INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE MINAS GERAIS - *CAMPUS* BAMBUÍ

Mickael Oswaldo de Oliveira
Artur Francisco Pereira Carvalho
Deivison Oliveira Costa

TRABALHO FINAL - COMPILADORES

Bambuí-MG
2023

1. A LINGUAGEM

O Python é uma linguagem de programação amplamente adotada e apreciada por sua simplicidade e legibilidade. No entanto, uma das críticas recorrentes à linguagem é a falta de tipagem estática e símbolos ambíguos, o que pode tornar o código propenso a erros difíceis de detectar e depurar [5]. Para abordar esse desafio, foi desenvolvido o *tyPython*, um novo compilador *Python* que visa tornar a linguagem mais segura e estável por meio da introdução de tipagem estática. Apesar de se basear e claramente ter inspirações, funciona de forma totalmente diferente da programação genérica do interpretador *CPython*, sendo totalmente reformulada.

É importante ressaltar que o projeto ainda se encontra em fases iniciais de desenvolvimento e, portanto, não é uma versão finalizada. Neste momento, o *tyPython (TPY)* não oferece suporte à programação orientada a objetos, o que representa uma limitação significativa. Além disso, o compilador ainda carece de funcionalidades essenciais, como alguns loops, funções e outras estruturas fundamentais presentes no *Python* tradicional. Mesmo com essas limitações, o *tyPython* é promissor e busca preencher uma lacuna no ecossistema, oferecendo uma alternativa tipada e segura para desenvolvedores que desejam usufruir desses benefícios, mesmo que com algumas restrições.

