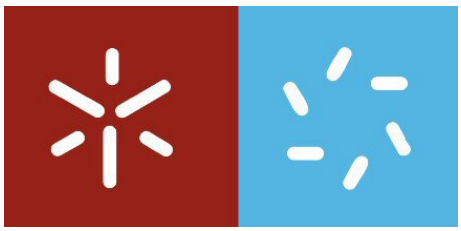


Processamento de Linguagens e Compiladores (3º ano de LCC)

Compilador de uma Linguagem Imperativa Simples

Relatório de Desenvolvimento

Bruna Carvalho
A87982

Márlon Ferreira
A81735

20 de janeiro de 2021

Resumo

Um compilador é uma ferramenta que permite traduzir código que está escrito numa linguagem de programação em código de uma outra linguagem, normalmente para uma de baixo nível. Quando executado, o compilador efetua uma análise léxica com a ajuda de expressões regulares, reconhecendo os tokens da linguagem em questão e removendo espaços em branco e possíveis comentários. Sempre que um token é dado como inválido é gerada uma mensagem de erro.

O analisador léxico funciona juntamente com o analisador sintático comparando a stream de tokens gerados com as regras de produção via gramáticas independentes de contexto (GIC's), resultando na criação de uma árvore de parsing com os tokens mencionados anteriormente. Sempre que o texto fonte não respeitar as regras de produção é gerado um erro.

Tendo uma árvore de parsing com os tokens do texto fonte, passamos à análise semântica onde é verificada a consistência de acordo com a definição da linguagem. É nesta fase que são, por exemplo, verificadas variáveis não declaradas e conflitos de tipos comparando com a informação que é armazenada na tabela de símbolos. Esta tabela será também útil para etiquetar as variáveis de acordo com a sua localização na stack. Neste processo é também gerado o código de acordo com as especificações da linguagem final.

Utilizando ferramentas que temos ao nosso dispor como Flex (Fast Lexical Analyzer Generator) e Yacc/Bison e implementando a ideia acima descrita, desenvolvemos uma linguagem imperativa simples e o respetivo compilador.

Conteúdo

Capítulo 1

Introdução

Supervisor: Prof. Pedro Rangel Henriques
Área: Processamento de Linguagens

1.1 Compilador de uma linguagem imperativa simples

Neste relatório são descritas a especificação da linguagem criada, a estratégia de resolução utilizada e as decisões que lideraram o desenho da linguagem e gramática. A linguagem desenhada é muito semelhante à linguagem já existente C. No capitulo 2 é descrita a especificação da linguagem que inclui a declaração e atribuição de variáveis inteiras, arrays e até funções simples. Inclui também a especificação de expressões, condições e instruções condicionais e de ciclo. Nesse mesmo capítulo é também documentada a análise léxica e semântica e a gramática da linguagem. No capitulo 3 é descrito o processo de compilação em que é feita a tradução para o pseudo-código Assembly da Máquina Virtual VM, respeitando a documentação disponibilizada pelo professor. Neste relatório disponibilizaremos o nosso código assim como alguns exemplos de programas e respetivos outputs.

1.2 Enunciado proposto

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 *cíclicas* 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** conforme o Número do seu Grupo módulo 3 seja 0, 1 ou 2.

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 ao Gerador **Yacc/Flex**.
O compilador deve gerar **pseudo-código**, Assembly da Máquina Virtual VM cuja documentação completa está disponibilizada no Bb.

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. Esse conjunto terá de conter, no mínimo, os 4 primeiros exemplos abaixo e um dos 2 últimos conforme a sua escolha acima:

- ler 4 números e dizer se podem ser os lados de um quadrado.
- ler um inteiro N, depois ler N números e escrever o menor deles.
- ler N (constante do programa) números e calcular e imprimir o seu produtório.
- contar e imprimir os números ímpares de uma sequência de números naturais.
- ler e armazenar N números num array; imprimir os valores por ordem inversa.
- invocar e usar num programa seu uma função 'potencia()', que começa por ler do input a base B e o expoente E e retorna o valor B^E.

Capítulo 2

Especificação da Linguagem

2.1 Sintaxe

Os únicos tipos desta linguagem são inteiros e arrays de inteiros. Os tipos boolean são representados pelo inteiro 1 ou 0 para verdadeiro ou falso, respetivamente.

2.1.1 Expressões

Existem dois tipos de expressões nesta linguagem:

- Expressões aritméticas;
- Expressões condicionais.

