Processamento de Linguagens e Compiladores

Trabalho Prático 2

Relatório de Desenvolvimento

Gilberto Cunha
A89142

Tomás Carneiro
A82552

21 de janeiro de 2021

Conteúdo

Capítulo 1

Introdução

Este trabalho tem como objetivo desenvolver um compilador para a nossa própria linguagem, fazendo a tradução entre o código da nossa linguagem, de acordo com as regras estipuladas para a mesma, para código `Assembly` da `VM`.

Para atingir este objetivo, foi desenvolvido um analisador lexical com o `flex`, que retorna uma stream de tokens para um analisador semântico, desenvolvido em `yacc`. Este analisador semântico por sua vez traduz a linguagem para o código executável final `Assembly` da `VM`.

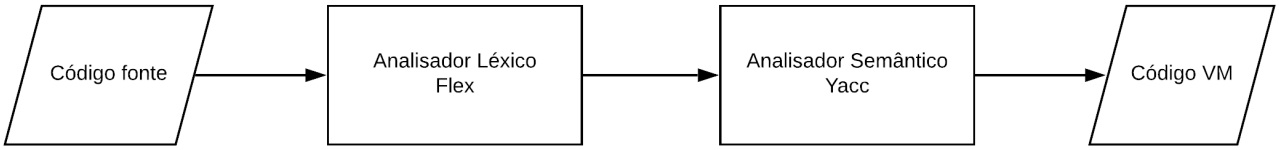

Figura 1.1: Fluxograma que ilustra o processo de tradução código fonte para código `VM`.

Capítulo 2

Introdução à `VM`

A `VM`, ou "Virtual Machine", é uma máquina que funciona por pilhas. A pilha mais importante para o desenvolvimento deste projeto é a chamada "operand stack", que é responsável por armazenar valores que sejam utilizados ao longo de todo o código, como o valor de variáveis declaradas ou valores intermédios do cálculo de uma certa expressão.

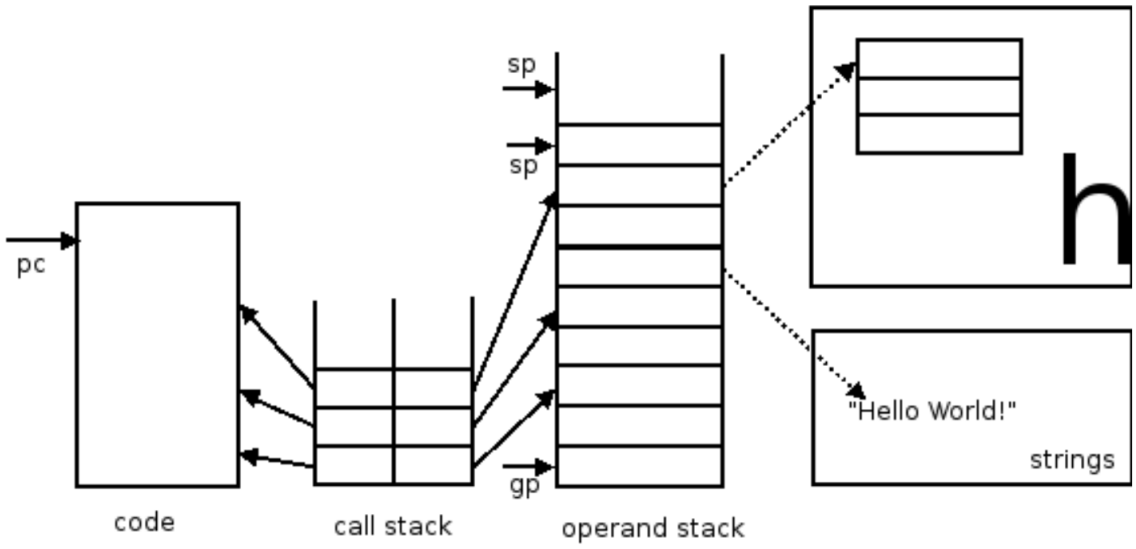

Figura 2.1: Ilustração do sistema de funcionamento da `VM` por pilhas.

Esta máquina permite, então, o empilhamento de valores na stack, bem como a realização de operações de adição (`add`), multiplicação (`mul`), entre várias outras. Realizando estas operações e tendo uma posição reservada na pilha para cada variável declarada, podemos então criar um programa que permita alterar o estado destas variáveis.

Capítulo 3

Sintaxe da Linguagem

A nossa linguagem é baseada na grande maioria no `C`, mas com modificações baseadas no `Python` que permitem (a nosso ver), tornar certas funcionalidades do `C` mais simpáticas e simples de serem escritas.

3.1 Cálculo de expressões

Para o cálculo de expressões, as seguintes operações são permitidas, por *ordem* ***crescente*** *de prioridade*:

1. Operações booleanas de conjunção (and), disjunção (or) e negação (not)
2. As típicas operações de verificação de igualdades/desigualdades ("==", "!=", ">", "<", ">=", "<="). Estas operações são efetuadas sobre *inteiros*, tal como a linguagem `C`
3. Operações sobre inteiros de soma ("+") e subtração ("−")
4. Operações sobre inteiros de divisão ("/"), multiplicação ("∗") e módulo ("%")
5. É também permitido o uso de "(" e ")" para definir uma maior prioridade nestas operações

A título de exemplo, uma atribuição a uma variável pode ser feita da seguinte forma:

```
a = 2 * (x / 4 - 2) and (x != 0 or y <= 3)
```

Esta variável ficará então com o valor lógico da expressão apresentada, consoante os valores das variáveis `x` e `y`.
A definição de expressões em `yacc` encontra-se na secção 4.3.1

3.2 Declaração de variáveis e atribuições

A declaração de variáveis e atribuições são em tudo semelhante ao `C`. Podem ser declaradas várias variáveis do mesmo tipo numa única linha. Cada linha deve terminar com um "\n"ou com um ";". Um ponto muito importante é que as declarações *devem ser todas feitas antes de qualquer outra instrução.*

A linguagem também suporta `arrays`, que são muito semelhantes também ao `C`. Para os arrays, pode-lhes ser atribuída uma lista quando este é declarado, caso o tamanho desta seja o mesmo do `array`.

É também permitida a declaração e atribuição simultânea, numa única linha, a cada uma das variáveis. Dentro destas atribuições é possível utilizar a leitura do *standard input*, através da função `read`. Por convenção, sempre que uma variável é declarada sem lhe ser feita uma atribuição, é-lhe atribuído o valor 0.

```
# Declarações #
int a, b = 233 / 5, c = 20 + 4, d = read("Valor para d: ")
int v[5] = [-3, 4, 27, 3, 15]

# Atribuições #
x = a + d * 3 / (1 + c)
v[2] = x + d
```

A definição de declarações em `yacc` encontra-se na secção 4.3.2 e a das atribuições na secção 4.3.4.

3.3 Standard input e standard output

Para fazer a leitura e escrita devem ser utilizadas as funções `read` e `write`. Estas funções podem ter como argumento uma `string` ou uma `fstring` (tal como no `Python`):

```
# Código #
int a, b = 37, c = read("Qual o valor de c? ")
write("Valores das variáveis: \n")
write(f"a: {a} | b: {b} | c: {c}")

# Output #
>> Qual o valor de c? 42
>> Valores das variáveis:
>> a: 0 | b: 37 | c: 42
```

A sua implementação em `yacc` encontra-se na secção 4.3.3.

3.4 Condicionais

A definição dos condicionais é também semelhante ao `C`. São usadas as palavras reservadas `if` e `else`, que podem também ser encadeados (`else if`), seguidas de uma expressão, tal como foi explicado em 3.1. Caso estas condições se verifiquem, o corpo dos condicionais é executado. Para delimitar o início e fim dos condicionais são utilizados, respetivamente, o **::** e **:;**. *Nota:* caso se usem condicionais encadeados (`else if`), apenas o `else` final deve ser fechado (com **:;**).

```
# Condicionais #
if x == 0 and y != 1 ::
    ...
:;

# Condicionais encadeados #
if x == 0 ::
    ...
:: else if x == 1 ::
    ...
:;
```

O código `yacc` dos condicionais encontra-se descrito na secção 4.3.5.

3.5 Ciclos

Para os ciclos, foi implementado o `for`. Este pode ser escrito de várias maneiras:

1. Iterar uma variável já declarada desde um inteiro inicial até um inteiro final, incrementando-a em 1 valor por cada iteração
2. Igual ao tópico anterior, mas com um incremento definido pelo utilizador. Este incremento também pode ser um valor *negativo*, permitindo percorrer arrays pela ordem inversa
3. Tal como o `C`, com uma atribuição inicial, seguida de uma condição que termina o ciclo quando quebrada, seguida de uma instrução de incremento
4. Percorrer os índices e valores de um array, utilizando duas variáveis já declaradas

```
int i, v, a[4] = [10, 7, 14, -20]

# 1: Iterar i entre 0 e 4 (exclusive) de 1 em 1 #
for (i, 0, 4) ::
    ...
:;

# 2: Iterar i entre 0 e 4 (exclusive) de 2 em 2 #
for (i, 0, 4, 2) ::
    ...
:;

# 2: Iterar i entre 4 e -1 (exclusive) decrescendo i em 1 por iteração #
for (i, 4, -1, -1) ::
    ...
:;

# 3: Ciclo for estilo C #
for (i=0, i<4, i=i+1) ::
    ...
:;

# 4: Iterar todos os indices (0, 1, 2, 3) #
# e valores (10, 7, 14, -20) do array "a" #
for (i, v) -> a ::
    ...
:;
```

A implementação de ciclos em `yacc` encontra-se disponível na secção 4.3.6.

3.6 Comentários

De modo a permitir ao utilizador compreender melhor o seu código após já o ter escrito, foram adicionados comentário de uma única linha, começados e terminados por "#", e de múltiplas linhas, onde os comentários devem se encontrar entre três ".

```
# Este código não é executado, é um comentário de uma linha #
x = 2 * a

"""
Para comentários mais extensos, também os podemos fazer
desta forma.
Assim podemos deixar detalhes mais claros no nosso código,
caso necessitemos
"""
a = 23 + 47
```

3.7 Funções

Na nossa linguagem, à exceção da função `main`, cujo tipo não deve ser explicitado, todas as funções devem ter o tipo `void`. À exceção da `main`, nenhuma função pode declarar variáveis, apenas alterar o seu estado, e também não podem receber argumentos.
Tendo isto em conta, o código deve ser estruturado da seguinte forma:

1. Declarações
2. Definição de funções auxiliares
3. Definição da `main`

```
# Declarações #
int a, b = 233 / 5, c = 20 + 4, d = read("Valor para d: ")

# Definir funções auxiliares #
:: void faux1 ::
    ...
:;

# Definir a main #
|| main ||
    ...
    # Chamada à função faux1, o seu código vai ser executado #
    faux1 ()
    ...
:;
```

Na secção 4.3.8 encontram-se as produções `yacc` para as funções.

Capítulo 4

Concepção da Solução

4.1 Analisador léxico (FleX)

Inicialmente foi criado um analisador léxico, utilizando a ferramenta `flex`, que permite processar o código fonte, contendo um programa escrito na nossa linguagem. Durante este processamento, são retornados diferentes *tokens*, correspondentes a expressões regulares que identificam padrões do nosso código.

