**Criando uma linguagem de programação**

**APS – Lógica da Computação**

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

1. **Contextualização**

Foi proposto a implementação de uma linguagem de programação própria. Essa linguagem deveria conter pelo menos todas as estruturas básicas de uma linguagem de programação: variáveis, condicionais, loops e funções.

Existem inúmeras bibliotecas para auxiliar no desenvolvimento de uma linguagem própria. Nesse projeto foi utilizado o Python, com a biblioteca *RPLY* para realizar os passos de *tokenização* e análise sintática; as classes Node (e classes que extendem Node) foram mantidas do projeto de compilador desenvolvido em aula. Sua utilidade continua a mesma, montar uma AST e verificar a semântica do programa de entrada.

No entanto, 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, a linguagem base escolhida foi o *JavaScript*. Mais especificamente um JavaScript com tokens e palavras reservadas que remetem a língua italiana. A seguir, um exemplo da linguagem de programação proposta:

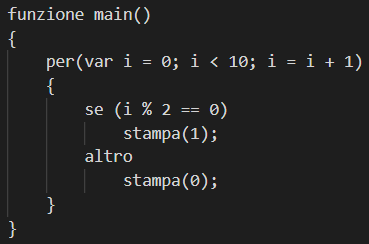


Figura 1: exemplo básico da linguagem; programa que imprime 1 se ‘*i*’ for par e 0 se for ímpar, dado um range de valores (loop for – per).

Por simplicidade, são utilizados somente números inteiros. Mesmo assim, é possível criar programas interessantes, realizar comparações, entre outras *features* básicas presentes nas demais linguagens de programação.

Além das estruturas básicas requeridas no projeto, foram implementados também operações bit a bit (shift para esquerda - << , shift para direita - >> , and - & , or - | , xor - ^), além de ter uma função pronta para exponencial (n\*\*m – mesma sintaxe de Python) e para capturar inputs do usuário (*leggere*() – *sprintf* em C).

A EBNF completa dessa linguagem é a seguinte:

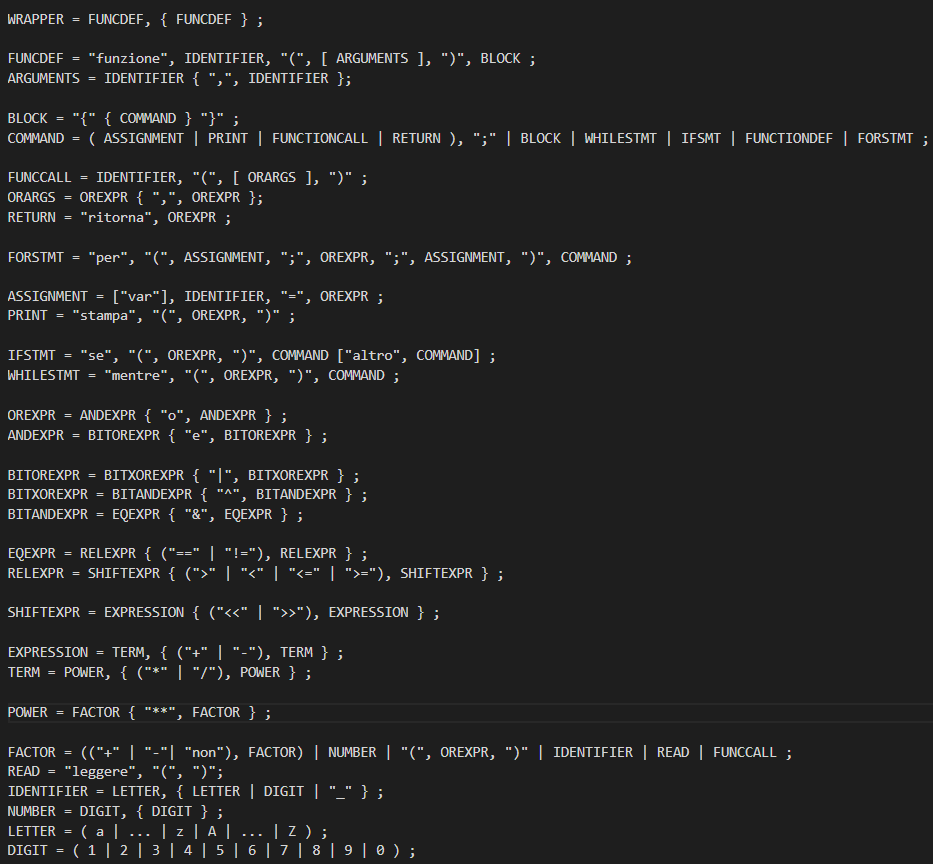


Figura 2: EBNF

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 ocorre a limpeza do código fonte, removendo comentários ou traduzindo macros. Como não existem/não foram implementadas macros na linguagem proposta, os comentários são removidos utilizando RegExp. Ou seja, a string do código é processada removendo casos em que há comentários dos tipos:

* /\* comentário \*/
* // comentário

Como não foi implementada a operação “//” (parte inteira da divisão), pois só foi trabalhado com números inteiros, remover todos os casos em que há “//” não compromete a integridade do código.

