Processamento de Linguagens e Compiladores (3º ano de LCC)

Trabalho Prático 2

Relatório de Desenvolvimento

André Lucena Ribas Ferreira
(A94956)

Carlos Eduardo da Silva Machado
(A96936)

Gonçalo Manuel Maia de Sousa
(A97485)

15 de janeiro de 2023

Resumo

Este relatório aborda o desenvolvimento de um compilador para uma linguagem original utilizando os módulos *lex* e *ply* de *Python* no contexto do 2º trabalho prático da UC *Processamento de Linguagens e Compiladores*.

Conteúdo

Capítulo 1

Introdução

Para que seja possível garantir a comunicação de humanos com máquinas é necessária a existência de linguagens estruturadas para que possam ser sujeitas a análise sintática, na qual o texto de origem é atomizado. Este processo é conhecido por parsing e é precedido por um processo de análise léxica, na qual sequências de caracteres são convertidas em *t*okens.
No âmbito do desenvolvimento da nossa linguagem, definimos uma gramática independente de contexto (*GIC*) e um compilador em *python* que gera um código *Assembly* que é executado por uma máquina de *stack* virtual(VM).
Das duas opções de implementação que nos foram fornecidas decidimos fazer ambas com algumas alterações. Por um lado, no que toca a *Arrays*, decidimos que seriam implementados com um número arbitrário de dimensões.
Por outro, lado no que toca à declaração de subprogramas, foram implementados na forma de funções com vários argumentos de entrada e um de saída que podem ser chamadas em qualquer ponto do programa, até com recursividade, e declarados por qualquer ordem.

Estrutura do Relatório

Para além deste, o relatório compreende diferentes Capítulos. Em 2 descreve-se o enunciado do projeto. Em 3, descreve-se a conceção da Linguagem e a sua sintaxe. Em 4 apresenta-se o código do Analisador Léxico e do Sintático. Em 5 apresenta-se a Conclusão do projeto.

Capítulo 2

Enunciado

Pretende-se que comece por definir uma linguagem de programação imperativa simples, a seu gosto. Apenas deve ter em consideração que essa linguagem terá de permitir:

- *declarar* variáveis atómicas do tipo *inteiro*, com os quais se podem realizar as habituais operações aritméticas, relacionais e lógicas;
- *efetuar* instruções algorítmicas básicas como a *atribuição do valor de expressões numéricas a variáveis*;
- *ler* do *standard input* e *escrever* no *standard output*;
- *efetuar* instruções *condicionais* para controlo do fluxo de execução;
- *efetuar* instruções cílicas para controlo do fluxo de execução, permitindo o seu aninhamento.
  Note que deve implementar pelo menos o ciclo **while-do**, **repeat-until** ou **for-do**.

Adicionalmente deve ainda suportar, à sua escolha, uma das duas funcionalidades seguintes:

- *declarar e manusear* variáveis estruturadas do tipo *array (a 1 ou 2 dimensões)* de inteiros, em relação aos quais é apenas permitida a operação de indexação (índice inteiro);
- *definir e invocar subprogramas* sem parâmetros mas que possam retornar um resultado do tipo inteiro.

Como é da praxe neste tipo de linguagens, as variáveis deverão ser declaradas no início do programa e não pode haver re-declarações, nem utilizações sem declaração prévia. Se nada for explicitado, o valor da variável após a declaração é 0 (zero). Desenvolva, então, um compilador para essa linguagem com base na GIC criada acima e com recurso aos módulos Yacc/ Lex do PLY/Python. O compilador deve gerar pseudo-código, Assembly da Máquina Virtual VM.
Muito Importante:
Para a entrega do TP deve preparar um conjunto de testes (programas-fonte escritos na sua linguagem) e mostrar o código *Assembly* gerado bem como o programa a correr na máquina virtual *VM*.

Capítulo 3

Linguagem LIGMA

Para a linguagem compilada, demos o nome de **LIGMA**, isto é, Linguagem Inventada por Gonçalo, Machado e André. A sua sintaxe é inspirada por **C**, com a adição de detalhes tirados de outras linguagens além de uma nova estruturas de controlo o *switch* que é único da nossa linguagem. Para além disso **LIGMA** apenas opera sobre inteiros.

3.1 Estrutura do programa

Qualquer programa escrito em **LIGMA** deve seguir a seguinte estrutura: As declarações de variáveis podem ser feitas em qualquer ponto do programa e a sua localidade irá depender de onde foram declaradas. As funções não podem ser declaradas dentro de qualquer estrutura de controlo ou função mas podem ser chamadas em qualquer ponto. Todas as estruturas de controlo são limitadas por chavetas, no entanto é permitida a sua omissão no caso em que é executado apenas um comando, e todo o comando deve ser terminado com ";".

3.2 Declarações e Atribuições

A declaração de variáveis é feita de forma semelhante a **C**. Inicialmente é indicado o tipo de dados e seguidamente o nome da variável. As Atribuições, no entanto são feitas de forma mais próxima a **R**, na qual uma atribuição é feita através do uso de uma seta (->,<-) na base da qual se encontra o valor a atribuir e na ponta a variável. Além disso permitimos que as setas tenham um comprimento arbitrário. Para mais, interpretamos uma atribuição dupla do tipo, <->, que também pode ter comprimento arbitrário, será um operador de troca de valores de variáveis.

```
1  int a;
2  a <- 5;
3  int b <- 2;
4  int c;
5  c <---------------- 40;
6  int d;
7  2 -> d;
8  d ----------> b;
9  c <-> d
10 b <-------> a
```

Listing 3.1: Exemplo de Declaração e Atribuição

Se uma variável for declarada sem atribuição inicial, assume o valor de 0, o mesmo acontece com todas as declarações de *arrays*.

3.3 Operações aritméticas, lógicas e condicionais

Como qualquer linguagem, **LIGMA** é capaz de realizar as operações aritméticas habituais como a adição (+), subtração (-), divisão (/), multiplicação (*), módulo (%) e potência (^), esta última está definida como função e apenas é carregada para o output se for utilizada. Além disso estão implementadas as operações lógicas, *and* (&), *or* (|) e *not* (~, !). Para mais, a linguagem contém também as operações usuais de relação como maior (>), maior ou igual (>=), menor (<), menor ou igual (<=), igual (=), o igual pode ter comprimento arbitrário, e diferente (!=, ~=).

```
1
2  a + b
3
4  a - b
5
6  a / b
7
8  a * b
9
10 a^b
11
12 a & b
13
14 a | b
15
16 !a
17 ~a
18
19 a >= b
20 a <= b
21 a > b
22 a < b
23
24 a = b
25 a == b
26 a ======= b
27
28 a != b
29 a ~= b
30 a !========== b
```

Listing 3.2: Exemplo de Operações

3.4 Estruturas de controlo de execução

Em termos de estruturas de controlo, **LIGMA** possui a estrutura mais usual, *if*, escrito de modo totalmente análogo àquele de **C**, permitindo o encadeamento da palavra reservada *else*

Para além disso, está implementada uma estrutura especial que age como uma forma mais elegante de escrever uma série de condições lógicas, o *switch* escrito do seguinte modo:

Começa-se por uma das duas palavras reservadas *switchcase* e *switchcond*, seguida de várias condições, entre parênteses, separados por vírgulas e opcionalmente precedidos por um identificador a que chamamos *label*, que identifica a condição que precede.
As condições são seguidas de um bloco entre chavetas, com estrutura especial.
O bloco de código é, por sua vez, composto por várias instâncias de ":" seguido de código entre chavetas e precedidas opcionalmente por uma das *label* cuja condição pretendemos associar ao código.
Em todos os casos em que o código não está associado a uma *label*, é lhe associado uma das condições sem *label*, pela ordem que aparecem. Note-se que todas as condições devem ter código a elas associado.
A ordem de verificação das condições depende de qual das palavras reservadas é utilizada, *switchcase* ou *switchcond*, são testadas as condições pela ordem em que aparecem antes das chavetas ou pela ordem em que aparecem dentro das chavetas, respetivamente.

```
1
2  if ( a != 2) {
3      (...)
4  } else {
5      (...);
6  }
7
8  if (a == 2) {
9      (...)
10 }
11
12 switchcase label1(x >= 0), (x = 4), label2(3){
13     :{
14     (...)
15     }
16     label1:{
17     (...)
18     }
19     label2:{
20     (...)
21     }
22 }
23
24 \vfill
25 \newpage
```

Listing 3.3: Exemplo estruturas de controlo

3.5 Instruções de repetição

Como estrutura de repetição apenas foi implementado o *while*, que, tal como o *if*, tem funcionamento semelhante ao da linguagem **C**.

```
1 while(x < 10){
2     (...)
3 }
4
5
6 while(i<2){
7     (...)
```

```
8       while(j<2){
9           (...)
10      }
11      (...)
12  }
```

Listing 3.4: Exemplo While

3.6 Input e Output

Como operadores de input output foi utilizado o carácter "?" precedido por ">" ou "<" no caso do input ou output respetivamente.

```
1  int x;
2  x <- 15;
3  (x)>?;
4
5  int y;
6  y <- <?;
```

Listing 3.5: Exemplo de IO

3.7 Arrays

É possível declarar e manusear variáveis estruturada do tipo *array* de dimensões arbitrárias. A maneira como declaramos é igual à da linguagem **C**, ou seja, indicamos o tipo seguido do nome da variável e por fim o número de dimensões que vai corresponder ao número de "[ ]". Dentro de cada "[ ]", temos de indicar o tamanho daquela dimensão.

```
1  int ar[1];
2  int ar2[2][2];
3  int ar3[3][3][3];
4  int ar4[4][4][4][4];
5
6  ar[0] <- 2 + 15;
7  a <- ar2[0][3];
```

Listing 3.6: Exemplo de Arrays

3.8 Funções

A implementação de funções foi desenhada de forma a que seja semelhante à notação matemática. Para tal, a declaração de funções é feita do seguinte modo: Uma palavra seguida do carácter ":" denota o nome da função, seguidos de uma série de nomes de variáveis separadas por vírgulas, os argumentos da função. Por fim, uma seta para a direita (->) de tamanho variável seguida por um único nome denota a variável na qual deve ser posto o valor de retorno da função.
Uma função pode ter qualquer número de argumentos ou zero e pode adicionalmente retornar ou não um valor.

```
f:a,b,c -> res{
    (...)
    z <- a+b
}

f:->x{
x <- 3
}

f:{
(...)
}
```

Listing 3.7: Exemplo de Funções

3.9 Comentários

É importante, para uma linguagem de programação, possuir comentários, pois permite a explicação do código junto deste, tal como permite manter excertos de códigos dentro do ficheiro que não se pretendem compilar. Podemos fazer comentários através de um "#" no início e outro "#" no fim, podemos, portanto, fazer comentários multi-linha. Para além disso decidimos deixar ao utilizador decidir como indentar o código, para tal caracteres como espaço, *tab* e \n, são sempre ignorados pelo compilador podendo ser colocados em qualquer ponto do programa.

```
# Exemplo de comentario de uma linha #

# Exemplo
de comentario
multi-linha
#
```

Listing 3.8: Exemplo de Comentários

Capítulo 4

Codificação e Testes

O nosso trabalho está dividido em dois ficheiros distintos, correspondentes ao Analisador Léxico e ao Analisador Sintático, com este último também a realizar as ações semânticas necessárias à geração do código.

A invocação do programa faz-se através de uma linha de comandos a partir do *python* e do ficheiro *yacc.py*, utilizando o redirecionamento da escrita do *out* para o ficheiro desejado.

NOTA: Devido à utilização do *print* para a escrita dos erros, estes irão ser também escritos, quando ocorrem, no ficheiro de saída desejado.

De seguida, deve-se introduzir o ficheiro de leitura escrito na linguagem **LIGMA**.

```
1    py .\yacc.py > test1.vm
2    test1.ligma
```

Listing 4.1: Exemplo da execução do programa (léxico e parser)

4.1 Lexer

O código do ficheiro **Lexer** encontra-se em A.1. Nele, foram definidos os *tokens* e os *literais* necessários. Para além disso, foram determinadas palavras reservadas, que não são capturadas como **ID**, um identificador. Tal foi feito a partir de um dicionário com todas estas palavras como chave e o respetivo Símbolo Literal como valor.

```
1  reserved = {
2      'if' : 'IF',
3      'else' : 'ELSE',
4      'while' : 'WHILE',
5      'switchcond' : 'SWITCHCOND',
6      'switchcase' : 'SWITCHCASE'
7  }
8
9  def t_ID(t):
10     r'[a-zA-Z]\w*'
11     t.type = reserved.get(t.value, 'ID')
12     return t
```

Listing 4.2: Definição do dicionário *reserved* e do terminal **ID**

Pela definição de **ID**, pode-se notar que as variáveis da linguagem devem começar por uma letra e só depois podem ser seguidas por outro tipos de carateres habituais.

Por fim, também se definiu a regra para comentários, através da utilização da *keyword pass*, encapsulando qualquer letra entre ocorrências de #.

```
def t_ANY_COMMENT(t):
    r'\#[^\#]*\#'
    pass
```

Listing 4.3: Definição do *token* **COMMENT**

4.2 Parser

O ficheiro *yacc.py*, que contém o código do *Parser* encontra-se em A.2.

4.2.1 Estrutura Geral

A nossa linguagem, tal como descrito anteriormente, não obriga a que as declarações sejam feitas no início de qualquer programa, nem que as funções sejam declaradas antes do resto do código. Também temos em consideração a convenção da linguagem **C** de permitir uma função *main* que é executada como o cerne do programa. Dessa forma, a função mais base tem em consideração se essa função existe ou não.

```
def p_axiom_start(p):
    "Axiom : Start"
    main = "pusha main\ncall\n" if p.parser.main else ""
    p.parser.final_code = "start\n" + \
        p[1] + main + pop_local_vars(p) + "stop\n" + \
        "".join(p.parser.function_buffer)


def p_start_empty(p):
    "Axiom : "
    p.parser.final_code = ""
```

Listing 4.4: Produções a partir do Axioma

Aqui dá-se uso a três variáveis subjacentes ao *parser*:

- *final_code*, que armazena o código final no final da análise.
- *main* que dita se a função *main* ocorreu em alguma altura durante a análise.
- lista *function_buffer* que armazena o código todo das funções declaradas, de modo a que sejam colocadas após o *stop* para não serem executados fora de ordem.

