse contexto é um conjunto de pares $(f, [v_1 : \tau_1, ..., v_n : \tau_n] \rightarrow [\tau'_1, ..., \tau'_m])$, em que f é o nome da função e $[v_1 : \tau_1, ..., v_n : \tau_n] \rightarrow [\tau'_1, ..., \tau'_m]$ o seu tipo. Note que o tipo dos argumentos é representado por uma sequência de pares de identificadores e seus tipos e o tipo de retorno como uma lista de tipos.
- Δ: contexto contendo informações sobre os tipos definidos pelo usuário. O contexto Δ é um conjunto de pares da forma (x, S) em que x é o nome do tipo e S é um conjunto formado por pares (v, τ) em que $v :: \tau$ é um campo do tipo x.
- Γ: contexto contendo definições de variáveis e operadores de *lang*. Esse contexto é formado por pares (x, τ) em que x é um identificador e τ o seu respectivo tipo. Nesse contexto armazemos os tipos de todos os operadores binários e unários de *lang*.

A notação $\Gamma(x) = \tau$ denota que $(x, \tau) \in \Gamma$ e $\Gamma, x : \tau$ denota $\Gamma \cup \{(x, \tau)\}$. Essas operações aplicam-se também aos contextos Δ e Θ. Finalmente, representamos um contexto vazio por $\bullet$.

3.1.2 Semântica Estática de Expressões

A semântica estática de expressões *lang* são definidas como um julgamento de forma $\Theta; \Delta; \Gamma \vdash_e e : \tau$ que representa que a expressão e possui o tipo τ sobre os contextos Θ, Δ e Γ.

Iniciaremos a descrição com regras para constantes, que possuem regras de tipos imediatas: por exemplo, a primeira regra mostra que constantes inteiras (n) possuem o tipo **Int**. A única pecurialidade é que a constante **null** pode possuir qualquer tipo de arranjo ou de registro, mas não tipos primitivos da linguagem.

$$
\frac{}{\Theta; \Delta; \Gamma \vdash_e n : \mathbf{Int}} \qquad \frac{}{\Theta; \Delta; \Gamma \vdash_e f : \mathbf{Float}} \qquad \frac{}{\Theta; \Delta; \Gamma \vdash_e c : \mathbf{Char}}
$$

$$
\frac{}{\Theta; \Delta; \Gamma \vdash_e \mathbf{true} : \mathbf{Bool}} \quad \frac{}{\Theta; \Delta; \Gamma \vdash_e \mathbf{false} : \mathbf{Bool}} \quad \frac{\tau \notin \{\mathbf{Int}, \mathbf{Char}, \mathbf{Float}, \mathbf{Bool}\}}{\Theta; \Delta; \Gamma \vdash_e \mathbf{null} : \tau}
$$

Variáveis possuem o tipo que é atribuído a elas pelo contexto de tipos Γ.

$$\frac{\Gamma(x) = \tau}{\Theta; \Delta; \Gamma \vdash_e x : \tau}$$

A verificação de acesso a campos em valores de registros é verificada da seguinte forma:

1. Primeiro, obtemos o tipo da variável x_1, $\Theta; \Delta; \Gamma \vdash_e x_1 : \tau_1$;
2. Em seguida, obtemos o conjunto de campos de τ_1, $\Delta(\tau_1) = S_1$;
3. Finalmente, obtemos o tipo de x_2 no conjunto de campos de τ_1, S_1.

$$\frac{\Theta; \Delta; \Gamma \vdash_e x_1 : \tau_1 \quad \Delta(\tau_1) = S_1 \quad (x_2, \tau) \in S_1}{\Theta; \Delta; \Gamma \vdash_e x_1.x_2 : \tau}$$

A verificação de operadores binários e unários são como se segue:

$$\frac{\Theta; \Delta; \Gamma \vdash_e e_1 : \tau_1 \quad \Theta; \Delta; \Gamma \vdash_e e_2 : \tau_2 \quad \Theta; \Delta; \Gamma(\circ) = [\tau_1, \tau_2] \to \tau}{\Theta; \Delta; \Gamma \vdash_e e_1 \circ e_2 : \tau}$$

$$\frac{\Theta; \Delta; \Gamma \vdash_e e_1 : \tau_1 \quad \Theta; \Delta; \Gamma(\surd) = \tau_1 \to \tau}{\Theta; \Delta; \Gamma \vdash_e \surd e_1 : \tau}$$

Uma expressão $e_1 \circ e_2$ possui o tipo τ se:

1. O operador $\circ$ possui o tipo $[\tau_1, \tau_2] \to \tau$;
2. a expressão e_1 possui tipo τ_1, isto é, $\Theta; \Delta; \Gamma \vdash_e e_1 : \tau_1$;
3. a expressão e_2 possui o tipo τ_2, isto é, $\Theta; \Delta; \Gamma \vdash_e e_2 : \tau_2$.

A verificação dos operadores unários é similar.