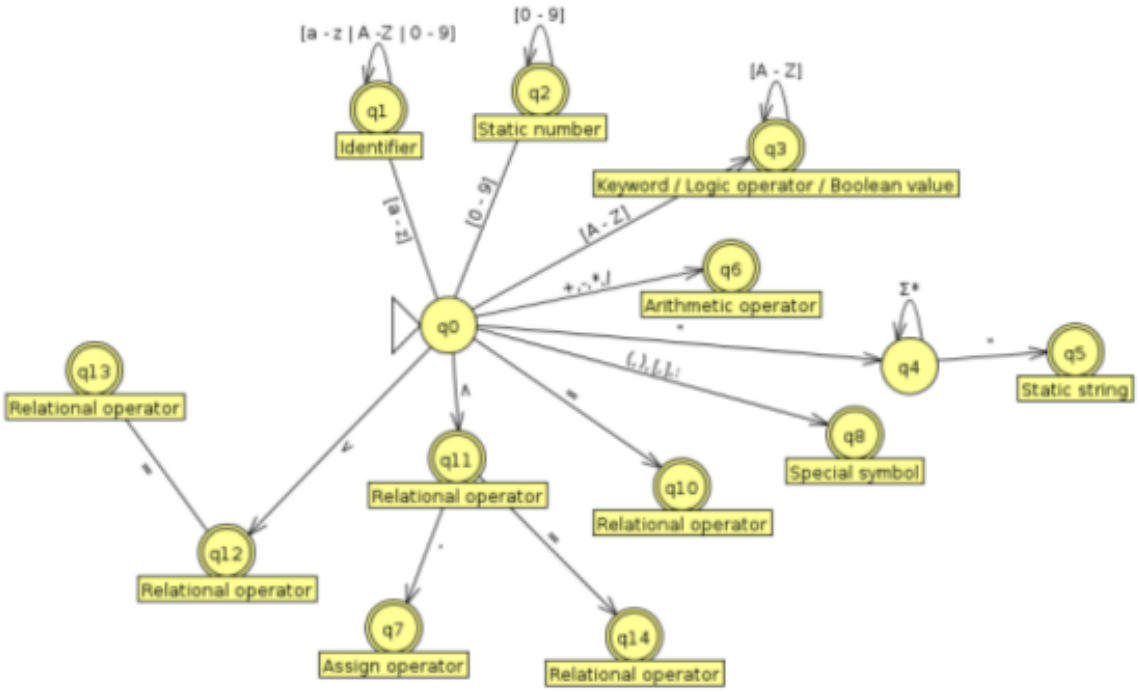
1.1. TABELA DE TOKENS

RegEx	Token	Token Category
Num [0:9]	-	-
Char [a:z | A:Z]	-	-
(a-z) (Char | Num)*	<id, ?>	Identifier
(Num)+	<staticNum, ?>	Static number
"(Char | Num)*"	<staticString, ?>	Static string
INTEGER	<INTEGER, ?>	Keyword
STRING	<STRING, ?>	Keyword
IF	<IF, ?>	Keyword
ELIF	<ELIF, ?>	Keyword
ELSE	<ELSE, ?>	Keyword
WHILE	<WHILE, ?>	Keyword
WRITE	<WRITE, ?>	Keyword
READ	<READ, ?>	Keyword
RETURN	<RETURN, ?>	Keyword
$	-	Commentary
(;	<, ?>	Special symbol
)	<), ?>	Special symbol
[	<[, ?>	Special symbol
]	<], ?>	Special symbol
:	<-<<, ?>	Special symbol
AND	<AND, ?>	Logic operator
OR	<OR, ?>	Logic operator
NOT	<NOT, ?>	Logic operator
TRUE	<TRUE, ?>	Boolean operator
FALSE	<FALSE, ?>	Boolean operator
+	<+, ?>	Arithmetic operator
-	<←, ?>	Arithmetic operator
*	<*, ?>	Arithmetic operator
/	</<, ?>	Arithmetic operator
<-	<-<<, ?>	Assign operator
>	<>, ?>	Relational operator
<=	<<<=, ?>	Relational operator
>=	<>>=, ?>	Relational operator
=	<<=, ?>	Relational operator
!=	<!=, ?>	Relational operator

Como observado na tabela acima, todas as palavras reservadas pela linguagem são escritas em maiúsculo. Isso faz com que a leitura do código seja mais fácil, deixando bem evidente as funções nativas. A tabela de tokens é uma estrutura de dados que mantém informações sobre os tokens reconhecidos em um programa ou código-fonte [1]. Os tokens são unidades léxicas, como palavras-chave, identificadores, números, operadores e símbolos especiais, que formam a estrutura da linguagem [1]. Cada entrada na tabela de tokens geralmente contém informações como [1]:

- O tipo do token (por exemplo, identificador, número, palavra-chave).
- O valor associado ao token (por exemplo, o nome de uma variável ou o valor de um número).
- A posição no código-fonte onde o token foi encontrado (linha e coluna).
- Outras informações relevantes, como escopo, nível de aninhamento, etc.
- A tabela de tokens é usada para armazenar informações sobre os tokens identificados durante a análise léxica, e essas informações são posteriormente utilizadas durante a análise sintática e semântica da linguagem.

2. O AUTÔMATO



O Autômato Finito Determinístico (AFD) acima é uma máquina de estados finitos que descreve o processo de reconhecimento de tokens em um código-fonte [2]. Ele define as regras e padrões que governam como os caracteres no código-fonte são agrupados em tokens

válidos [2]. O AFD funciona lendo os caracteres do código-fonte um a um e, com base em seu estado atual e entrada, transita para um novo estado ou reconhece um token [2].

O AFD é projetado para ser capaz de diferenciar os diferentes tipos de tokens em um código-fonte. Cada estado no AFD representa um possível estado de reconhecimento, e as transições entre estados são determinadas pelas regras da linguagem. Quando o AFD atinge um estado de aceitação, ele reconhece o token correspondente.