Neste relatório, quando são referidas expressões ou `expressions` deve-se entender expressões aritméticas e quando são referidas condições ou `conditions` deve-se entender expressões condicionais. Neste último tipo de expressões estão incluídos os operadores relacionais e lógicos.

- Nas **expressões** vamos encontrar dois cenários distintos que serão componentes unários. É o caso dos `inteiros`, `identificadores`, elementos de um `array` ou a própria função `read()`. Estes componentes com a introdução dos operadores +, `-`, `*`, `/` e `%` permitem a criação de expressões mais complexas. A utilização de parêntesis também é possível. Exemplos:

  ```
  1 + 2 * 4    var * 2    10 / 5      4 % 2       var - 1
  ```

  A linguagem respeita as regras de precedência da matemática. Exemplo:

  ```
  6 * 2 / (2 + 1 * 2 / 3 + 6) + 8 * (8 / 4) =
  = 6 * 2 / (2 + 2 / 3 + 6) + 8 * (8 / 4) =
  = 6 * 2 / (2 + 0 + 6) + 8 * (8 / 4) =
  = 6 * 2 / 8 + 8 * (8 / 4) = 6 * 2 / 8 + 8 * 2 =
  = 12 / 8 + 8 * 2 = 1 + 8 * 2 =
  = 1 + 16 = 17
  ```

- À semelhança das expressões, as `condições` têm os mesmos componentes unários mas com a exceção do `read()`. São utilizados os operadores `>`, `<`, `>=`, `<=`, `==`, `!=`, `not`, `and` e `or`. Também é respeitada a ordem de precedência. Exemplos:

```
(x < 6) or (y > 10)    not (x >= 4 and x <= 7)    (x != y and y == z)
```

2.1.2 Declarações

As declarações são efetuadas no inicio do programa antes da função main(). As declarações de variáveis podem ser do tipo inteiro e do tipo array de inteiros. Nesta mesma secção do programa são também declaradas as funções. As declarações seguem o seguinte formato:

- Sem atribuição de valor:

```
int identificador;
int identificador[constante];
```

  Exemplos:

```
int input;
int lista[7];
```

- Com atribuição de valor[1]:

```
int identificador = expressão;
```

  Exemplos:

```
int semana = 7;
int mes = semana * 4;
```

- Declaração de funções sem parâmetros[2]:

```
int identificador() {
  ...
  instruções
  ...
  return expressão;
}
```

2.1.3 Instruções

As instruções ou `statements` existentes são as atribuições, condições, ciclos e também a função pré definida `print()`. As instruções só podem existir dentro de uma função.

[1] Os arrays são sempre declarados com todos os elementos inicializados a 0.

[2] Nesta linguagem as funções nunca têm parâmetros.

Atribuição

As atribuições só podem ser efetuadas dentro de funções e apenas em variáveis anteriormente declaradas. As atribuições seguem o seguinte formato:

- Atribuição de valor a uma variável:

  ```
  identificador = expressão;
  ```

  Exemplos:

  ```
  duzia = 12;
  meiaDuzia = duzia / 2;
  resultado = media();
  input = read();
  ```

- Atribuição de valor a um elemento de um array:

  ```
  identificador[expressão] = expressão;
  ```

  Exemplos:

  ```
  lista[0] = 5;
  lista[duzia] = 7;
  lista[6] = lista[0] + 1;
  lista[2] = foo();
  ```

Condições

Existem dois tipos de condições muito semelhantes: uma do tipo `if then else` e a outra só do tipo `if then`. Estas condições seguem o seguinte formato:

- Na condição `if then else` existe um conjunto de instruções a serem realizadas se a condição for verdadeira e um outro conjunto de instruções a serem realizadas se a condição for falsa.

  ```
  if(condição) {
    ...
    instruções
    ...
  }
  else {
    ...
    instruções
    ...
  }
  ```

- A condição `if then` por sua vez tem um conjunto de instruções a serem realizadas se a condição for verdadeira, mas não tem um conjunto de instruções específico se a condição for falsa. Neste último cenário é executada a próxima instrução no programa.

```
if(condição) {
  ...
  instruções
  ...
}
```

Ciclos

Os ciclos permitem-nos repetir instruções enquanto uma condição se mantiver verdadeira ou falsa. Esta linguagem suporta três formatos diferentes de ciclos: **while-do**, **repeat-until** e **for-do**.

