Sintaxe Concreta de L1

Sumário

[Introdução 2](#_Toc465500938)

[Sintaxe Corrente 3](#_Toc465500939)

[Termos 3](#_Toc465500940)

[Termos Básicos 3](#_Toc465500941)

[Tipos 4](#_Toc465500942)

[Tipos Básicos 4](#_Toc465500943)

[Traits 4](#_Toc465500944)

[Operadores 5](#_Toc465500945)

[Operadores Básicos 5](#_Toc465500946)

[Operadores Adicionais 5](#_Toc465500947)

[Prioridade e Associatividade 6](#_Toc465500948)

[Identificadores 6](#_Toc465500949)

[Indentação e Espaçamento 7](#_Toc465500950)

[Açúcar Sintático 7](#_Toc465500951)

[Extensões Práticas 8](#_Toc465500952)

[Comentários 8](#_Toc465500953)

[Bibliotecas 8](#_Toc465500954)

[stdlib 9](#_Toc465500955)

[Argumentos de Linha de Comando 13](#_Toc465500956)

[Extensões de L1 14](#_Toc465500957)

[Igualdade Genérica 14](#_Toc465500958)

[Trait Orderable 16](#_Toc465500959)

[Input e Output 18](#_Toc465500960)

# Introdução

Este documento tem como objetivo especificar a sintaxe concreta feita da linguagem L1. É presumido conhecimento da gramática, semântica e sistema de tipos de L1, focando apenas nos detalhes específicos a essa implementação da linguagem.

Originalmente, a implementação possuía apenas as construções básicas de L1, porém foram sendo adicionados tanto açúcares sintáticos quanto extensões práticas, que não modificam a definição da linguagem (como comentários, bibliotecas, etc). Mais tarde, a própria linguagem L1 foi expandida, adicionando novos termos e tipos.

A seção *Sintaxe Corrente* mostra todas as construções básicas presentes na linguagem, separando cada tipo de construção em básico (ou seja, presente na definição padrão de L1) e adicional (ou seja, criada nessa implementação específica de L1). Além disso, ela possui informações sobre restrições impostas para cada construção (identificadores, por exemplo).

Embora a seção *Sintaxe Corrente* mostre todas as construções gramaticais da linguagem, é necessário ler a seção *Extensões Práticas* para saber a sintaxe completa dessa implementação de L1. Essas construções não adicionam nenhuma funcionalidade à linguagem, mas apenas facilitam o trabalho de programação com comentários, reuso de código, etc. A única exceção é a adição de argumentos de linha de comando, permitindo alterar o resultado de um programa sem mexer em seu código fonte. Porém, como essa é uma construção externa ao ambiente de codificação, decidi categorizá-la como uma extensão pratica.

# Sintaxe Corrente

## Termos

### Termos Básicos

|  |  |
| --- | --- |
| Termo | Sintaxe |
|  |  |
|  |  |
| op | op |
| if then else | if then else |
|  |  |
|  |  |
| fn | fn |
| fn | fn |
| let in | let |
| let in | let |
| let rec fn in | let rec |
| let rec fn in | let rec |
|  |  |
|  |  |
| isempty | empty? |
| hd | head |
| tl | tail |
| raise | raise |
| try with | try except |

### Termos Adicionais

|  |  |
| --- | --- |
| Termo | Sintaxe |
|  | ‘’ |

## Tipos

### Tipos Básicos

|  |  |
| --- | --- |
| Tipo | Sintaxe |
| int | Int |
| bool | Bool |
|  | -> |
| list |  |

Na especificação de funções, o tipo é associativo à direita. Dessa forma, é o mesmo que . É possível utilizar parênteses para obter a associatividade desejada.

### Tipos Adicionais

|  |  |
| --- | --- |
| Tipo | Sintaxe |
| char | Char |

### Traits

Traits são características que certos tipos podem ter. Essas características servem para criar restrições de tipos em funções e operadores, permitindo-as a serem polimórficas com certas restrições.

#### Equatable

Termos de tipos que possuem a trait *Equatable* podem ser testados igualdade. A regra para a construção de tipos *Equatable* é a seguinte.

Dos tipos básicos, apenas funções não são *Equatable*. Além disso, para uma lista ser *Equatable*, os elementos da mesma precisam ser *Equatable*.

#### Orderable

Essa trait permite que termos sejam comparados por ordem (menor, maior). Para um tipo ser *Orderable*, ele também precisa ser *Equatable*.