3. A GRAMÁTICA

A implementação da gramática fornece uma compreensão da estrutura e da sintaxe da linguagem *tyPython*. Ela define as regras que governam a formação de programas .tpy especificando como os diferentes elementos da linguagem podem ser combinados. A seguir, apresentamos a gramática da respectiva linguagem com suas regras e símbolos:

P -> : S :
S -> A | I | L | W | R | F | V | ε
V -> T "id"; S
T -> INTEGER | STRING | BOOLEAN | REAL
A -> "id" <- D; S | "id" <- EXP; S
D -> int | str | bool | real | "id"
EXP -> D OPA D | "id" OPA "id" | D OPA "id" | "id" OPA D
I -> "IF" "(" CONDITION ")" ":" S ":" S | "IF" "(" CONDITION ")" ":" S ":" E
CONDITION -> id OPR C' | D OPR C'
C' -> "id" | D | "id" OPL CONDITION | D OPL CONDITION
OPR -> "<"|">"|"="|"!="|"<="|">="
OPL -> "AND" | "OR" | "NOT"
OPA -> "+" | "-" | "*" | "/"
E -> "ELSE" ":" S ":" S | "ELIF" "(" CONDITION ")" ":" S ":" E | "ELIF" "(" CONDITION ")" ":" S ":" S
L -> "WHILE" "(" CONDITION ")" ":" S ":" S
W -> "WRITE" "(" D ")" ";"
R -> "READ" "(" "id" ")" ";"
F -> "FUNCTION" "id" "(" PARAMETERS ")" ":" S "RETURN" D ":" S
PARAMETERS -> T "id" | T "id" "," PARAMETERS

4. TABELA *FIRST E FOLLOW*

A tabela *FIRST* e *FOLLOW* é utilizada na construção e análise de gramáticas livres de contexto (GLCs) como parte do processo de compilação e análise de linguagens de programação [2]. Ela é particularmente importante na geração de analisadores sintáticos, como analisadores preditivos e de descida recursiva, que são usados para determinar a estrutura sintática de um programa escrito em uma linguagem de programação.

A tabela *FIRST* e *FOLLOW* ajuda nas seguintes atividades [2]:

- Determina os primeiros símbolos (terminais) que podem começar uma cadeia de símbolos derivados em uma gramática.
- Resolve conflitos na análise sintática e ajuda a determinar qual produção de uma regra deve ser usada durante a análise.
- Determina os símbolos que podem seguir imediatamente uma cadeia de símbolos derivados em uma gramática.
- Detecta erros sintáticos durante a análise de um programa, permitindo que o analisador saiba quando um símbolo não é seguido por um símbolo esperado.
- Detecta e resolve conflitos de redução em analisadores de tabela LR (analisadores de análise sintática bottom-up).

Símbolo	**First**	**Follow**
P	{ ":" }	{ $ }
S	{ "id", "IF", "WHILE", "WRITE", "READ", "FUNCTION", ε }	{ "id", "IF", "WHILE", "WRITE", "READ", "FUNCTION", "ELSE", "ELIF", ε }
V	{ INTEGER, STRING, BOOLEAN, REAL }	{ "id", "IF", "WHILE", "WRITE", "READ", "FUNCTION", "ELSE", "ELIF", ":" }
T	{ INTEGER, STRING, BOOLEAN, REAL }	{ "id" }

A	{ "id" }	{ "id", "IF", "WHILE", "WRITE", "READ", "FUNCTION", "ELSE", "ELIF", ":" }
D	{ "int", "str", "bool", "real", "id" }	{ ";", ":", "=", "!=", "<", ">", "<=", ">=", "+", "-", "*", "/", "AND", "OR", "NOT", ",", ")", "RETURN" }
EXP	{ "int", "str", "bool", "real", "id" }	{ ";", ":", "=", "!=", "<", ">", "<=", ">=", "+", "-", "*", "/", "AND", "OR", "NOT", ",", ")", "RETURN" }
I	{ "IF" }	{ "id", "IF", "WHILE", "WRITE", "READ", "FUNCTION", "ELSE", "ELIF", ":" }
CONDITION	{ "id", "int", "str", "bool", "real" }	{ ":", ")", "ELSE", "ELIF" }
C’	{ "id", "int", "str", "bool", "real", ε }	{ ":", ")", "ELSE", "ELIF" }
OPR	{ "<", ">", "=", "!=", "<=", ">=" }	{ "id", "int", "str", "bool", "real" }
OPL	{ "AND", "OR", "NOT" }	{ "id", "int", "str", "bool", "real" }
OPA	{ "+", "-", "*", "/" }	{ "id", "int", "str", "bool", "real" }
E	{ "ELSE", "ELIF", ε }	{ "id", "IF", "WHILE", "WRITE", "READ",

