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

Gramática 1: Sintaxe da linguagem *lang*

# A Linguagem Lang

Neste documento é apresentada a especificação da linguagem de programação que será usada na implementação dos trabalhos da disciplina CSI506-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 descrito na seção 2, sintático descrito na seção 2, 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.

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 são detalhadas nas subseções subsequentes.

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

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

divmod(num :: Int, div :: Int) : Int, Int {
  q = num / div;
  r = num % div;
  return q, r;
}
```

Figura 1: Exemplos de funções e procedimentos na linguagem *lang*.

## 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:

```
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.

## 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. Após os parâmetros, segue 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 bloco de comandos. Um exemplo de programa na linguagem *lang* é apresentado na Figura 1, 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. Esta função recebe dois parâmetros inteiros e retorna o quociente e o resto da divisão do primeiro pelo segundo parâmetro.

## 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 no qual 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 com a seguinte estrutura:

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

O comando **iterate** especifica um trecho de código que será executado por uma quantidade de vezes determinada pela avaliação da expressão delimitada entre parêntesis. Ressalta-se que 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.

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* da Figure 1 fica:

```
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>;
```

Este 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 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:

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

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

```
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, 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*. 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 $y$.

## 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;
- 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.

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

# 3 Semântica "Informal" de Lang

A semântica de um programa *lang*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: $M/\{I_1, \ldots, I_n\}$.

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 $\alpha$ 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, floats, booleanos, a constante null, vetores ou registros.

O conjunto de valores $\mathcal{L}$ da linguagem *lang*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. $\alpha$: 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 possui 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 do estado, podemos descrever a semântica de cada comando da linguagem *lang*. 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 por 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 operador binário qualquer, $\langle M, v_1 : P, =\rangle S(e_1, \langle H, M, P\rangle)$, $\langle M, v_2 : v_1 : P, =\rangle S(e_2, \langle H, M, v_1 : P\rangle)$ e a função $f_{op}$ mapeia cada operador em sua semântica correspondente na álgebra. Exemplo $f_{op}(+, 1, 2) = 3$. O operador $f_{op}$ está definido para os operadores aritméticos, booleanos e relacionais assim como para os 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 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 $(ID, V) \in R$.
- $S(ID, \langle H, M, P\rangle) = \langle H, M, M(ID) : P\rangle$.
- $S(ID(e_1, \ldots, e_n)[e], \langle H, M, P\rangle) = \langle H, M, P\rangle$ onde
  - $\langle H, M, V_1 : ... : V_n : P\rangle = S(e_1, S(e_2, ..., S(e_n, \langle H, M, P\rangle)))$.
  - Construímos uma nova memória $M'[ID_1 = V_1, ...., ID_n = V_n]$, onde $ID_1, ..., ID_n$ são os identificadores dos parâmetros da função chamada. O corpo da função deverá então ser executado no estado $\langle H, M', P\rangle$ .
  - Após a execução da função, o estado resultante, supondo que a função retorne $m$ valores, será $\langle H, M, k : V_1 : ... : V_m : P\rangle$ onde $0 \leq k \leq m$.
  - Finalmente, salvamos apenas o $k$-ésimo valor do retorno na pilha, removendo os demais valores da pilha: $\langle H, M, V_k : P\rangle$.
- $S(new\ \ TYPE\ \ n, \langle H, M, P\rangle) = \langle H[\alpha = V], M, \alpha : P\rangle$, onde o valor $V$ é obtido por uma instanciação do tipo $TYPE$ e o valor $n$ pode ser omitido caso $TYPE$ seja um nome de registro. Note que a operação new insere, na memória global, o valor associado a algum identificador $\alpha$ 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 o nome de tipo definido pelo usuário e o tamanho for omitido, então 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 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 preenchido com o valor $null$.