Palavras reservadas

Ao longo do código, existem expressões utilizadas diretamente pela linguagem, designadas por "palavras reservadas". Quando estas são identificadas, os seus *tokens* são devolvidos. Podemos visualizar abaixo:

```
main                                    { return MAIN;  }
int                                     { return INT;   }
void                                    { return VOID;  }
for                                     { return FOR;   }
if                                      { return IF;    }
else                                    { return ELSE;  }
or                                      { return OR;    }
and                                     { return AND;   }
not                                     { return NOT;   }
read                                    { return READ;  }
write                                   { return WRITE; }
```

De forma a permitir a declaração de **números inteiros** e **variáveis**, utilizam-se as seguintes expressões regulares:

```
DIGIT [0-9]
LETTER [a-zA-Z]
(...)
%%
{DIGIT}+                            { yylval.num = atoi(yytext); return NUM; }
{LETTER}+_*{LETTER}*{DIGIT}*        { yylval.id = strdup(yytext); return ID; }
```

Os valores dos *tokens* `NUM` e `ID` são atribuídos à variável predefinida `yylval`, que está associada a uma `union`, que armazena os valores `int num`, `char *id` com os respetivos valores dos *tokens* referidos.

Para comentários de uma linha (identificados com um # no início e no fim) ou de múltiplas linhas (delimitados por três " no início e no fim):

```
\#[^\n]*\#                                  { ; }
["]{3}(["]{0,2}([^\\"]|\\(.|\n)))*["]{3}    { ; }
```

Foram criados dois *tokens* que identificam os caracteres **::** e **:;** correspondentes ao início e fim de instruções condicionais, ciclos `for` e funções:

```
\:\:                                        { return START; }
\:\;                                        { return END;   }
```

De forma a poder identificar operadores lógicos, utilizam-se a seguintes expressões:

```
\=\=                                        { return EQ;  }
\!\=                                        { return NEQ; }
\<\=                                        { return LE;  }
\>\=                                        { return GE;  }
```

Os caracteres < e >, não necessitam de conversão uma vez que, nestes casos, o caracter é o próprio *token*.

Se for detetada uma `string`, todos os caracteres (incluindo as aspas) são devolvidos com o *token* `STR`.
O *token* `FSS` é devolvido aquando da deteção do seu início, com os caracteres **f"**:

```
\"[^\"]*\"                          { yylval.id = strdup(yytext); return STR; }
f\"                                 { BEGIN FSTRING; return FSS;              }
```

Quando o início de uma `fstring` é detetado, é inicializada a sua *Start Condition* , com a seguinte estrutura:

```
<FSTRING>\"                         { BEGIN INITIAL; return yytext[0];         }
<FSTRING>\{                         { BEGIN CBRACES; return yytext[0];         }
<FSTRING>[^\{\"]*                   { yylval.id = strdup(yytext); return FSTR; }
```

Os caracteres que não sejam { ou " são devolvidos pelo *token* `FSTR`.
Se for detetado o caracter de término da `fstring` (**"**), o programa regressa para a *Start Condition* `INITIAL`.
Se forem detetadas chavetas, a *Start Condition* `CBRACES` é inicializada:

```
<CBRACES>\}                             { BEGIN FSTRING; return yytext[0];       }
<CBRACES>{DIGIT}+                       { yylval.num = atoi(yytext); return NUM; }
<CBRACES>{LETTER}+_*{LETTER}*{DIGIT}*   { yylval.id = strdup(yytext); return ID; }
<CBRACES>[ ]                            { ; }
<CBRACES>(.|\n)                         { return yytext[0]; }
```

O objetivo desta *Start Condition* está em permitir a identificação de variáveis do programa e de valores inteiros quando nos encontramos numa `fstring`. Como tal, quando estes são identificados, os *tokens* `NUM` e `ID` são devolvidos, para posterior processamento no analisador semântico em 4.3. Caso seja encontrado o caracter de término **}** , o programa regressa à *Start Condition* `FSTRING`.

De forma a eliminar espaços, indentações e `carriage returns`, estes são simplesmente ignorados pelo programa:

```
[\ \t\r]                                    { ; }
```

Por último, todos os caracteres que não sejam detetados pelas expressões definidas acima serão devolvidos:

```
(.|\n)                                      { return yytext[0]; }
```

4.2 Estruturas de dados

Ao desenvolver o analisador semântico para a nossa linguagem vai ser necessário verificar e avisar o utilizador dos seus erros no código. Para além disso, é também necessário realizar operações sobre variáveis declaradas. De modo a viabilizar o acesso a informações dessas variáveis a fazer a tradução do programa original, estas devem ser armazenadas, bem como informação relativa às mesmas, numa estrutura de dados.
A informação que queremos guardar das variáveis é:

- O **nome** da variável
- A **"classe"** dessa variável (se é um valor, array ou função)
- O **tipo** da variável (inteiro ou `void` para funções)
- O ***stack pointer*** dessa variável, para podermos aceder à mesma na stack da `VM`
- O **tamanho** dessa variável (útil para dar erros de segmentation fault em arrays)

Para armazenarmos estes dados, escolhemos utilizar árvores AVL cujos elementos são ordenados como numa árvore de pesquisa, permitindo-nos realizar inserções e pesquisa de elementos em tempo $\mathcal{O}(\log n)$. As variáveis inseridas na árvore são então ordenadas de acordo com o seu nome.
Veja-se a título de exemplo, o programa seguinte que apenas declara variáveis, arrays e funções:

```
int i, j, a, l, k, var[20]

:: void fswap ::
:;

|| main ||
:;
```

Pode-se reparar que a função `fswap` também tem um *stack pointer* nulo armazenado. No entanto, as funções não são acedidas no código através do seu *stack pointer*, logo este facto não interfere com nenhuma outra tarefa a ser feita. A função `main` não é nem necessita ser armazenada.

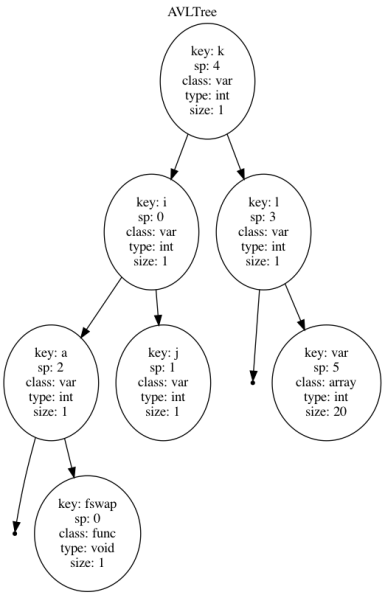

Figura 4.1: Árvore AVL do programa à esquerda.

4.3 Analisador Semântico (`Yacc`)

Uma vez que a stream de tokens está a ser corretamente enviada, é então necessário estabelecer regras relativamente à sua organização, usando estas regras para definir a sintaxe da nossa linguagem.

Para definir os tipos dos símbolos terminais e não terminais a usar, foi feita uma `union` para a variável `yylval` do `yacc`:

```
%union {
  int num;
  char *id;
  char *inst;
}
```

É também importante referir que é usada uma variável global que conta o número de variáveis já declaradas para determinar o valor do *stack pointer* de cada variável quando esta é inserida na árvore AVL. Similarmente, devido aos comandos `jz label`, que implicam a criação de *labels*, recorre-se a uma variável global para contar o número de *labels* já utilizadas, de modo a não existirem colisões entre *labels* com propósitos distintos.

4.3.1 Cálculo de expressões

De modo a implementar o cálculo de expressões (símbolo não terminal `expr`) da secção 3.1, definiram-se várias produções, todas com o tipo `char *`:

```
%type <inst> factor par term arithm lexpr expr
```

Serão então agora detalhadas as produções de cada símbolo não terminal, por ordem *decrescente* de prioridade no cálculo de expressões.

O último símbolo não terminal de uma expressão é um fator, isto é, um valor. Este fator pode ser uma variável, um número ou um array, e todos estes podem ser negativos também:

```
factor : NUM                    { asprintf (&$$, "pushi %d\n", $1); }
       | ID                     { factorId (&$$, $1, &vars); }
       | ID '[' expr ']'        {
         factorArray (&$$, $1, $3, &vars, &func_count, @3.last_line);
       }
       | '-' NUM                { asprintf (&$$, "pushi %d\n", -$2); }
       | '-' ID                 { negfactorId (&$$, $2, &vars); }
       | '-' ID '[' expr ']'    {
         negfactorArray (&$$, $2, $4, &vars, &func_count, @3.first_line);
       };
```

Caso tenhamos um número, apenas precisamos de o empilhar com o comando `pushi` da `VM`.

Para variáveis (`ID`), é verificado se esta está declarada e se a sua "classe" é "var" e, em caso afirmativo, o seu *stack pointer* é extraído da árvore AVL usando o comando `pushg` para empilhar o seu valor.

Para os arrays (`ID '[' expr ']'`), é verificado também se este já foi declarado e se pertence à classe "array". Para além disso, o seu índice é avaliado e verifica-se (em *runtime*) usando comandos `jz`, se este está entre 0 e o tamanho declarado do array (exclusive). Em caso afirmativo, o valor do array é extraído usando os comandos `pushgp`, `padd` e `loadn`.

Em `par`, são avaliados tanto fatores como expressões com prioridade (entre parêntesis):

```
par : '(' expr ')'        { asprintf (&$$, "%s", $2); }
    | factor              { asprintf (&$$, "%s", $1); };
```

Com menos prioridade temos ainda as multiplicações, divisões e módulos, implementados com os comandos `mul`, `div` e `mod` da `VM`.

```
term : term '*' par    { asprintf (&$$, "%s%smul\n", $1, $3); }
     | term '/' par    { asprintf (&$$, "%s%sdiv\n", $1, $3); }
     | term '%' par    { asprintf (&$$, "%s%smod\n", $1, $3); }
     | par             { asprintf (&$$, "%s", $1); };
```

Para finalizar a aritmética tradicional de inteiros, são também adicionadas operações de adição e subtração, com os comandos `add` e `sub`:

```
arithm : arithm '+' term    { asprintf (&$$, "%s%sadd\n", $1, $3); }
       | arithm '-' term    { asprintf (&$$, "%s%ssub\n", $1, $3); }
       | term               { asprintf (&$$, "%s", $1); };
```

No entanto, como já referido, adicionamos sobre a aritmética de inteiros expressões lógicas, tal como no `C`. São então agora suportadas operações de igualdade, desigualdade e de ordenação, usando os comandos `equal`, `sup`, `inf`, `supeq` e `infeq`

```
lexpr : lexpr EQ arithm     { asprintf (&$$, "%s%sequal\n", $1, $3); }
      | lexpr NEQ arithm    { asprintf (&$$, "%s%sequal\nnot\n", $1, $3); }
      | lexpr GE arithm     { asprintf (&$$, "%s%ssupeq\n", $1, $3); }
      | lexpr LE arithm     { asprintf (&$$, "%s%sinfeq\n", $1, $3); }
      | lexpr '>' arithm    { asprintf (&$$, "%s%ssup\n", $1, $3); }
      | lexpr '<' arithm    { asprintf (&$$, "%s%sinf\n", $1, $3); }
      | arithm              { asprintf (&$$, "%s", $1); };
```

Para terminar as expressões, são também suportadas as disjunções, conjunções e negações. Estas são definidas no símbolo não terminal que vai avaliando aos poucos os outros símbolos acima referidos, pois estas operações (`and`, `or` e `not`) são as de menor prioridade.

```
expr : expr AND lexpr    {
       asprintf (&$$, "%sdup 1\njz func%d\n%smul\nfunc%d:\n", $1, fcount, $3, fcount);
       fcount++;
     }
     | expr OR lexpr     {
       asprintf (&$$, "%snot\ndup 1\njz func%d\n%snot\nmul\nnot\nfunc%d:\n",
                 $1, fcount, $3, fcount);
       fcount++;
     }
     | NOT lexpr         { asprintf (&$$, "%snot\n", $2); }
     | lexpr             { asprintf (&$$, "%s", $1); };
```

Para a negação é utilizado o comando `not` da `VM`.