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.

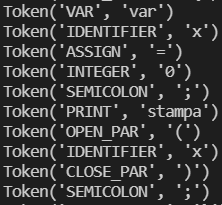


Figura 4: Tokens gerados a partir do código da Figura 3.

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, as regras de produção e como as cadeias de Tokens devem existir dentro de um código fonte já foram pré-determinados. 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.

Para facilitar o entendimento dessa parte, segue o exemplo de um loop *while* (*mentre*). Segundo a ENBNF, um loop *while* é definido por:

WHILESTMT = “*mentre*”, “(“, OREXPR, “)”, COMMAND ;

Essa estrutura implica que é necessário que existam os tokens “*mentre*”, seguido de um “(“, seguido de tokens relacionados a uma OREXPR, um “)” e finalmente as expressões (ou expressão) que fazem parte desse loop.

* 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”, essas verificações devem ser feitas uma a uma e erros devem ser levantados caso algo não seja compatível com o que foi definido.

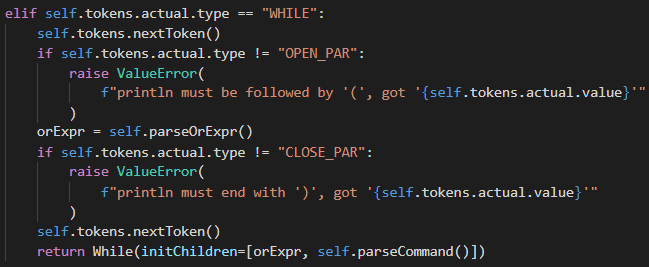


Figura 5: verificação da sintaxe de um loop while.

Utilizando o *ParserGenerator*, a estrutura de código é mais simplificada; se tudo for feito corretamente, evitando ambiguidade na língua, é possível chegar em códigos simples e de fácil legibilidade.

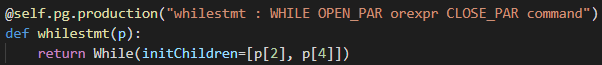


Figura 6: verificação da sintaxe de um loop while utilizando ferramentas do *RPLY*.

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, as etapas do Lexer e do Parser foram implementadas de uma forma simplificada. Quando o código fonte é processado, objetos da classe Node (como While das Figuras 5 e 6) são encadeados para montar uma *Abstract Syntax Tree* (AST). Para exemplificar: dada a operação (2+3)/(5\*1) (lembrando que não é possível ter um comando como uma expressão dessas na linguagem proposta), a AST gerada deveria ser:

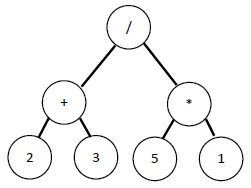


Figura 7: AST para a expressão (2+3)/(5\*1).

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

Os objetos Node possuem um método especial responsável por resolver a AST. Para o exemplo existem 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:

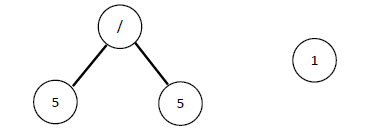


Figura 8: resolução intermediária e final da expressão (2+3)/(5\*1).

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 (praticamente um código em assembly – linguagem de máquina).

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

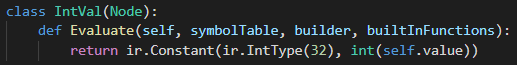


Figura 9: declaração de uma constante usando LLVMLITE.

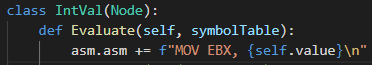


Figura 10: declaração de uma constante em assembly x86

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 otimizar ambos os códigos, e o módulo do LLVM permite passar argumentos para o construtor, de modo a otimizar o código sem muitos esforços, que existiriam caso fosse implementado um código assembly do zero.

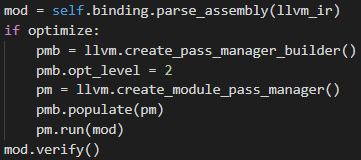


Figura 11: exemplo simples de otimização

1. **Conclusão**

Todas as etapas que envolvem um compilador foram explicadas de maneira simples e sucinta. Dado um código fonte, as etapas do pré-processamento, das análises léxica, sintática e semântica, da geração do código e otimização do código gerado foram completados. Finalmente, é possível compilar e executar um programa qualquer, que possua estruturas básicas implementadas na linguagem de programação feita, como mostra a Figura 10.

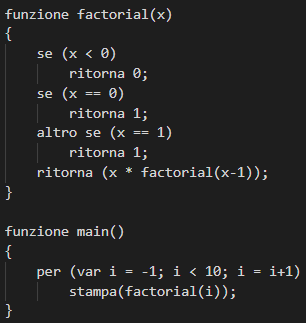


Figura 12: outro exemplo de código da linguagem desenvolvida.