**Semântica de comandos:** A semântica de comandos será dada pela função $C : c \rightarrow \langle H, M, P\rangle \rightarrow \langle H', M', P'\rangle$ definida por 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, então 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 dada 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 $C(stmtBlock, \langle H', M, P\rangle)$. Senão $\langle H', M, P\rangle$.
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 $C(stmtBlock_1, \langle H', M, P\rangle)$. Senão, $C(stmtBlock_2, \langle H', M, P\rangle)$.
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 um nova memória que mapeia um novo identificador $I_c$, que não ocorre em lugar algum do programa[2], para o valor $V$.
   (a) Caso $M[I_c] = 0$ (o valor seja o inteiro zero), então o resultado é: $\langle H', M'/\{I_c\}, P\rangle$, concluindo a execução do iterate.
   (b) Caso $M[I_c] = i$ (o valor seja um inteiro maior que zero), 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 resultando é $\langle H'', M''', P'\rangle$. Considerando a memória deste novo estado, retomamos a execução a partir da regra 4a.
   (c) Caso $V = A$ (o valor seja um arranjo de tamanho $n$): Considere o tamanho de $A$ como um valor inteiro $V$ e prossiga de acordo com as regras 4a e 4b.
5. $C(iterate\ id : exp\ stmtBlock, \langle H, M, P\rangle)$: Neste caso, devemos considerar $id$ como uma variável de acesso ao contador. Também devemos considerar que a variável $id$, se declarada no comando iterate, é local ao bloco do comando $iterate$, ou seja, a variável não deve ser acessível fora do bloco do comando. Além disso, caso a variável já tenha sido previamente delcarada 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 novo identificador $fresh\ I_c$.

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

(a) Caso $M[I_c] = 0$ (o valor seja o inteiro zero), então o resultado é: $\langle H', M'/\{I_c\}, P\rangle$, se $id$ foi declarada previamente ao comando iterate. O resultado será $\langle H', M'/\{I_c, id\}, P\rangle$ se $id$ foi declarada no comando iterate. Este passo conclui a execução do iterate.

(b) Caso $M[I_c] = i$ (o valor seja um inteiro maior que zero), 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 resultando é $\langle H'', M''', P'\rangle$. Considerando a memória deste novo estado retomamos a execução a partir da regra 5a.

(c) Caso $V = A$ (o valor seja um arranjo de tamanho $n$): Neste caso, devemos considerar $M' = M[I_c = n, id = A[n - M[I_c]]$

i. Caso $M[I_c] = 0$ (o valor seja o inteiro zero), então o resultado é: $\langle H', M'/\{I_c\}, P\rangle$, se $id$ foi declarada previamente ao comando iterate, e $\langle H', M'/\{I_c, id\}, P\rangle$ se $id$ foi declarada no comando iterate. Este passo conclui a execução do iterate.

ii. Caso $M[I_c] = i$ (o valor seja um inteiro maior que zero), 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 resultando é $\langle H'', M''', P'\rangle$. Considerando a memória deste novo estado, retomamos a execução a partir da regra 5(c)i.

6. $C(print\ exp, \langle H, M, P\rangle)$: Seja $\langle H', M, V : P\rangle = S(exp, \langle H, M, P\rangle)$. O valor $V$ deve ser impresso na saída padrão (que geralmente é a tela). O estado resultante é $\langle H', M, V : P\rangle$.

7. $C(read\ lvalue, \langle H, M, P\rangle)$: Realiza-se a leitura de um valor a partir da entrada padrão (geralmente o teclado), inserindo o valor na pilha e resultando no estado $\langle H, M, V : P\rangle$. Em seguida, $update(\langle H, M, P\rangle, V)$.

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

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

   O estado resultante é $\langle H_n, M, V_n \ldots V_1 : P\rangle$. 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.

9. A semântica da chamada de função ao nível de comando é análoga à chamada de função para expressões, exceto pelo fato de que os valores de retorno removidos da pilha devem ser associados aos lvalues anotados entre $<$ e $>$. Deve-se fazer com que as posições desses lvalues correspondam à mesma ordem dos retornos declarados na função.

10. A sementica de uma squência $C(\{c_1\, c_2 \ldots\, c_n\}, \langle H, M, P\rangle) = \langle H'', M'', P''\rangle$, onde $\langle H', M', P'\rangle = C(c_1, \langle H, M, P\rangle)$ e $\langle H'', M'', P''\rangle = C(\{c_2 \ldots\, c_n\}, \langle H', M', P'\rangle)$

Além da parte formal apresentada neste texto, considere também que programas corretos em *lang*sempre satisfazem as seguintes afirmações.

- Os operadores aritméticos são homogêneos em relação aos tipos. Ou seja, um operador aritmético $a \oplus b$ pode operar apenas sobre argumentos inteiros (ambos $a$ e $b$) ou sobre ambos argumentos de ponto flutuante, mas nunca sobre um argumento de ponto flutuante e um argumento inteiro.