Nas conjunções é verificada a expressão à esquerda e, no caso de este ser falso, a segunda expressão já não é avaliada. Para esta operação apenas são multiplicados os números com o comando `mul`.

A disjunção é implementada à semelhança da conjunção: caso a expressão à esquerda seja verdadeira, a expressão à direita já não precisa de ser avaliada. Esta operação é realizada negando cada uma das expressões, multiplicando-as e negando-as novamente.

4.3.2 Declaração de variáveis

De forma a permitir a declaração de variáveis (símbolo não terminal `declrs`), da secção 3.2, definiram-se produções com o tipo `char *`:

```
%type <inst> declrs declr decllist singdecl list
```

Para implementar as regras detalhadas em 3.2, foram então definidas as seguintes produções:

```
singdecl : ID                       { declaration (&$$, $1, &sp_count, &vars);    }
         | ID '[' NUM ']'           { declrArray (&$$, $1, $3, &sp_count, &vars); }
         | ID '=' expr              { declrExpr (&$$, $1, $3, &vars, &sp_count);  }
         | ID '=' read              { declrRead (&$$, $1, $3, &vars, &sp_count);  }
         | ID '[' NUM ']' '=' '[' list ']'  {
           decList (&$$, $1, $3, $7, &vars, &sp_count, &list_size);
         }
         ;

list : expr ',' list                { asprintf (&$$, "%s%s", $1, $3); list_size++;  }
     | expr                         { asprintf (&$$, "%s", $1); list_size++;        }
     ;
```

As produções acima referidas permitem fazer a declaração singular de variáveis. Uma **variável inteira** poderá então ser declarada de três formas: sem lhe ser atribuído um valor (neste caso irá tomar o valor 0); atribuindo-lhe um valor, podendo este resultar de uma expressão; através da leitura do *standard input*, com recurso à função `read` da nossa linguagem. Um **array de inteiros** pode ser declarado de outras duas formas: se o seu valor não for explicitado, **todos** os seus elementos serão inicializados a 0; alternativamente, pode ser-lhe atribuída uma lista (delimitada pelos caracteres '[' e ']', utilizando a vírgula para separação dos

seus elementos).
Aquando da sua declaração, a informação da variável, explicitada em 4.2, é inserida na estrutura **AVL**, caso não existam conflitos de redeclarações. Após a sua inserção, dependendo da forma como a variável foi declarada, esta irá ser acrescentada à *Stack* da `VM`.

Para valores inteiros:

1. Se estes forem declarados sem qualquer atribuição, é efetuado um `pushn 1`, que empilha no topo da *Stack* o valor 0 (zero) uma vez.

2. Se lhes for atribuída uma expressão, é efetuado um `pushn 1`, o valor da expressão é posteriormente calculado e empilhado no topo da stack e armazenado na variável correspondente, com recurso ao comando `storeg sp`, em que `sp` corresponde ao *stack pointer* associado à variável criada.

3. Em último caso, se à variável for atribuída uma função `read`, as instruções da `VM` a executar serão semelhantes à anterior declaração, mas em vez de ser calculado um valor de uma expressão, é efetuado um `read` seguido de um `atoi`, empilhando desta forma o valor inteiro correspondente à *string* lida no *standard input* no topo da stack. A estas instruções segue-se novamente um `storeg sp`, que guarda o valor no endereço da variável designada.

Para arrays de inteiros:

1. Em declarações sem atribuição, são empilhados `n` valores 0 (zero) com a instrução `pushn`, de acordo com o tamanho que for especificado pelo inteiro correspondente à *token* `NUM`. O `sp` que marca o início da lista, assim como o seu tamanho, serão guardados na estrutura de dados que alberga as variáveis.

2. Nas declarações com atribuição, é detetado o *token* `list`, cujas produções efetuam sucessivos `pushi` a cada valor da lista na `VM`, contando ao mesmo tempo o seu tamanho, através do inteiro `list_size`. Na função `decList`, é verificado se os valores dos inteiros `list_size` e `NUM` (correspondentes ao tamanho da lista e ao valor da *token* `NUM`), são iguais, devolvendo uma mensagem de erro caso não se verifique.

```
decllist : singdecl ',' decllist    { asprintf (&$$, "%s%s", $1, $3); }
         | singdecl                 { asprintf (&$$, "%s", $1); }
         ;
```

De forma a permitir a declaração de **múltiplas variáveis** na **mesma linha**, estas têm de estar separadas por vírgulas.

```
endline : '\n' | ';' ;

declr : INT decllist endline        { asprintf (&$$, "%s", $2); }
      | '\n'                        { asprintf (&$$, "%s", ""); }
      ;
```

Cada linha correspondente à declaração de uma ou múltiplas variáveis contém um *token* `INT` no início, sendo o seu término assinalado por um `endline`, correspondente ao caracter **n** ou **;**.

```
declrs : declrs declr              { asprintf (&$$, "%s%s", $1, $2); }
       | error endline             { asprintf (&$$, "%s",""); }
       |                           { asprintf (&$$, "%s", ""); }
       ;
```

As declarações serão então compostas pelas produções acima. Caso sejam detetados erros numa delas, esta será ignorada até ser encontrado o `endline`.

4.3.3 Standard input e standard output

A implementação das funções `read` e `write` da secção 3.3 necessita do uso de `strings` e de `fstrings`. Foram então criados os seguintes símbolos não terminais:

```
%type <inst> string fstring read write

string : FSS fstring '"'    { asprintf (&$$, "%s", $2); }
       | STR                { asprintf (&$$, "pushs %s\nwrites\n", $1); }
       |                    { asprintf (&$$, "%s", ""); };

fstring : fstring '{' expr '}' { asprintf (&$$, "%s%swritei\n", $1, $3); }
        | fstring FSTR         { asprintf (&$$, "%spushs \"%s\"\nwrites\n", $1, $2); }
        | '{' expr '}'         { asprintf (&$$, "%swritei\n", $2); }
        | FSTR                 { asprintf (&$$, "pushs \"%s\"\nwrites\n", $1); };
```

O nosso símbolo não terminal `string` abrange tanto as `strings` tradicionais como as `fstrings`, ficando com o valor destas.

As `fstrings` são constituídas por `strings` e por expressões (colocadas entre chavetas), como definidas anteriormente. Usando então os comandos `writei` para escrever o valor das expressões e `writes` para escrever `strings`, fazemos a sua tradução para código `VM`.

Após termos então as `strings` e `fstrings`, os comandos `read` e `write` tomam o seu valor:

```
write : WRITE '(' string ')'   { asprintf (&$$, "%s", $3); };

read : READ '(' string ')'     { asprintf (&$$, "%sread\natoi\n", $3); };
```

Para a função `read` é também usado o comando `VM read` que lê uma `string`, que é convertida para inteiro com o comando `atoi`.

4.3.4 Atribuições a variáveis

Após declaração das variáveis, queremos poder alterar o seu valor com atribuições como em 3.2. Foram então desenvolvidas as seguintes produções:

```
%type <instr> atr
atr : ID '=' expr                 { exprAtr (&$$, $1, $3, &vars); }
    | ID '=' read                 { readAtr (&$$, $1, $3, &vars); }
    | ID '[' expr ']' '=' expr    {
      arrayAtr (&$$, $1, $3, $6, &vars, &fcount, @3.first_line);
    }
    | ID '[' expr ']' '=' read    {
      readArrayAtr (&$$, $1, $3, $6, &vars, &fcount, @3.first_line);
    };
```

As atribuições podem ser apenas expressões ou leituras do *standard input* que modificam o valor das variáveis. Estas atribuições podem ser feitas sobre arrays e sobre inteiros. As variáveis a ser modificadas são então procuradas na árvore AVL para verificar se já foram declaradas. Em caso afirmativo, e novamente verificando os limites do índice dos arrays, o valor da expressão ou da leitura é atribuído à variável, usando o seu *stack pointer* guardado na árvore e os comandos `storeg` para inteiros e `storen` para arrays.

4.3.5 Condicionais

Os `if` e `else` *statements* descritos na secção 3.4 começam pelo *token* `IF` ou `ELSE`, seguidos de uma expressão cujo valor lógico deve ser avaliado e das instruções no seu corpo.

Suponhamos que temos um símbolo não terminal `instrs` com o tipo `<instr>` que contém a string correspondente a uma sequência de instruções. As produções deste símbolo serão explicadas mais à frente, mas partindo do princípio que este funciona, temos então as seguintes produções:

```
%type <instr> cond
cond : IF expr START instrs END                          {
       ifInstr (&$$, $2, $4, &fcount);
     }
     | IF expr START instrs START ELSE START instrs END  {
       ifElse (&$$, $2, $4, $8, &fcount);
     }
     | IF expr START instrs START ELSE cond              {
       ifElseif (&$$, $2, $4, $7, &fcount);
     };
```

Pela ordem das produções acima, para converter em código `VM`:

1. Num `if`, é calculado o valor da expressão e executado um comando `jz` com label "cond{fcount}". Caso a expressão seja falsa, então o `jz` faz o código saltar para o fim do ciclo. Caso seja verdadeira, o corpo do ciclo é executado. O contador de labels `fcount` é por fim incrementado em 1.

2. Para o caso de um `else`, é colocada na stack a negação da condição do `if`. Caso a expressão negada seja falsa, então entramos no corpo `if`, que está inicializado com uma label "cond{fcount}". No final deste corpo é executado um `jump` para a *label* de fim do condicional "cond{fcount+2}". Caso a condição do início do `if` seja falsa, então é executado um `jump` após o `jz` para a *label* "cond{fcount+1}", executando assim o corpo do `else`, que termina com um comando `jump` para a label "cond{fcount+2}". Por fim, `fcount` é incrementada em duas unidades.

3. O `if ... else if ...`, é em tudo semelhante ao caso anterior, mas é feito um salto no final do corpo do if e do else para a *label* "cond{fcount-1}", que será a *label* do final do condicional.

4.3.6 Ciclos

Para terminar os diferentes tipos de instruções que a nossa linguagem permite que sejam feitas, temos os **ciclos**. O ciclo que foi implementado é um `for`, que pode ser executado de várias maneiras distintas de acordo com a preferência do utilizador, tal como descrito em 3.5.

Foram implementadas as seguintes produções:

```
%type <inst> cycle

cycle : FOR '(' ID ',' expr ',' expr ')' START instrs END                {
        forStartEnd (&$$, $3, $5, $7, $10, &vars, &fcount);
      }
      | FOR '(' ID ',' expr ',' expr ',' expr ')' START instrs END       {
        forStep (&$$, $3, $5, $7, $9, $12, &vars, &fcount);
      }
      | FOR '(' ID ',' ID ')' '-' '>' ID START instrs END                {
        forArrayIV (&$$, $3, $5, $9, $11, &vars, &fcount);
      }
      | FOR '(' ID '=' expr ',' expr ',' ID '=' expr ')' START instrs END {
        forCond (&$$, $3, $9, $5, $7, $11, $14, &vars, &fcount);
      };
```

Vejamos o que é feito em cada um dos casos, pela ordem das produções acima:

1. Caso seja detetada a primeira produção, é colocado (caso a variável `ID` esteja na árvore `AVL`) antes do corpo do ciclo o valor da primeira `expression` na variável `ID`, através de um `pushi` seguido de `storeg`. O ciclo começa pela sua *label* e no seu corpo são executadas as suas instruções. No final deste é incrementada a variável `ID` em 1 unidade e a condição do ciclo (o valor de `ID` ser inferior à segunda expressão) é novamente avaliada: caso seja verdadeira, o comando `jz` faz um salto de início de ciclo novamente, caso contrário o ciclo é abandonado.