Também se dá uso à função auxiliar *pop_local_vars* que, serve, neste caso específico, para limpar a *stack* das variáveis (globais, nomeadamente) no final da execução do programa.

4.2.2 Declaração de Funções

A declaração de funções tem consideração quatro possibilidades: Nenhum argumento nem valor de retorno; apenas valor de retorno; apenas argumentos; tanto argumentos como retorno.
As funções são armazenadas na tabela *function_table* cuja chave é o identificador da função. Os valores são:

- *num_args* : número de argumentos da função;
- *return* : se a função tem valor de retorno ou não;
- *label* : delimitador o início do código da função, no código da máquina virtual.

As funções que têm mais que um argumento e um valor de retorno colocam-no, no final da sua execução, na mesma localização do primeiro argumento, enquanto as que não têm argumentos deixam o valor de retorno no topo da pilha antes do retorno, no mesmo local onde a variável de retorno foi declarada. Esta variável é a mesma que a descrita na declaração da função, não existindo palavra reservada *return* para o efeito.

Após a salvaguarda do retorno, as variáveis do âmbito local são retiradas, gerando-se código para tal. Importante realçar que esta derivação não utiliza a função *pop_local_vars*, cujo objetivo é retirar todas as variáveis do contexto local e recuperar o endereço local, porque necessita de uma ligeira diferença no caso de não existirem argumentos mas existir valor de retorno.

Também é de notar tanto a utilização da variável *internal_label*, que garante que todas as *labels* são únicas, como a reposição da variável *local_adress*, que, no caso do âmbito geral, mantém-se igual à *global_adress*.

```
def p_function_header(p):
    "FunctionHeader : ID FunScope FunCases"
    label = p.parser.internal_label
    num_args = len(p[3][1])

    s_label = f"F{label}"
    if p[1] == 'main':
        p.parser.main = True
        s_label = f"main"

    p.parser.function_table[p[1]] = {'num_args': num_args,
                                     'return': p[3][0],
                                     'label': s_label}

    #Se a fun   o devolve ou n o algo. p[3]    um tuplo.

    p[0] = (p[3][0], num_args, s_label + ":\n")
```

Listing 4.5: Definição do cabeçalho de uma função

O cabeçalho da função trata de colocar a função na *function_table*, permitindo que o *Body* da função a possa chamar já que já se encontra declarada.

O *FunctionHeader* devolve um tuplo com três elementos, o retorno, o *num_args* e a *label* pronta a ser colocada à cabeça do pseudo-código máquina.

```
def p_function(p):
    "Function : FunctionHeader Body"
    s = ""
    local_args = len(p.parser.id_table_stack[-1])-int(p[1][1])
    if local_args > 0:
        #deixar se houver retorno e n o houver argumentos
        s += "pop %d\n" % (local_args - int(p[1][1] == 0 and p[3][0]))
    p.parser.local_adress = 0
    p.parser.id_table_stack.pop()

    if p[1][0]:
        p.parser.function_buffer.append(
```

```
            p[1][2] + "pushi 0\n" + p[2] + \
            f"pushl 0\nstorel {-p[1][1]}\n" + s + "return\n")
    else:
        p.parser.function_buffer.append(
            p[1][2] + p[2] + s + "return\n")

    p[0] = ""
    p.parser.internal_label += 1
    p.parser.local_adress = p.parser.global_adress
```

Listing 4.6: Produção de reconhecimento da declaração de Funções

Notavelmente, a parte do código da função consta da derivação *Body*, que pode tanto ser código vazio, várias blocos ou apenas um que, tal como a linguagem **C** em certos aspetos, não necessita de chavetas.

Stack de Tabelas de Variáveis

As variáveis de todas as localidades, como é exemplo as da função declarada, são armazenadas na lista de tabelas *id_table_stack*, que merece atenção especial. Esta variável começa como uma lista de um único dicionário: a tabela das variáveis globais. Sempre que uma nova localidade emerge, é necessário adicionar uma tabela nova, que representa localidade. Isto ocorre nas estruturas de controlo e nas declarações das funções. Quando estas são derivadas, a função *pop_local_vars* trata tanto de retirar a última tabela desta lista como de gerar código para retirar as variáveis que estão na *stack*.

A derivação para *FunScope* determina o início da declaração da função, com o seu âmbito local. Nela, é adicionada uma nova localidade específica que regista apenas as variáveis locais à função. Para além disso, esta localidade vai ser única da Função, sem consideração pelas locais anteriores, ao contrário das estruturas de controlo.

```
def p_funscope(p):
    "FunScope : ':'"
    p.parser.id_table_stack.append(dict())
    p[0] = p[1]
```

Listing 4.7: Início do âmbito da declaração da Função

Deste modo, as variáveis declaradas como parâmetros da função encontram-se dentro do âmbito da mesma.

FunCases

A produção do não literal *FunCases* gera um tuplo com os parâmetros da função e a variável booleana que dita se a função tem retorno. Entretanto, essas variáveis já foram colocadas na nova tabela de variáveis com os endereços respetivos à *frame* que irá existir quando a função for invocada, que são retiradas da produção de *FunExtra*.

```
def p_funcases_funextra_rarrow(p):
    "FunCases : FunExtra RARROW ID"
    p.parser.id_table_stack[-1][p[3]] = {'classe': 'var',
                                         'endereco': 0,
                                         'tamanho': [1],
                                         'tipo': 'int'}

    for i in range(1, len(p[1])+1):
        if p[1][-i] not in p.parser.id_table_stack[-1]:
            p.parser.id_table_stack[-1][p[1][-i]] = {'classe': 'var',
                                                     'endereco': -i,
                                                     'tamanho': [1],
                                                     'tipo': 'int'}
        else:
            print("ERROR: Variable %s already declared locally."
            % p[1][-i],file=sys.stderr)
            p_error(p)
    p[0] = (True, p[1])
    p.parser.local_adress = 1
```

Listing 4.8: Derivação da declaração da função no caso em que existe retorno e parâmetros

Neste exemplo, pode-se reparar na forma como são introduzidas as variáveis na última tabela, a mais recente. Como as variáveis deste contexto são todas do tipo inteiro, serão idênticas nos valores que se introduzem, nomeadamente a classe, como *var*, o endereço, tendo em consideração a distância aos argumentos que serão colocados pela chamadora em relação ao novo apontador *fp*, e o tipo *int*, apenas aplicável para expansão futura.

4.2.3 Blocos de Código

As restantes derivações de *Block*, que também é derivado por *Code* mas que tem recursividade em si e é utilizado apenas no contexto do *Body*. As possibilidades para um *Block*, que representa maior parte das expressões do programa, são:

- *Exp ';'*, onde *Exp* pode ser qualquer expressão considerada mais básica e limitada;
- *FunCall ';'* que representa a chamada de uma função, devolva ou não resultados. Esta derivação serve especialmente para as funções que não devolvem valores para poderem ser invocadas;
- *If*, *IfElse*, *While* e *Switch* que representa a estrutura de controlo com o mesmo nome (com o *Switch* a nossa estrutura de controlo original).

A explicação do símbolo *FunCall* será feita no contexto do símbolo *Base*, que representa o fator mais simples de uma operação, já que as funções podem ser invocadas nesse contexto. No entanto, esse contexto vai apenas aceitar funções com retorno, enquanto que este existe especialmente para aquelas que não os devolvem, descartando o resultado se devolverem

```
def p_block_funcall(p):
    "Block : FunCall ';'"
    if p[1][1]:
        p[0] = p[1][0] + "pop 1\n"
    else:
        p[0] = p[1][0]
```

Listing 4.9: Produção de *FunCall* para *Block*

É de notar que, nesta derivação, considera-se que o valor definido não é numérico pois não será colocado em nenhuma variável nem guardado na *stack*, apesar de se aceitar instruções que sejam valores numéricos, que serão descartados.

If, IfElse

Estas estruturas de controlo são muito semelhantes e também o são codificadas. A sua lógica provém da manipulação de saltos e de *labels* no pseudo-código máquina. Para controlar a localidade dentro de cada bloco, utiliza-se a mesma lógica que na declaração de funções, embora não se redefina o endereço local porque as variáveis do âmbito local anterior ainda está a vigor. Neste caso, as palavras reservadas *if* e *else* representam a mudança de localidade. A condição definida é do tipo *AtribOp*, como serão também as restantes, uma produção que representa uma expressão númerica e será explicada mais à frente.

```
def p_if_scope(p):
    "IfScope : IF"
    p.parser.id_table_stack.append(dict())
    p[0] = p[1]


def p_if(p):
    "If : IfScope AtribOp Body"
    label = p.parser.internal_label
    p[0] = p[2] + \
        f"jz I{label}\n" + \
        p[3] + \
        f"I{label}:\n"
    p.parser.internal_label += 1

    p[0] += pop_local_vars(p)


def p_else_scope(p):
    "ElseScope : ELSE"
    pop_local_vars(p)  # pop do if
    p.parser.id_table_stack.append(dict())
    # limpar scope anterior (if anterior)
    p[0] = p[1]


def p_ifelse(p):
    "IfElse : IfScope AtribOp Body ElseScope Body"

    label = p.parser.internal_label
    p[0] = p[2] + f"jz I{label}\n" + \
```

```
32          p[3] + \
33          f"jump E{label}\n" + \
34          p[4] + \
35          f"I{label}:\n" + \
36          p[5] + \
37          f"E{label}:\n"
38      p.parser.internal_label += 1
39      #p[4] s o as local_vars do if
40      p[0] += pop_local_vars(p)  # pop do else
```

Listing 4.10: Produções subjacentes a *If* e *IfElse*

O salto funciona da seguinte forma: a condição é executada é o seu resultado deixado no topo da pilha, que será testado pelo primeiro salto, utilizando o *jz*. Se for falso, salta-se para o código de *else*/o fim da condição, dependendo de qual a estrutura. Se não, a execução continua e o primeiro corpo é executado. Notavelmente, como é utilizado *Body* em cada zona, é possível encadear diretamente sequências de *IfElse* sem a utilização de chavetas diretamente pelo *Block*.

While

De formas semelhante às instruções condicionais, a instrução de repetição *while* cuja lógica é proveniente da exploração de saltos e labels no pseudo-código máquina. O controlo da localidade semelhante com a ligeira diferença de que caso haja mais um ciclo temos de repetir o processo de criar uma nova localidade que no final do mesmo será novamente apagada através da função *pop_local_vars*. Temos, novamente, a condição a ser definida por um tipo *AtribOp* mencionado mais abaixo.

```
1  def p_while_scope(p):
2      "WhileScope : WHILE"
3      p.parser.id_table_stack.append(dict())
4      p[0] = p[1]
5
6  def p_while(p):
7      "While : WhileScope '(' AtribOp ')' Body"
8
9      lable_num = p.parser.internal_label
10
11     pop_local = pop_local_vars(p)
12
13     p[0] = f"W{lable_num}:\n" + \
14         p[3] + \
15         f"jz WE{lable_num}\n" + \
16         p[5] + \
17         pop_local + \
18         f"jump W{lable_num}\n" + \
19         f"WE{lable_num}:\n" + \
20         pop_local
21
22     p.parser.internal_label += 1
```

Listing 4.11: Produções correspondentes ao *while*

O pseudo-código máquina começa por marcar uma *label* que dita o primeiro passo do ciclo *while*, que é, efetivamente, testar a condição, caso a mesma falhe, recorremos ao *jz* para saltar para a marca que simboliza que o não estamos mais no ciclo e prosseguimos por "limpar"as variáveis locais. Caso contrário, estamos

elegíveis a entrar no corpo do ciclo. No final da execução do corpo, um salto incondicional repete este processo.

Switch

A nossa estrutura original resume-se a tentar colocar as condições de um *IfElse* juntas à cabeça de uma estrutura, com os blocos de código de seguida, marcados ou não por *labels* que os identifiquem.

Tal como as restantes estruturas de controlo, define-se um marcador de início de localidade, que neste caso poderá ser 2 símbolos diferentes, **SWITCHCOND** ou **SWITCHCASE**. Cada um dos casos terá comportamento diferente.

Para tal, percorrem-se as listas devolvidas em *Conds* ou em *Cases*, dependendo de qual a escolhida, que serão constituídas pelas *labels*, pela ordem que aparecem em cada uma das posições. No entanto, é também necessário uma forma de recordar quais as *labels* em utilização, no caso do aninhamento desta estrutura, utilizando então a stack de tuplos *label_table_stack*.

Cada componente destes tuplos é constituída por um dicionário que ora corresponde uma condição/código do caso para cada *label* definida, ora corresponde uma lista de condições/códigos dos casos para as que não foram definidas *labels*.

```
1  def p_switch(p):
2      # aqui eu ja tenho as duas tabelas
3      "Switch : SwitchScope Conds '{' Cases '}'"
4
5      cond_table = p.parser.label_table_stack[-1][0]
6      case_table = p.parser.label_table_stack[-1][1]
7
8      # testes de integridade das tabelas
9      if cond_table.keys() != case_table.keys():
10         print("ERROR: Condition labels don't match case labels",file=sys.stderr)
11         p_error(p)
12     if len(cond_table[':']) != len(case_table[':']):
13         print("ERROR: Number of unlabeled conditions doesn't match number of
       unlabeled cases",file=sys.stderr)
14         p_error(p)
15
16     end_label_num = p.parser.internal_label
17     p.parser.internal_label += 1
18
19     p[0] = ""
20
21     for label in p[p[1]]:  # percorrer ap[0]chamadas
22         lab_num = p.parser.internal_label
23         if label == ':':
24             cond = cond_table[':'].pop(0)
25             case = case_table[':'].pop(0)
26
27         else:
28             cond = cond_table[label]
29             case = case_table[label]
30
31         p[0] += cond + f"jz S{lab_num}\n" + case + \
32             f"jump SE{end_label_num}\n" + f"S{lab_num}:\n"
33         p.parser.internal_label += 1
```

```
34
35      p[0] += f"SE{end_label_num}:\n"
36      p[0] += pop_local_vars(p)
37      p.parser.label_table_stack.pop()  # tirar as duas tabelas da stack
```

Listing 4.12: Produção de *switch*

O código garante que o número de condições com/sem *label* seja o mesmo que os Blocos de código com/sem *label*, respetivamente.

