Relatório de desenvolvimento do List

PPGTI / IMD / UFRN / DSL

Alunos:

- Diogo Eugenio
- Willie Lawrece

Índice

Questões Gerais

A forma escolhida pra fazer o transpilador foi usando Visitors e a linguagem de desenvolvimento do transpilador foi Python3. O transpilador em si está no arquivo `TranspilerVisitor.py`. No arquivo de gramatica nós colocamos labels nas expressões pra permitir fazer os tratamentos mais facilmente.

Os tipos primitivos que são compartilhados entre List e Python3 foram simples de fazer, bastando no visitor retornar o valor "puro", foi o caso dos literais inteiros e booleanos.

Soma

As operações de soma em inteiros são as mesmas do python, então bastaria somar as expressões da esquerda e da direita, porém, na especificação de List está definido que não pode ocorrer uma soma entre um inteiro e um bool, por exemplo. Então decidimos fazer o tratamento para que esse erro semântico seja estourado em tempo de execução pelo interpretador do Python3.

Como na especificação de List está definido a soma somente para inteiros e lista, a regra que adotamos pra fazer a verificação foi simplesmente verificar que o tipo das expressões não pode ser `bool` **e** os tipos devem ser iguais. Como List só possui três tipos, remover o bool da regra da soma faz com que os dois outros tipos (int e lista) sejam suportados.

Como a soma é uma expressão que pode ocorrer isolada numa linha como em

```
print 1 + False
```

mas também pode ocorrer numa atribuição, como em:

```
a = 1 + False
```

o resultado da expressão soma pelo transpilador não poderia ter mais de uma linha e deveria ser considerado como uma única expressão pelo Python3, por isso, usamos o operador ternário do Python pra conseguir fazer essa verificação dos tipos e ao mesmo tempo retornar a soma.

O esquema geral do operador ternário em Python é:

```
variavel = valor if expressao_booleana else outro_valor
```

é similar à:

```
if expressao_booleana:
    variavel = valor
else:
    variavel = outro_valor
```

Então o esquema foi fazer:

```
(L_EXP + R_EXP) if (tipos_iguais() and not_bool()) else estoure_erro()
```

desde que `estoure_erro` seja uma funcao, seria preciso criar uma função que estoura um erro. Decidimos não criar essa função porque ao gerar vários arquivos Python3 a partir de vários script List essa função acabaria sendo redefinida várias vezes. Apesar de termos criado um script extra, como será falado mais abaixo, preferimos ainda assim não se utilizar desse artifício. Assim, `estoure_erro` teria de ser uma função que pudesse ser definida também em apenas uma linha. A forma sintaticamente aceita de fazer isso em Python é usando lambda, porém é erro sintático fazer:

```
lambda e: raise Exception()
```

porque `raise Exception()` não é uma expressão que retorna algum valor. Encontramos a solução à essa problema no stackoverflow: https://stackoverflow.com/a/8294654, onde a solução foi criar um generator. Assim foi possível chamar a função `throw` do generator passando como argumento o objeto do erro que se quer retornar:

```
lambda e: (_ for _ in ()).throw(e)
```

Soma e multiplicação de listas

No List, a soma de listas retorna uma outra lista com seus elementos somados. Vimos que pra fazer esse tratamento no transpilador seria preciso descobrir o valor exato de cada expressão na soma, então se tivéssemos:

```
print [1, [2], a] + [b, [c]]
```

O transpilador precisaria saber que o valor na segunda posição da expressão da esquerda (`[2]`) é também uma lista e a soma entre ele e `[c]` também deveria seguir a mesma regra da soma definida na especificação de List. Imaginamos que pra isso seria preciso fazer uma recursidade que seria muito difícil de fazer numa linha só, então preferimos implementar a operação de soma no código final de Python, aproveitando que o Python permite sobrescrever as operações de soma, multiplicação etc.

Então criamos uma classe `ListList` que se comporta exatamente igual à uma lista normal do Python, porém sobrescrevendo o magic-method `__add__`, de tal forma que, em Python, ao fazer `[1, 2] + [2, 3] == [1, 2, 2, 3]`, sobrescrevendo a operação de adição, conseguimos rescrever a lógica e fazer: `[1, 2] + [2, 3] == [3, 5]`. Pra isso criamos um arquivo acessório em python que chamamos de `lislang.py` que contém a definição da nossa classe List. Sobrescrevemos os magic-methods: `__getitem__`, `__repr__`, `__str__`, `__contains__` pra que o comportamento dos objetos dessa classe sejam muito similares à uma lista normal do Python, de tal forma que agora o transpilador só precisou retornar `ListList([lista_em_python])`.

Sobrescrevendo a operação de soma numa classe não foi preciso criar nenhuma função recursiva pra soma, porque a lógica foi simplesmente usar o operador `+` do Python em todos os elementos da lista, se a lista for um outro objeto `ListList` segue a mesma regra.

Pra concatenação em List (`.`) um princípio parecido foi usado. Como tínhamos uma classe própria pra lista e já tinhamos usado o operador `+` do Python, decidimos traduzir o operador `.` de List para qualquer outro operador binário em Python. O operador da multiplicação foi escolhido de forma aleatória, sem um motivo específico. Como a operação de concatenação, em List, está definida somente para objetos lista, foi preciso realizar o mesmo tratamento com o operador ternário em Python pra garantir que caso os tipos não sejam corretos um erro é estourado. Assim, o código:

```
print a . b
```

seria traduzido para o Python como:

```
print(a * b)
```

seguindo a mesma regra do operador ternário pra garantir que a multiplicação só ocorreria se as duas expressões fossem em cima de objetos `ListList`.

A título de exemplo, uma tradução de `print a . b` em List para Python, fica:

```
from listlang import *

print(a * b if (type(a) is ListList and type(b) is ListList) else (lambda e: (_ for _ in ()).throw(e))(ValueError('It is not possible concatenate "a" with "b"')))
```

Arquivos de exemplo

Os arquivos de teste estão armazenados na pasta `examples`. Cada um deles serve pra testar uma funcionalidade implementada do transpilador.

arquivo	Descrição
ex01	testa a funcionalidade de concatenar listas e criar listas com numeros e variaveis
ex02	testa um erro sintático
ex03	testa a funcionalidade de "soma" de listas, somando cada valor individualmente
ex04	testa a funcionalidade de erro semântico ao tentar somar um inteiro com um valor booleano
ex05	testa um erro semântico de concatenar uma lista com um inteiro