2. Se for detetada uma expressão adicional para além da anterior, então esta é referente ao incremento da variável `ID`. É então novamente inicializada esta variável e avaliada a condição do ciclo (que verifica o sinal do passo). Dentro do ciclo tudo é semelhante ao caso anterior, com a exceção de que o incremento da variável é agora dado pela terceira expressão.

3. O terceiro caso corresponde a percorrer os índices e valores de um array. Para este caso, as variáveis $\texttt{ID}_1$ e $\texttt{ID}_2$ são inicializadas com 0 e com o primeiro valor do array, respetivamente. Dentro do ciclo as instruções do corpo são executadas. No fim do corpo, a variável $\texttt{ID}_1$, relativa aos índices, é incrementada em 1 unidade. Caso esta ainda seja inferior ao tamanho do array, obtido a partir da árvore `AVL`, então é colocado o valor seguinte do array na variável $\texttt{ID}_2$ e é feito um salto para o início do ciclo. Caso contrário, o ciclo é terminado.

4. Para implementar um ciclo como no `C`, é empilhado o valor da 1ª expressão e armazenado no $\texttt{ID}_1$. A condição do ciclo é avaliada e, caso falsa, é dado um salto usando o comando `jz` para o final do ciclo, com *label* "cycle{fcount+1}". Dentro do corpo do ciclo são executadas as suas instruções e, no final, é

atribuído o valor da última expressão à variável `ID`. Após a atribuição, a condição do ciclo é novamente avaliada [1], caso seja verdadeira é feito um salto para o início do ciclo, caso contrário este termina.

4.3.7 Função *Main*

Uma vez que já se conhecem quase todas as instruções que podem ser executadas na nossa linguagem, devemos então conseguir usá-las na nossa função `main`. Para tal, foram então empacotadas todas as instruções até agora definidas num único símbolo não terminal `instr`, que por sua vez permite definir uma sequência de instruções `instrs`. Esta sequência de instruções permite-nos então criar a nossa função `main`:

```
%type <inst> main instrs instr

main : '|' MAIN '|' '|' instrs      { asprintf (&$$, "%s", $5); };

instrs : instrs instr   { asprintf (&$$, "%s%s", $1, $2); }
       | error endline  { asprintf (&$$, "%s",""); }
       |                { asprintf (&$$, "%s", ""); };

instr : atr endline      { asprintf (&$$, "%s", $1); }
      | write endline    { asprintf (&$$, "%s", $1); }
      | cond endline     { asprintf (&$$, "%s", $1); }
      | cycle endline    { asprintf (&$$, "%s", $1); }
      | fcall endline    { asprintf (&$$, "%s", $1); }
      | '\n'             { asprintf (&$$, "%s", ""); };
```

A única instrução nova que aqui aparece é a do símbolo não terminal `fcall`, que implementa a chamada de funções. Esta será explicada na próxima subsecção.

4.3.8 Funções auxiliares do tipo *void*

As únicas funções implementadas são do tipo *void* e não recebem argumentos. Estas são utilizadas quase apenas como macros, permitindo-nos não repetir código ou torná-lo mais legível, separando-o por partes. Estas são implementadas da seguinte forma:

[1] repare-se que nas disjunções e conjunções poder ser avaliado apenas o primeiro argumento nos ciclos é muito útil. Exemplo: `for(i=10, i>0 and v[i-1]!=3, i=i-1)`. Caso tal não fosse permitido, então avaliar a condição do ciclo no caso de paragem i=0 iria resultar num erro de *segmentation fault* por acedermos à posição `-1` de `v`.

```
%type <inst> funcs

funcs : START VOID ID START instrs END funcs    {
        declrFunc (&$$, $3, $5, $7, &vars, "void");
      }
      | VOID ID START instrs END funcs          {
        declrFunc (&$$, $2, $4, $6, &vars, "void");
      }
      | '\n' funcs                              { asprintf (&$$, "%s", $2); }
      | error endline                           { asprintf (&$$, "%s", ""); }
      |                                         { asprintf (&$$, "%s", ""); };

fcall : ID '(' ')'            { funcCall (&$$, $1, &vars); };
```

Aqui a segunda produção é relativa à primeira função a ser declarada e a primeira produção a todas as restantes. Esta função é então inserida na àrvore `AVL` com um *stack pointer* de -1, uma vez que esta é uma variável que não é empilhada.

O código `VM` das funções consiste na sua *label* seguida de um `nop`, o corpo da função (valor do `instr`) e terminada em `return`. O código das funções auxiliares será sempre inserido no fim do programa `VM` gerado.

4.3.9 Axioma

Por fim, tendo já todas as componentes da nossa linguagem implementadas, resta apenas definir o seu **axioma**. Este pode ser definido da seguinte forma:

```
%start L

L : declrs START funcs '|' main END {
    fprintf (vm, "%sstart\n%sstop\n%s", $1, $5, $3);
  }
  | declrs '|' main END             {
    fprintf (vm, "%sstart\n%sstop\n", $1, $3);
  }
  | error '\n';
```

Resumindo, os nossos programas devem ser constituídos por declarações e pela função `main` ou por declarações, seguidas de definições de funções auxiliares e por fim a `main`.

Em termos de código `VM`, as primeiras instruções a serem inseridas no ficheiro `.vm` são as declarações, seguidas da função `main` e por fim as labels e o corpo das funções auxiliares. De notar que o início da `main` tem de ser assinalado com uma instrução `start` e terminar com uma instrução `stop` no ficheiro de output `.vm`.

4.3.10 Erros

Ao longo de toda a geração de código máquina vários erros são detetados, a grande maioria utilizando a informação das árvores `AVL`. Alguns dos erros gerados são:

1. Variáveis não declaradas caso não sejam encontradas na árvore ou re-declaradas caso estejam a ser declaradas e já lá se encontrem

2. Declaração de arrays com uma lista de tamanho diferente. Aqui é utilizada uma variável que conta os elementos da lista e a compara com o tamanho passado como índice ao array.

3. Erros de mistura de tipos/classes, através da informação contida nas árvores.

4. Erros de *segmentation fault*, onde o índice de um array é ou inferior a 0 ou superior/igual ao seu tamanho. Estes erros são distintos dos restantes pois apenas são detetados em *run time*, e não em *compile time*. Para este erro gerou-se código máquina sempre que um array é chamado e indexado, que verifica se o índice pertence à gama de índices do array, caso contrário retorna um erro com o comando `err` da `VM`.

Uma possível melhoria a esta parte do trabalho seria tentar solucionar o problema do `yacc` não ser muito preciso na linha onde os erros ocorrem. Tirando isso, achamos que os erros são bem apresentados, com fácil deteção do problema, e amigáveis com o utilizador, tentando dar-lhe sempre o máximo de informação.
De modo a exemplificar os erros, apresentamos aqui uma compilação de vários destes.

```
int i, x, b, array[10] = [1, 2, 3] # array e lista com tamanhos diferentes #
int w[2] = read()                  # Erro de sintaxe #
int v[4] = [-18, 23, 4, 7], i      # Redeclaração do i #

:: void fswap ::

:;

|| main ||
    z = i                # Variável não declarada #
    fswap = x            # Atribuir a funções void #
    write (f"{x[i]}\n")  # Indexar um inteiro #
    b()                  # Chamar uma variável #

    # A variável que itera entre os índices não é inteira #
    for (v, 0, 10) ::
        write (f"{v}\n")
    ;;
:;

# Output da compilação #
--------------------------------------------------------------------------------
Line 1: Array "array" declared with size 10 but list has size 3.
--------------------------------------------------------------------------------
Line 2: syntax error, unexpected READ, expecting '['
--------------------------------------------------------------------------------
Line 4: Variable "i" redeclared.
--------------------------------------------------------------------------------
Line 11: Can't access variable "z" because it hasn't been declared.
--------------------------------------------------------------------------------
Line 12: Function "fswap" value not accessable.
--------------------------------------------------------------------------------
Line 12: Integer "x" can't be indexed.
--------------------------------------------------------------------------------
Line 13: Variable not callable.
--------------------------------------------------------------------------------
Line 17: Array "v" can't be treated as an integer.
--------------------------------------------------------------------------------
Line 18: Can't iterate variable of array, use integer instead.
--------------------------------------------------------------------------------
Line 22: Array "v" can't be treated as an integer.
--------------------------------------------------------------------------------
Line 22: Function "fswap" value not accessable.
--------------------------------------------------------------------------------
Line 23: Must use integer to hold array indices.
--------------------------------------------------------------------------------
```

Para os erros de *segmentation fault*:

```
int v[3] = [-11, 23, 3]

|| main ||
    v[-11] = 24
:;

# Compilação não dá erros #

# Ao correr o programa na VM #
> vms program.vm

--------------------------------------------------------------------------------
Line 4: Index of array 'v' smaller than zero.
--------------------------------------------------------------------------------
```

```
int v[3] = [-11, 23, 3]

|| main ||
    v[23] = 24
:;

# Compilação não dá erros #

# Ao correr o programa na VM #
> vms program.vm

--------------------------------------------------------------------------------
Line 4: Index of array 'v' too high for its size.
--------------------------------------------------------------------------------
```

Capítulo 5

Testes e exemplos

Primeiramente, para se compilar um programa da nossa linguagem, deve ser executado o script bash "bobwc", que permite que se usem como argumentos:

1. `verbose=yes|no`: no modo verboso, o compilador vai informando o utilizador das tarefas que executa
2. `debug=yes|no`: no modo *debug*, é gerada uma imagem da árvore `AVL` das variáveis do programa, à semelhança da Figura 4.1. É também mantido o ficheiro `.vm` gerado, mesmo que o compilador detete erros
3. `file=[filename]`: selecionar o ficheiro que deve ser compilado
4. `man`: é escrito no terminal um programa exemplo da nossa linguagem, permitindo a verificação do seu funcionamento a nível sintático a um utilizador que não esteja familiarizado.

