**Criando uma linguagem de programação**

**APS – Lógica da Computação**

**INSPER – 7° Semestre – Matheus Pellizzon**

1. **A implementação**

Existem inúmeras bibliotecas com inúmeras funcionalidades para auxiliar no desenvolvimento de uma linguagem própria. Escolhi o Python, utilizando a biblioteca *RPLY* para *tokenizar* e fazer a análise sintática; ainda criei classes Node para montar uma AST e fazer a verificação semântica. Ao avaliar (*Evaluate*) a árvore montada, a biblioteca *llvmlite* é utilizada para gerar o código de máquina e otimizá-lo. Esse relatório tem como objetivo mostrar um pouco do que foi feito e como foi feito (de uma maneira simplificada).

1. **Definindo a linguagem**

Para a realização desse projeto, me inspirei na sintaxe do *JavaScript*, embora tenha fugido um pouco do proposto por essa linguagem. Decidi também aproveitar um pouco da minha descendência italiana para escrever uma linguagem de programação em italiano. Assim, minha linguagem de programação (ainda sem um nome específico) tem essa cara:

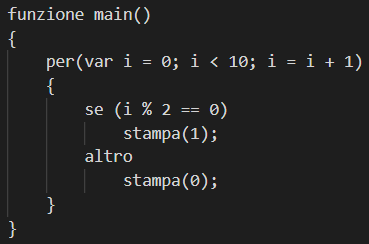


Figura 1: exemplo básico da linguagem; programa que printa 1 se ‘i’ for par e 0 caso contrário, dado um range de valores (loop for – per)

Por simplicidade, trabalhei somente com variáveis inteiras, mas ainda sim é possível criar programas interessantes, realizar comparações, entre outras *features* básicas da maioria das linguagens de programação. A EBNF completa dessa linguagem é a seguinte:

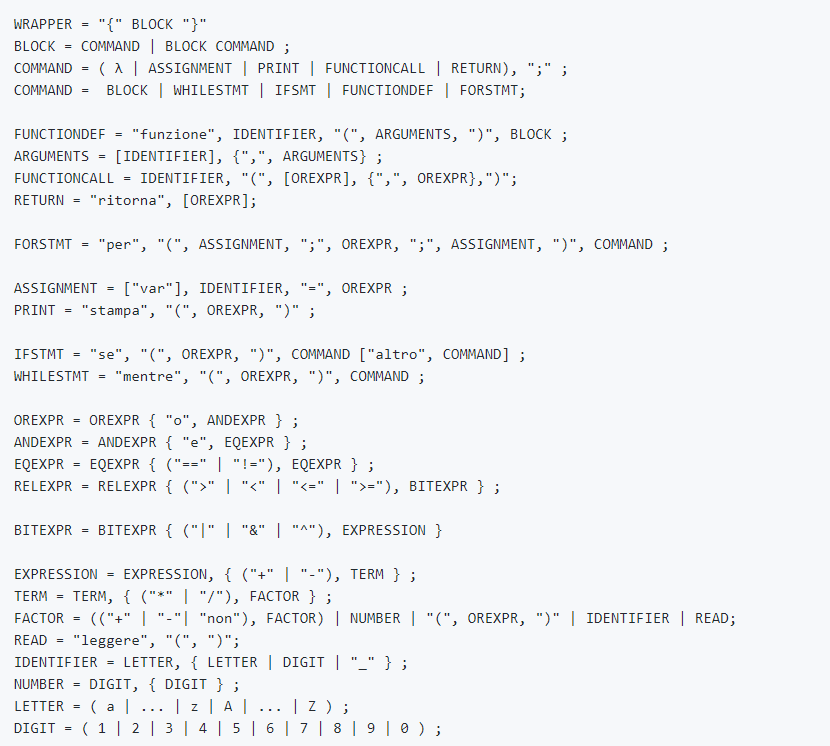


Figura 2: EBNF

Com a EBNF definida, fica claro o que deve ser *tokenizado* e quais são as expressões regulares para os tokens. Ela também nos ajuda a entender quais são as regras de produção para um determinado comando ou estruturas básicas que serão interpretadas pelo meu compilador.

1. **Pré-processamento**

Nessa etapa limpamos o código fonte, removendo comentários ou traduzindo macros. Como não tenho macros na minha linguagem, apenas removo comentários utilizando regex no próprio python. Ou seja, processo a string do código removendo casos onde há comentários dos tipos:

- /\*comentário\*/

- //comentário

Como não implementei a operação “//” (parte inteira da divisão), porque só estou trabalhando com inteiros, remover todos os casos onde há “//” sem comprometer a integridade do código é simples.

1. **Lexer ou tokenizador**

Tendo em vista os tokens especiais e palavras reservadas da linguagem, podemos *tokenizar* um código fonte. Por exemplo, o código fonte da Figura 3 passa a ser uma lista de Tokens (Figura 4).



Figura 3: exemplo simples de código.

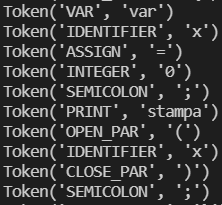


Figura 4: Tokens gerados a partir do código da Figura 3.

Os resultados da Figura 4 foram gerados e guardados pelo *LexerGenerator* da biblioteca *RPLY.* A próxima seção aborda um pouco do parser (também do *RPLY*), assim a integração entre os módulos *LexerGenerator* e *ParserGenerator* é simples e facilita o trabalho de especificar a sintaxe da linguagem.