De resto, as condições no pseudo-código máquina serão semelhantes ao *IfElse*, com uma única condição de término no caso de alguma condição se verificar e algum código for acedido. Deste modo, *Conds* deriva eventualmente numa sequência de condições separadas por vírgula que são em si também *AtribOp*.

```
1  def p_conds_rec(p):
2      "Conds : Conds ',' Cond"
3      p[0] = p[1]
4      p[0].append(p[3])
5
6
7  def p_conds_base(p):
8      "Conds : Cond"
9      p[0] = list(p[1])
10
11
12 def p_cond_id(p):
13     "Cond : ID '(' AtribOp ')'"
14     p.parser.label_table_stack[-1][0][p[1]] = p[3]
15     p[0] = p[1]
16
17
18 def p_cond_empty(p):
19     "Cond : '(' AtribOp ')'"
20     p.parser.label_table_stack[-1][0][':'].append(p[2])
21     p[0] = ':'
22
23
24 def p_cases_rec(p):
25     "Cases : Cases Case "
26     p[0] = p[1]
27     p[0].append(p[2])
28
29
30 def p_cases_base(p):
31     "Cases : Case"
32     p[0] = list(p[1])
33
34
35 def p_case_id(p):
36     "Case : ID ':' Body"
37     # preciso ver se ja tem la para dar erro
38     p.parser.label_table_stack[-1][1][p[1]] = p[3]
39     p[0] = p[1]  # ~acho que podemos ignorar isto mas whatever
40
41
42 def p_case_empty(p):
```

```
43      "Case : ':' Body"
44      # o par no label stack seria cond,case
45      p.parser.label_table_stack[-1][1][':'].append(p[2])
46      p[0] = ':'
```

Listing 4.13: Produções complementares do *switch*

4.2.4 Expressões

Uma expressão pode ser derivada em 5 tipos diferentes:

- *Atrib*
- *Decl*
- *DeclArray*
- *DeclAtrib*
- *Op*
- **Str PRINT**

A derivação de **Str PRINT** apenas gera código para poder escrever constantes derivadas em símbolos terminais **STRING**.

```
1  def p_exp_print(p):
2      "Exp : Str PRINT"
3      p[0] = "pushs " + p[1] + "\nwrites\n" + r'pushs "\n"' + "\nwrites\n"
4
5  def p_Str_Aspas(p):
6      "Str : '(' STRING ')'"
7      p[0] = p[2]
8
9  def p_Str_SemAspas(p):
10     "Str : STRING"
11     p[0] = p[1]
```

Listing 4.14: Derivação da escrita de constantes *string*

Temos a opção de escrever a string entre parênteses ou não.

Atrib

Numa atribuição estamos a atribuir um certo valor a uma variável, podendo ser à esquerda, à direita, dupla, ou, ainda, para um elemento de um *array*.
Como já sabemos, a sintaxe das atribuições é constituída por uma seta que aponta para o **ID** da variável que vai ser atribuída. Portanto, uma atribuição à esquerda é quando o **ID** está mais à esquerda (por isso, a seta terá a forma "<-", não esquecendo que o comprimento dos traços é variável), a atribuição à direita é exatamente o oposto (logo a seta será do estilo "->"). Uma dupla atribuição não é nada mais que um *swap* entre duas variáveis. Além das atribuições entre variáveis simples, podemos efetua-las numa variável mais complexa que é o *array*. Para tal, teremos de derivar o *Atrib* num *AtribArray* e este novo, pode derivar-se em atribuições à esquerda ou à direita.

```
def p_atrib_left(p):
    "Atrib : ID LARROW AtribOp"
    p[0] = p[3] + gen_atrib_code_stack(p, p[1], p[3])


def p_atrib_right(p):
    "Atrib : AtribOp RARROW ID"
    p[0] = p[1] + gen_atrib_code_stack(p, p[3], p[1])


def p_atrib_equiv(p):
    "Atrib : ID SWAP ID"


    flag1 = flag2 = True

    for i in range(len(p.parser.id_table_stack)-1, 0, -1):

        if flag1 and p[1] in p.parser.id_table_stack[i]:
            end = p.parser.id_table_stack[i][p[1]]['endereco']
            end1 = "pushl %d\n" % end
            store1 = "storel %d\n" % end
            flag1 = False
        if flag2 and p[3] in p.parser.id_table_stack[i]:
            end = p.parser.id_table_stack[i][p[3]]['endereco']
            end2 = "pushl %d\n" % end
            store2 = "storel %d\n" % end
            flag2 = False
        if not (flag1 or flag2):
            p[0] = end1 + end2 + store1 + store2
            return

    if flag1:
        if p[1] in p.parser.id_table_stack[0]:
            end = p.parser.id_table_stack[0][p[1]]['endereco']
            end1 = "pushg %d\n" % end
            store1 = "storeg %d\n" % end
        else:
            print("ERROR: Variable %s not in scope" % p[1],file=sys.stderr)
    if flag2:
        if p[3] in p.parser.id_table_stack[0]:
            end = p.parser.id_table_stack[0][p[3]]['endereco']
            end2 = "pushg %d\n" % end
            store2 = "storeg %d\n" % end
        else:
            print("ERROR: Variable %s not in scope" % p[3],file=sys.stderr)

    p[0] = end1 + end2 + store1 + store2

    return


def p_atrib_array(p):
```

```
    "Atrib : AtribArray"
    p[0] = p[1] + "pop 1\n"
```

Listing 4.15: Atrib

Ambas as *p_atrib_left* e *p_atrib_right* após irem buscar o código correspondente à expressão numérica ao *AtribOp*, utilizam a função *gen_atrib_code_stack* que nos produz o pseudo-código máquina de atribuição correto seguindo as regras de localidade.

A dupla atribuição é um pouco mais complexa, em vez de procurar por o endereço de uma variável, procuramos para duas. Ora, de forma a tornar o nosso código um pouco mais eficiente recorremos a *flags*, procurando evitar fazer dois ciclos for, a lógica é semelhante à da função *gen_atrib_code_stack*, se encontrarmos a variável dentro do for vamos, desta vez, vamos criar uma string com o *pushl* e o *storel*. Se não for encontrada no ciclo, então a variável não existe e, portanto, resulta em erro, ou é uma variável global e usa-mos os comandos *pushg* e *storeg*. Embora, a VMtenha um comando *swap*, não precisamos de recorrer a esse comando para efetuar a troca, pois após fazer *pushl*/*pushg* dos valores das variáveis, por ordem, basta fazer*storel*/*storeg* nessa mesma ordem.Desse modo, os valores das variáveis estarão trocados.

O último tipo de atribuição é a atribuição de *arrays*, a mesma estará explicada de forma mais específica na subsecção a seguir à explicação da função *gen_atrib_code_stack*. É de notar o valor que é retirado à pilha, pois o *AtribArray* é considerado como expressão numérica, enquanto que o *Atrib* não, então tem de se descartar esse valor.

Gerar código das Atribuições

A função *gen_atrib_code_stack* apenas gera código para identificadores que são da classe *variável*, efetuando a verificação se o identificador existir.

```
def gen_atrib_code_stack(p, id, atribop):
    s = ""
    for tamanho in range(len(p.parser.id_table_stack)-1, 0, -1):
        if id in p.parser.id_table_stack[tamanho]:
            if p.parser.id_table_stack[tamanho]['classe'] == 'var':
                s = "storel %d\n" % p.parser.id_table_stack[tamanho][id]['endereco']
                break
            else:
                print("ERROR: %s is not of variable class" % id,file=sys.stderr)
                p_error(p)
    else:
        if id not in p.parser.id_table_stack[0]:
            print("ERROR: Name %s not defined." % id,file=sys.stderr)
            p_error(p)
        else:
            if p.parser.id_table_stack[0]['classe'] == 'var':
                s = "storeg %d\n" % p.parser.id_table_stack[0][id]['endereco']
            else:
                print("ERROR: %s is not of variable class" % id,file=sys.stderr)
                p_error(p)
    return s
```

Listing 4.16: Função Gen_atrib_code_stack

Assim, dentro da função percorremos a *stack* de tabelas à procura da ocorrência de **ID** mais recente. Se for encontrado dentro do *for*, então quer dizer que é uma variável local e então utilizamos o comando do

pseudo-código máquina *storel*, senão temos duas opções: Ou é global e, por isso, recorremos ao *storeg*, ou não existe nenhuma variável com aquele **ID**, disparando um erro.

AtribArray — Acessos a *arrays*

A maneira como atribuímos para a esquerda ou para a direita tem uma sintaxe semelhante à atribuição entre variáveis, a diferença é que temos que indexar os índices do *array* junto com o seu **ID** da forma como explicamos na secção dos *arrays* no Capítulo 3.

Em primeiro lugar, é necessário saber qual o elemento a se aceder dentro do *array*, que existe na *stack* completamente consecutivo. Para tal, a produção de *ArraySize* dita a posição a aceder, mas em ordem contrária à escrita, algo útil durante o cálculo da posição.

```
def p_arraysize_rec(p):
    "ArraySize : ArraySize '[' AtribOp ']'"
    p[0] = p[3] + p[1]


def p_arraysize_empty(p):
    "ArraySize : '[' AtribOp ']'"
    p[0] = p[2]
```

Listing 4.17: Produção de *ArraySize*

```
def p_atribarray_Rightatribop(p):
    "AtribArray : AtribOp RARROW ID ArraySize"
    # 5+7 <-- x[2][5][4]
    # X[a][b][c]
    # X[x][y][z]
    # X + (x*c + y)*b + z
    # coloca o valor do atribop no topo da stack
    for i in range(len(p.parser.id_table_stack)-1, 0, -1):
        if p[3] in p.parser.id_table_stack[i]:
            if p.parser.id_table_stack[i][p[3]]['classe'] == 'array':
                endereco = p.parser.id_table_stack[i][p[3]]['endereco']
                s = "pushfp\n"
                sizes = p.parser.id_table_stack[i][p[3]]['tamanho'][1:]
                break
            else:
                print("ERROR: Variable %s is not of array type" % p[3],file=sys.
    stderr)
                p_error(p)
    else:
        if p[3] in p.parser.id_table_stack[0]:
            if p.parser.id_table_stack[0][p[3]]['classe'] == 'array':
                endereco = p.parser.id_table_stack[0][p[3]]['endereco']
                s = "pushgp\n"
                sizes = p.parser.id_table_stack[0][p[3]]['tamanho'][1:]
            else:
                print("ERROR: Variable %s is not of array type" % p[3],file=sys.
    stderr)
                p_error(p)
        else:
            print("ERROR: Variable %s not in scope" % p[3],file=sys.stderr)
            p_error(p)
```

```
30      if endereco != 0:
31          s += f"pushi {endereco}\npadd\n"
32      s += p[4]
33      for size in sizes:
34          s += f"pushi {size}\nmul\nadd\n"
35
36      end = p.parser.local_adress
37
38      p[0] = p[1] + s + f"pushl {end}\n" + "storen\n"
```

Listing 4.18: Uma das produções de AtribArray

A lógica da procura do endereço a partir do **ID** é a mesma que na atribuição de variáveis, porém utilizamos os comandos *pushfp* e *pushgp* para as variáveis locais e globais, respetivamente, acedendo assim a partir desses apontadores.

Para além disso, guardamos os tamanho de cada dimensão à exceção da primeira numa variável *sizes*. Após isso, se o endereço for diferente de zero, temos de calcular o endereço, somando ao valor do *pointer* o valor que guardamos no endereço, o que nos indica o endereço da primeira célula.

Após isso, vamos acrescentar à string os comandos gerados pelo *ArraySize*. A string é produzida no *ArraySize* é a junção de todos os *AtribOps*, em ordem inversa. Estes *AtribOps* são os índices do array que queremos aceder. Após isso, percorremos a variável *sizes* através de um ciclo *for*, a cada iteração colocamos o tamanho da dimensão no topo da *stack* com um *pushi*, multiplicamos com o primeiro índice que queremos aceder e somamos o resultado com o segundo índice.

Por exemplo para o *array* $a[4][4][4]$, se quisermos aceder $a[1][2][3]$, então a conta será: $(4 * 1 + 2) * 4 + 3$. Após isso, basta guardar o valor através de um *storen*.

Antes de retirar o valor que é calculado para a atribuição, colocado em $p[1]$, da pilha, realiza-se um *pushl* com o endereço atual da stack, o que irá colocar esse mesmo valor duplicado, garantindo a sua existência após o *storen* na pilha. Isto faz-se pois o *AtribArray* é considerado como expressão numérica no contexto do *AtribOp*, o que exige que mantenha o seu valor correspondente no topo.

Decl

Em *Decl* efetuam-se as declarações das variáveis, exclusivamente do tipo inteiro, com dois **ID** seguidamente colocados, o do tipo e o nome a dar à variável. A variável do *parser local_adress* é o próximo endereço onde será colocada a próxima variável, que deve ser incrementada.