5.1 Verificar se 4 números são lados de um quadrado

```
int i, r, N[4]

|| main ||
    for (i, 0, 4) ::
        # Adicionar uso de variáveis como índices de arrays #
        N[i] = read(f"{i+1}º lado do quadrado: ")

        if i>0 ::
            if N[i] != N[i-1] ::
                r = 1
            ;;
        ;;
    ;;

    if r != 1 ::
        write("São lados de um quadrado\n")
    :: else ::
        write("Não são lados de um quadrado\n")
    ;;
;;
```

```
> ./bobwc file=Examples/1_square_sides.txt

> vms program.vm
1º lado do quadrado: 3
2º lado do quadrado: 3
3º lado do quadrado: 3
4º lado do quadrado: 3
São lados de um quadrado

> vms program.vm
1º lado do quadrado: 3
2º lado do quadrado: 3
3º lado do quadrado: 3
4º lado do quadrado: 2
Não são lados de um quadrado
```

5.2 Menor elemento de N números

```
int N = read("Quer descobrir o menor elemento de quantos números? ")
int i, r, var

|| main ||
    r = read("1º número: ")

    for (i, 0, N-1) ::
        var = read(f"{i+2}º número: ")

        if var < r ::
            r = var
        ;;
    ;;
    write(f"O menor dos números é {r}\n")
;;
```

```
> ./bobwc file=Examples/2_smaller_number.txt

> vms program.vm
Quer descobrir o menor elemento de quantos números? 5
1º número: 37
2º número: 2
3º número: 24
4º número: -3
5º número: 4
O menor dos números é -3
```

5.3 Produto de N números

```
int i, r = 1, tmp
int N = read("Quer calcular o produto de quantos números? ")

|| main ||
    for (i , 0 , N) ::
        tmp = read(f"{i+1}º número: ")
        r = r * tmp
    ;;

    write (f"O produto dos números é {r}\n")
;;
```

```
> ./bobwc file=Examples/3_productory.txt

> vms program.vm
Quer calcular o produto de quantos números? 4
1º número: 20
2º número: 20
3º número: 3
4º número: 17
O produto dos números é 20400
```

5.4 Contar elementos ímpares num array

```
int i, v, count
int seq[5] = [1, 10, 7, 6, 3]

|| main ||
    write ("Array: ")
    for (i, v) -> seq ::
        write (f"{v} ")
    ;;
    write ("\n")
    for (i, 0, 5) ::
        if seq[i]%2 == 1 ::
            write(f"O {i+1}º elemento do array (com valor {seq[i]}) é ímpar\n")
            count = count + 1
        ;;
    ;;
    write(f"O número de elementos ímpares é {count}\n")
;;
```

```
> ./bobwc file=Examples/4_odd.txt

> vms program.vm
Array: 1 10 7 6 3
O 1º elemento do array (com valor 1) é ímpar
O 3º elemento do array (com valor 7) é ímpar
O 5º elemento do array (com valor 3) é ímpar
O número de elementos ímpares é 3
```

5.5 Ler um array e imprimi-lo por ordem inversa

```
int i, u, v[5]

|| main ||
    for (i, 0, 5) ::
        v[i] = read (f"Insira o {i+1}º elemento do array: ")
    ;;
    write ("Array:        : ")
    for (i, u) -> v ::
        write (f"{u} ")
    ;;
    write ("\n")
    write ("Inverted array: ")
    for (i, 4, -1, -1) ::
        write (f"{v[i]} ")
    ;;
    write ("\n")
;;
```

```
> ./bobwc file=Examples/5_array.txt

> vms program.vm
Insira o 1º elemento do array: 3
Insira o 2º elemento do array: 27
Insira o 3º elemento do array: -4
Insira o 4º elemento do array: 3
Insira o 5º elemento do array: 5
Array:        : 3 27 -4 3 5
Inverted array: 5 3 -4 27 3
```

5.6 Cálculo de uma potência

```
int i, e, b, p

:: void potencia ::
    p = 1
    for (i=e, i>0, i=i-1) ::
        p = p * b
    ;;
;;

|| main ||
    write ("Cálculo da potência. Escolha a sua base e expoente.\n")
    b = read ("Base: ")
    e = read ("Expoente: ")
    potencia ()
    write (f"{b}^{e}={p}\n")
;;
```

```
> ./bobwc file=Examples/6_power.txt

> vms program.vm
Cálculo da potência. Escolha a sua base e expoente.
Base: 4
Expoente: 4
4^4=256
```

5.7 Máximo Divisor Comum

```
int a = read("Insira o primeiro número: ")
int b = read("Insira o segundo número: ")
int aux, i , a0 = a , b0 = b

:: void myswap ::
    aux = a
    a = b
    b = aux
:;

|| main ||
    for(i=0, b, i = i+1) ::
        a = a % b
        myswap()
    :;
    write(f"O valor do máximo divisor comum entre {a0} e {b0} é {a}.\n")
:;
```

```
> ./bobwc file=Examples/Euclidean_GCD.txt

> vms program.vm
Insira o primeiro número: 1904
Insira o segundo número: 24
O valor do máximo divisor comum entre 1904 e 24 é 8.
```

5.8 Insertion Sort

```
int i, j, vaux, v[10] = [-3, 4, 7, 21, 43, 6, 14, -33, 9, 0]

:: void fswap ::
    vaux = v[j-1]
    v[j-1] = v[j]
    v[j] = vaux
:;

|| main ||
    write ("Array:        ")
    for (i, vaux) -> v ::
        write (f"{vaux} ")
    :;

    # Ordenar o array
    for (i, 1, 10) ::
        for (j=i, j>0 and v[j-1]>v[j], j=j-1) ::
            fswap ()
        :;
    :;

    write ("\nSorted array: ")
    for (i, vaux) -> v ::
        write (f"{vaux} ")
    :;
    write ("\n")
:;
```

```
> ./bobwc file=Examples/insertionsort.txt

> vms program.vm
Array:        -3 4 7 21 43 6 14 -33 9 0
Sorted array: -33 -3 0 4 6 7 9 14 21 43
```

5.9 Selection Sort

```
int i, j, aux, vaux, v[10] = [-3, 4, 7, 21, 43, 6, 14, -33, 9, 0]

:: void fswap ::
    vaux = v[i]
    v[i] = v[aux]
    v[aux] = vaux
:;

|| main ||
    write (f"Array:        ")
    for (i, vaux) -> v ::
        write (f"{vaux} ")
    :;

    write ("\nSorted array: ")
    for (i=0, i<10, i=i+1) ::
        aux = i
        for (j=i+1, j<10, j=j+1) ::
            if v[j] < v[aux] ::
                aux = j
            :;
        :;
        fswap ()
        write (f"{v[i]} ")
    :;
    write ("\n")
:;
```

```
> ./bobwc file=Examples/selectionsort.txt

> vms program.vm
Array:        -3 4 7 21 43 6 14 -33 9 0
Sorted array: -33 -3 0 4 6 7 9 14 21 43
```

Capítulo 6

Conclusão

O desenvolvimento do nosso próprio compilador permitiu-nos comprovar a surpreendente potencialidade das ferramentas `Flex` e `Yacc` para o desenvolvimento tanto de analisadores léxicos, como de gramáticas tradutoras.

O aspeto mais instigante na construção deste trabalho foi a "liberdade" que nos foi proporcionada no que respeitava à edificação da nossa própria linguagem. Tomámos a iniciativa de implementar funcionalidades práticas e úteis, com o intuito de tornar a linguagem numa que fosse fácil de escrever, e ao mesmo tempo permitisse a resolução de problemas com recurso a diferentes implementações possíveis.

À medida que vamos ganhando experiência no mundo da computação, apercebemo-nos de algumas características que distinguem diferentes linguagens de programação, por exemplo no *"debug"* de código, na sua organização estrutural e simbólica, na exigência de rigor na sua escrita, entre outras. Por conseguinte, um dos aspetos do trabalho nos quais mais nos focámos, residiu em proporcionar uma experiência *"user-friendly"* para qualquer pessoa que pretendesse utilizar a nossa linguagem. Para isto, implementámos a deteção de erros sintáticos, criámos um ficheiro `Man` (com informação útil para alguém com dúvidas relativamente à sintaxe da linguagem) e permitimos também que um programa que tenha sido escrito na nossa linguagem seja compilado sem dificuldades, com recurso ao *script* `bobwc` desenvolvido.

A maior dificuldade deste projeto foi organizar tantas peças diferentes ao mesmo tempo. Desde o lexer ao parser, às árvores `AVL` ou mesmo à tradução do código para `VM`, ter uma boa organização no projeto foi uma tarefa exigente mas fulcral para o bom funcionamento e bom *debug* de cada parte constituinte.

Caso decidamos continuar a aprimorar este projeto, existem várias direções que ainda poderemos tomar. Desde a implementação de funções com argumentos e variáveis locais à introdução de novos tipos ou a criação de estruturas, a lista de possíveis funcionalidades que podem ser ainda adicionadas é vasta e o único limite é o nosso conhecimento tanto em técnicas de parsing como outras ferramentas e linguagens de programação.

Apesar das limitações em termos de funcionalidades da nossa linguagem, todo o seu desenvolvimento foi um processo muito instrutivo e a simples gratificação de criar *a nossa* própria linguagem, permitindo prontamente a sua utilização, é uma enorme recompensa.

Apêndice A

Código do programa

A.1 Código Flex

```
%{
#include <stdio.h>
#include "y.tab.h"
#define YY_USER_ACTION yylloc.first_line = yylloc.last_line = yylineno;
%}

DIGIT [0-9]
LETTER [a-zA-Z]

%x FSTRING CBRACES
%option noyywrap
%option yylineno

%%
main                                        { return MAIN; }
int                                         { return INT; }
void                                        { return VOID; }
for                                         { return FOR; }
if                                          { return IF; }
else                                        { return ELSE; }
or                                          { return OR; }
and                                         { return AND; }
not                                         { return NOT; }
read                                        { return READ; }
write                                       { return WRITE; }
{DIGIT}+                                    { yylval.num = atoi(yytext); return NUM; }
{LETTER}+{DIGIT}*_*{LETTER}*{DIGIT}*        { yylval.id = strdup(yytext); return ID; }
\#[^\n]*\#                                  { ; }
["]{3}(["]{0,2}([^\\"]|\\(.|\n)))*["]{3}    { ; }
\:\:                                        { return START; }
\:\;                                        { return END; }
\=\=                                        { return EQ; }
```

```
\!\=                                    { return NEQ; }
\<\=                                    { return LE; }
\>\=                                    { return GE; }
\"[^\"]*\"                              { yylval.id = strdup(yytext); return STR; }
f\"                                     { BEGIN FSTRING; return FSS; }
[\ \t\r]                                { ; }
(.|\n)                                  { return yytext[0]; }

<FSTRING>\"                             { BEGIN INITIAL; return yytext[0]; }
<FSTRING>\{                             { BEGIN CBRACES; return yytext[0]; }
<FSTRING>[^\{\"]*                       { yylval.id = strdup(yytext); return FSTR; }

<CBRACES>\}                             { BEGIN FSTRING; return yytext[0]; }
<CBRACES>or                             { return OR; }
<CBRACES>and                            { return AND; }
<CBRACES>not                            { return NOT; }
<CBRACES>{DIGIT}+                       { yylval.num = atoi(yytext); return NUM; }
<CBRACES>{LETTER}+_*{LETTER}*{DIGIT}*   { yylval.id = strdup(yytext); return ID; }
<CBRACES>\=\=                           { return EQ; }
<CBRACES>\!\=                           { return NEQ; }
<CBRACES>\<\=                           { return LE; }
<CBRACES>\>\=                           { return GE; }
<CBRACES>[ ]                            { ; }
<CBRACES>(.|\n)                         { return yytext[0]; }
%%
```

A.2 AVLTrees

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "AVLTrees.h"

void ShowAVLTree (AVLTree a) {
    if (a != NULL) {
        printf ("(%s, %d) ", a->key, a->sp);
        ShowAVLTree (a->left);
        ShowAVLTree (a->right);
    }
}

void GraphAVLTreeAux (AVLTree a, FILE *f) {
    if (a != NULL) {
        fprintf (f, "%s [label=\"key: %s\nsp: %d\nclass: %s\ntype: %s\nsize: %d\"];\n",
                 a->key, a->key, a->sp, a->class, a->type, a->size);
        if (a->left != NULL) fprintf (f, "%s -> %s;\n", a->key, a->left->key);
        else if (a->right != NULL) {
```