Evidentemente, o tipo dos operadores deve estar presente no contexto Γ. A tabela a seguir, descreve o tipo de cada um dos operadores disponível em *lang*. O contexto inicial formado pelas definições dessa tabela será chamado de Θ_0.

Além de tipos básicos, *lang* permite a definição de arranjos. A verificação de acesso a arranjos é feita conforme a seguinte regra:

Operador	Tipo
$+, -, *, /,$	$[a, a] \rightarrow a$, em que $a \in \{\mathbf{Int}, \mathbf{Float}\}$
%	$[\mathbf{Int},\mathbf{Int}] \rightarrow \mathbf{Int}$
$==, != , <$	$[a, a] \rightarrow \mathbf{Bool}$, em que $a \in \{\mathbf{Int}, \mathbf{Float}, \mathbf{Char}\}$
&&	$[\mathbf{Bool},\mathbf{Bool}] \rightarrow \mathbf{Bool}$
!	$\mathbf{Bool} \rightarrow \mathbf{Bool}$
-	$a \rightarrow a$, em que $a \in \{\mathbf{Int}, \mathbf{Float}\}$
print	$a \rightarrow \mathbf{Void}$, em que $a \in \{\mathbf{Int}, \mathbf{Char}, \mathbf{Bool}, \mathbf{Float}\}$

$$\frac{\Theta; \Delta; \Gamma \vdash_e e_1 : \tau\,[\,] \quad \Theta; \Delta; \Gamma \vdash_e e_2 : \mathbf{Int}}{\Theta; \Delta; \Gamma \vdash_e e_1[e_2] : \tau}$$

A regra de verificação de chamadas de funções é feita da seguinte forma:

1. Primeiro obtemos o tipo da função utilizando o contexto Θ e seu identificador: $\Theta(v) = [x_1 : \tau_1, ..., x_m : \tau_m] \rightarrow \tau$;
2. Na sequência verificamos os tipos de cada argumento da chamada de função com o seu respectivo tipo: $\Theta; \Delta; \Gamma \vdash_e e_i : \tau_i$, em que m é o número de argumentos da função, $1 \leq i \leq m$.
3. Verificar que os indicadores de acesso a valores de retorno estão dentro do limite do número de valores de retorno, p.

$$\frac{\Theta(v) = [x_1 : \tau_1, ..., x_m : \tau_m] \rightarrow [\tau_0, ..., \tau_p] \quad \Theta; \Delta; \Gamma \vdash_e e_i : \tau_i \quad 0 \leq n \leq p \quad 1 \leq i \leq m}{\Theta; \Delta; \Gamma \vdash_e v(e_1, ..., e_m)[n] : \tau_n}$$

A expressão para alocação dinâmica pode ser utilizada sobre tipos de arranjos ou tipos de registros. A regra seguinte mostra como verificar a alocação de arranjos, em que $\mathbf{dom}(\Delta)$ representa o domínio do contexto Δ, isto é $\mathbf{dom}(\Delta) = \{x \mid \exists S.\Delta(x) = S\}$.

$$\frac{\Theta; \Delta; \Gamma \vdash_e e : \mathbf{Int} \quad \tau \notin \{\mathbf{Int}, \mathbf{Float}, \mathbf{Char}, \mathbf{Bool}\} \cup \mathbf{dom}(\Delta)}{\Theta; \Delta; \Gamma \vdash_e \mathbf{new}\ \tau\ e : \tau}$$

A regra funciona da seguinte forma:

1. A expressão e deve ter tipo **Int**, para definição de limites de arranjos.
2. O tipo τ deve ser um tipo de arranjo, isto é, não deve ser um tipo primitivo nem um tipo definido pelo usuário.

A próxima regra mostra como verificar a alocação de tipos definidos pelo usuário.

$$\frac{\tau \in \mathbf{dom}(\Delta)}{\Theta; \Delta; \Gamma \vdash_e \mathbf{new}\ \tau\ : \tau}$$

Note que a única restrição é que o tipo a ser alocado faz parte do domínio do ambiente Δ.

3.1.3 Semântica Estática de Comandos

A semântica estática de comandos é dada por um julgamento $\Theta; \Delta; \Gamma; V \vdash_c\ c \rightsquigarrow V'; \Gamma'$ que denota que o comando c é bem formado nos contextos Θ, Δ e Γ. Note que o comando c pode modificar o contexto Γ por incluir uma nova variável. Por isso, este julgamento produz, como resultado, um novo contexto Γ' e conjuntos de variáveis incluídas V e V'. Usaremos este conjunto para controlar o escopo de visibilidade de identificadores. Adicionalmente, o julgamento $\Theta; \Delta; \Gamma; V \vdash_{cs}\ c \rightsquigarrow V' \times \Gamma'$ lida com a verificação de blocos.