A variável deve ser guardada na última tabela da *stack* das variáveis locais, podendo ser guardada no domínio global, que é sempre a posição 0.

```
1  def p_decl(p):
2      "Decl : ID ID"
3      if p[1].lower() not in p.parser.type_table:
4          print("ERROR : invalid type",file=sys.stderr)
5          p_error(p)
6      else:
7          p[0] = "pushi 0\n"
8          if len(p.parser.id_table_stack) == 1 and p[2] not in p.parser.id_table_stack
       [0]:
9              p.parser.id_table_stack[0][p[2]] = {'classe': 'var',
10                                                 'endereco': p.parser.global_adress,
11                                                 'tamanho': [1],
12                                                 'tipo': p[1].lower()}
```

```
            p.parser.global_adress += 1
            p.parser.local_adress += 1
        elif p[2] not in p.parser.id_table_stack[-1]:
            p.parser.id_table_stack[-1][p[2]] = {'classe': 'var',
                                                 'endereco': p.parser.local_adress,
                                                 'tamanho': [1],
                                                 'tipo': p[1].lower()}
            p.parser.local_adress += 1
        else:
            print("ERROR : Variable %s already declared locally." % p[2],file=sys.stderr)
            p_error(p)
```

Listing 4.19: Declaração de variáveis

A consideração mais importante a registar é a evolução da variável *local_adress* mesmo no âmbito global, já que, na eventualidade de existirem estruturas de controlo escritas globalmente, é necessário garantir continuidade da variável.

DeclArray

A declaração de *arrays* é semelhante à das variáveis, mas é necessário ter cuidado com a declaração do tamanho e do armazenamento desse na *stack* das tabelas. O pseudo-código máquina utilizado é o *pushn* com o tamanho sendo o produto de todas as dimensões, dimensões essas que serão guardadas numa lista na tabela. No entanto, esta implementação não permite que *arrays* sejam declarados com tamanhos vindos de variáveis, já que estes não são conhecidos a tempo de compilação.

```
def p_declarray(p):
    "DeclArray : ID ID DeclArraySize"
    # int x[1][1][2]
    if p[1].lower() not in p.parser.type_table:
        print("ERROR: invalid type",file=sys.stderr)
        p_error(p)
    else:
        res = 1
        for s in p[3]:
            if s <= 0:
                print("ERROR: Dimension non-positive for array %s" % p[2],file=sys.stderr)
                p_error(p)
            res *= s
        p[0] = f"pushn {res}\n"
        if len(p.parser.id_table_stack) == 1 and p[2] not in p.parser.id_table_stack[0]:
            p.parser.id_table_stack[0][p[2]] = {'classe': 'array',
                                                'endereco': p.parser.global_adress,
                                                'tamanho': p[3],
                                                'tipo': p[1]}
            p.parser.global_adress += res
            p.parser.local_adress += res
        elif p[2] not in p.parser.id_table_stack[-1]:
            p.parser.id_table_stack[-1][p[2]] = {'classe': 'array',
                                                 'endereco': p.parser.local_adress,
                                                 'tamanho': p[3],
```

```
                                                'tipo': p[1]}
            p.parser.local_adress += res
        else:
            print("ERROR : Variable %s already declared locally." % p[2],file=sys.
    stderr)
            p_error(p)
```

Listing 4.20: Declaração de arrays

As dimensões do array provém da produção que deriva em *DeclArraySize*, obrigatoriamente constantes numéricas positivas, nunca podendo ser omitidas.

```
def p_declarraysize_rec(p):
    "DeclArraySize : DeclArraySize '[' NUM ']'"
    p[0] = p[1]
    p[0].append(p[3])


def p_declarraysize_empty(p):
    "DeclArraySize : '[' NUM ']'"
    p[0] = [p[2]]
```

Listing 4.21: Derivação das dimensões de um array

DeclAtrib

Esta produção vai para além do pedido no enunciado, possibilitando a declaração e atribuição direta de variáveis, apenas do tipo inteiro. O único código gerado é o valor de *AtribOp*, que é colocado diretamente na *stack* na posição do topo que será exatamente o local da nova variável acabada de declarar.
No contexto da *stack* das tabelas da localidade, o funcionamento é o mesmo que na declaração.

```
def p_declarray(p):
    "DeclArray : ID ID DeclArraySize"
    # int x[1][1][2]
    if p[1].lower() not in p.parser.type_table:
        print("ERROR: invalid type",file=sys.stderr)
        p_error(p)
    else:
        res = 1
        for s in p[3]:
            if s <= 0:
                print("ERROR: Dimension non-positive for array %s" % p[2],file=sys.
    stderr)
                p_error(p)
            res *= s
        p[0] = f"pushn {res}\n"
        if len(p.parser.id_table_stack) == 1 and p[2] not in p.parser.id_table_stack
    [0]:
            p.parser.id_table_stack[0][p[2]] = {'classe': 'array',
                                                'endereco': p.parser.global_adress,
                                                'tamanho': p[3],
                                                'tipo': p[1]}
            p.parser.global_adress += res
            p.parser.local_adress += res
```

```
22          elif p[2] not in p.parser.id_table_stack[-1]:
23              p.parser.id_table_stack[-1][p[2]] = {'classe': 'array',
24                                                  'endereco': p.parser.local_adress,
25                                                  'tamanho': p[3],
26                                                  'tipo': p[1]}
27              p.parser.local_adress += res
28          else:
29              print("ERROR : Variable %s already declared locally." % p[2],file=sys.
    stderr)
30              p_error(p)
```

Listing 4.22: Uma das derivações de *DeclAtrib*

Op

As produções do símbolo não terminal *Op* são descritas na subsecção seguinte. Neste contexto de derivação para *Exp*, o seu valor é descartado apesar de calculado.

```
1 def p_exp_op(p):
2     "Exp : Op"
3     p[0] = p[1] + "pop 1\n"
```

Listing 4.23: Derivação de *Op* para *Exp*

4.2.5 AtribOp

Este símbolo não terminal já apareceu em maior parte das derivações anteriores, simbolizando uma expressão que, no seu final, tem um valor número que pode ser avaliado/atribuído. As suas derivações são tanto *Op* como *AtribNum*, uma forma reduzida do *Atrib*.

AtribNum

A produção do *AtribNum* são as mesmas que em *Atrib*, exceto o **SWAP** que não tem consenso para qual o valor que deve ser devolvido.

```
1  def p_atribnum_left(p):
2      "AtribNum : ID LARROW AtribOp"
3      p[0] = p[3] + "dup 1\n" + gen_atrib_code_stack(p, p[1], p[3])
4
5
6  def p_atribnum_right(p):
7      "AtribNum : AtribOp RARROW ID"
8      # 2+4->x++
9      p[0] = p[1] + "dup 1\n" + gen_atrib_code_stack(p, p[3], p[1])
10
11
12 def p_atribnum_array(p):
13     "AtribNum : AtribArray"
14     p[0] = p[1]
```

Listing 4.24: Produções de *AtribNum*

Op

A produção *Op* define as operações do programa. Estas operações são númericas por natureza, com um ou dois operandos, cumprindo então o requerimento que *AtribOp* seja uma produção com valor numérico. As operações aparecem em seguida por ordem crescente de prioridade numa única expressão:

- *OpBin* : Operações Lógicas, as primeiras operações binárias definidas;
- *TermMod* : Operação Módulo;
- *TermPlus* : Operações de soma/subtração;
- *TermMult* : Operações de multiplicação/divisão inteira;
- *TermPow* : Operação de exponenciação;
- *Base* : Fator base das operações, que pode constituir uma operação unária.

A operação de exponenciação é considerada como uma função que o compilador tem a responsabilidade de inserir no código. Quando a primeira ocorrência do operador aparecer, o *parser* consegue o código do ficheiro **??**, cuja função está escrita nos anexos.

```
def p_oppow(p):
    "OpPow : POW"

    fp_pow = open("pow.vm", "r")  # retorna erro se ficheiro nao existir

    pow_function_string = fp_pow.read()

    if p.parser.pow_flag:
        # adiciona ao buffer mas nao  a  tabela
        p.parser.function_buffer.append(pow_function_string)
        p.parser.pow_flag = False
    p[0] = "pusha P\ncall\npop 1\n"
```

Listing 4.25: Produção de *OpPow*

Para definir esta hierarquia, com recursividade à esquerda, para se efetuarem as operações da esquerda para a direita, as com maior prioridade são reconhecidas primeiro. Isso inclui o uso de parênteses para a definição de prioridade superior, reconhecido em *Base*.

```
def p_opbin_rec(p):
    "OpBin : OpBin OpLogico TermMod"
    p[0] = p[1] + p[3] + p[2]


def p_opbin_base(p):
    "OpBin : TermMod"
    p[0] = p[1]
```

Listing 4.26: Exemplo de uma das produções das operações

Base

O símbolo não terminal *Base* deriva em *AtribOp*, formando um ciclo onde expressões numéricas diferentes podem ser encadeadas.

```
def p_base_exp(p):
    "Base : '(' AtribOp ')'"
    p[0] = p[2]
```

As bases de uma operação podem ser tanto **ID** ou **NUM**, números por si. A derivação de **ID** para *Base* representa o acesso a uma variável, cuja classe deve ser testada,

```
def p_base_id(p):
    "Base : ID"
    for i in range(len(p.parser.id_table_stack)-1, 0, -1):
        if p[1] in p.parser.id_table_stack[i]:
            ele = p.parser.id_table_stack[i][p[1]]
            if ele['classe'] == 'var':
                p[0] = "pushl %d\n" % ele['endereco']
                return
            else:
                print("ERROR: %s is not a variable." % p[1],file=sys.stderr)
                p_error(p)
    if p[1] not in p.parser.id_table_stack[0]:
        print("ERROR: variable %s not in scope" % p[1],file=sys.stderr)
        p_error(p)
    elif p.parser.id_table_stack[0][p1]['classe'] == 'var' :
        p[0] = "pushg %d\n" % p.parser.id_table_stack[0][p[1]]['endereco']
    else:
        print("ERROR: %s is not a variable." % p[1],file=sys.stderr)
        p_error(p)
    return
```

Listing 4.27: Produção de *Base* para **ID**

Como podemos observar no excerto de código *Python*, fazemos uma procura de uma variável com o **ID** seguindo a lógica das localidades, com o adicional de testar se é uma variável simples, isto é, não é um *array*. A derivação de *Base* para **NUM** representa um simples *pushi* de um número inteiro.

```
def p_base_num(p):
    "Base : NUM"
    p[0] = "pushi %d\n" % p[1]
```

Listing 4.28: Produção de *Base* para **NUM**

A derivação de *Base* para **READ** representa a funcionalidade de *Input* do programa, que obrigatoriamente deve ser de inteiros, já que são a única variável que o programa trabalha com. Como está escrito dentro do contexto da *Base*, a funcionalidade de *input* pode ser utilizada como um fator de uma operação.

```
def p_base_read(p):
    "Base : READ"
    p[0] = "read\natoi\n"
```

Listing 4.29: Produção de *Base* para **READ**

FunCall

FunCall é o símbolo não terminal que representa a chamada de uma função, como já foi nomeado no contexto de *Block*. Esta exige o identificador da função a chamar, que deve existir na tabela de funções, e os argumentos que lhe serão passados. Esses argumentos serão derivados por *AtribOp*, o que permite atribuições na passagem de argumentos de funções.

```
def p_funarg_funrec(p):
    "FunArg : FunRec"
    p[0] = p[1]


def p_funarg_empty(p):
    "FunArg : "
    p[0] = []


def p_funrec_rec(p):
    "FunRec : FunRec ',' AtribOp"
    p[0] = p[1]
    p[0].append(p[3])


def p_funrec_base(p):
    "FunRec : AtribOp"
    p[0] = [p[1]]
```

Listing 4.30: Produções que representam os argumentos de uma chamada de função

A chamada da função deve colocar, por ordem de ocorrência dos argumentos, os valores associados aos parâmetros passados. Depois, deve colocar o endereço da função, que foi colocada no fim do código, através do *pusha* e da *label* guardada na tabela das funções.

O código da chamada também deve limpar os argumentos da função, que foram colocados na pilha, exceto um se existir valor de retorno.

A produção deve devolver o código da chamada, se a função devolve ou não um valor (para tratamento no contexto do *Base*) e o nome da função.

```
def p_funcall(p):
    "FunCall : ID '(' FunArg ')'"
    # print(p.parser.function_table)
    if p[1] not in p.parser.function_table:
        print("ERROR: Function %s not defined" % p[1],file=sys.stderr)
        p_error(p)
    label = p.parser.function_table[p[1]]['label']
    var_num = p.parser.function_table[p[1]]['num_args']
    if var_num == len(p[3]):
        p[0] = ("".join(p[3]) +
                f"pusha {label}\n" +
                "call\n" +
                f"pop {var_num-int(p.parser.function_table[p[1]]['return'])}\n",
                p.parser.function_table[p[1]]['return'],
                p[1])  # Nao esquecer de por o return em cima da primeira variavelF
    else:
        print("ERROR: Number of arguments given %d is not equal to needed %d, for
```

```
    function %s"
18                % (len(p[3]), var_num, p[1]),file=sys.stderr)
19          p_error(p)
```

Listing 4.31: Produção de *FunCall*

OpUno

As últimas operações definidas são as unárias, nomeadamente:

- **NEG** *Base* : Negação lógica de algum valor numérico base. A utilização de *Base* implica a obrigatoriedade de parênteses quando usado com expressões mais complexas: *AtribOp*;
- *AccessArray* : Acesso a um *array* para recolher o seu valor;
- **SUB** *Base* : O simétrico de um valor numérico. É feito a partir de um cálculo subtraido a 0 o valor;
- *Base* **PRINT** : Funcionalidade de *output* de um valor numérico.

```
1  def p_opuno_neg(p):
2      "OpUno : NEG Base"
3      p[0] = p[2] + 'not\n'
4
5
6  def p_opuno_accessarray(p):
7      "OpUno : AccessArray"
8      p[0] = p[1]
9
10
11 def p_opuno_minus(p):
12     "OpUno : SUB Base"
13     p[0] = "pushi 0\n" + p[2] + "sub\n"
14
15
16 def p_opuno_print(p):
17     "OpUno : Base PRINT"
18     p[0] = p[1] + "dup 1\n" + "writei\n" + r'pushs "\n"' + "\nwrites\n"
```

Listing 4.32: Produções de *OpUno*

AccessArray

O acesso a *arrays* é feito de forma análoga à do *atribarray* à exeção do *storen*, que no caso do acesso a *arrays* é subtituido por um *loadn*, garantindo assim que o valor numérico é colocado no topo da pilha.

```
1  def p_accessarray(p):
2      "AccessArray : ID ArraySize"
3      s = ""
4      for i in range(len(p.parser.id_table_stack)-1, 0, -1):
5          if p[1] in p.parser.id_table_stack[i]:
6              if p.parser.id_table_stack[i][p[1]]['classe'] == 'array':
7                  endereco = p.parser.id_table_stack[i][p[1]]['endereco']
```

```
                s = "pushfp\n"
                sizes = p.parser.id_table_stack[i][p[1]]['tamanho'][1:]
                break
            else:
                print("ERROR: Variable %s is not of array type" % p[1],file=sys.
    stderr)
                p_error(p)
     else:
         if p[1] in p.parser.id_table_stack[0]:
             if p.parser.id_table_stack[0][p[1]]['classe'] == 'array':
                 endereco = p.parser.id_table_stack[0][p[1]]['endereco']
                 s = "pushgp\n"
                 sizes = p.parser.id_table_stack[0][p[1]]['tamanho'][1:]
             else:
                 print("ERROR: Variable %s is not of array type" % p[1],file=sys.
    stderr)
                 p_error(p)
         else:
             print("ERROR: Variable %s not in scope" % p[1],file=sys.stderr)
             p_error(p)
     if endereco != 0:
         s += f"pushi {endereco}\npadd\n"
     s += p[2]
     for size in sizes:
         s += f"pushi {size}\nmul\nadd\n"
     p[0] = s + "loadn\n"
```

Listing 4.33: Produções de *OpUno*

4.3 Alternativas, Decisões e Problemas de Implementação

A nossa solução foi concebida permitindo uma rápida e facilitada expansão futura, tal como uma mais abrangente e flexível estrutura de escrita por parte do utilizador.