		"FUNCTION", "ELSE", "ELIF", ":" }
L	{ "WHILE" }	{ "id", "IF", "WHILE", "WRITE", "READ", "FUNCTION", "ELSE", "ELIF", ":" }
W	{ "WRITE" }	{ "id", "IF", "WHILE", "WRITE", "READ", "FUNCTION", "ELSE", "ELIF", ":" }
R	{ "READ" }	{ "id", "IF", "WHILE", "WRITE", "READ", "FUNCTION", "ELSE", "ELIF", ":" }
F	{ "FUNCTION" }	{ "id", "IF", "WHILE", "WRITE", "READ", "FUNCTION", "ELSE", "ELIF", ":" }
PARAMETERS	{ INTEGER, STRING, BOOLEAN, REAL }	{ "id" }

5. O COMPILADOR

O compilador da linguagem é um tipo de software que traduz o código-fonte de um programa escrito em uma linguagem de programação de alto nível para um código de máquina ou código intermediário que pode ser executado por um computador [1]. Em outras palavras, um compilador converte o código escrito por um programador em uma linguagem que os computadores possam entender e executar.

5.1. REQUISITOS FUNCIONAIS

- **Análise Léxica** [3]**:**

    - Reconhecimento e tokenização de palavras-chave, identificadores, literais, operadores e símbolos em conformidade com as regras da linguagem de programação.
- **Análise Sintática** [3]**:**
    - Verificação da sintaxe do código-fonte para garantir que ele siga a estrutura e a gramática corretas da linguagem.
- **Análise Semântica** [3]**:**
    - Verificação de erros semânticos, como tipos incompatíveis, escopo de variáveis, uso de variáveis não declaradas, etc.
    - Geração de tabelas de símbolos para rastrear informações sobre variáveis, funções e outros elementos do programa.
    - Verificação de conformidade com regras semânticas da linguagem, como atribuição de tipos de dados adequados.
- **Geração de Código** [3]**:**
    - Tradução do código-fonte em linguagem de alto nível para código de máquina ou código intermediário.
    - Otimização do código gerado para melhorar o desempenho e a eficiência da execução do programa.
- **Manipulação de Entrada e Saída** [3]**:**
    - Leitura do código-fonte a partir de arquivos ou outras fontes de entrada.
    - Escrita do código de máquina ou código intermediário gerado em arquivos de saída.
- **Manipulação de Erros** [3]**:**
    - Detecção e relatório de erros no código-fonte.
    - Possibilitar a recuperação de erros e a continuação da compilação sempre que possível.
- **Gerenciamento de Memória** [3]**:**
    - Alocação e liberação de memória para estruturas de dados internas do compilador.
    - Gerenciamento eficiente da memória durante a compilação.
- **Suporte a Recursos da Linguagem** [3]**:**
    - Implementação de recursos da linguagem de programação, como estruturas de controle, estruturas de dados, funções, classes, herança, polimorfismo, etc.

- **Integração com Ambiente de Desenvolvimento** [3]**:**
    - Integração com ferramentas de desenvolvimento, como editores de código e ambientes de depuração.
- **Geração de Relatórios e Documentação** [3]**:**
    - Geração de informações úteis, como mensagens de erro, warnings e documentação associada ao código gerado.