- 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 sempre são realizadas observando-se o número correto de argumentos esperados na definição.
- A quantidade de expressões passadas ao comando Códigoreturn sempre corresponde ao número e aos tipos expressos na anotação de tipo da função.
- Toda variável deve ser delcarada antes de ser usada.
- Todo tipo de dados definido pelo usuário deve ser declarado antes de ser usado, sendo permitido o uso de tipos de dados recursivos e mutuamente recursivos.

# 4 Semântica estática de Lang

A semântica estática de Lang (ou sistema de tipos 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 3.1.1 descreve a sintaxe abstrata e os contextos utilizados para definir todas as regras semânticas. A seções 3.1.2, 3.1.3 e 3.1.4 descrevem regras para expressões, comandos e declarações, respectivamente.

## 4.1 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, $f$ constantes de ponto flutuante, $\circ$ denota operadores binários, $\surd$ denota operadores unários e $v$ denota um identificador.

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

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.

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

Comandos podem ser atribuições, declaração de variáveis, condicionais, repetição, leitura de valores, impressão no console, retorno de função e chamada de função.

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.

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

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

- $\Theta$: contexto contendo os tipos de funções definidas pelo usuário. Este 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.
- $\Delta$: contexto contendo informações sobre os campos de registros definidos pelo usuário. O contexto $\Delta$ é um conjunto de pares da forma $(x, S)$ em que $x$ é o nome do tipo e $S$ é um conjunto de formado por pares $(v, \tau)$ em que $v :: \tau$ é um campo do tipo $x$.
- $\Gamma$: contexto contendo definições de variáveis e operadores de *lang*. Este contexto é formado por pares $(x, \tau)$ em que $x$ é um identificador e $\tau$ o seu respectivo tipo. Neste 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 $\Gamma \cup \{(x, \tau)\}$. Estas operações aplicam-se também aos contextos $\Delta$ e $\Theta$. Finalmente, representamos um contexto vazio por $\bullet$.

## 4.2 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 $\tau$ sobre os contextos $\Theta$, $\Delta$ e $\Gamma$.

Iniciaremos a descrição com regras para constantes, que possuem regras para 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.

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

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

Variáveis possuem o tipo que é atribuído a elas pelo contexto de tipos $\Gamma$.

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

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 $\tau_1$, $\Delta(\tau_1) = S_1$;
3. Finalmente, obtemos o tipo de $x_2$ no conjunto de campos de $\tau_1$, $S_1$.

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

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 $\tau$ se:

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

3. a expressão $e_2$ possui o tipo $\tau_2$, isto é, $\Theta; \Delta; \Gamma \vdash_e e_2 : \tau_2$.

A verificação dos operadores unários é similar.

Evidentemente, que o tipo dos operadores deve estar presente no contexto $\Gamma$. A tabela a seguir, descreve o tipo de cada um dos operadores disponível em *lang*.

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

O contexto inicial formado pelas definições da tabela anterior será chamado de $\Theta_0$.

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

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

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

1. Primeiro obtemos o tipo da função utilizando o contexto $\Theta$ 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, $p$.

$$\frac{\Theta(v) = [x_1 : \tau_1, ..., x_m : \tau_m] \rightarrow [\tau_1, ..., \tau_p] \quad \Theta; \Delta; \Gamma \vdash_e e_i : \tau_i \quad 1 \leq n_j \leq p \quad 1 \leq i \leq m \quad 1 \leq j \leq p}{\Theta; \Delta; \Gamma \vdash_e v(e_1, ..., e_m)[n_1, ..., n_p] : \tau}$$

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 $\textbf{dom}(\Delta)$ representa o domínio do contexto $\Delta$, isto é $\textbf{dom}(\Delta) = \{x \mid \exists S. \Delta(x) = S\}$.

$$\frac{\forall e. e \in es \rightarrow \Theta; \Delta; \Gamma \vdash_e e : \textbf{Int} \quad \tau \notin \{\textbf{Int}, \textbf{Float}, \textbf{Char}, \textbf{Bool}\} \cup \textbf{dom}(\Delta)}{\Theta; \Delta; \Gamma \vdash_e \textbf{new}\ \tau\ es : \tau}$$

A regra funciona da seguinte forma:

1. Todas as expressões $e \in es$ devem ter tipo **Int**, para definição de limites de arranjos.
2. O tipo $\tau$ 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 $\Delta$.