Decidimos que não se podem fazer redeclarações de variáveis, apesar de ser possível implementar, sendo necessário uma limpeza da pilha de cada variável retirada e atribuído novo espaço à redeclaração, já que a sua classe/tipo podem mudar.

Também decidimos que não é possível a declaração de funções dentro de outras estruturas de controlo, apenas em condições globais.

4.4 Testes realizados e Resultados

4.4.1 *Swap* com *scopes* diferentes

```
int x <- 10;
if(1){
    int y <- 0;
    y <- 12;
    x <-> y;
}
```

Listing 4.34: Teste de *Swap* com diferentes âmbitos

Este teste demonstra, não só a dupla atribuição, mas também a sua utilização com variáveis de localidade diferente. Além diso é um exemplo do uso de um *if*.

```
start
pushi 10
pushi 1
jz I0
pushi 0
pushi 12
storel 1
pushg 0
pushl 1
storeg 0
storel 1
I0:
pop 1
pushi 0
pushn 4
W2:
pushg 1
pushi 2
inf
jz WE2
pushi 0
W1:
pushl 6
pushi 2
inf
jz WE1
pushgp
pushi 2
padd
pushl 6
pushg 1
pushi 2
mul
add
loadn
dup 1
writei
pushs "\n"
writes
pop 1
pushl 6
pushi 1
add
storel 6
jump W1
WE1:
pop 0
pushg 1
pushi 1
add
storeg 1
jump W2
```

```
WE2:
pop 1
pop 2
stop
```

Listing 4.35: Código do teste anterior.

4.4.2 *Switch*

```
f : x,y ------> z
x-y+2+2->z;
int x;
switchcond x( (x<-f(x, 1)) >= 1), (<?) {

    : {
        f(x,1)>?;
    }

    x: x>?;
}
```

Listing 4.36: Teste da estrutura original *Switch*

Este é um teste do *switch* em que são mostradas condições com e sem *label*.
Além disso mostramos o resultado do código quando é usado *switchcond* e *switchcase*, respetivamente à direita e há esquerda. Podemos então ver que a ordem de execução é feita corretamente.
Para mais, mostra tanbém definição, e chamada de funções e a capacidade da linguagem de utilizar expressões complexas como condição. Adicionalmente, mostra a possibilidade de utilizar o mesmo nome para uma variável e uma label sem que isto cause problemas de execução.

```
start
pushi 0
pushg 0
pushi 1
pusha F0
call
pop 1
dup 1
storeg 0
pushi 1
supeq
jz S2
pushg 0
dup 1
writei
pushs "\n"
writes
pop 1
jump SE1
S2:
read
atoi
jz S3
pushg 0
pushi 1
pusha F0
call
pop 1
dup 1
writei
pushs "\n"
writes
pop 1
jump SE1
S3:
SE1:
pop 0
pop 1
stop
F0:
pushi 0
pushl -2
pushl -1
sub
pushi 2
add
pushi 2
add
storel 0
pushl 0
storel -2
pop 1
return
```

```
start
pushi 0
read
atoi
jz S2
pushg 0
pushi 1
pusha F0
call
pop 1
dup 1
writei
pushs "\n"
writes
pop 1
jump SE1
S2:
pushg 0
pushi 1
pusha F0
call
pop 1
dup 1
storeg 0
pushi 1
supeq
jz S3
pushg 0
dup 1
writei
pushs "\n"
writes
pop 1
jump SE1
S3:
SE1:
pop 0
pop 1
stop
F0:
pushi 0
pushl -2
pushl -1
sub
pushi 2
add
pushi 2
add
storel 0
pushl 0
storel -2
pop 1
return
```

Listing 4.37: Código do teste *Switchcond*

4.4.3 Declaração de um *array* multi-dimencional

```
int ar[12][12][10];

ar[1][2][3] <- 17;

(ar[1][2][3])>?;
```

Listing 4.39: Teste While e Array

Neste teste é feita uma declaração, atribuição e um acesso a um array com três dimenções. Podemos ver como o array é declarado como uma zona contigua de memória e como a aritmética de acesso é feita.

```
start
pushn 1440
pushi 17
pushgp
pushi 3
pushi 2
pushi 1
pushi 12
mul
add
pushi 10
mul
add
pushl 1440
storen
pop 1
pushgp
pushi 3
pushi 2
pushi 1
pushi 12
mul
add
pushi 10
mul
add
loadn
dup 1
writei
pushs "\n"
writes
pop 1
pop 1440
stop
```

Listing 4.40: Teste While e Array

4.4.4 Exemplo de um *array* e um *while*

Este é um exemplo simples de um ciclo *while*, em que a cada ciclo, é atribuído ao *array* no índice i o valor do mesmo.

```
int i;
int ar[4];

i <- 0;
while(i<4){
    ar[i] <- i;
    i <- i + 1;
}
```

Listing 4.41: Teste While e Array

```
start
pushi 0
pushn 4
pushi 0
storeg 0
W0:
pushg 0
pushi 4
inf
jz WE0
pushg 0
pushgp
pushi 1
padd
pushg 0
pushl 5
storen
pop 1
pushg 0
pushi 1
add
storeg 0
pop 0
jump W0
WE0:
pop 5
stop
```

Listing 4.42: pseudo-código máquina do While e Array

4.4.5 Exemplo da utilização do comando de exponenciação

Neste exemplo são executados dois comandos *pow* se forma a demonstrar os seguintes aspetos sobre este operador:
O código VMde potênciação apenas é colocado no ficheiro de output quando é utilizado. Todas as chamadas do comando são feitas através de um *call* de modo a não escrever o código máquina da potência mais do que uma vêz.

```
int x;
int y;

x <- 2^2;
```

```
y <- 3^2;
```

Listing 4.43: Teste Pow

```
start
pushi 0
pushi 0
pushi 2
pushi 2
pusha P
call
pop 1
storeg 0
pushi 3
pushi 2
pusha P
call
pop 1
storeg 1
pop 2
stop
P:
pushi 1
P1:
pushl -1
jz P2
pushl -2
pushl 0
mul
storel 0
pushl -1
pushi 1
sub
storel -1
jump P1
P2:
storel -2
return
```

Listing 4.44: pseudo-código máquina Teste Pow

4.4.6 Exemplo de *Input/Output* e Comentários

Aqui encontra-se um exemplo de input, output com strings. Além disso mostramos a utilização de comentários e mostramos o modo como estes são ignorados pelo compilador.

```
int x;

x <- <?;

x <- x + 2;

(x)>?;

```

```
"Fim do programa">?;

#
Codigo em comentario
if(10){
    x <- 0;
}
#
```

Listing 4.45: Teste IO e Comentários

```
start
pushi 0
read
atoi
storeg 0
pushg 0
pushi 2
add
storeg 0
pushg 0
dup 1
writei
pushs "\n"
writes
pop 1
pushs "Fim do programa"
writes
pushs "\n"
writes
pop 1
stop
```

Listing 4.46: pseudo-código máquina do Teste IO e Comentários

4.4.7 Exemplo de Aninhamento de *IfElse*

```
int d <- 1^0;
int y <- 2^0;
int x <- 0^2;

if (1>0) {
    int x <- <?;
    while ((x>?) != 2) {
        int y <- 1;
        x <- y + (x = 1);
    }
    if (1>0)
        (!0)>?;
    else
        1>?;
} else if (1>2) {
    int y;
```

```
    2>?;
} else if (1>0) {
    int z;
    if (5) {int w; 5>?;}
    3>?;
} else {
    4>?;
}

"lmao">?;
```

Listing 4.47: Teste de aninhamento de condicionais

Este teste tem em consideração a declaração de variáveis com o mesmo identificador em âmbitos diferentes e as considerações que se tem de ter para que não haja corrupção da *stack*.

```
start
pushi 1
pushi 0
pusha P
call
pop 1
pushi 2
pushi 0
pusha P
call
pop 1
pushi 0
pushi 2
pusha P
call
pop 1
pushi 1
pushi 0
sup
jz I5
read
atoi
W0:
pushl 3
dup 1
writei
pushs "\n"
writes
pushi 2
equal
not
jz WE0
pushi 1
pushl 4
pushl 3
pushi 1
equal
add
storel 3
```

```
pop 1
jump W0
WE0:
pushi 1
pushi 0
sup
jz I1
pushi 0
not
dup 1
writei
pushs "\n"
writes
pop 1
pop 0
jump E1
I1:
pushi 1
dup 1
writei
pushs "\n"
writes
pop 1
pop 0
E1:
pop 1
jump E5
I5:
pushi 1
pushi 2
sup
jz I4
pushi 0
pushi 2
dup 1
writei
pushs "\n"
writes
pop 1
pop 1
jump E4
I4:
pushi 1
pushi 0
sup
jz I3
pushi 0
pushi 5
jz I2
pushi 0
pushi 5
dup 1
writei
pushs "\n"
```

```
writes
pop 1
I2:
pop 1
pushi 3
dup 1
writei
pushs "\n"
writes
pop 1
pop 1
jump E3
I3:
pushi 4
dup 1
writei
pushs "\n"
writes
pop 1
pop 0
E3:
pop 0
E4:
pop 0
E5:
pushs "lmao"
writes
pushs "\n"
writes
pop 3
stop
P:
pushi 1
P1:
pushl -1
jz P2
pushl -2
pushl 0
mul
storel 0
pushl -1
pushi 1
sub
storel -1
jump P1
P2:
storel -2
return
```

Listing 4.48: Pseudo-código máquina do teste do aninhamento

4.5 Gramática da Linguagem

```
1
2       Axiom        :=  Axiom Start
3                    |   ε
4
5       Start        :=  Start Block
6                    |   Start Function
7                    |   Block
8                    |   Function
9
10      Code         :=  Code Block
11                   |   Block
12
13      Block        :=  Exp ';'
14                   |   FunCall ';'
15                   |   If
16                   |   IfElse
17                   |   While
18                   |   Switch
19
20      Body         :=  Block
21                   |   '{' Code '}'
22                   |   '{' '}'
23
24      FunctionHeader  :=  ID FunScope FunCases
25
26      Function     :=  FunctionHeader Body
27
28      FunScope     := ':'
29
30      FunCases     :=  FunExtra RARROW ID
31                   |   RARROW ID
32                   |   FunExtra
33                   |   ε
34
35      FunExtra     :=  FunExtra ',' ID
36                   |   ID
37
38      IfScope      :=  'if'
39
40      If           :=  IfScope AtribOp Body
41
42      ElseScope    := 'else'
43
44      IfElse       :=  IfScope AtribOp Body ElseScope Body
45
46      WhileScope   :=  'while'
47
48      While        :=  WhileScope AtribOp Body
49
50      SwitchScope  :=  'switchcond'
51
52      SwitchScope  :=  'switchcase'
```

```
53
54      Switch      :=  SwitchScope Conds '{' Cases '}'
55
56      Conds       :=  Conds ',' Cond
57                  |   Cond
58
59      Cond        :=  ID '(' AtribOp ')'
60                  |   '(' AtribOp ')'
61
62      Cases       :=  Cases Case
63                  |   Case
64
65      Case        :=  ID ':' Body
66                  |   ':' Body
67
68      Exp         :=  Atrib | Op | Decl | DeclArray | DeclAtrib | str PRINT
69
70      Str         :=  '(' STRING ')'
71                    | STRING
72
73      AtribOp     :=  AtribNum
74                  |   Op
75
76      Decl        :=  ID ID
77
78      DeclArray   :=  ID ID DeclArraySize
79
80      DeclArraySize   :=  DeclArraySize '[' NUM ']'
81                      |   '[' NUM ']'
82
83      AtribArray  :=  ID ArraySize LARROW AtribOp
84                  |   AtribOp RARROW ID ArraySize
85
86      ArraySize   :=  ArraySize '[' AtribOp ']'
87                  |   '[' AtribOp ']'
88
89      DeclAtrib   :=  ID ID LARROW AtribOp
90                  |   AtribOp RARROW ID ID
91
92      AtribNum    :=  ID LARROW AtribOp
93                  |   AtribOp RARROW ID
94                  |   AtribArray
95
96      Atrib       :=  ID LARROW AtribOp
97                  |   AtribOp RARROW ID
98                  |   ID SWAP ID
99                  |   AtribArray
100
101     Op          :=  OpBin
102
103     OpUno       :=  NEG Base
104                 |   AccessArray
105                 |   SUB Base
106                 |   Base PRINT
```

```
107
108     AccessArray :=  ID ArraySize
109
110     OpBin       :=  OpBin OpLogico TermMod
111                 |   TermMod
112
113     TermMod     :=  TermMod OpMod TermPlus
114                 |   TermPlus
115
116     TermPlus    :=  TermPlus OpPlus TermMult
117                 |   TermMult
118
119     TermMult    :=  TermMult OPMult TermPow
120                 |   TermPow
121
122     TermPow     :=  TermPow OpPow Base
123                 |   Base
124
125     Base        :=  '(' AtribOp ')' | ID | NUM | FunCall | READ | OpUno
126
127     FunCall     :=  ID '(' FunArg ')'
128
129     FunArg      :=  FunRec
130                 |   ε
131
132     FunRec      :=  FunRec ',' AtribOp
133                 |   AtribOp
134
135     ArrayAtrib  :=  ArrayNum ']'
136                 |   '[' ε ']'
137
138     ArrayNum    :=  '[' NUM
139                 |   ArrayNum ',' NUM
140
141     OpLogico    :=  AND | OR | LESSER | GREATER | LEQ | GEQ | EQUAL | DIFF
142
143     OpMais      :=  ADD | SUB
144
145     OpMult      :=  MUL | DIV
146
147     OPPow       :=  POW
```

Listing 4.49: Gramática

Capítulo 5

Conclusão

Este trabalho aborda a criação de uma linguagem de programação bem como a implementação de um compilador da mesma, através da construção da sua Gramática, de um Analisador Léxico e de um Sintático.