5.2. REQUISITOS NÃO-FUNCIONAIS

- **Desempenho** [4]**:**
    - Define a velocidade e a eficiência do compilador.
    - Isso pode incluir a velocidade de compilação e a eficiência do código gerado.
    - Um compilador rápido pode ser essencial para desenvolvimento eficiente.
- **Confiabilidade** [4]**:**
    - Refere-se à capacidade do compilador de funcionar corretamente e gerar código correto de forma consistente.
    - Um compilador confiável não deve gerar erros inesperados.
- **Segurança** [4]**:**
    - Em relação à capacidade do compilador de proteger contra vulnerabilidades de segurança.
    - Por exemplo, um compilador deve evitar gerar código suscetível a ataques de injeção de código.
- **Portabilidade** [4]**:**
    - A capacidade do compilador de gerar código que seja portável entre diferentes sistemas e arquiteturas.
    - Um compilador portátil é capaz de compilar programas para serem executados em diferentes ambientes.
- **Manutenibilidade** [4]**:**
    - Define o quão fácil é manter e estender o compilador ao longo do tempo.
    - Isso inclui a clareza e a documentação do código-fonte do compilador.
- **Usabilidade** [4]**:**
    - Refere-se à facilidade de uso do compilador, tanto para desenvolvedores que escrevem código-fonte quanto para aqueles que mantêm o compilador em si.
    - Uma interface de usuário amigável pode ser importante em alguns contextos.

- **Consumo de Recursos** [4]**:**
    - Os recursos de hardware e memória que o compilador consome durante a compilação.
    - O compilador não deve ser excessivamente exigente em termos de recursos.
- **Compatibilidade** [4]**:**
    - A capacidade de um compilador gerar código que seja compatível com outras versões do mesmo compilador ou com compiladores de outros fornecedores.
- **Tolerância a Falhas** [4]**:**
    - A capacidade de um compilador lidar com erros de maneira elegante, apresentando mensagens de erro compreensíveis e permitindo que a compilação continue em situações de erro não críticas.
- **Documentação e Suporte** [4]**:**
    - A disponibilidade de documentação completa e suporte ao usuário para o compilador, incluindo manuais, tutoriais e recursos de ajuda.

5.3. O ANALISADOR LÉXICO

O analisador léxico é uma das componentes essenciais de um compilador, que desempenha um papel fundamental na fase de análise do processo de compilação [3]. Sua principal função é analisar o código-fonte de um programa e transformá-lo em uma sequência de tokens, ou seja, unidades significativas da linguagem de programação em questão.

A análise léxica é a primeira etapa do processo de compilação, que ocorre antes da análise sintática e da geração de código intermediário. Ela lida com a identificação e classificação dos diferentes elementos lexicais presentes no código-fonte, tais como palavras-chave, identificadores, operadores, números, símbolos e constantes. Durante a análise léxica, o analisador léxico realiza as seguintes tarefas:

- **Tokenização:**
    - O código-fonte é dividido em tokens, onde cada token é uma unidade léxica com um significado específico. Por exemplo, em uma linguagem de programação como C, a palavra "int" é tokenizada como uma palavra-chave que representa um tipo de dado.
- **Eliminação de espaços em branco:**

    - O analisador léxico remove espaços em branco, tabulações e quebras de linha que não são relevantes para a estrutura do programa. Isso simplifica o código para a análise subsequente.
- **Contagem de linhas e colunas:**
    - O analisador léxico também mantém informações sobre a posição de cada token no código-fonte, o que pode ser útil para a geração de mensagens de erro e depuração.
- **Tratamento de comentários:**
    - Comentários no código-fonte são geralmente ignorados pelo analisador léxico, uma vez que não têm impacto na execução do programa final. Comentários podem ser de várias formas, como comentários de uma única linha (//) ou comentários de várias linhas (/* ... */).

O analisador léxico gera uma saída que é uma sequência de tokens reconhecidos, que são posteriormente passados para o analisador sintático. O analisador sintático verifica a estrutura gramatical do programa e constrói uma árvore de análise sintática, que serve como base para a geração de código intermediário ou a execução do programa.