### 4.3 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 $\Theta$, $\Delta$ e $\Gamma$. Note que o comando $c$ pode modificar o contexto $\Gamma$ por incluir uma nova variável. Por isso, este julgamento produz, como resultado, um novo contexto $\Gamma'$ 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, $\Gamma'$, todos os identificadores que foram introduzidos no bloco atual e que estão armazenados no conjunto de variáveis $V$.

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

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. Esta regra será dividia em dois casos e é formada pelos seguintes passos:

1. Verificamos o tipo do lado esquerdo da atribuição;
2. Caso 1: O lado esquerdo da atribuição seja uma **variável que não** possui tipo definido no contexto $\Gamma$.
   (a) Verificamos o tipo do lado direito da atribuição.

(b) A regra retorna o contexto modificado contendo o tipo da nova variável e a inclui no conjunto de variáveis definidas no bloco atual.

3. Caso 2: O lado esquerdo da atribuição seja um ***lvs*** que **possui tipo definido no contexto** $\Gamma$.
   (a) Verificamos o tipo do lado direito da atribuição.
   (b) Asseguramos que os tipos do lado esquerdo e direito são iguais.

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

Figura 2: Regra de atribuição para o caso 1

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

Figura 3: Regra de atribuição para o caso 2

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 $cs_2$ ignorando as eventuais declarações introduzidas nestes blocos.

$$\frac{\Theta;\Delta;\Gamma \vdash_e e : \textbf{Bool} \quad \Theta;\Delta;\Gamma;V \vdash_c cs_1 \rightsquigarrow V_1;\Gamma_1 \quad \Theta;\Delta;\Gamma;V \vdash_c cs_2 \rightsquigarrow V_2;\Gamma_2}{\Theta;\Delta;\Gamma;V \vdash_c \textbf{if } e \textbf{ then } cs_1 \textbf{ else } cs_2 \rightsquigarrow V;\Gamma}$$

A seguir, apresentamos a regra para verificar comandos de repetição. Iniciamos a verificação por demandar que a expressão $e$ possua tipo **Int** ou tipo arranjo $\tau[]$ e na sequência verificamos o bloco de comandos do comando **iterate**, descartando modificações no contexto de tipos.

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

Uma variação do comando de repetição permite o uso de uma varáivel que conta a iteração corrente. Se esta variável já estiver presente no contexto com tipo inteiro, a seguinte regra se aplica.

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

Caso a variável não esteja no contexto então a seguinte regra se aplica.

$$\frac{\Theta;\Delta;\Gamma \vdash_e e : \tau \quad \tau \in \{\textbf{Int}, \tau[]\} \quad v \notin \Gamma \quad V' = \{v\} \cup V \quad \Gamma' = \Gamma, v : \textbf{Int} \quad \Theta;\Delta;\Gamma';V' \vdash_c cs \rightsquigarrow V'';\Gamma''}{\Theta;\Delta;\Gamma;V \vdash_c \textbf{iterate } v \ e \ cs \rightsquigarrow V;\Gamma}$$

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

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

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.

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

## 4.4 Semântica de declarações

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

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

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

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

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

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;

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

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 $\Gamma$ com os parâmetros formais da função definida, junto com o tipo desta função, para permitir chamadas recursivas.
3. Inicializamos $\Theta_\tau$ 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, este deve ser um return.

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

Outra possibilidade válida para término de uma função é chamando outra que possua o mesmo tipo de retorno.

$$\frac{\Theta(v) = (\tau_1, ..., \tau_m) \to (\tau_1, ..., \tau_n)}{(\tau_1, ..., \tau_n) \vdash_{ret} v(e_1, ..., e_m)\ \bullet}$$

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.

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

## 4.5 Semântica de programas completos

Programas Lang são formados por uma ou mais declarações. Dizemos que um programa Lang é bem formado se este possui pelo menos uma função de nome **main** de tipo $String[] \rightarrow Void$, em que o arranjo de strings representa possíveis argumentos de linha de comando fornecidos ao programa. A regra a seguir faz essa validação, inicializando o contexto $\Theta$ com as definições iniciais presentes em $\Theta_0$:

$$\frac{\Theta_0; \bullet \vdash_p prog \rightsquigarrow \Theta; \Delta \quad \Theta(\textbf{main}) = \textbf{String[]} \rightarrow \textbf{Void}}{\vdash_{wf} \ prog}$$

Com isso, finalizamos a especificação da semântica estática de Lang. Na próxima seção apresentaremos a semântica dinâmica de programas Lang.