A primeira regra para blocos lida com a marcação de final de um bloco, $\bullet$. Esta regra simplesmente remove do contexto de tipos de resultado, Γ', todos os identificadores que foram introduzidos no bloco atual e que estão armazenados no conjunto de variáveis V.

$$\frac{\Gamma' = \{(x, \tau) \mid x \notin V \wedge \Gamma(x) = \tau\}}{\Theta; \Delta; \Gamma; V \vdash_{cs} \bullet \rightsquigarrow \emptyset; \Gamma'}$$

A próxima regra lida com blocos não vazios e basicamente é responsável por processar o primeiro comando do bloco e repassar os resultados deste processamento para o restante dos comandos deste bloco.

$$\frac{\Theta; \Delta; \Gamma; V \vdash_c c \rightsquigarrow V_1; \Gamma_1 \quad \Theta; \Delta; \Gamma_1; V_1 \vdash_{cs} cs \rightsquigarrow V'; \Gamma'}{\Theta; \Delta; \Gamma; V \vdash_{cs} c\ cs \rightsquigarrow V'; \Gamma'}$$

A primeira regra para comandos lida com a atribuição. Essa regra é formada pelos seguintes passos:

1. Verificamos o tipo do lado esquerdo da atribuição;
2. Verificamos o tipo do lado direito da atribuição;

$$\frac{\Theta; \Delta; \Gamma \vdash_e lv : \tau \quad \Theta; \Delta; \Gamma \vdash_e e : \tau}{\Theta; \Delta; \Gamma; V \vdash_c lv\ =\ e : V; \Gamma}$$

É importante notar que esta regra não modifica o conjunto de variáveis introduzidas e nem o contexto de tipos, pois não há inclusão de novas variáveis no contexto.

A verificação de declaração de variáveis locais é feita de acordo com os seguintes passos:

1. Iniciamos verificando que a expressão de inicialização da variável v possui o mesmo tipo τ de sua declaração.
2. A regra retorna o contexto modificado contendo o tipo da nova variável e a inclui no conjunto de variáveis definidas no bloco atual.

$$\frac{\Theta; \Delta; \Gamma \vdash_e e : \tau \quad V' = \{v\} \cup V \quad \Gamma' = \Gamma, v : \tau}{\Theta; \Delta; \Gamma; V \vdash_c \tau \ v \ = \ e \rightsquigarrow V'; \Gamma'}$$

Na sequência, apresentamos a regra para validação de comandos condicionais. Primeiro, verificamos que o tipo da expressão e deve ser **Bool** e, na sequência, validamos os blocos de comandos cs_1 e cs_2 (caso exista) ignorando as eventuais declarações introduzidas nesses blocos.

$$\frac{\Theta; \Delta; \Gamma \vdash_e e : \textbf{Bool} \quad \Theta; \Delta; \Gamma; V \vdash_c c_1 \rightsquigarrow V_1; \Gamma_1 \quad \Theta; \Delta; \Gamma; V \vdash_c c_2 \rightsquigarrow V_2; \Gamma_2}{\Theta; \Delta; \Gamma; V \vdash_c \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \rightsquigarrow V; \Gamma}$$

$$\frac{\Theta; \Delta; \Gamma \vdash_e e : \textbf{Bool} \quad \Theta; \Delta; \Gamma; V \vdash_c c_1 \rightsquigarrow V_1; \Gamma_1}{\Theta; \Delta; \Gamma; V \vdash_c \textbf{if } e \textbf{ then } c_1 \rightsquigarrow V; \Gamma}$$

A seguir, apresentamos as regras para verificar comandos de repetição. Na primeira forma, iniciamos a verificação por demandar que a expressão e possua tipo **Int** ou arranjo e, na sequência, verificamos o bloco de comandos do comando **iterate**, descartando modificações no contexto de tipos. Na segunda forma, quando e tem tipo **Int**, verificamos que v não está no contexto ou está com tipo **Int**. Caso e seja um vetor, a regra é similar, porém v deve ter o mesmo tipo dos elementos do vetor.

$$\frac{\Theta; \Delta; \Gamma \vdash_e e : \textbf{Int} \quad \Theta; \Delta; \Gamma; V \vdash_c c \rightsquigarrow V'; \Gamma'}{\Theta; \Delta; \Gamma; V \vdash_c \textbf{iterate } e \, c \rightsquigarrow V; \Gamma} \quad \frac{\Theta; \Delta; \Gamma \vdash_e e : \tau\,[\,] \quad \Theta; \Delta; \Gamma; V \vdash_c c \rightsquigarrow V'; \Gamma'}{\Theta; \Delta; \Gamma; V \vdash_c \textbf{iterate } e \, c \rightsquigarrow V; \Gamma}$$

$$\frac{\Theta; \Delta; \Gamma \vdash_e e : \textbf{Int} \quad \Gamma(v) = \textbf{Int} \vee \nexists \tau.(v, \tau) \in \Gamma \quad \Theta; \Delta; \Gamma, v : \textbf{Int}; V \vdash_c c \rightsquigarrow V'; \Gamma'}{\Theta; \Delta; \Gamma; V \vdash_c \textbf{iterate } v \, e \, c \rightsquigarrow V; \Gamma}$$

$$\frac{\Theta; \Delta; \Gamma \vdash_e e : \tau\,[\,] \quad \Gamma(v) = \tau \vee \nexists \tau_1.(v, \tau_1) \in \Gamma \quad \Theta; \Delta; \Gamma, v : \tau; V \vdash_c c \rightsquigarrow V'; \Gamma'}{\Theta; \Delta; \Gamma; V \vdash_c \textbf{iterate } v \, e \, c \rightsquigarrow V; \Gamma}$$