5.4. O ANALISADOR SINTÁTICO

O analisador sintático, também conhecido como *parser*, é uma das partes fundamentais de um compilador, desempenhando um papel crucial na fase de análise da compilação [3]. Sua principal função é verificar a estrutura gramatical do código-fonte de um programa e construir uma representação hierárquica chamada de árvore de análise sintática ou árvore de *parse*. Esta árvore é uma representação intermédia que reflete a organização e a sintaxe do código-fonte, o que é fundamental para a próxima etapa do processo de compilação.

A análise sintática ocorre após a análise léxica, que transforma o código-fonte em uma sequência de *tokens*, identificando palavras-chave, identificadores, operadores e outros elementos léxicos. O analisador sintático trabalha com esses *tokens* para verificar se eles estão organizados de acordo com as regras gramaticais da linguagem de programação em questão. Ele realiza as seguintes tarefas:

- **Verificação da sintaxe:**
    - O analisador sintático verifica se a sequência de tokens está de acordo com a gramática da linguagem.

    - Isso envolve a identificação de estruturas sintáticas válidas, como declarações, expressões, comandos condicionais, loops, entre outros.
- **Construção da árvore de análise sintática:**
    - Quando uma sequência de tokens corresponde a uma estrutura sintática válida, o analisador sintático cria um nó na árvore de análise sintática para representar essa estrutura.
    - A árvore é construída de forma hierárquica, onde os nós pais representam estruturas de nível superior e os nós filhos representam estruturas aninhadas.
- **Tratamento de erros sintáticos:**
    - Se o código-fonte contém erros de sintaxe, o analisador sintático é responsável por detectá-los e relatar esses erros.
    - Ele pode gerar mensagens de erro indicando a localização dos problemas no código, facilitando a depuração e a correção.

A árvore de análise sintática gerada pelo analisador sintático é usada como base para as próximas etapas do processo de compilação. Pode servir como entrada para o gerador de código intermediário, que cria representações intermediárias do programa, facilitando a otimização e a geração de código de máquina.

Além disso, a árvore de análise sintática também é útil para a geração de mensagens de erro mais informativas e para a análise semântica, que verifica se o código segue as regras semânticas da linguagem.

5.5. O ANALISADOR SEMÂNTICO

Um analisador semântico é uma parte essencial de um compilador, desempenhando um papel crítico na fase de análise da compilação [3]. Sua principal função é verificar se o código-fonte de um programa está de acordo com as regras semânticas da linguagem de programação em questão.

Enquanto o analisador léxico cuida dos aspectos lexicais (*tokens*) e o analisador sintático trata da estrutura gramatical do código, o analisador semântico se concentra na significação e nas implicações do código.

A análise semântica ocorre após a análise léxica e a análise sintática, quando o código-fonte já foi convertido em uma estrutura hierárquica (árvore de análise sintática) que representa a organização do programa. O analisador semântico executa as seguintes tarefas:

- **Verificação de tipos:**

    - Um dos principais aspectos da análise semântica é verificar se os tipos de dados usados nas expressões e declarações são compatíveis de acordo com as regras da linguagem. Por exemplo, verifica se você não está tentando somar uma string com um número em linguagens que não permitem essa operação.
- **Verificação de escopo:**
    - O analisador semântico verifica se as variáveis e identificadores usados no código estão definidos nos escopos apropriados. Ele garante que variáveis locais estejam acessíveis apenas dentro de funções onde foram declaradas, que variáveis globais sejam acessíveis em todo o programa, e que não ocorram conflitos de nome.
- **Resolução de sobrecarga:**
    - Em linguagens que suportam sobrecarga de funções (ter várias funções com o mesmo nome, mas com parâmetros diferentes), o analisador semântico determina qual função é chamada com base nos argumentos fornecidos.
- **Verificação de coerência semântica:**
    - O analisador semântico garante que o código siga as regras e convenções semânticas da linguagem, como a proibição de utilizar variáveis não inicializadas, a detecção de loops infinitos, entre outras.
- **Geração de mensagens de erro:**
    - Quando o código não está em conformidade com as regras semânticas, o analisador semântico é responsável por gerar mensagens de erro informativas que indicam a natureza e a localização dos problemas no código.