```
20              fprintf (f, "%d [shape=point];\n", (int) &(a->left));
21              fprintf (f, "%s -> %d;\n", a->key, (int) &(a->left));
22          }
23          if (a->right != NULL) fprintf (f, "%s -> %s;\n", a->key, a->right->key);
24          else if (a->left != NULL){
25              fprintf (f, "%d [shape=point];\n", (int) &(a->right));
26              fprintf (f, "%s -> %d;\n", a->key, (int) &(a->right));
27          }
28          GraphAVLTreeAux (a->left, f);
29          GraphAVLTreeAux (a->right, f);
30      }
31  }
32
33  void GraphAVLTree (AVLTree a) {
34      if (a != NULL) {
35          FILE *f = fopen ("avl.dot", "w");
36          fprintf (f, "digraph G {\n");
37          fprintf (f, "\tlabelloc=\"t\";\n");
38          fprintf (f, "\tlabel=\"AVLTree\";\n");
39          GraphAVLTreeAux (a, f);
40          fprintf (f, "}");
41          fclose (f);
42          system ("dot -Tpng avl.dot > avl.png");
43      }
44  }
45
46  int max (int a, int b) {
47      int r;
48      if (a > b) r = a;
49      else r = b;
50      return r;
51  }
52
53  int height (AVLTree a) {
54      int r;
55      if (a == NULL) r = 0;
56      else r = a->height;
57      return r;
58  }
59
60  int get_balance (AVLTree a) {
61      int r;
62      if (a == NULL) r = 0;
63      else r = height (a->left) - height (a->right);
64      return r;
65  }
66
67  AVLTree Left (AVLTree a) {
```

```
68      AVLTree r = a->right;
69      AVLTree aux = r->left;
70
71      r->left = a;
72      a->right = aux;
73
74      a->height = 1 + max (height (a->left), height (a->right));
75      r->height = 1 + max (height (r->left), height (r->right));
76
77      return r;
78  }
79
80  AVLTree Right (AVLTree a) {
81      AVLTree r = a->left;
82      AVLTree aux = r->right;
83
84      r->right = a;
85      a->left = aux;
86
87      a->height = 1 + max (height (a->left), height (a->right));
88      r->height = 1 + max (height (r->left), height (r->right));
89
90      return r;
91  }
92
93  int size (AVLTree a) {
94      int r;
95      if (a == NULL) r = 0;
96      else r = 1 + size (a->left) + size(a->right);
97      return r;
98  }
99
100 void insertAVL (AVLTree *a, char *key, char *class, char *type, int size, int x) {
101     if ((*a) == NULL) {
102         *a = (AVLTree) malloc (sizeof (struct node));
103         (*a)->sp = x;
104         (*a)->key = strdup(key);
105         (*a)->class = strdup(class);
106         (*a)->type = strdup(type);
107         (*a)->size = size;
108         (*a)->height = 1;
109         (*a)->left = NULL;
110         (*a)->right = NULL;
111     }
112     else if (strcmp(key, (*a)->key) < 0)
113         insertAVL (&((*a)->left), key, class, type, size, x);
114     else if (strcmp(key, (*a)->key) > 0)
115         insertAVL (&((*a)->right), key, class, type, size, x);
```

```
116
117     (*a)->height = 1 + max (height ((*a)->left), height ((*a)->right));
118     int balance = get_balance (*a);
119
120     if (balance > 1 && strcmp(key, (*a)->left->key) <= 0) *a = Right (*a);
121     else if (balance > 1 && strcmp(key, (*a)->left->key) > 0) {
122         (*a)->left = Left ((*a)->left);
123         *a = Right (*a);
124     }
125     else if (balance < -1 && strcmp(key, (*a)->right->key) >= 0) *a = Left (*a);
126     else if (balance < -1 && strcmp(key, (*a)->right->key) < 0) {
127         (*a)->right = Right ((*a)->right);
128         *a = Left (*a);
129     }
130 }
131
132 int isBSTree (AVLTree a) {
133     int r;
134     if (a == NULL) r = 1;
135     else if (a->left != NULL && strcmp(a->left->key, a->key) > 0) r = 0;
136     else if (a->right != NULL && strcmp(a->right->key, a->key) < 0) r = 0;
137     else if (!isBSTree (a->right) || !isBSTree (a->left)) r = 0;
138     else r = 1;
139     return r;
140 }
141
142 int searchAVL (AVLTree a, char *key, char **class, char **type, int *size, int *sp) {
143     int r = 1;
144     if (a == NULL) r = 0;
145     else if (strcmp (key, a->key) < 0) r = searchAVL (a->left, key, class, type, size, sp);
146     else if (strcmp (key, a->key) > 0) r = searchAVL (a->right, key, class, type, size, sp);
147     else {
148         asprintf (class, "%s", a->class);
149         asprintf (type, "%s", a->type);
150         *sp = a->sp;
151         *size = a->size;
152     }
153     return r;
154 }
```

A.3 Código Yacc

```
1  %{
2  #include "translator.h"
3
4  int DEBUG, VERBOSE;
5  int ERROR = 0, FUNC = 0;
```

```
6  int sp_count = 0;
7  int fcount = 0;
8  int list_size = 0;
9  AVLTree vars = NULL;
10 FILE *vm;
11
12 int yylex ();
13 void yyerror (char *s);
14 %}
15 %error-verbose
16 %locations
17
18 %union {
19   int num;
20   char *id;
21   char *inst;
22 }
23
24 %token INT VOID
25 %token <id> ID STR FSTR
26 %token <num> NUM
27
28 %token AND OR NOT
29 %token EQ NEQ GE LE
30
31 %token FOR IF ELSE
32 %token START END
33
34 %token MAIN READ WRITE
35 %token FSS
36
37 %type <inst> main funcs declrs declr decllist singdecl list
38 %type <inst> instrs instr atr read write cond cycle fcall
39 %type <inst> par factor term expr lexpr arithm fstring string
40
41 %start L
42
43 %%
44
45 L : declrs START funcs '|' main END { fprintf (vm, "%sstart\n%sstop\n%s", $1, $5, $3); }
46   | declrs '|' main END             { fprintf (vm, "%sstart\n%sstop\n", $1, $3); }
47   | error '\n'
48   ;
49
50 endline : '\n' | ';' ;
51
52 main : '|' MAIN '|' '|' instrs      { asprintf (&$$, "%s", $5); }
53      ;
```

```
54
55  funcs : START VOID ID START instrs END funcs  { declrFunc (&$$, $3, $5, $7, &vars, "void"); }
56        | VOID ID START instrs END funcs        { declrFunc (&$$, $2, $4, $6, &vars, "void"); }
57        | '\n' funcs                            { asprintf (&$$, "%s", $2); }
58        | error endline                         { asprintf (&$$, "%s", ""); }
59        |                                       { asprintf (&$$, "%s", ""); }
60        ;
61
62  instrs : instrs instr   { asprintf (&$$, "%s%s", $1, $2); }
63         | error endline  { asprintf (&$$, "%s",""); }
64         |                { asprintf (&$$, "%s", ""); }
65         ;
66
67  instr : atr endline      { asprintf (&$$, "%s", $1); }
68        | write endline    { asprintf (&$$, "%s", $1); }
69        | cond endline     { asprintf (&$$, "%s", $1); }
70        | cycle endline    { asprintf (&$$, "%s", $1); }
71        | fcall endline    { asprintf (&$$, "%s", $1); }
72        | '\n'             { asprintf (&$$, "%s", ""); }
73        ;
74
75  fcall : ID '(' ')'            { funcCall (&$$, $1, &vars); }
76        ;
77
78  cycle : FOR '(' ID ',' expr ',' expr ')' START instrs END                {
79          forStartEnd (&$$, $3, $5, $7, $10, &vars, &fcount);
80        }
81        | FOR '(' ID ',' expr ',' expr ',' expr ')' START instrs END        {
82          forStep (&$$, $3, $5, $7, $9, $12, &vars, &fcount);
83        }
84        | FOR '(' ID ',' ID ')' '-' '>' ID START instrs END                {
85          forArrayIV (&$$, $3, $5, $9, $11, &vars, &fcount);
86        }
87        | FOR '(' ID '=' expr ',' expr ',' ID '=' expr ')' START instrs END  {
88          forCond (&$$, $3, $9, $5, $7, $11, $14, &vars, &fcount);
89        }
90        ;
91
92  cond : IF expr START instrs END                          { ifInstr (&$$, $2, $4, &fcount); }
93       | IF expr START instrs START ELSE START instrs END {
94         ifElse (&$$, $2, $4, $8, &fcount);
95       }
96       | IF expr START instrs START ELSE cond             {
97         ifElseif (&$$, $2, $4, $7, &fcount);
98       }
99       ;
100
101 write : WRITE '(' string ')'   { asprintf (&$$, "%s", $3); }
```

```
102       ;
103
104 read : READ '(' string ')'     { asprintf (&$$, "%sread\natoi\n", $3); }
105      ;
106
107 string : FSS fstring '"'    { asprintf (&$$, "%s", $2); }
108        | STR                { asprintf (&$$, "pushs %s\nwrites\n", $1); }
109        |                    { asprintf (&$$, "%s", ""); }
110        ;
111
112 fstring : fstring '{' expr '}'   { asprintf (&$$, "%s%swritei\n", $1, $3); }
113         | fstring FSTR           { asprintf (&$$, "%spushs \"%s\"\nwrites\n", $1, $2); }
114         | '{' expr '}'           { asprintf (&$$, "%swritei\n", $2); }
115         | FSTR                   { asprintf (&$$, "pushs \"%s\"\nwrites\n", $1); }
116         ;
117
118 atr : ID '=' expr                 { exprAtr (&$$, $1, $3, &vars); }
119     | ID '=' read                 { readAtr (&$$, $1, $3, &vars); }
120     | ID '[' expr ']' '=' expr    {
121       arrayAtr (&$$, $1, $3, $6, &vars, &fcount, @3.first_line);
122     }
123     | ID '[' expr ']' '=' read    {
124       readArrayAtr (&$$, $1, $3, $6, &vars, &fcount, @3.first_line);
125     }
126     ;
127
128 declrs : declrs declr    { asprintf (&$$, "%s%s", $1, $2); }
129        | error endline   { asprintf (&$$, "%s", ""); }
130        |                 { asprintf (&$$, "%s", ""); }
131        ;
132
133 declr : INT decllist endline       { asprintf (&$$, "%s", $2); }
134       | '\n'                       { asprintf (&$$, "%s", ""); }
135       ;
136
137 decllist : singdecl ',' decllist   { asprintf (&$$, "%s%s", $1, $3); }
138          | singdecl                { asprintf (&$$, "%s", $1); }
139          ;
140
141 singdecl : ID                              { declaration (&$$, $1, &sp_count, &vars); }
142          | ID '[' NUM ']'                  { declrArray (&$$, $1, $3, &sp_count, &vars); }
143          | ID '=' expr                     { declrExpr (&$$, $1, $3, &vars, &sp_count); }
144          | ID '=' read                     { declrRead (&$$, $1, $3, &vars, &sp_count); }
145          | ID '[' NUM ']' '=' '[' list ']'  {
146            decList (&$$, $1, $3, $7, &vars, &sp_count, &list_size);
147          }
148          ;
149
```