A verificação de comandos **read** e **print** é bastante direta e realizada pelas regras a seguir.

$$\frac{\Theta;\Delta;\Gamma \vdash_e lv : \tau \quad \tau \in \{\mathbf{Int}, \mathbf{Float}, \mathbf{Char}, \mathbf{Float}\}}{\Theta;\Delta;\Gamma;V \vdash_c \mathbf{read}\ lv \rightsquigarrow V;\Gamma} \qquad \frac{\Theta;\Delta;\Gamma \vdash_e e : \tau \quad \tau \in \{\mathbf{Int}, \mathbf{Float}, \mathbf{Char}, \mathbf{Float}\}}{\Theta;\Delta;\Gamma;V \vdash_c \mathbf{print}\ e \rightsquigarrow V;\Gamma}$$

Para verificarmos o comando **return**, precisamos determinar se a lista de valores retornados possui o mesmo tipo que o retorno anotado no cabeçalho da função. Denotamos por $\Theta_\tau = [\tau_1, ..., \tau_m]$ a tupla de tipos do retorno da função. A regra seguinte mostra como validar o comando **return**.

$$\frac{\Theta_\tau = [\tau_1, ..., \tau_m] \quad \Theta;\Delta;\Gamma \vdash_e e_i : \tau_i \quad 1 \leq i \leq m}{\Theta;\Delta;\Gamma;V \vdash_c \mathbf{return}\ e_1 ... e_m \rightsquigarrow V;\Gamma}$$

Chamadas de funções podem ser feitas a nível de comandos. A verificação de chamadas de função se dá pelos seguintes passos:

1. Primeiro obtemos o tipo da função v, $\Theta(v) = [x_1 : \tau_1, ..., x_m : \tau_m] \rightarrow [\tau_1, ..., \tau_p]$.
2. Na sequência, verificamos que os argumentos possuem o tipo exigido pela definição de função, $\Theta;\Delta;\Gamma \vdash_e e_i : \tau_i$
3. Finalmente, verificamos que as expressões passadas para os componentes da tupla de retorno devem possui o mesmo tipo dos retornos da função.

$$\frac{\Theta(v) = [x_1 : \tau_1, ..., x_m : \tau_m] \rightarrow [\tau_1, ..., \tau_p] \quad \Theta;\Delta;\Gamma \vdash_e e_i : \tau_i \quad \Theta;\Delta;\Gamma \vdash_e lv_j : \tau_j \quad 1 \leq j \leq p \quad 1 \leq i \leq m}{\Theta;\Delta;\Gamma;V \vdash_c v(e_1, ..., e_m)\ lv_1 ... lv_p \rightsquigarrow V;\Gamma}$$

3.1.4 Semântica de Declarações

A semântica estática de declarações é dada por um julgamento $\Theta;\Delta \vdash_d prog \rightsquigarrow \Theta';\Delta'$ que denota que a sequência de declarações *prog* produz os contextos Θ', Δ' a partir de contextos Θ e Δ.

A primeira regra, mostra que um conjunto vazio de declarações não modifica os contextos.

$$\frac{}{\Theta;\Delta \vdash_d \bullet \rightsquigarrow \Theta;\Delta}$$

A próxima regra mostra como verificar conjuntos de declarações. A cada passo, verificamos cada uma das declarações produzindo novos contextos.

$$\frac{\Theta; \Delta \vdash_d def \rightsquigarrow \Theta_1; \Delta_1 \quad \Theta_1; \Delta_1 \vdash_d prog \rightsquigarrow \Theta'; \Delta'}{\Theta; \Delta \vdash_d def \;\; prog \rightsquigarrow \Theta'; \Delta'}$$

A validação de definição de novos tipos de dados se dá pelos seguintes passos:

1. Verificar que não há nomes de campos repetidos no tipo de dados definido;
2. Verificar que não há tipo com o mesmo nome definido previamente;
3. Verificar as funções dos tipos abstratos;

$$\frac{v \notin \mathbf{dom}(\Delta) \quad \exists! x. \exists \tau. x :: \tau \in fds}{\Theta; \Delta \vdash_d v \; fds \rightsquigarrow \Theta; \{(v, fds)\} \cup \Delta}$$

$$\frac{v \notin \mathbf{dom}(\Delta) \quad \exists! x. \exists \tau. x :: \tau \in fds \quad \Theta; \{(v, fds)\} \cup \Delta \vdash_d funs \rightsquigarrow \Theta'; \Delta'}{\Theta; \Delta \vdash_d v \; fds \; funs \rightsquigarrow \Theta'; \{(v, \bullet)\} \cup \Delta}$$

Observe que para tipos abstratos, o contexto resultante contém o nome do tipo sem as informações dos campos. Apenas as funções internas são tipas num contexto incluindo informações do campos do tipo abstrato.