O principal objetivo do analisador semântico é garantir que o código-fonte seja correto do ponto de vista semântico, ou seja, que ele funcione de acordo com as regras da linguagem e que não leve a comportamentos indefinidos ou erros durante a execução. Ao fazê-lo, ele ajuda a evitar problemas comuns de programação, como erros de tipo, erros de escopo e outros erros semânticos que podem ser difíceis de depurar.

6. MANUAL DA LINGUAGEM

Aqui demonstramos e descrevemos o funcionamento dos recursos até então implementados, desde atribuições e estruturas de controle de fluxo até entrada e saída de dados.

6.1. EXEMPLO DE DECLARAÇÃO E ATRIBUIÇÃO

Este exemplo demonstra a declaração e atribuição de uma variável de *string* 'x'. Segue a sintaxe do *TyPython*, onde deve haver um espaço entre o identificador e o símbolo de atribuição <-.

```
$ TyPython example of declaration and assignment of vars $
$ between the identifier and the assign symbol "<-" there must be a space $
:
STRING x;
x <- "Var assignment";
:
```

6.2. EXEMPLO DE ESTRUTURA CONDICIONAL

Este exemplo destaca o uso de declarações condicionais (*IF*, *ELIF* e *ELSE*). Declara duas variáveis inteiras 'a' e 'b', lê a entrada para ambas e, em seguida, as compara. Dependendo dos resultados da comparação, imprime a saída apropriada.

```
$ TyPython example of if/elif/else $
:
INTEGER a;
INTEGER b;
READ(a);
READ(b);
IF (a>b):
    WRITE(a);
:ELIF(b>a):
    WRITE(b);
:ELSE:
    WRITE("a equals b");
:
:
```

Saída:

```
1
3
3
```

6.3. EXEMPLOS DE E/S

6.3.1. ESCRITA

Este exemplo demonstra a declaração *WRITE*. Imprime um literal de *string* e o valor armazenado na variável ‘x’. O comentário especifica que qualquer dado na instrução *WRITE* pode ser uma *string*, inteiro, booleano, real ou um identificador.

```
$ TyPython example of WRITE $
$ WRITE '(' any data ')' ';'$
$ any data is string value, int value, bool value, real value or identifier $
:
STRING x;
WRITE("Write example");
WRITE(x);
:
```

Saída:

```
Write example
```

6.3.2. LEITURA

Isso ilustra o uso da declaração *READ* no *TyPython*. Declara uma variável de *string* ‘x’ e lê um valor para ela do usuário.

```
$ TyPython example of READ $
$ READ '(' "identifier" ')' ';'$
:
STRING x;
READ(x);
:
```

Saída:

```
"Read example"
```

6.4. ESTRUTURA DE REPETIÇÃO WHILE

Isso mostra o uso de um loop WHILE no TyPython. Inicializa uma variável booleana x com o valor 1 e, em seguida, entra em um loop que continua enquanto x for igual a 1. Dentro do loop, imprime "LOOP" e lê um novo valor em x.

```
$ TyPython example of loop with while $
:
BOOLEAN x;
x <- 1;
WHILE (x = 1):
   WRITE("LOOP");
   READ(x);
:
:
```

Saída:

```
LOOP
0
```

REFERÊNCIAS

[1] HOE, Alfred V.; SETHI, Ravi; ULLMAN, Jeffrey D. **Compilers—principles, techniques, and tools**. 1986.

[2] COOPER, Keith D. TORCZON, Linda. **Engineering a compiler**. Elsevier, 2011.

[3] GRUNE, Dick et al. **Modern compiler design**. Springer Science & Business Media, 2012.

[4] SCOTT, Michael. **Programming language pragmatics**. Morgan Kaufmann, 2000.

[5] GUTH, Dwight. **A formal semantics of Python 3.3**. 2013.