Neste relatório explicamos as decisões de design feitas ao longo do projeto bem como detalhes da implementação das mesmas.

Em retrospetiva, obtivemos uma linguagem elegante e liberal que implementa, na nossa opinião, muitos dos melhores aspetos de várias linguagens.

Consideramos que a linguagem é, no entanto, algo simplista e propomos como trabalho futuro os seguintes aspetos:

- A implementação de outros tipos de dados como floats, bem como a correta implementação das operações sobre estes e a implementação de operações de conversão entre tipos;
- A implementação de lógica de apontadores, com qualquer profundidade de indireção;
- Operações entre *arrays*.

Apêndice A

Código do Programa

A.1 Lex

```
import ply.lex as lex

import sys

tokens = (
'ID',
'NUM',
'STRING',
'RARROW',
'LARROW',
'SWAP',
'IF',
'ELSE',
'WHILE',
'SWITCHCOND',
'SWITCHCASE',
'NEG',
'AND',
'OR',
'LESSER',
'GREATER',
'LEQ',
'GEQ',
'EQUAL',
'DIFF',
'MOD',
'ADD',
'SUB',
'MUL',
'DIV',
'POW',
'READ',
'PRINT'
)

t_ANY_ignore = ' \n\t'
```

```
37
38 literals = ['(',')','[',']','{','}',':',',',';']
39
40 reserved = {
41     'if' : 'IF',
42     'else' : 'ELSE',
43     'while' : 'WHILE',
44     'switchcond' : 'SWITCHCOND',
45     'switchcase' : 'SWITCHCASE'
46 }
47
48 t_STRING = r'"[^"\n]*"'
49
50 t_RARROW = r'-+>'
51
52 t_LARROW = r'<-+'
53
54 t_SWAP = r'<-+>'
55
56 t_NEG = r'~|!'
57
58 t_AND = r'&'
59
60 t_OR = r'\|'
61
62 t_LESSER = r'<'
63
64 t_GREATER = r'>'
65
66 t_LEQ = r'<='
67
68 t_GEQ = r'>='
69
70 t_EQUAL = r'=+'
71
72 t_DIFF = r'!=+|~=+'
73
74 t_ADD = r'\+'
75
76 t_MOD = r'\%'
77
78 #O lex n o consegue apanhar o a -1 porque acha que (-1)    um n mero oops
79 t_SUB = r'-'
80
81 t_MUL = r'\*'
82
83 t_DIV = r'/'
84
85 t_POW = r'\^'
86
87 t_PRINT = r'\>\?'
88
89 t_READ = r'\<\?'
90
```

```
91 def t_ANY_error(t):
92     print('Illegal character: %s', t.value[0])
93 
94 def t_ID(t):
95     r'[a-zA-Z]\w*' # \w cont m o _ e n o queremos vars a come ar por _
96     t.type = reserved.get(t.value, 'ID')
97     return t
98 
99 def t_NUM(t):
100     r'[0-9]+'
101     t.value = int(t.value)
102     return t
103 
104 def t_ANY_COMMENT(t):
105     r'\#[^\#]*\#'
106     pass
107 
108 lexer = lex.lex()
109 
110 #for linha in sys.stdin:
111 #     lexer.input(linha)
112 #     tok = lexer.token()
113 #     while tok:
114 #         print(tok)
115 #         tok = lexer.token()
```

Listing A.1: Código pertencente ao lex

A.2 Parser

```
import ply.yacc as yacc
import sys
from lex import tokens
from math import prod

def p_axiom_start(p):
    "Axiom : Start"
    main = "pusha main\ncall\n" if p.parser.main else ""
    p.parser.final_code = "start\n" + \
        p[1] + main + pop_local_vars(p) + "stop\n" + \
        "".join(p.parser.function_buffer)


def p_start_empty(p):
    "Axiom : "
    p.parser.final_code = ""


def p_axiom_code(p):
    "Start : Start Block"
    #p[0] = "\n".join(p.parser.function_buffer) + "start" + p[1] + p[2] + "stop"
    p[0] = p[1] + p[2]


def p_axiom_function(p):
    "Start : Start Function"
    # p.parser.function_buffer
    p[0] = p[1]


def p_axiom_empty(p):
    "Start : "
    p[0] = ""


def p_code_block(p):
    "Code : Code Block"
    p[0] = p[1] + p[2]


def p_code_empty(p):
    "Code : Block"
    p[0] = p[1]


def p_block_funcall(p):
    "Block : FunCall ';'"
    if p[1][1]:
        p[0] = p[1][0] + "pop 1\n"
    else:
        p[0] = p[1][0]