A próxima regra mostra como verificar conjuntos de funções. A cada passo, verificamos cada uma das funções, produzindo novos contextos.

$$\frac{\Theta; \Delta \vdash_d fun \rightsquigarrow \Theta_1; \Delta_1 \quad \Theta_1; \Delta_1 \vdash_d funs \rightsquigarrow \Theta'; \Delta'}{\Theta; \Delta \vdash_d fun \;\; funs \rightsquigarrow \Theta'; \Delta'}$$

A última regra da semântica mostra como uma função é verificada em um programa. A validação de uma declaração de função segue os seguintes passos:

1. Verifica-se se não há declaração de outra função de mesmo nome;
2. Inicializamos o contexto Γ com os parâmetros formais da função definida, junto com o tipo desta função, para permitir chamadas recursivas.
3. Inicializamos Θ_τ com o tipo de retorno desta função.
4. Verificamos o corpo da função e validamos se todos os caminhos de execução terminam com um comando **return** com o tipo apropriado.

$$\frac{\begin{array}{ccc} v \notin \mathbf{dom}(\Theta) & \Gamma = \{(x_i, \tau_i) \,|\, 1 \leq i \leq m\} & \Theta'; \Delta; \Gamma \vdash_{cs} cs \rightsquigarrow \Gamma'; V' \\ \textbf{Seja } \Theta_\tau = (\tau_1, ..., \tau_n) & \Theta' = \Theta, v : (\tau_1, ...\tau_m) \to (\tau_1, ..., \tau_n) & (\tau_1, ..., \tau_n) \vdash_{ret} cs \end{array}}{\Theta; \Delta \vdash_d v \ (x_1 \ \tau_1, ..., x_m \ \tau_m) \ (\tau_1, ..., \tau_n) \ cs \rightsquigarrow \Theta'; \Delta}$$

As regras $(\tau_1, ..., \tau_n) \vdash_{ret} \ cs$ validam que todos os caminhos do bloco cs terminam com um **return** apropriado. A primeira regra verifica que em um bloco contendo apenas um comando, esse deve ser um return.

$$\frac{\Theta; \Delta; \Gamma \vdash_e e_i : \tau_i \quad 1 \leq i \leq n}{(\tau_1, ..., \tau_n) \vdash_{ret} \ \textbf{return} \, e_1, ..., e_n \ \bullet}$$

Em seguida, caso o último comando de um bloco seja um **if**, devemos garantir que ambos os blocos do if devem possuir um return.

$$\frac{(\tau_1, ..., \tau_n) \vdash_{ret} \ cs_1 \quad (\tau_1, ..., \tau_n) \vdash_{ret} \ cs_2}{(\tau_1, ..., \tau_n) \vdash_{ret} \ cs \, \textbf{if} \, e \, \textbf{then} \, cs_1 \, \textbf{then} \, cs_2}$$

Na situação do último comando de um bloco ser um **iterate**, devemos garantir que o último comando de seu bloco seja um **return**.

$$\frac{(\tau_1, ..., \tau_n) \vdash_{ret} \ cs}{(\tau_1, ..., \tau_n) \vdash_{ret} \ cs \, \textbf{iterate} \, e \, cs}$$

Caso o bloco seja formado por uma sequência de dois ou mais comandos, devemos ignorar o primeiro comando e verificar a cauda do bloco.

$$\frac{(\tau_1, ..., \tau_n) \vdash_{ret} \ cs}{(\tau_1, ..., \tau_n) \vdash_{ret} c \ cs}$$

3.1.5 Semântica de Programas Completos

Programas *lang* são formados por uma ou mais declarações. Dizemos que um programa Lang é bem formado se esse possui pelo menos um procedimento de nome **main** sem argumento. A regra a seguir faz essa validação, inicializando o contexto Θ com as definições iniciais presentes em Θ_0:

$$\frac{\Theta_0; \bullet \vdash_d prog \rightsquigarrow \Theta; \Delta \quad \Theta(\textbf{main}) = [] \to []}{\vdash_{wf} \ prog}$$

Com isso, finalizamos a especificação da semântica estática de *lang*. Na próxima seção apresentaremos a sua semântica dinâmica.

3.2 Semântica dinâmica

Assim como na semântica estática, a semântica dinâmica de *lang* também será dividida em diferentes níveis: expressões, comandos e funções.