## Operadores

### Operadores Básicos

|  |  |  |
| --- | --- | --- |
| Operador | Sintaxe | Tipos Aceitos |
|  |  | Int |
|  |  | Int |
|  |  | Int |
| div |  | Int |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

### Operadores Adicionais

Além dos operadores definidos na gramática de L1, essa implementação possui os seguintes operadores:

|  |  |  |
| --- | --- | --- |
| Operador | Uso | Significado |
| (prefixo) |  | Oposto numérico de \*\* |
|  |  | Resto da divisão inteira de por \*\* |
|  |  | Concatena a lista com a lista \*\* |
|  |  | Aplica em |
|  |  | Aplica em |
|  |  | Operação logica “e” em e \*\* |
|  |  | Operação logica “ou” em e \*\* |

\*\* Requer a *stdlib* para funcionar

### Prioridade e Associatividade

Abaixo está a prioridade de cada operador, além de sua associatividade. É importante ressaltar que é possível mudar tanto a prioridade quando a associatividade com o uso de parênteses.

|  |  |  |
| --- | --- | --- |
| Prioridade | Operador | Associatividade |
| 1 | * (prefixo) | Nenhuma |
| 2 | (aplicação de função) | Esquerda |
|  | head, tail, empty? | Esquerda |
| 3 | ,, | Esquerda |
| 4 | , (infixo) | Esquerda |
| 5 |  | Direita |
| 6 |  | Esquerda |
| 7 |  | Nenhuma |
|  | , | Esquerda |
| 8 |  | Esquerda |
| 9 |  | Esquerda |

## Identificadores

Identificadores são strings de tamanho arbitrário, sem espaços em brancos, compostos de quaisquer caracteres com exceção dos seguintes:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| . | , | ; | : | \ | + |
| - | / | \* | % | > | < |
| = | ! | ( | ) | { | } |
| [ | ] | @ | ‘ | “ |  |

* Numerais não são permitidos no começo de um identificador, porém podem ser utilizados depois.
* A interrogação é permitida em identificadores, e o uso mais indicado para a mesma é no final de funções que retornem um booleano.

Existem termos reservados na linguagem L1, e esses termos não podem ser utilizados como identificadores:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| fn | let | rec | for | in | head |
| tail | empty? | nil | try | except | raise |
| true | false | if | then | else |  |

## Indentação e Espaçamento

Indentação não é considerada na hora de avaliar termos. Dessa forma, é possível colocar a indentação que o programador desejar.

Existem algumas construções que requerem espaços entre seus termos (, por exemplo), e esses “espaços” podem ser quaisquer um dos seguintes caracteres (além do próprio espaço):

\n, \t, \r

# Caracteres e Strings

Literais de caracteres (*chars*) são delimitados por aspas simples (‘), e literais de strings, por aspas duplas (“). O caractere de barra invertida (\) é usado para escapar certos caracteres especiais. Esses são:

*\b, \n, \r, \t, \\, \”, \’*

Literais de caractere possuem tamanho exatamente igual a 1, enquanto strings não possuem restrições.

# Açúcar Sintático

|  |  |
| --- | --- |
| Construção | Equivalência |
|  |  |
| let | let |
| let | let |
|  | **\*** |
|  | **\*** |
|  | \* |
| fn | fn |
| fn | fn |
|  |  |
|  |  |
| let rec | let rec \*\* |
| let rec | let rec \*\* |
| let | let |
| let | let |
|  |  |

\* Requer a *stdlib*

\*\* Apenas o nível mais externo é definido como recursivo.

# Extensões Práticas

## Comentários

São permitidos comentários de linha única. Tudo que vem depois da string é ignorado pelo interpretador até o final da linha.

## Bibliotecas

Foi implementada uma versão simples de bibliotecas externas para permitir o reuso de código e (principalmente) para permitir a criação de um conjunto de funções básicas para o funcionamento da linguagem (a *stdlib*).

Para importar uma biblioteca, é usada a expressão

onde é o caminho do arquivo a ser importado, relativo ao caminho do executável do interpretador. Vale ressaltar que é obrigatório o uso de aspas duplas na especificação do caminho. Por fim, a extensão do arquivo pode ser omitida, porém ela precisa ser .

Em L1, bibliotecas são arquivos normais de código (*nome.l1*) que possuem uma sequência de *let*s e *let rec*s, sendo que o último desses está incompleto (isto é, não possui ). Ao importar a biblioteca, o interpretador irá substituir a expressão pelo conteúdo da biblioteca, adicionando assim seus identificadores ao escopo atual.

### stdlib

Como em toda linguagem real, existem certas funcionalidades que faz mais sentido implementar através de funções do que construções da linguagem. Para isso, existe uma biblioteca de funções, automaticamente incluída em todos os programas, que fornece uma base mais robusta de funcionalidades. Abaixo estão todas as funções da *stdlib*, organizadas pelo tipo de manipulação que fazem.