```

```
53
54 def p_block_exp(p):
55     "Block : Exp ';'"
56     p[0] = p[1]
57
58
59 def p_block_if(p):
60     "Block : If "
61     p[0] = p[1]
62
63
64 def p_block_ifelse(p):
65     "Block : IfElse "
66     p[0] = p[1]
67
68
69 def p_block_while(p):
70     "Block : While "
71     p[0] = p[1]
72
73
74 def p_block_switch(p):
75     "Block : Switch "
76     p[0] = p[1]
77
78
79 # def p_block_empty(p):
80 #     "Block : "
81 #     p[0] = ""
82
83 def p_body_empty(p):
84     "Body : '{' '}'"
85     p[0] = ""
86
87
88 def p_body_block(p):
89     "Body : Block"
90     p[0] = p[1]
91
92
93 def p_body_code(p):
94     "Body : '{' Code '}'"
95     p[0] = p[2]
96
97 def p_function_header(p):
98     "FunctionHeader : ID FunScope FunCases"
99     label = p.parser.internal_label
100     num_args = len(p[3][1])
101
102     s_label = f"F{label}"
103     if p[1] == 'main':
104         p.parser.main = True
105         s_label = f"main"
106
```

```
107     p.parser.function_table[p[1]] = {'num_args': num_args,
108                                      'return': p[3][0],
109                                      'label': s_label}
110 
111     #Se a fun   o devolve ou n o algo. p[3]    um tuplo.
112 
113     p[0] = (p[3][0], num_args, s_label + ":\n")
114 
115 
116 def p_function(p):
117     "Function : FunctionHeader Body"
118     s = ""
119     local_args = len(p.parser.id_table_stack[-1])-int(p[1][1])
120     if local_args > 0:
121         #deixar se houver retorno e n o houver argumentos
122         s += "pop %d\n" % (local_args - int(p[1][1] == 0 and p[1][0]))
123     p.parser.local_adress = 0
124     p.parser.id_table_stack.pop()
125 
126     if p[1][0]:
127         p.parser.function_buffer.append(
128             p[1][2] + "pushi 0\n" + p[2] + \
129             f"pushl 0\nstorel {-p[1][1]}\n" + s + "return\n")
130     else:
131         p.parser.function_buffer.append(
132             p[1][2] + p[2] + s + "return\n")
133 
134     p[0] = ""
135     p.parser.internal_label += 1
136     p.parser.local_adress = p.parser.global_adress
137 
138 
139 def p_funscope(p):
140     "FunScope : ':'"
141     p.parser.id_table_stack.append(dict())
142     p[0] = p[1]
143 
144 
145 def p_funcases_funextra_rarrow(p):
146     "FunCases : FunExtra RARROW ID"
147     p.parser.id_table_stack[-1][p[3]] = {'classe': 'var',
148                                         'endereco': 0,
149                                         'tamanho': [1],
150                                         'tipo': 'int'}
151 
152     for i in range(1, len(p[1])+1):
153         if p[1][-i] not in p.parser.id_table_stack[-1]:
154             p.parser.id_table_stack[-1][p[1][-i]] = {'classe': 'var',
155                                                     'endereco': -i,
156                                                     'tamanho': [1],
157                                                     'tipo': 'int'}
158         else:
159             print("ERROR: Variable %s already declared locally."
160             % p[1][-i],file=sys.stderr)
```

```
161             p_error(p)
162     p[0] = (True, p[1])
163     p.parser.local_adress = 1
164
165
166 def p_funcases_rarrow(p):
167     "FunCases : RARROW ID"
168     p.parser.id_table_stack[-1][p[2]] = {'classe': 'var',
169                                         'endereco': 0,
170                                         'tamanho': [1],
171                                         'tipo': 'int'}
172     p[0] = (True, [])
173     p.parser.local_adress = 1
174
175
176 def p_funcases_funextra(p):
177     "FunCases : FunExtra"
178     for i in range(1, len(p[1])+1):
179         if p[1][-i] not in p.parser.id_table_stack[-1]:
180             p.parser.id_table_stack[-1][p[1][-i]] = {'classe': 'var',
181                                                     'endereco': -i,
182                                                     'tamanho': [1],
183                                                     'tipo': 'int'}
184         else:
185             print("ERROR: Variable %s already declared locally."
186             % p[1][-i],file=sys.stderr)
187             p_error(p)
188     p.parser.local_adress = 0
189     p[0] = (False, p[1])
190
191
192 def p_funcases_empty(p):
193     "FunCases : "
194     p.parser.local_adress = 0
195     p[0] = (False, [])
196
197
198 def p_funextra_rec(p):
199     "FunExtra : FunExtra ',' ID"
200     p[0] = p[1]
201     p[0].append(p[3])
202
203
204 def p_funextra_empty(p):
205     "FunExtra : ID"
206     p[0] = [p[1]]
207
208
209 def p_if_scope(p):
210     "IfScope : IF"
211     p.parser.id_table_stack.append(dict())
212     p[0] = p[1]
213
214
```

```
215 def p_if(p):
216     "If : IfScope AtribOp Body"
217     label = p.parser.internal_label
218     p[0] = p[2] + \
219         f"jz I{label}\n" + \
220         p[3] + \
221         f"I{label}:\n"
222     p.parser.internal_label += 1
223 
224     p[0] += pop_local_vars(p)
225 
226 
227 def p_else_scope(p):
228     "ElseScope : ELSE"
229     # limpar scope anterior (if anterior)
230     # pop do if
231     p[0] = pop_local_vars(p)
232     p.parser.id_table_stack.append(dict())
233 
234 
235 
236 def p_ifelse(p):
237     "IfElse : IfScope AtribOp Body ElseScope Body"
238     label = p.parser.internal_label
239     else_vars = pop_local_vars(p) #pop_else
240     p[0] = p[2] + f"jz I{label}\n" + \
241         p[3] + \
242         p[4] + \
243         f"jump E{label}\n" + \
244         f"I{label}:\n" + \
245         p[5] + \
246         else_vars + \
247         f"E{label}:\n"
248     p.parser.internal_label += 1
249     # p[4] s o as local_vars do if
250 
251 
252 def p_while_scope(p):
253     "WhileScope : WHILE"
254     p.parser.id_table_stack.append(dict())
255     p[0] = p[1]
256 
257 
258 def p_while(p):
259     "While : WhileScope '(' AtribOp ')' Body"
260 
261     lable_num = p.parser.internal_label
262 
263     pop_local = pop_local_vars(p)
264 
265     p[0] = f"W{lable_num}:\n" + \
266         p[3] + \
267         f"jz WE{lable_num}\n" + \
268         p[5] + \
```

```
269         pop_local + \
270         f"jump W{lable_num}\n" + \
271         f"WE{lable_num}:\n"
272 
273     p.parser.internal_label += 1
274 
275 
276 def p_switch_scopecond(p):
277     "SwitchScope : SWITCHCOND"
278     p.parser.id_table_stack.append(dict())
279     p.parser.label_table_stack.append(
280         (  # -----isto  e  um tuplo
281             {':': list()},
282             {':': list()}
283         )  # -----
284     )  # cond,cases
285     # inicializar com o caracter especial e uma lista vazia
286     p[0] = 2
287 
288 def p_switch_scopecase(p):
289     "SwitchScope : SWITCHCASE"
290     p.parser.id_table_stack.append(dict())
291     p.parser.label_table_stack.append(
292         (  # -----isto  e  um tuplo
293             {':': list()},
294             {':': list()}
295         )  # -----
296     )  # cond,cases
297     # inicializar com o caracter especial e uma lista vazia
298     p[0] = 4
299 
300 
301 def p_switch(p):
302     # aqui eu ja tenho as duas tabelas
303     "Switch : SwitchScope Conds '{' Cases '}'"
304 
305     cond_table = p.parser.label_table_stack[-1][0]
306     case_table = p.parser.label_table_stack[-1][1]
307 
308     # testes de integridade das tabelas
309     if cond_table.keys() != case_table.keys():
310         print("ERROR: Condition labels don't match case labels",file=sys.stderr)
311         p_error(p)
312     if len(cond_table[':']) != len(case_table[':']):
313         # estou a fazer com que todas as condi oes apare am e sejam chamadas uma
    vez (pode ser mudado)
314         print("ERROR: Number of unlabeled conditions doesn't match number of
    unlabeled cases",file=sys.stderr)
315         p_error(p)
316 
317     end_label_num = p.parser.internal_label
318     p.parser.internal_label += 1
319 
320     p[0] = ""
```

```
321
322     for label in p[p[1]]:  # percorrer ap[0]chamadas
323         lab_num = p.parser.internal_label
324         if label == ':':
325             cond = cond_table[':'].pop(0)
326             case = case_table[':'].pop(0)
327
328         else:
329             cond = cond_table[label]
330             case = case_table[label]
331
332         p[0] += cond + f"jz S{lab_num}\n" + case + \
333             f"jump SE{end_label_num}\n" + f"S{lab_num}:\n"
334         p.parser.internal_label += 1
335
336     p[0] += f"SE{end_label_num}:\n"
337     p[0] += pop_local_vars(p)
338     p.parser.label_table_stack.pop()  # tirar as duas tabelas da stack
339
340
341 # nas conds passar para cima um par (conds_com_lable (dict?), conds_sem_lable (lista
        ?))
342 def p_conds_rec(p):
343     "Conds : Conds ',' Cond"
344     p[0] = p[1]
345     p[0].append(p[3])
346
347
348 def p_conds_base(p):
349     "Conds : Cond"
350     p[0] = list(p[1])
351
352
353 def p_cond_id(p):
354     "Cond : ID '(' AtribOp ')'"
355     p.parser.label_table_stack[-1][0][p[1]] = p[3]
356     p[0] = p[1]
357
358
359 def p_cond_empty(p):
360     "Cond : '(' AtribOp ')'"
361     p.parser.label_table_stack[-1][0][':'].append(p[2])
362     p[0] = ':'
363
364
365 def p_cases_rec(p):
366     "Cases : Cases Case "
367     p[0] = p[1]
368     p[0].append(p[2])
369
370
371 def p_cases_base(p):
372     "Cases : Case"
373     p[0] = list(p[1])
```

```
374
375
376 def p_case_id(p):
377     "Case : ID ':' Body"
378     # preciso ver se ja tem la para dar erro
379     if p[1] not in p.parser.label_table_stack[-1][0]:
380         print("ERROR: %s not label in current scope" % p[1],file=sys.stderr)
381         p_error(p)
382     p.parser.label_table_stack[-1][1][p[1]] = p[3]
383     p[0] = p[1]  # ~acho que podemos ignorar isto mas whatever
384
385
386 def p_case_empty(p):
387     "Case : ':' Body"
388     # o par no label stack seria cond,case
389     p.parser.label_table_stack[-1][1][':'].append(p[2])
390     p[0] = ':'
391
392 def p_exp_print(p):
393     "Exp : Str PRINT"
394     # funciona para tudo que n o seja array
395     p[0] = "pushs " + p[1] + "\nwrites\n" + r'pushs "\n"' + "\nwrites\n"
396
397 def p_Str_Aspas(p):
398     "Str : '(' STRING ')'"
399     p[0] = p[2]
400
401 def p_Str_SemAspas(p):
402     "Str : STRING"
403     p[0] = p[1]
404
405 def p_exp_atrib(p):
406     "Exp : Atrib"
407     p[0] = p[1]
408
409
410 def p_exp_op(p):
411     "Exp : Op"
412     p[0] = p[1] + "pop 1\n"
413
414
415 def p_exp_decl(p):
416     "Exp : Decl"
417     p[0] = p[1]
418
419
420 def p_exp_declarray(p):
421     "Exp : DeclArray"
422     p[0] = p[1]
423
424
425 def p_exp_declatrib(p):
426     "Exp : DeclAtrib"
427     p[0] = p[1]
```

```
428
429
430 def p_atribop_atribnum(p):
431     "AtribOp : AtribNum"
432     p[0] = p[1]
433
434
435 def p_atribop_op(p):
436     "AtribOp : Op"
437     p[0] = p[1]
438
439
440 def p_decl(p):
441     "Decl : ID ID"
442     if p[1].lower() not in p.parser.type_table:
443         print("ERROR : invalid type",file=sys.stderr)
444         p_error(p)
445     else:
446         p[0] = "pushi 0\n"
447         if len(p.parser.id_table_stack) == 1 and p[2] not in p.parser.id_table_stack
    [0]:
448             p.parser.id_table_stack[0][p[2]] = {'classe': 'var',
449                                                'endereco': p.parser.global_adress,
450                                                'tamanho': [1],
451                                                'tipo': p[1].lower()}
452             p.parser.global_adress += 1
453             p.parser.local_adress += 1
454         elif p[2] not in p.parser.id_table_stack[-1]:
455             p.parser.id_table_stack[-1][p[2]] = {'classe': 'var',
456                                                 'endereco': p.parser.local_adress,
457                                                 'tamanho': [1],
458                                                 'tipo': p[1].lower()}
459             p.parser.local_adress += 1
460         else:
461             print("ERROR : Variable %s already declared locally." % p[2],file=sys.
    stderr)
462             p_error(p)
463
464
465 def p_declarray(p):
466     "DeclArray : ID ID DeclArraySize"
467     # int x[1][1][2]
468     if p[1].lower() not in p.parser.type_table:
469         print("ERROR: invalid type",file=sys.stderr)
470         p_error(p)
471     else:
472         res = 1
473         for s in p[3]:
474             if s <= 0:
475                 print("ERROR: Dimension non-positive for array %s" % p[2],file=sys.
    stderr)
476                 p_error(p)
477             res *= s
478         p[0] = f"pushn {res}\n"
```

```
479         if len(p.parser.id_table_stack) == 1 and p[2] not in p.parser.id_table_stack
    [0]:
480             p.parser.id_table_stack[0][p[2]] = {'classe': 'array',
481                                                'endereco': p.parser.global_adress,
482                                                'tamanho': p[3],
483                                                'tipo': p[1]}
484             p.parser.global_adress += res
485             p.parser.local_adress += res
486         elif p[2] not in p.parser.id_table_stack[-1]:
487             p.parser.id_table_stack[-1][p[2]] = {'classe': 'array',
488                                                 'endereco': p.parser.local_adress,
489                                                 'tamanho': p[3],
490                                                 'tipo': p[1]}
491             p.parser.local_adress += res
492         else:
493             print("ERROR : Variable %s already declared locally." % p[2],file=sys.
    stderr)
494             p_error(p)
495
496
497 def p_declarraysize_rec(p):
498     "DeclArraySize : DeclArraySize '[' NUM ']'"
499     p[0] = p[1]
500     p[0].append(p[3])
501
502
503 def p_declarraysize_empty(p):
504     "DeclArraySize : '[' NUM ']'"
505     p[0] = [p[2]]
506
507
508 def p_atribarray_Leftatribop(p):
509     "AtribArray : ID ArraySize LARROW AtribOp"
510     for i in range(len(p.parser.id_table_stack)-1, 0, -1):
511         if p[1] in p.parser.id_table_stack[i]:
512             if p.parser.id_table_stack[i][p[1]]['classe'] == 'array':
513                 endereco = p.parser.id_table_stack[i][p[1]]['endereco']
514                 s = "pushfp\n"
515                 sizes = p.parser.id_table_stack[i][p[1]]['tamanho'][1:]
516                 break
517             else:
518                 print("ERROR: Variable %s is not of array type" % p[1],file=sys.
    stderr)
519                 p_error(p)
520     else:
521         if p[1] in p.parser.id_table_stack[0]:
522             if p.parser.id_table_stack[0][p[1]]['classe'] == 'array':
523                 endereco = p.parser.id_table_stack[0][p[1]]['endereco']
524                 s = "pushgp\n"
525                 sizes = p.parser.id_table_stack[0][p[1]]['tamanho'][1:]
526             else:
527                 print("ERROR: Variable %s is not of array type" % p[1],file=sys.
    stderr)
528                 p_error(p)
```

```
529         else:
530             print("ERROR: Variable %s not in scope" % p[1],file=sys.stderr)
531             p_error(p)
532     if endereco != 0:
533         s += f"pushi {endereco}\npadd\n"
534     s += p[2]
535     for size in sizes:
536         s += f"pushi {size}\nmul\nadd\n"
537 
538     end = p.parser.local_adress
539 
540     p[0] = p[4] + s + f"pushl {end}\n" + "storen\n"
541 
542 
543 def p_atribarray_Rightatribop(p):
544     "AtribArray : AtribOp RARROW ID ArraySize"
545     # 5+7 <—— x[2][5][4]
546     # X[a][b][c]
547     # X[x][y][z]
548     # X + (x*c + y)*b + z
549     # coloca o valor do atribop no topo da stack
550     for i in range(len(p.parser.id_table_stack)-1, 0, -1):
551         if p[3] in p.parser.id_table_stack[i]:
552             if p.parser.id_table_stack[i][p[3]]['classe'] == 'array':
553                 endereco = p.parser.id_table_stack[i][p[3]]['endereco']
554                 s = "pushfp\n"
555                 sizes = p.parser.id_table_stack[i][p[3]]['tamanho'][1:]
556                 break
557             else:
558                 print("ERROR: Variable %s is not of array type" % p[3],file=sys.
    stderr)
559                 p_error(p)
560     else:
561         if p[3] in p.parser.id_table_stack[0]:
562             if p.parser.id_table_stack[0][p[3]]['classe'] == 'array':
563                 endereco = p.parser.id_table_stack[0][p[3]]['endereco']
564                 s = "pushgp\n"