1. **Parser**

Baseado na EBNF da linguagem, podemos segui-la para definir as regras de produção e como as cadeias de Tokens devem existir dentro de um código fonte. Caso um Token seja encontrado em um local onde não deveria existir, o programa não faz sentido sintaticamente, logo não será possível compila-lo e executá-lo. Como exemplo vamos utilizar um loop while (mentre, na minha linguagem).

Segundo a ENBNF, um loop while é definido por:

WHILESTMT = "mentre", "(", OREXPR, ")", COMMAND ;

A lógica por trás de um compilador implica que devemos ter uma sequência dos seguintes tokens:

Token(“WHILE”, “mentre”)

Token(“OPEN\_PAR”, “(“)

- Tokens relacionados a operação condicional (exemplo: x < 10)

Token(“CLOSE\_PAR”, “)“)

- Tokens relacionados aos comandos do while (exemplo: x = x + 1; stampa(x);)

Em formato de código “tradicional”, teríamos que fazer essas verificações uma a uma e acusando erros caso algo não esteja compatível com o que foi definido.

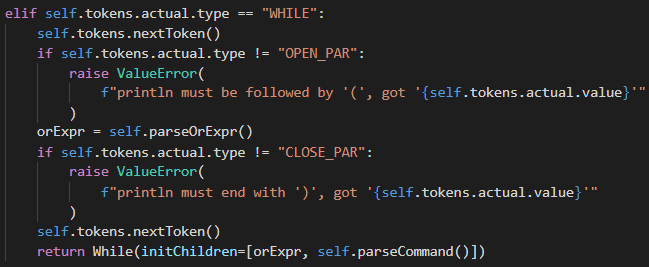


Figura 5: verificação da sintaxe de um loop while.

Já se utilizarmos o *ParserGenerator*, temos algo bem mais simplificado.

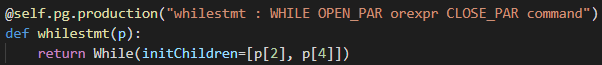


Figura 6: verificação da sintaxe de um loop while utilizando ferramentas do *RPLY*.

Assim como no código da Figura 5, existem outras definições e outras funções com regras de produção diferentes para garantir que tudo será tratado adequadamente. A chamada de “self.parseOrExpr()” será responsável por fazer a mesma verificação que “orexpr” da Figura 6.

1. **AST**

Como visto, temos as etapas do lexer e do parser implementadas de uma forma simples. Quando o código fonte vai sendo processado, objetos da classe Node (como While das Figuras 5 e 6) são encadeados para montar uma *Abstract Syntax Tree*. Para exemplificar: dada a operação (2+3)/(5\*1), temos a seguinte AST:

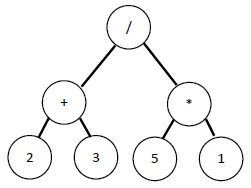


Figura 7: AST para a expressão (2+3)/(5\*1).

* 1. **Resolução dos nós (Evaluate):**

Os objetos Node possuem um método especial responsável por resolver a AST. Para o exemplo anterior temos três operadores binários (+, /, \*) e 4 nós de valor. Ao chamar o método Evaluate da raiz (/), todos os nós serão resolvidos para obter, nesse caso, o valor final da operação:

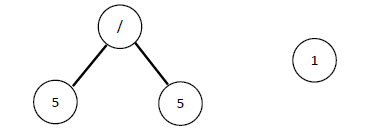


Figura 7: resolução intermediária e final da expressão (2+3)/(5\*1).

* 1. **Como prosseguir:**

Tendo em mente a estrutura recursiva da resolução de uma AST, e tendo definido os nós relevantes para a linguagem sendo implementada, basta definir o que o método Evaluate de cada nó deverá fazer, e assim gerar um programa em linguagem de máquina para ser compilado e executado.

1. **LLVM**

A biblioteca LLVMLITE possui diversos wrappers para escrever em uma linguagem de mais alto nível (python) para um IR do LLVM (sucintamente, um código em assembly).

A declaração de um Node IntVal (que retorna um constante) é feita como mostra a Figura 8, enquanto a Figura 9 mostra como seria o método alternativo.

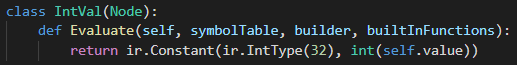


Figura 8: declaração de uma constante usando LLVMLITE.

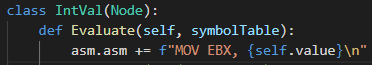


Figura 8: declaração de uma constante em assembly x86

Embora ambos os métodos funcionem e são relativamente simples, os códigos produzidos pelo segundo método não serão necessariamente os mais eficientes em termos de compilação. É possível passar argumentos para o módulo LLVM quer será capaz de otimizar o código sem muitos esforços.

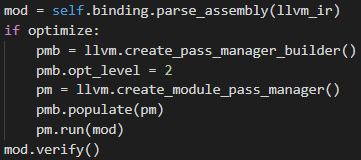


Figura 9: exemplo simples de otimização

1. **Conclusão**

Tratamos de todas as etapas que constituem um compilador. Dado um código fonte, tratamos do pré-processamento, das análises léxica, sintática e semântica, da geração do código e otimização do código gerado. Finalmente, podemos compilar e executar um programa.