#### Operações aritméticas

|  |
| --- |
|  |
| Resto da divisão inteira |

|  |
| --- |
|  |
| Negação unária (oposto numérico) |

|  |
| --- |
|  |
| Valor absoluto |

#### Operações lógicas

|  |
| --- |
|  |
| Negação lógica |

|  |
| --- |
|  |
| “e” lógico |

|  |
| --- |
|  |
| “ou” lógico |

|  |
| --- |
|  |
| “ou exclusivo” lógico |

#### Listas

##### Funções básicas

|  |
| --- |
|  |
| Adiciona um elemento no final de uma lista |

|  |
| --- |
|  |
| Concatena duas listas, colocando todos os elementos da segunda após os elementos da primeira |

|  |
| --- |
|  |
| Retorna o último elemento de uma lista não vazia |

|  |
| --- |
|  |
| Retorna todos os elementos de uma lista não vazia exceto o último |

|  |
| --- |
|  |
| Retorna o número de elementos de uma lista |

##### Geração de listas

|  |
| --- |
|  |
| retorna uma lista de inteiros da forma  onde caso e caso |

##### Transformações

|  |
| --- |
|  |
| Inverte a ordem dos elementos de uma lista |

|  |
| --- |
|  |
| Retorna uma lista obtida ao aplicar a função a cada argumento da lista. Ou seja: |

##### Reduções

|  |
| --- |
|  |
| Aplica uma função a um *acumulador* e cada elemento de uma lista em ordem, retornando sempre o novo valor do *acumulador* |

|  |
| --- |
| **reduce** |
| Mesma funcionalidade de *fold*, porém utilizando o primeiro elemento da lista como *acumulador.* Por causa disso, requer uma lista não vazia. |

|  |
| --- |
|  |
| termina se qualquer elemento de uma lista satisfaz um predicado |

|  |
| --- |
|  |
| Determina se todos os elementos de uma lista satisfazem um predicado |

|  |
| --- |
|  |
| O maior elemento de uma lista não vazia |

|  |
| --- |
|  |
| O menor elemento de uma lista não vazia |

##### Sublistas

|  |
| --- |
|  |
| retorna os primeiros elementos de , ou a lista inteira caso |

|  |
| --- |
|  |
| retorna o final de após pular elementos, ou uma lista vazia caso |

|  |
| --- |
|  |
| Retorna todos os elementos de uma lista até o primeiro que não satisfaz o predicado |

|  |
| --- |
|  |
| Retorna uma lista, pulando todos os elementos até o primeiro que não satisfaz o predicado. |

|  |
| --- |
|  |
| retorna a sublista de que começa no índice e termina no índice . Caso , aciona uma exceção. |

##### Buscas

|  |
| --- |
|  |
| Verifica se um determinado elemento ocorre na lista |

|  |
| --- |
|  |
| Retorna todos os elementos de uma lista que satisfazem um predicado |

##### Indexações

|  |
| --- |
|  |
| Acessa uma lista de forma indexada, começando a contagem em 0 |

|  |
| --- |
|  |
| Retorna o índice da primeira ocorrência de um elemento em uma lista, ou -1 caso ele não ocorra. |

##### Ordenação

|  |
| --- |
|  |
| Ordena os elementos de uma lista em ordem crescente. Usa algoritmo de bubble sort (só que ao contrário). |

## Argumentos de Linha de Comando

É possível passar argumentos para programas de L1 ao interpretá-los em uma linha de comando. Esses argumentos estarão disponíveis no programa da seguinte forma:

# Extensões de L1

## Igualdade Genérica

Inicialmente, L1 permitia apenas a comparação de igualdade (e desigualdade) para inteiros. Com essa extensão, é possível comparar termos de todos os tipos, exceto funções.

Para permitir isso, foi criada a ideia de *Traits* para o sistema de tipos. Para implementar igualdade, foi necessário criar apenas uma trait (*Equatable*), mas o sistema permite expansão para outras traits no futuro (*Iterable, Orderable, etc*).