3.2.1 Semântica de Expressões

Antes de apresentarmos a semântica de expressões, devemos introduzir o conceito de valor. De maneira simples, um **valor** representa o resultado final da computação de uma expressão. Valores são definidos pela seguinte gramática livre de contexto:

$$val \quad \rightarrow \quad n \,|\, c \,|\, f \,|\, \textbf{true} \,|\, \textbf{false} \,|\, \textbf{null} \,|\, A \,|\, D$$

Os não terminais A e D representam valores de arranjos e tipos de dados definidos pelo usuário (registros). Usando o conceito de valor, especificaremos a semântica de expressões como um conjunto de regras da forma:

$$\Delta; \theta; \sigma; e \Rightarrow_e val$$

em que θ é o **ambiente de funções**, que consiste de uma associação entre nomes de funções e sua respectiva definição. Por sua vez, σ é o **ambiente de execução**, que consiste de uma associação entre nomes de variáveis e seu respectivo valor, e é a expressão a ser executada e val o valor retornado. Observe que a semântica utiliza o ambiente Δ para permitir a correta criação de valores de registros. De maneira simples, o ambiente σ consiste de uma abstração da memória utilizada por um programa. A notação $\sigma(x) = val$ denota que o valor val está associado ao nome x em σ. Por sua vez, representamos por $\sigma[x \mapsto val]$ o ambiente σ' tal que $\sigma'(i) = \sigma(i)$, para todo $i \neq x$ e $\sigma'(x) = val$.

As primeiras regras lidam com a avaliação de valores e de variáveis.

$$\frac{}{\Delta; \theta; \sigma; val \Rightarrow_e val} \qquad \frac{}{\Delta; \theta; \sigma; x \Rightarrow_e \sigma(x)}$$

A execução de valores é imediata e a de variáveis apenas obtém o valor associado a ela no ambiente σ. A execução de acesso a elementos de arranjos e a campos de registros são definidas pelas regras a seguir.

$$\frac{\Delta;\theta;\sigma;e_1 \Rightarrow_e A_1 \quad \Delta;\theta;\sigma;e_2 \Rightarrow_e n_1 \quad A_1[n] = val}{\Delta;\theta;\sigma;e_1[e_2] \Rightarrow_e val} \quad \frac{\Delta;\theta;\sigma;e_1 \Rightarrow_e D_1 \quad D_1(v) = val}{\Delta;\theta;\sigma;e_1.v \Rightarrow_e val}$$

Para o acesso a arranjos, primeiro avaliamos a expressão e_1 que retorna um valor de arranjo, A_1. Em seguida, avaliamos a expressão do índice da posição acessada, e_2, resultando uma constante n_1. Finalmente, o valor val é obtido pelo valor presente na posição n_1 do arranjo A_1. A avaliação do acesso a campo de registros é similar. Primeiro, executamos a expressão e_1 e obtemos um valor D_1 de registro. Em seguida, obtemos o valor val usando como chave o nome do campo acessado, v. Observe que registros são representados como tabelas formadas por pares chave/valor.

Em seguida, temos as regras para execução de expressões com operadores binários.

$$\frac{\Delta;\theta;\sigma;e_1 \Rightarrow_e val_1 \quad \Delta;\theta;\sigma;e_2 \Rightarrow_e val_2}{\Delta;\theta;\sigma;e_1 \circ e_2 \Rightarrow_e val_1 \oplus val_2}$$

O operador $\oplus$ denota uma função correspondente a $\circ$ sobre valores. A execução de operadores unários é similar, em que $\angle$ é o operador equivalante a $\surd$ que opera sobre valores.

$$\frac{\Delta;\theta;\sigma;e_1 \Rightarrow_e val_1}{\Delta;\theta;\sigma;\surd\, e_1 \Rightarrow_e \angle val_1}$$

Para realizarmos a execução de funções, devemos definir como realizar a execução de sua lista de argumentos. Faremos isso definindo regras para avaliar sequências de expressões.

$$\frac{}{\Delta;\theta;\sigma;\bullet \Rightarrow_{es} \bullet} \quad \frac{\Delta;\theta;\sigma;e \Rightarrow_e val \quad \Delta;\theta;\sigma;es \Rightarrow_{es} vals}{\Delta;\theta;\sigma;e\ es \Rightarrow_{es} val\ vals}$$

A primeira das regras anteriores mostra que uma sequência vazia de expressões reduz para uma sequência vazia de valores. A segunda regra mostra que sequências não vazias de expressões são reduzidas executando-se o primeiro elemento da lista e, em seguida, executamos a lista contendo o restante das expressões da lista original.

Utilizando as regras para avaliação de listas de expressões, podemos definir a execução de funções como a seguir:

1. Primeiro, executamos os argumentos es da chamada retornando os valores $vals_1$.
2. Na sequência, executamos o argumento da posição de retorno, e, resultando no valor val_2.
3. Obtemos a definição da função v no ambiente θ.
4. Em seguida, criamos o ambiente σ' que estende σ com os valores obtidos pela avaliação dos argumentos.

5. O próximo passo envolve executar o corpo da função v, cs, usando o ambiente σ' retornando os valores como um arranjo A.