```
150 list : expr ',' list          { asprintf (&$$, "%s%s", $1, $3); list_size++; }
151      | expr                   { asprintf (&$$, "%s", $1); list_size++; }
152      ;
153
154 expr : expr AND lexpr    {
155        asprintf (&$$, "%sdup 1\njz func%d\n%smul\nfunc%d:\n",
156                  $1, fcount, $3, fcount);
157        fcount++;
158      }
159      | expr OR lexpr     {
160        asprintf (&$$, "%snot\ndup 1\njz func%d\n%snot\nmul\nnot\nfunc%d:\n",
161                  $1, fcount, $3, fcount);
162        fcount++;
163      }
164      | NOT lexpr         { asprintf (&$$, "%snot\n", $2); }
165      | lexpr             { asprintf (&$$, "%s", $1); }
166      ;
167
168 lexpr : lexpr EQ arithm     { asprintf (&$$, "%s%sequal\n", $1, $3); }
169       | lexpr NEQ arithm    { asprintf (&$$, "%s%sequal\nnot\n", $1, $3); }
170       | lexpr GE arithm     { asprintf (&$$, "%s%ssupeq\n", $1, $3); }
171       | lexpr LE arithm     { asprintf (&$$, "%s%sinfeq\n", $1, $3); }
172       | lexpr '>' arithm    { asprintf (&$$, "%s%ssup\n", $1, $3); }
173       | lexpr '<' arithm    { asprintf (&$$, "%s%sinf\n", $1, $3); }
174       | arithm              { asprintf (&$$, "%s", $1); }
175       ;
176
177 arithm : arithm '+' term    { asprintf (&$$, "%s%sadd\n", $1, $3); }
178        | arithm '-' term    { asprintf (&$$, "%s%ssub\n", $1, $3); }
179        | term               { asprintf (&$$, "%s", $1); }
180        ;
181
182 term : term '*' par    { asprintf (&$$, "%s%smul\n", $1, $3); }
183      | term '/' par    { asprintf (&$$, "%s%sdiv\n", $1, $3); }
184      | term '%' par    { asprintf (&$$, "%s%smod\n", $1, $3); }
185      | par             { asprintf (&$$, "%s", $1); }
186      ;
187
188 par : '(' expr ')'        { asprintf (&$$, "%s", $2); }
189     | factor              { asprintf (&$$, "%s", $1); }
190     ;
191
192 factor : NUM                    { asprintf (&$$, "pushi %d\n", $1); }
193        | ID                     { factorId (&$$, $1, &vars); }
194        | ID '[' expr ']'        { factorArray (&$$, $1, $3, &vars, &fcount, @3.last_line); }
195        | '-' NUM                { asprintf (&$$, "pushi %d\n", -$2); }
196        | '-' ID                 { negfactorId (&$$, $2, &vars); }
197        | '-' ID '[' expr ']'    {
```

```
198           negfactorArray (&$$, $2, $4, &vars, &fcount, @3.first_line);
199         }
200         ;
201
202 %%
203
204 #include "lex.yy.c"
205
206 void yyerror (char *s) {
207     if (!ERROR) printf ("\n%s\n", repeatChar ('-', 90));
208     fprintf (stderr,"Line %d: %s\n", yylineno, s);
209     printf ("%s\n", repeatChar ('-', 90));
210     ERROR = 1;
211 }
212
213 int main(int argc, char **argv) {
214     if (!strcmp (argv[1], "yes")) DEBUG = 1;
215     else DEBUG = 0;
216     if (!strcmp (argv[2], "yes")) VERBOSE = 1;
217     else VERBOSE = 0;
218     vm = fopen ("program.vm", "w");
219
220     if (VERBOSE) printf ("-> Started parsing\n");
221     yyparse ();
222     fclose (vm);
223
224     if (!ERROR && VERBOSE) {
225         printf ("-> Parsing complete with no compile time errors.\n");
226         printf ("-> VM program generated\n");
227     }
228     else if (ERROR && VERBOSE) {
229         system ("rm program.vm");
230         printf ("\n-> VM program file deleted. Errors found while parsing.\n");
231         printf ("-> Correct them in order to be able to run the program.\n");
232     }
233     else if (ERROR) system ("rm program.vm");
234
235     system ("rm a.out lex.yy.c y.tab.h y.tab.c");
236
237     if (DEBUG && !ERROR) {
238             GraphAVLTree (vars);
239             system ("rm avl.dot");
240             if (VERBOSE)
241                 printf ("-> Debug mode detected. VM file kept and variables
242                         AVLTree image generated.\n");
243     }
244     else if (DEBUG) {
245         GraphAVLTree (vars);
```

```
246         system ("rm avl.dot");
247         if (VERBOSE)
248             printf ("-> Debug mode detected. Variables AVLTree image generated.\n");
249     }
250 
251     return 0;
252 }
```

A.4 Ficheiro de tradução

```
1  #include "translator.h"
2  
3  void ifInstr (char **r, char *expr, char *instr, int *count) {
4      asprintf (r, "%sjz cond%d\n%scond%d:\n", expr, *count, instr, *count);
5      *count = *count + 1;
6  }
7  
8  void ifElse (char **r, char *expr, char *instr1, char *instr2, int *count) {
9      asprintf (r, "%sjz cond%d\n%sjump cond%d\ncond%d:\n%scond%d:\n",
10                expr, *count, instr1, *count + 1, *count, instr2, *count + 1);
11     *count = *count + 2;
12 }
13 
14 void ifElseif (char **r, char *expr, char *instr, char *cond, int *count) {
15     asprintf (r, "%sjz cond%d\n%sjump cond%d\ncond%d:\n%s",
16                expr, *count, instr, *count - 1, *count, cond);
17     *count = *count + 1;
18 }
19 
20 void exprAtr (char **r, char *id, char *expr, AVLTree *vars) {
21     int sp, size;
22     char *class, *type;
23     if (!searchAVL (*vars, id, &class, &type, &size, &sp)) notDeclared (r, id, "variable");
24     else if (strcmp (class, "var")==0) asprintf (r, "%sstoreg %d\n", expr, sp);
25     else if (strcmp (class, "array")==0) assignIntArray (r, id);
26     else if (strcmp (class, "func")==0) assignFunc (r, id);
27 }
28 
29 void arrayAtr (char **r, char *id, char *instr, char *expr, AVLTree *vars,
30                int *count, int line) {
31     int sp, size;
32     char *class, *type;
33     if (!searchAVL (*vars, id, &class, &type, &size, &sp)) notDeclared (r, id, "array");
34     else if (strcmp (class, "array")==0) {
35         char *error_str = outOfRange (id, instr, size, count, line);
36         asprintf (r, "%spushgp\npushi %d\npadd\n%s%sstoren\n",
37                   error_str, sp, instr, expr);
```

```
38      }
39      else if (strcmp (class, "var")==0) intIndex (r, id);
40      else if (strcmp (class, "func")==0) assignFunc (r, id);
41  }
42
43  void readAtr (char **r, char *id, char *instr, AVLTree *vars) {
44      int sp, size;
45      char *class, *type;
46      if (!searchAVL (*vars, id, &class, &type, &size, &sp)) notDeclared (r, id, "variable");
47      else if (strcmp (class, "var")==0) asprintf (r, "%sstoreg %d\n", instr, sp);
48      else if (strcmp (class, "array")==0) assignIntArray (r, id);
49      else if (strcmp (class, "func")==0) assignFunc (r, id);
50  }
51
52  void readArrayAtr (char **r, char *id, char *instr1, char *instr2, AVLTree *vars,
53                     int *count, int line) {
54      int sp, size;
55      char *class, *type;
56      if (!searchAVL (*vars, id, &class, &type, &size, &sp)) notDeclared (r, id, "array");
57      else if (strcmp (class, "array")==0) {
58          char *error_str = outOfRange (id, instr1, size, count, line);
59          asprintf (r, "%spushgp\npushi %d\npadd\n%s%s\nstoren\n",
60                    error_str, sp, instr1, instr2);
61      }
62      else if (strcmp (class, "var")==0) intIndex (r, id);
63      else if (strcmp (class, "func")==0) assignFunc (r, id);
64  }
65
66  void declaration (char **r, char *id, int *count, AVLTree *vars) {
67      char *class, *type; int size, sp;
68      if (searchAVL (*vars, id, &class, &type, &size, &sp)) reDeclaration (r, id);
69      else {
70          insertAVL (vars, id, "var", "int", 1, *count);
71          asprintf (r, "pushn 1\n");
72          *count = *count + 1;
73      }
74  }
75
76  void declrArray (char **r, char *id, char *index, char *count, AVLTree *vars) {
77      char *class, *type; int size, sp;
78      if (searchAVL (*vars, id, &class, &type, &size, &sp)) reDeclaration (r, id);
79      else if (index <= 0) myyyerror (r, "Array size must be a natural number.");
80      else {
81          insertAVL (vars, id, "array", "int", index, *count);
82          asprintf (r, "pushn %d\n", index);
83          *count = *count + index;
84      }
85  }
```

```
86
87  void declrExpr (char **r, char *id, char *expr, AVLTree *vars, int *count) {
88      char *class, *type; int size, sp;
89      if (searchAVL (*vars, id, &class, &type, &size, &sp)) reDeclaration (r, id);
90      else {
91          insertAVL (vars, id, "var", "int", 1, *count);
92          asprintf (r, "pushn 1\n%sstoreg %d\n", expr, *count);
93          *count = *count + 1;
94      }
95  }
96
97  void declrRead (char **r, char *id, char *instr, AVLTree *vars, int *count) {
98      char *class, *type; int size, sp;
99      if (searchAVL (*vars, id, &class, &type, &size, &sp)) reDeclaration (r, id);
100     else {
101         insertAVL (vars, id, "var", "int", 1, *count);
102         asprintf (r, "pushn 1\n%sstoreg %d\n", instr, *count);
103         *count = *count + 1;
104     }
105 }
106
107 void decList (char **r, char *id, int index, char *instr, AVLTree *vars,
108               int *count, int *size) {
109     if (*size == index) {
110         char *class, *type; int size, sp;
111         if (searchAVL (*vars, id, &class, &type, &size, &sp)) reDeclaration (r, id);
112         else {
113             insertAVL (vars, id, "array", "int", index, *count);
114             asprintf (r, "%s", instr);
115             *count = *count + index;
116         }
117     }
118     else indexSizeDontMatch (r, id, index, *size);
119     *size = 0;
120 }
121
122 void declrFunc (char **r, char *id, char *instrs1, char *instrs2,
123                 AVLTree *vars, char *ftype) {
124     char *class, *type; int size, sp;
125     if (searchAVL (*vars, id, &class, &type, &size, &sp)) reDeclaration (r, id);
126     else {
127         insertAVL (vars, id, "func", ftype, 1, -1);
128         if (strcmp (ftype, "void")==0)
129             asprintf (r, "\n%s:\nnop\n%sreturn\n%s", id, instrs1, instrs2);
130         else if (strcmp (ftype, "int")==0)
131             asprintf (r, "\n%s:\nnop\n%sstorel -1\nreturn\n%s", id, instrs1, instrs2);
132     }
133 }
```