565                 sizes = p.parser.id_table_stack[0][p[3]]['tamanho'][1:]
566             else:
567                 print("ERROR: Variable %s is not of array type" % p[3],file=sys.
    stderr)
568                 p_error(p)
569         else:
570             print("ERROR: Variable %s not in scope" % p[3],file=sys.stderr)
571             p_error(p)
572     if endereco != 0:
573         s += f"pushi {endereco}\npadd\n"
574     s += p[4]
575     for size in sizes:
576         s += f"pushi {size}\nmul\nadd\n"
577 
578     end = p.parser.local_adress
579 
580     p[0] = p[1] + s + f"pushl {end}\n" + "storen\n"
```

```
581
582
583 def p_accessarray(p):
584     "AccessArray : ID ArraySize"
585     s = ""
586     for i in range(len(p.parser.id_table_stack)-1, 0, -1):
587         if p[1] in p.parser.id_table_stack[i]:
588             if p.parser.id_table_stack[i][p[1]]['classe'] == 'array':
589                 endereco = p.parser.id_table_stack[i][p[1]]['endereco']
590                 s = "pushfp\n"
591                 sizes = p.parser.id_table_stack[i][p[1]]['tamanho'][1:]
592                 break
593             else:
594                 print("ERROR: Variable %s is not of array type" % p[1],file=sys.
    stderr)
595                 p_error(p)
596     else:
597         if p[1] in p.parser.id_table_stack[0]:
598             if p.parser.id_table_stack[0][p[1]]['classe'] == 'array':
599                 endereco = p.parser.id_table_stack[0][p[1]]['endereco']
600                 s = "pushgp\n"
601                 sizes = p.parser.id_table_stack[0][p[1]]['tamanho'][1:]
602             else:
603                 print("ERROR: Variable %s is not of array type" % p[1],file=sys.
    stderr)
604                 p_error(p)
605         else:
606             print("ERROR: Variable %s not in scope" % p[1],file=sys.stderr)
607             p_error(p)
608     if endereco != 0:
609         s += f"pushi {endereco}\npadd\n"
610     s += p[2]
611     for size in sizes:
612         s += f"pushi {size}\nmul\nadd\n"
613     p[0] = s + "loadn\n"
614
615
616 def p_arraysize_rec(p):
617     "ArraySize : ArraySize '[' AtribOp ']'"
618     p[0] = p[3] + p[1]  # array nao  e  uma lista
619
620
621 def p_arraysize_empty(p):
622     "ArraySize : '[' AtribOp ']'"
623     p[0] = p[2]
624
625
626 def p_declatrib_left(p):
627     "DeclAtrib : ID ID LARROW AtribOp"
628     # int x <---------------------- 5+8
629     if p[1].lower() not in p.parser.type_table:
630         print("ERROR: invalid type",file=sys.stderr)
631         p_error(p)
632     else:
```

```
633         p[0] = p[4]
634         if len(p.parser.id_table_stack) == 1 and p[2] not in parser.id_table_stack
    [0]:
635             p.parser.id_table_stack[0][p[2]] = {'classe': 'var',
636                                                'endereco': p.parser.local_adress,
637                                                'tamanho': [1],
638                                                'tipo': p[1]}
639             p.parser.global_adress += 1
640             p.parser.local_adress += 1
641         elif p[2] not in parser.id_table_stack[-1]:
642             p.parser.id_table_stack[-1][p[2]] = {'classe': 'var',
643                                                 'endereco': p.parser.local_adress,
644                                                 'tamanho': [1],
645                                                 'tipo': p[1]}
646             p.parser.local_adress += 1
647         else:
648             print("ERROR: Variable %s already declared locally" % p[2],file=sys.
    stderr)
649             p_error(p)
650 
651 
652 def p_declatrib_right(p):
653     "DeclAtrib : AtribOp RARROW ID ID"
654     # 7+5 -> int  INT Int iNt x? #####vao permitir isto???#####
655     if p[3].lower() not in p.parser.type_table:
656         print("ERROR: invalid type",file=sys.stderr)
657     else:
658         p[0] = p[1]
659         if len(p.parser.id_table_stack) == 1:
660             p.parser.id_table_stack[0][p[4]] = {'classe': 'var',
661                                                'endereco': p.parser.global_adress,
662                                                'tamanho': [1],
663                                                'tipo': p[3]}
664             p.parser.global_adress += 1
665             p.parser.local_adress += 1
666         else:
667             p.parser.id_table_stack[-1][p[4]] = {'classe': 'var',
668                                                 'endereco': p.parser.local_adress,
669                                                 'tamanho': [1],
670                                                 'tipo': p[3]}
671             p.parser.local_adress += 1
672 
673 
674 def p_atribnum_left(p):
675     "AtribNum : ID LARROW AtribOp"
676     p[0] = p[3] + "dup 1\n" + gen_atrib_code_stack(p, p[1], p[3])
677 
678 
679 def p_atribnum_right(p):
680     "AtribNum : AtribOp RARROW ID"
681     # 2+4->x++
682     p[0] = p[1] + "dup 1\n" + gen_atrib_code_stack(p, p[3], p[1])
683 
684 
```

```
685 def p_atribnum_array(p):
686     "AtribNum : AtribArray"
687     p[0] = p[1]
688 
689 
690 def p_atrib_left(p):
691     "Atrib : ID LARROW AtribOp"
692     p[0] = p[3] + gen_atrib_code_stack(p, p[1], p[3])
693 
694 
695 def p_atrib_right(p):
696     "Atrib : AtribOp RARROW ID"
697     p[0] = p[1] + gen_atrib_code_stack(p, p[3], p[1])
698 
699 
700 def p_atrib_equiv(p):
701     "Atrib : ID SWAP ID"
702 
703 
704     flag1 = flag2 = True
705 
706     for i in range(len(p.parser.id_table_stack)-1, 0, -1):
707 
708         if flag1 and p[1] in p.parser.id_table_stack[i]:
709             end = p.parser.id_table_stack[i][p[1]]['endereco']
710             end1 = "pushl %d\n" % end
711             store1 = "storel %d\n" % end
712             flag1 = False
713         if flag2 and p[3] in p.parser.id_table_stack[i]:
714             end = p.parser.id_table_stack[i][p[3]]['endereco']
715             end2 = "pushl %d\n" % end
716             store2 = "storel %d\n" % end
717             flag2 = False
718         if not (flag1 or flag2):
719             p[0] = end1 + end2 + store1 + store2
720             return
721 
722     if flag1:
723         if p[1] in p.parser.id_table_stack[0]:
724             end = p.parser.id_table_stack[0][p[1]]['endereco']
725             end1 = "pushg %d\n" % end
726             store1 = "storeg %d\n" % end
727         else:
728             print("ERROR: Variable %s not in scope" % p[1],file=sys.stderr)
729     if flag2:
730         if p[3] in p.parser.id_table_stack[0]:
731             end = p.parser.id_table_stack[0][p[3]]['endereco']
732             end2 = "pushg %d\n" % end
733             store2 = "storeg %d\n" % end
734         else:
735             print("ERROR: Variable %s not in scope" % p[3],file=sys.stderr)
736 
737     p[0] = end1 + end2 + store1 + store2
738 
```

```
739     return
740
741
742 def p_atrib_array(p):
743     "Atrib : AtribArray"
744     p[0] = p[1] + "pop 1\n"
745
746
747 # def p_op_opuno(p):
748 #     "Op : OpUno"
749 #     p[0] = p[1]
750
751
752 def p_op_opbin(p):
753     "Op : OpBin"
754     p[0] = p[1]
755
756
757 def p_opuno_neg(p):
758     "OpUno : NEG Base"
759     p[0] = p[2] + 'not\n'
760
761
762 def p_opuno_accessarray(p):
763     "OpUno : AccessArray"
764     p[0] = p[1]
765
766
767 def p_opuno_minus(p):
768     "OpUno : SUB Base"
769     p[0] = "pushi 0\n" + p[2] + "sub\n"
770
771
772 def p_opuno_print(p):
773     "OpUno : Base PRINT"
774     # funciona para tudo que n o seja array
775     p[0] = p[1] + "dup 1\n" + "writei\n" + r'pushs "\n"' + "\nwrites\n"
776
777
778
779 def p_opbin_rec(p):
780     "OpBin : OpBin OpLogico TermMod"
781     p[0] = p[1] + p[3] + p[2]
782
783
784 def p_opbin_base(p):
785     "OpBin : TermMod"
786     p[0] = p[1]
787
788
789 def p_opmod_rec(p):
790     "TermMod : TermMod OpMod TermPlus"
791     p[0] = p[1] + p[3] + p[2]
792
```

```
793
794 def p_opmod_base(p):
795     "TermMod : TermPlus"
796     p[0] = p[1]
797
798
799 def p_termplus_rec(p):
800     "TermPlus : TermPlus OpPlus TermMult"
801     p[0] = p[1] + p[3] + p[2]
802
803
804 def p_termplus_base(p):
805     "TermPlus : TermMult"
806     p[0] = p[1]
807
808
809 def p_termmult_rec(p):
810     "TermMult : TermMult OpMult TermPow"
811     p[0] = p[1] + p[3] + p[2]
812
813
814 def p_termmult_base(p):
815     "TermMult : TermPow"
816     p[0] = p[1]
817
818
819 def p_termpow_rec(p):
820     "TermPow : TermPow OpPow Base"
821     p[0] = p[1] + p[3] + p[2]
822
823
824 def p_termpow_base(p):
825     "TermPow : Base"
826     p[0] = p[1]
827
828
829 def p_base_opuno(p):
830     "Base : OpUno"
831     p[0] = p[1]
832
833
834 def p_base_exp(p):
835     "Base : '(' AtribOp ')'"
836     p[0] = p[2]
837
838
839 def p_base_id(p):
840     "Base : ID"
841     for i in range(len(p.parser.id_table_stack)-1, 0, -1):
842         if p[1] in p.parser.id_table_stack[i]:
843             ele = p.parser.id_table_stack[i][p[1]]
844             if ele['classe'] == 'var':
845                 p[0] = "pushl %d\n" % ele['endereco']
846                 return
```

```
            else:
                print("ERROR: %s is not a variable." % p[1],file=sys.stderr)
                p_error(p)
    if p[1] not in p.parser.id_table_stack[0]:
        print("ERROR: variable %s not in scope" % p[1],file=sys.stderr)
        p_error(p)
    elif p.parser.id_table_stack[0][p[1]]['classe'] == 'var' :
        p[0] = "pushg %d\n" % p.parser.id_table_stack[0][p[1]]['endereco']
    else:
        print("ERROR: %s is not a variable." % p[1],file=sys.stderr)
        p_error(p)
    return


def p_base_num(p):
    "Base : NUM"
    p[0] = "pushi %d\n" % p[1]


def p_base_funcall(p):
    "Base : FunCall"
    if (p[1][1]):
        p[0] = p[1][0]
    else:
        print("ERROR: Function %s doesn't return any value" % p[1][2] +
              " and is used in an operation.",file=sys.stderr)
        p_error(p)


def p_base_read(p):
    "Base : READ"
    p[0] = "read\natoi\n"


def p_funcall(p):
    "FunCall : ID '(' FunArg ')'"
    # print(p.parser.function_table)
    if p[1] not in p.parser.function_table:
        print("ERROR: Function %s not defined" % p[1],file=sys.stderr)
        p_error(p)
    label = p.parser.function_table[p[1]]['label']
    var_num = p.parser.function_table[p[1]]['num_args']
    if var_num == len(p[3]):
        p[0] = ("".join(p[3]) +
                f"pusha {label}\n" +
                "call\n" +
                f"pop {var_num-int(p.parser.function_table[p[1]]['return'])}\n",
                p.parser.function_table[p[1]]['return'],
                p[1])  # Nao esquecer de por o return em cima da primeira variavelF
    else:
        print("ERROR: Number of arguments given %d is not equal to needed %d, for
    function %s"
              % (len(p[3]), var_num, p[1]),file=sys.stderr)
        p_error(p)
```

```
900
901
902 def p_funarg_funrec(p):
903     "FunArg : FunRec"
904     p[0] = p[1]
905
906
907 def p_funarg_empty(p):
908     "FunArg : "
909     p[0] = []
910
911
912 def p_funrec_rec(p):
913     "FunRec : FunRec ',' AtribOp"
914     p[0] = p[1]
915     p[0].append(p[3])
916
917
918 def p_funrec_base(p):
919     "FunRec : AtribOp"
920     p[0] = [p[1]]
921
922
923 def p_oplogico_diff(p):
924     "OpLogico : DIFF"
925     p[0] = "equal\nnot\n"
926
927 def p_oplogico_and(p):
928     "OpLogico : AND"
929     p[0] = "and\n"
930
931
932 def p_oplogico_or(p):
933     "OpLogico : OR"
934     p[0] = "or\n"
935
936
937 def p_oplogico_lesser(p):
938     "OpLogico : LESSER"
939     p[0] = "inf\n"
940
941
942 def p_oplogico_greater(p):
943     "OpLogico : GREATER"
944     p[0] = "sup\n"
945
946
947 def p_oplogico_leq(p):
948     "OpLogico : LEQ"
949     p[0] = "infeq\n"
950
951
952 def p_oplogico_geq(p):
953     "OpLogico : GEQ"
```

```
954     p[0] = "supeq\n"
955
956
957 def p_oplogico_equal(p):
958     "OpLogico : EQUAL"
959     p[0] = "equal\n"
960
961 def p_opmod_mod(p):
962     "OpMod : MOD"
963     p[0] = "mod\n"
964
965 def p_opplus_add(p):
966     "OpPlus : ADD"
967     p[0] = "add\n"
968
969
970 def p_opplus_sub(p):
971     "OpPlus : SUB"
972     p[0] = "sub\n"
973
974
975 def p_opmult_mul(p):
976     "OpMult : MUL"
977     p[0] = "mul\n"
978
979
980 def p_opmult_div(p):
981     "OpMult : DIV"
982     p[0] = "div\n"
983
984
985 def p_oppow(p):
986     "OpPow : POW"
987
988     fp_pow = open("pow.vm", "r")  # retorna erro se ficheiro nao existir
989
990     pow_function_string = fp_pow.read()
991
992     if p.parser.pow_flag:
993         # adiciona ao buffer mas nao  a  tabela
994         p.parser.function_buffer.append(pow_function_string)
995         p.parser.pow_flag = False
996     p[0] = "pusha P\ncall\npop 1\n"
997
998
999 def p_error(p):
1000     # print(p)
1001     # get formatted representation of stack
1002     stack_state_str = ' '.join([symbol.type for symbol in parser.symstack][1:])
1003     p.success = False
1004     print('Syntax error in input! Parser State: {} {}\n . {}\n'
1005           .format(parser.state,
1006                   stack_state_str,
1007                   p),file=sys.stderr)
```

```
1008     raise SyntaxError
1009
1010
1011 # eu pus este codigo aqui em baixo para nao misturar
1012 # as cenas da gramatica com outro codigo
1013
1014
1015 def gen_atrib_code_stack(p, id, atribop):
1016     s = ""
1017     for tamanho in range(len(p.parser.id_table_stack)-1, 0, -1):
1018         if id in p.parser.id_table_stack[tamanho]:
1019             if p.parser.id_table_stack[tamanho][id]['classe'] == 'var':
1020                 s = "storel %d\n" % p.parser.id_table_stack[tamanho][id]['endereco']
1021                 break
1022             else:
1023                 print("ERROR: %s is not of variable class" % id,file=sys.stderr)
1024                 p_error(p)
1025     else:
1026         if id not in p.parser.id_table_stack[0]:
1027             print("ERROR: Name %s not defined." % id,file=sys.stderr)
1028             p_error(p)
1029         else:
1030             if p.parser.id_table_stack[0][id]['classe'] == 'var':
1031                 s = "storeg %d\n" % p.parser.id_table_stack[0][id]['endereco']
1032             else:
1033                 print("ERROR: %s is not of variable class" % id,file=sys.stderr)
1034                 p_error(p)
1035     return s
1036
1037
1038 def pop_local_vars(p):
1039     s = ""
1040     min = float("inf")
1041     pop_size = 0
1042     for var in p.parser.id_table_stack[-1]:
1043         pop_size += prod(p.parser.id_table_stack[-1][var]['tamanho'])
1044         if (n := p.parser.id_table_stack[-1][var]['endereco']) < min:
1045             min = n
1046     if min != float("inf"):
1047         p.parser.local_adress = min if min > 0 else 0
1048     s += "pop %d\n" % pop_size
1049     p.parser.id_table_stack.pop()
1050     return s
1051
1052
1053 parser = yacc.yacc(debugfile="yacc.debug")
1054
1055 # 0->global; 1+->local
1056 # ++ x = (tipo, classe, localidade, endere o, dimen  o)
1057 parser.success = True
1058 parser.main = False
1059 parser.type_table = {'int'}
1060 parser.id_table_stack = list()
1061 parser.id_table_stack.append(dict())
```

```
1062 parser.label_table_stack = list()  # isto vai ser uma lista de pares
1063 parser.function_table = dict()
1064 parser.pow_flag = True  # leia-se  e  preciso por o texto do pow? Devia come ar a
     false?
1065 parser.internal_label = 0
1066 parser.global_adress = 0
1067 parser.local_adress = 0
1068 parser.function_buffer = []
1069 parser.final_code = ""
1070 # a ideia era a cond ter [label:cond ...] e a case ter [lable:body ...]
1071 # ambas tem uma chave special ":" onde pomos numa lista todas as conds e bodies sem
     lables
1072 # na label_table_stack podemos por o par
1073
1074 file_in = input()
1075
1076 f = open(file_in, "r")
1077 ligma_code = f.read()
1078
1079 parser.parse(ligma_code, debug=0)
1080
1081 f.close()
1082
1083 #f = open("test1.vm", "w")
1084 # f.write(parser.final_code)
1085 if parser.success:
1086     print(parser.final_code)
```

Listing A.2: Código pertencente ao Parser

A.3 Pseudo-código do comando *pow*

No código correspondente ao *yacc*, é aberto um ficheiro chamado *pow.vm* com o seguinte código:

```
P:
pushi 1
P1:
pushl -1
jz P2
pushl -2
pushl 0
mul
storel 0
pushl -1
pushi 1
sub
storel -1
jump P1
P2:
storel -2
return
```

Listing A.3: Código pertencente ao *Pow*

Bibliografia

[1] Tony Mason and Doug Brown *Lex & Yacc*, O'Reilly Associates, Inc, USA, 1990.