6. Finalmente, o valor é obtido a partir da posição val_2 no arranjo de valores de retorno, A.

$$\frac{\begin{array}{cc}\Delta;\theta;\sigma;es_1 \Rightarrow_{es} vals_1 & \Delta;\theta;\sigma;e \Rightarrow_e val_2 \\ \theta(v) = v(x :: \tau)\ ts\ cs & \sigma' = \sigma \cup \{(x_i, vals_{1i}) \mid 1 \leq i \leq |vals_1|\} \\ \Delta;\theta;\sigma';cs \Rightarrow_{cs} \sigma'';A & A[val_2] = val\end{array}}{\Theta;\theta;\sigma;v(es)\ [e] \Rightarrow_e val}$$

A próxima regra mostra como avaliar a alocação de arranjos. Primeiro, avaliamos a expressão que forma as dimensão do arranjo e a utilizamos para inicializar o valor de um arranjo usando um conjunto de regras auxiliares.

$$\frac{\Delta;\theta;\sigma;e \Rightarrow_e\ val \quad \tau;val \rightsquigarrow_A A}{\Delta;\theta;\sigma;\mathbf{new}\ \tau[]\ e \Rightarrow_e A}$$

Todos os arranjos em *lang* são unidimensionais. Para criar arranjos multidimensionais, deve-se arranjos cujos elementos são arranjos. Utilizando este fato, as regras paraa regra para criação de arranjos é apresentada a seguir.

$$\frac{}{\tau;val \rightsquigarrow_A \{\varphi(\tau)_1, ..., \varphi(\tau)_{val}\}}$$

A notação $\varphi(\tau)$ denota o valor **padrão** para o tipo τ.

Alocação de registros é feita pela regra a seguir que inicializa cada um dos seus campos com o valor padrão para o tipo do campo.

$$\frac{\Delta(\rho) = \{v_1 : \tau_1, ..., v_n : \tau_n\} \quad D = [v_1 : \varphi(\tau_1), ..., v_n : \varphi(\tau_n)]}{\Delta;\theta;\sigma;\mathbf{new}\ \rho \Rightarrow_e D}$$

Com isso terminamos a semântica dinâmica de expressões para *lang*.

3.2.2 Semântica de Comandos

Ao contrário da semântica de expressões que produz um valor como resultado, a execução de comandos apenas modifica o ambiente de execução σ. O comando **return** pode retornar um arranjo de resultados. Para evitar poluir a notação, vamos utilizar o arranjo de resultados apenas para o comando **return**.

Primeiro mostraremos as regras que lidam com a execução de sequências de comandos. A execução de uma sequência não vazia de comandos se dá pela execução do primeiro comando, c, que produz um novo ambiente σ_1 que é utilizado como entrada para a execução do restante do bloco de comandos que produz o ambiente final, σ'. Finalmente, observe que um bloco vazio produz como resultado um ambiente não modificado.

$$\frac{\Delta;\theta;\sigma;c \Rightarrow_c \sigma_1 \quad \Delta;\theta;\sigma_1;cs \Rightarrow_c \sigma'}{\Delta;\theta;\sigma;c\ cs \Rightarrow_c \sigma'} \qquad \frac{}{\Delta;\theta;\sigma;\bullet \Rightarrow_c \sigma}$$

A semântica de atribuições $v = e$ se dá pela avaliação da expressão e e a respectiva atualização do valor de v no ambiente σ. Para declarações de variáveis e sua respectiva inicialização, o processo é similar e também apresentado a seguir.

$$\frac{\Delta;\theta;\sigma;e \Rightarrow_e val}{\Delta;\theta;\sigma;\ v\ =\ e\ \Rightarrow_c \sigma[v \mapsto val]} \qquad \frac{\Delta;\theta;\sigma;e \Rightarrow_e val}{\Delta;\theta;\sigma;\ \tau\ v\ =\ e\ \Rightarrow_c \sigma[v \mapsto val]}$$

A avaliação de comandos condicionais se dá por executar a expressão e e, em seguida, executa-se o bloco correspondente ao **then** ou **else** (caso exista), dependendo do resultado da avaliação.

$$\frac{\Delta;\theta;\sigma;\ e \Rightarrow_e \mathbf{true} \quad \Delta;\theta;\sigma;\ c_1 \Rightarrow_c \sigma'}{\Delta;\theta;\sigma;\ \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \Rightarrow_c \sigma'} \qquad \frac{\Delta;\theta;\sigma;\ e \Rightarrow_e \mathbf{false} \quad \Delta;\theta;\sigma;\ c_2 \Rightarrow_c \sigma'}{\Delta;\theta;\sigma;\ \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \Rightarrow_c \sigma'}$$

$$\frac{\Delta;\theta;\sigma;\ e \Rightarrow_e \mathbf{true} \quad \Delta;\theta;\sigma;\ c_1 \Rightarrow_c \sigma'}{\Delta;\theta;\sigma;\ \mathbf{if}\ e\ \mathbf{then}\ c_1\ \Rightarrow_c \sigma'} \qquad \frac{\Delta;\theta;\sigma;\ e \Rightarrow_e \mathbf{false}}{\Delta;\theta;\sigma;\ \mathbf{if}\ e\ \mathbf{then}\ c_1\ \Rightarrow_c \sigma}$$

Comandos **iterate** são executados da seguinte forma:

1. Primeiro executamos a expressão e de forma a obter um valor inteiro n ou um arranjo de tamanho n.
2. Em seguida, realizamos n iterações do comando c.