```
134
135 void factorId (char **r, char *id, AVLTree *vars) {
136     int sp, size;
137     char *class, *type;
138     if (!searchAVL (*vars, id, &class, &type, &size, &sp)) notDeclared (r, id, "variable");
139     else if (strcmp (class, "var")==0) asprintf (r, "pushg %d\n", sp);
140     else if (strcmp (class, "array")==0) assignIntArray (r, id);
141     else if (strcmp (class, "func")==0) assignFunc (r, id);
142 }
143
144 void factorArray (char **r, char *id, char *instr, AVLTree *vars, int *count, int line) {
145     int sp, size;
146     char *class, *type;
147     if (!searchAVL (*vars, id, &class, &type, &size, &sp)) notDeclared (r, id, "array");
148     else if (strcmp (class, "array")==0) {
149         char *error_str = outOfRange (id, instr, size, count, line);
150         asprintf (r, "%spushgp\npushi %d\npadd\n%sloadn\n", error_str, sp, instr);
151     }
152     else if (strcmp (class, "var")==0) intIndex (r, id);
153     else if (strcmp (class, "func")==0) assignFunc (r, id);
154 }
155
156 void negfactorId (char **r, char *id, AVLTree *vars) {
157     int sp, size;
158     char *class, *type;
159     if (!searchAVL (*vars, id, &class, &type, &size, &sp)) notDeclared (r, id, "variable");
160     else if (strcmp (class, "var")==0) asprintf (r, "pushg %d\npushi -1\nmul\n", sp);
161     else if (strcmp (class, "array")==0) assignIntArray (r, id);
162     else if (strcmp (class, "func")==0) assignFunc (r, id);
163 }
164
165 void negfactorArray (char **r, char *id, char *instr, AVLTree *vars, int *count, int line) {
166     int sp, size;
167     char *class, *type;
168     if (!searchAVL (*vars, id, &class, &type, &size, &sp)) notDeclared (r, id, "array");
169     else if (strcmp (class, "array")==0) {
170         char *error_str = outOfRange (id, instr, size, count, line);
171         asprintf (r, "%spushgp\npushi %d\npadd\n%sloadn\npushi -1\nmul\n",
172                   error_str, sp, instr);
173     }
174     else if (strcmp (class, "var")==0) intIndex (r, id);
175     else if (strcmp (class, "func")==0) assignFunc (r, id);
176 }
177
178 void forStartEnd (char **r, char *id, char *expr1, char *expr2, char *instr,
179                   AVLTree *vars, int *count) {
180     int sp, size;
181     char *class, *type;
```

```
182     if (!searchAVL (*vars, id, &class, &type, &size, &sp)) notDeclared (r, id, "variable");
183     else if (strcmp (class, "var")==0) {
184         asprintf (r, "%s%sinf\njz cycle%d\n%sstoreg %d\ncycle%d:\n%spushg %d\npushi 1\nadd\n
185                   storeg %d\npushg %d\n%ssupeq\njz cycle%d\ncycle%d:\n",
186                   expr1, expr2, *count + 1, expr1, sp, *count, instr, sp,
187                   sp, sp, expr2, *count, *count + 1);
188         *count = *count + 2;
189     }
190     else if (strcmp (class, "array")==0)
191         myyyerror (r, "Can't iterate variable of array, use integer instead.");
192     else if (strcmp (class, "func")==0)
193         myyyerror (r, "Can't iterate function, use integer instead.");
194 }
195
196 void forStep (char **r, char *id, char *expr1, char *expr2, char *expr3, char *instr,
197               AVLTree *vars, int *count) {
198     int sp, size;
199     char *class, *type;
200     if (!searchAVL (*vars, id, &class, &type, &size, &sp)) notDeclared (r, id, "variable");
201     else if (strcmp (class, "var")==0) {
202         char *aux, *aux1;
203         asprintf (&aux, "%s%s%spushi 0\ninf\njz cycle%d\ncycle%d:\ninfeq\njz cycle%d\n
204                   jump cycle%d\ncycle%d:\nsupeq\njz cycle%d\njump cycle%d\n",
205                   expr1, expr2, expr3, *count + 1, *count, *count + 2, *count + 4,
206                   *count + 1, *count + 2, *count + 4);
207         asprintf (&aux1, "%spushi 0\ninf\njz cycle%d\ncycle%d:\ninfeq\njz cycle%d\n
208                   jump cycle%d\ncycle%d:\nsupeq\njz cycle%d\njump cycle%d\n",
209                   expr3, *count + 6, *count + 5, *count + 3, *count + 7,
210                   *count + 6, *count + 3, *count + 7);
211         asprintf (r, "%scycle%d:\n%sstoreg %d\ncycle%d:\n%spushg %d\n%sadd\n
212                   storeg %d\npushg %d\n%s%scycle%d:\n",
213                   aux, *count + 2, expr1, sp, *count + 3, instr, sp,
214                   expr3, sp, sp, expr2, aux1, *count + 7);
215         *count = *count + 8;
216     }
217     else if (strcmp (class, "array")==0)
218         myyyerror (r, "Can't iterate variable of array, use integer instead.");
219     else if (strcmp (class, "func")==0)
220         myyyerror (r, "Can't iterate function, use integer instead.");
221 }
222
223 void forArrayIV (char **r, char *index, char *v, char *id, char *instr,
224                  AVLTree *vars, int *count) {
225     int spi, spv, spa, size;
226     char *class, *type, *ind_id;
227     if (!searchAVL (*vars, index, &class, &type, &size, &spi))
228         notDeclared (r, index, "variable");
229     else if (strcmp (class, "var")!=0)
```

```
230         myyyerror (r, "Must use integer to hold array indices.");
231     else if (!searchAVL (*vars, v, &class, &type, &size, &spv))
232         notDeclared (r, v, "variable");
233     else if (strcmp (class, "var")!=0)
234         myyyerror (r, "Must use integer to hold array values.");
235     else if (!searchAVL (*vars, id, &class, &type, &size, &spa))
236         notDeclared (r, id, "array");
237     else if (strcmp (class, "array")==0) {
238         char *init, *update, *jump;
239         asprintf (&init, "pushi 0\nstoreg %d\npushgp\npushi %d\npadd\npushi 0\nloadn\n
240                   storeg %d\n", spi, spa, spv);
241         asprintf (&update, "pushi %d\npushg %d\npushi 1\nadd\nstoreg %d\npushgp\n
242                   pushi %d\npadd\npushg %d\nloadn\nstoreg %d\n",
243                   size, spi, spi, spa, spi, spv);
244         asprintf (&jump, "pushg %d\ninfeq\njz cycle%d\n", spi, *count);
245         asprintf (r, "%scycle%d:\n%s%s%s", init, *count, instr, update, jump);
246         *count = *count + 1;
247     }
248     else if (strcmp (class, "var")==0)
249         myyyerror (r, "Can't iterate integer, use array instead.");
250     else if (strcmp (class, "func")==0)
251         myyyerror (r, "Can't iterate function, use integer instead.");
252 }
253
254 void forCond (char **r, char *id1, char *id2, char *expr1, char *expr2, char *expr3,
255               char *instrs, AVLTree *vars, int *count) {
256     int sp, size;
257     char *class, *type;
258     if (!searchAVL (*vars, id1, &class, &type, &size, &sp))
259         notDeclared (r, id1, "variable");
260     else if (strcmp (class, "var")!=0)
261         myyyerror (r, "Must use integer to hold array indices.");
262     else if (strcmp (id1, id2) != 0)
263         myyyerror (r, "Update rule and declaration rule use different variables.");
264     else {
265         asprintf (r, "%sstoreg %d\n%sjz cycle%d\ncycle%d:\n%s%sstoreg %d\n%s
266                   not\njz cycle%d\ncycle%d:\n",
267                   expr1, sp, expr2, *count + 1, *count, instrs,
268                   expr3, sp, expr2, *count, *count + 1);
269         *count = *count + 2;
270     }
271 }
272
273 void funcCall (char **r, char *id, AVLTree *vars) {
274     int sp, size;
275     char *class, *type;
276     if (!searchAVL (*vars, id, &class, &type, &size, &sp))
277         notDeclared (r, id, "function");
```

```
278     else if (strcmp (class, "func")!=0)
279         myyyerror (r, "Variable not callable.");
280     else if (strcmp (type, "void")!=0)
281         myyyerror (r, "Non-void function not assigning its return.");
282     else asprintf (r, "pusha %s\ncall\nnop\n", id);
283 }
284
285 void funcAtr (char **r, char *id1, char *id2, AVLTree *vars) {
286     int sp, size;
287     char *class, *type1, *type2;
288     if (!searchAVL (*vars, id2, &class, &type1, &size, &sp))
289         notDeclared (r, id2, "function");
290     else if (strcmp (class, "func")!=0)
291         myyyerror (r, "Variable not callable.");
292     else if (strcmp (type1, "void")==0)
293         myyyerror (r, "Can't atribute value of void function");
294     else if (!searchAVL (*vars, id1, &class, &type2, &size, &sp))
295         notDeclared (r, id1, "variable");
296     else if (strcmp (type1, type2)!=0)
297         myyyerror (r, "Function and variable types don't match.");
298     else asprintf (r, "pushi 0\npusha %s\ncall\nnop\nstoreg %d\n", id2, sp);
299 }
```

A.5 Ficheiro C para erros

```
1  #include "errors.h"
2
3  void myyyerror (char **r, char *s) {
4      asprintf (r, "%s", "");
5      yyerror (s);
6  }
7
8  void notDeclared (char **r, char *id, char *type) {
9      char *error_str;
10     asprintf (&error_str, "Can't access %s \"%s\" because it hasn't been declared.",
11               type, id);
12     myyyerror(r, error_str);
13 }
14
15 char *outOfRange (char *id, char *instr, int size, int *count, int line) {
16     char *r, *inferior, *greater, *error_inf, *error_gr, *error_str;
17     asprintf (&error_gr, "\\n\\n%s\\nLine %d: Index of array \'%s\' too high for its
18               size.\\n%s\\n",
19               repeatChar ('-', 90), line, id, repeatChar ('-', 90));
20     asprintf (&error_inf, "\\n\\n%s\\nLine %d: Index of array \'%s\' smaller than
21               zero.\\n%s\\n",
22               repeatChar ('-', 90), line, id, repeatChar ('-', 90));
```

```
23      asprintf (&greater, "%spushi %d\nsupeq\njz func%d\nerr \"%s\"\nstop\nfunc%d:\n",
24                  instr, size, *count, error_gr, *count);
25      asprintf (&inferior, "%spushi 0\ninf\njz func%d\nerr \"%s\"\nstop\nfunc%d:\n",
26                  instr, *count + 1, error_inf, *count + 1);
27      asprintf (&error_str, "%s%s", inferior, greater);
28      *count = *count + 2;
29
30      return error_str;
31  }
32
33  void assignIntArray (char **r, char *id) {
34      char *error_str;
35      asprintf (&error_str, "Array \"%s\" can't be treated as an integer.", id);
36      myyyerror (r, error_str);
37  }
38
39  void intIndex (char **r, char *id) {
40      char *error_str;
41      asprintf (&error_str, "Integer \"%s\" can't be indexed.", id);
42      myyyerror (r, error_str);
43  }
44
45  void indexSizeDontMatch (char **r, char *id, int index, int size) {
46      char *error_str;
47      asprintf (&error_str, "Array \"%s\" declared with size %d but list has size %d.",
48                id, index, size);
49      myyyerror (r, error_str);
50  }
51
52  void reDeclaration (char **r, char *id) {
53      char *error_str;
54      asprintf (&error_str, "Variable \"%s\" redeclared.", id);
55      myyyerror (r, error_str);
56  }
57
58  void assignFunc (char **r, char *id) {
59      char *error_str;
60      asprintf (&error_str, "Function \"%s\" value not accessable.", id);
61      myyyerror (r, error_str);
62  }
```