A nova regra de tipos para a operação de igualdade e desigualdade fica:

Foi criado um novo tipo de constraint para o algoritmo de inferência de tipos (“Tipo possui trait ”). Agora, a função *collectEqs* retorna uma lista de constraints, que podem ser:

A função unify foi modificada para identificar qual o tipo de constraint na cabeça da fila e, caso ela seja uma contraint de Trait, ela vê se a constraint é para uma variável de tipo ou não. Caso positivo, ela chama o método de expansão de constraint, que procura por ocorrências da variável de tipo em outras constraints e gera novas constraints de trait. Caso contrário, ela tenta resolver a constraint, gerando novas constraints de trait caso não seja possível (ou seja, caso o tipo possua variáveis de tipo internas.

| Trait (typ, trait') as c ->

match typ with

| Type.X \_ ->

unify <| rest @ (expandTraitConstraint c rest)

| \_ ->

unify <| rest @ (getTraitRequirements typ trait')

A função *expandTraintConstraint* recebe uma constraint de Trait e uma lista de constraints. Para cada constraint de igualdade da lista que contenha a variável de tipo original, cria uma nova constraint de trait para adicionar no final da lista. Isso permite a verificação posterior das constraints de trait para variáveis de tipo que não tenham sido unificadas.

let rec expandTraitConstraint (Trait (typ, trait') as constraint') list =

match list with

| [] -> []

| first::rest ->

match first with

| Equals (s, t) | Equals (t, s) when s = typ ->

expandTraitConstraint constraint' <| rest @ [Trait (t, trait')]

| \_ ->

expandTraitConstraint constraint' rest

A função *getTraitRequirements* verifica se um tipo satisfaz a trait . Para isso, ela analisa recursivamente até encontrar a resposta (positiva ou negativa) ou encontrar uma nova variável de tipo. Caso encontre uma variável de tipo, ela cria um novo constraint de trait, que é então adicionado no final da lista de constraints do unify.

É importante ressaltar que cada trait vai possuir uma regra de construção diferente, então é impossível criar um método genérico para isso. A cada nova trait que for adicionado, será necessário aumentar esse método com mais um *match* e novas regras de recursão.

let rec getTraitRequirements typ trait' =

match trait' with

| Equatable ->

match typ with

| Int | Bool -> []

| Type.X x as typ' -> [Trait (typ', Equatable)]

| List typ' -> getTraitRequirements typ' trait'

| Function (\_, \_) ->

raise <| InvalidType "Did not meet equatable trait requirement"

As regras de semântica operacional são as seguintes (regras de propagação de raise não são mostradas):

As regras para a operação “=”, usada acima, são as seguintes:

## Trait *Orderable*

Seguindo das ideias da igualdade, foi criada uma nova trait *Orderable* para tipos que podem ser ordenáveis. Para ser *Orderable*, um tipo também deve ser .

A regra de tipo para todos os operadores de ordenação é a seguinte:

As regras da semântica operacional, com exceção das de propagação de raise, são as seguintes:

As regras para a operação *op’*, usada acima, são as seguintes:

(comparações numéricas normais)

## Input e Output

Para permitir entrada e saída, foram necessárias diversas modificações na linguagem. Primeiramente, foram introduzidos 3 novos tipos (e um pseudônimo de tipo).

O tipo *String* é, na realidade, apenas um açúcar sintático na definição, e é tratado exatamente como uma lista de *Char*.

Também foi criada uma nova *Trait* para tipos que podem ser convertidos de e para strings.

Foram introduzidas novos termos gramaticais:

Esses termos estão divididos, grosseiramente, em 2 categorias. A primeira categoria lida com entrada e conversões entre valores e strings. A segunda lida com saída e composição sequencial de operações.

Dos termos introduzidos, dois deles são valores:

Abaixo estão as regras do sistema de tipos para essa extensão:

Igualmente, as regras de avaliação:

A função *toString*, definida acima, tem o seguinte comportamento, definido em pseudocódigo:

let toString term =

let s =

match term with

| n -> “|n|”

| c -> c

| True -> “true”

| False -> “false”

| nil -> “[]”

| c::s -> “ + printString (c::s) + “

| v::s -> “[“ + printList + “]”

| \_ -> raise

fold (fun acc x => (C x)::acc) nil (reverse s)

let rec printString string =

match string with

| nil -> “”

| c::s -> c + printString s

let rec printList list =

match list with

| nil -> “”

| v::nil -> toString v

| v::s -> (toString v) + “,“ + printList s

Similarmente, a função fromString é definida como:

let fromString s =

let term, parsedString = parseString s

if parsedString != s then

Raise

else

term

let parseString s =

match s with

| ‘[‘::rest -> parseList rest

| ‘”’::rest -> parseString

| n::[] when n.isChar -> parseInt

| startsWith “true” -> True, “true”

| startsWith “false” -> False, “false”

| c::[] -> C c

| \_ -> Raise

let rec parseList s =

try

let t1 = fromString

| nil -> “”

| c::s -> c + printString s

let rec printList list =

match list with

| nil -> “”

| v::nil -> toString v

| v::s -> (toString v) + “,“ + printList s