$$\frac{\Delta;\theta;\sigma;e \Rightarrow_e n \quad \Delta;\theta;\sigma;c \Rightarrow_n \sigma'}{\Delta;\theta;\sigma;\ \mathbf{iterate}\ e\ c \Rightarrow_c \sigma'} \qquad \frac{\Delta;\theta;\sigma;e \Rightarrow_e A \quad |A| = n \quad \Delta;\theta;\sigma;c \Rightarrow_n \sigma'}{\Delta;\theta;\sigma;\ \mathbf{iterate}\ e\ c \Rightarrow_c \sigma'}$$

A iteração do comando é realizada pelas seguintes regras, cujo significado é imediato.

$$\frac{\Delta;\theta;\sigma;c \Rightarrow_c \sigma_1 \quad \Delta;\theta;\sigma_1;c \Rightarrow_n \sigma'}{\Delta;\theta;\sigma;c \Rightarrow_{n+1} \sigma'} \qquad \frac{}{\Delta;\theta;\sigma;c \Rightarrow_0 \sigma}$$

A execução do comando **iterate** contendo uma variável é similar:

1. Primeiro executamos a expressão e de forma a obter um valor inteiro n ou um arranjo A;
2. Caso e avalie para um inteiro, cria-se um arranjo A contendo os valores de n a 1, em ordem decrescente;
3. Em seguida, realizamos uma quantidade de iterações do comando c equivalente ao tamanho de A com a variável inicializada com o valor do próximo elemento do arranjo a cada iteração.

$$\frac{\Delta;\theta;\sigma;e \Rightarrow_e n \quad A = \{n, n-1, ..., 1\} \quad v;A;\Delta;\theta;\sigma;c \Rightarrow_n \sigma'}{\Delta;\theta;\sigma;\ \textbf{iterate}\ v\ e\ c \Rightarrow_c \sigma'}$$

$$\frac{\Delta;\theta;\sigma;e \Rightarrow_e A \quad v;A;\Delta;\theta;\sigma;c \Rightarrow_n \sigma'}{\Delta;\theta;\sigma;\ \textbf{iterate}\ v\ e\ c \Rightarrow_c \sigma'}$$

$$\frac{v;A;\Delta;\theta;\sigma[v \mapsto A[|A| - n + 1];c \Rightarrow_c \sigma_1 \quad v;A;\Delta;\theta;\sigma_1;c \Rightarrow_n \sigma'}{v;A;\Delta;\theta;\sigma;c \Rightarrow_{n+1} \sigma'} \quad \frac{}{v;A;\Delta;\theta;\sigma;c \Rightarrow_0 \sigma}$$

Para a semântica de comandos que realizam entrada e saída, vamos considerar funções **primitivas** para realizar a leitura e escrita utilizando a entrada / saída padrão. A função `print` é a função primitiva que realiza a impressão de valores no console e `read` realiza a leitura de valores.

$$\frac{\Delta;\theta;\sigma;e \Rightarrow_e val \quad \texttt{print}\ val}{\Delta;\theta;\sigma;\ \textbf{print}\ e \Rightarrow_c \sigma} \quad \frac{val = \texttt{read} \quad \Delta;\theta;\sigma;\ e \Rightarrow_e lv \quad \Delta;\theta;\sigma;lv = val \Rightarrow_c \sigma'}{\Delta;\theta;\sigma;\ \textbf{read}\ e \Rightarrow_c \sigma'}$$

A semântica do comando **return** avalia as expressões e as retorna em forma de um arranjo.

$$\frac{\Delta;\theta;\sigma;es \Rightarrow_{es} val_1...val_n\bullet \quad A = \{val_1, ..., val_n\}}{\Delta;\theta;\sigma;\ \textbf{return}\ es \Rightarrow_c \sigma;A}$$

Finalmente, chamadas de funções permitem utilizar acesso a campos ou variáveis para receber o valor produzido pela chamada da função em questão. A única diferença de chamadas de funções a nível de expressões é que enquanto comandos, funções podem realizar atribuições para receber o valor retornado.

$$\frac{\begin{array}{cc}\Delta;\theta;\sigma;es_1 \Rightarrow_{es} vals_1 & \Delta;\theta;\sigma;es_2 \Rightarrow_{es} lv_1...lv_n \\ \theta(v) = v(x :: \tau)\ ts\ cs & \sigma_1 = \sigma \cup \{(x_i, vals_{1i}) \mid 1 \leq i \leq |vals_1|\} \\ \Delta;\theta;\sigma_1;cs \Rightarrow_{cs} \sigma'';\{val_1, ..., val_n\} & cs_2 = \{lv_i = val_i \mid 1 \leq i \leq n\} \\ \multicolumn{2}{c}{\Delta;\theta;\sigma'';cs_2 \Rightarrow_{cs} \sigma'}\end{array}}{\Theta;\theta;\sigma;v(es_1)\ [es_2] \Rightarrow_c \sigma'}$$

3.2.3 Semântica de Programas

A execução de programas completos se dá pela avalição de sua função **main**.

Com isso, terminamos a especificação da linguagem *lang*.