- O ciclo `while`, como é possível deduzir do nome, repete um conjunto de instruções enquanto uma dada condição permanecer verdadeira. Deixando essa condição de ser verdadeira o programa sai do ciclo e executa a próxima instrução do programa.

  ```
  while(condição) {
    ...
    instruções
    ...
  }
  ```

- O ciclo `until` também é muito semelhante sintaticamente com o ciclo while mas este, em vez de repetir as instruções enquanto a condição for verdade, repete as instruções enquanto a condição for falsa.

  ```
  until(condição) {
    ...
    instruções
    ...
  }
  ```

- O ciclo `for` já difere mais dos anteriores sendo que este está mais indicado para ciclos iterativos. É dado um valor inicial e um valor final. As instruções irão repetir-se até que o valor inicial seja igual ao valor final, incrementando esse valor por cada iteração.

  ```
  for(identificador = expressão; expressão) {
    ...
    instruções
    ...
  }
  ```

Escrita

Também considerada uma instrução, a função pré definida `print()` permite que o programa tenha algum output. Este output será sempre um inteiro.

```
print(expressão);
```

2.2 Análise léxica

Com o objetivo de criar um analisador léxico para a nossa linguagem, utilizamos as capacidades da ferramenta Flex (Fast Lexical Analyzer Generator).

Com recurso às expressões regulares criamos um analisador que retorna o `TOKEN` de cada *keyword* pretendida. Estes tokens serão fundamentais para a definição da gramática da linguagem.

O token retornado é um valor numérico que está definido no ficheiro header gerado pelo Yacc/Bison, `y.tab.c` ou `"fileName".tab.c` respetivamente:

```
enum yytokentype
{
  INT = 258,
  MAIN = 259,
  IF = 260,
  ELSE = 261,
  WHILE = 262,
  UNTIL = 263,
  FOR = 264,
  READ = 265,
  PRINT = 266,
  RETURN = 267,
  INTEGER = 268,
  IDENTIFIER = 269,
  OR = 270,
  AND = 271,
  EQ = 272,
  NEQ = 273,
  SUPEQ = 274,
  INFEQ = 275,
  NOT = 276
};
```

Como se pode observar nem todos os símbolos têm um valor associado. Para os símbolos `[`, `]`, `{`, `}`, `(`, `)`, `;`, `+`, `-`, `*`, `/`, `<`, `>`, `=` e `%` o valor do token é a própria representação numérica do carácter pelo que não é preciso definir. Para conseguirmos retornar esse valor e como o valor do yytext é uma string, vamos precisar de extrair o carácter da string, para isso fazemos `return yytext[0]`.

Existem dois casos em que, para além de serem retornados os tokens, é retornado também o respetivo valor, no caso `INTEGER`, e a string em si no caso do `IDENTIFIER`, o próprio número inteiro.

No caso de encontrar um comentário, o compilador léxico ignora-o.

Por fim, todos os caracteres não previstos estão a ser retornados para mais tarde serem tratados pelo YACC e nele vão ser considerados `tokens` não previstos retornando um erro.

```
INTEGER                 \-?[0-9]+
IDENTIFIER              [_a-zA-Z][_a-zA-Z0-9]*
%%
"main"                  { return MAIN; }
"return"                { return RETURN; }
"read()"                { return READ; }
```

```
"print"                     { return PRINT; }
"int"                       { return INT; }
"if"                        { return IF; }
"else"                      { return ELSE; }
"while"                     { return WHILE; }
"for"                       { return FOR; }
"until"                     { return UNTIL; }
"or"                        { return OR; }
"and"                       { return AND; }
"=="                        { return EQ; }
"!="                        { return NEQ; }
">="                        { return SUPEQ; }
"<="                        { return INFEQ; }
"not"                       { return NOT; }
[\[\]{}();+\-*/<>=%]        { return yytext[0]; }
{INTEGER}                   { yylval.num = atoi(yytext); return INTEGER; }
{IDENTIFIER}                { yylval.str = strdup(yytext); return IDENTIFIER; }
"//".*                      { ; }
"/*"([^*]|\*+[^*/])*\*+"/"  { ; }
[ \t\n]                     { ; }
.                           { return yytext[0]; }
```

2.3 Gramática

Uma gramática é um modelo matemático que permite descrever uma linguagem identificando um mecanismo gerador dos seus elementos. Para a nossa linguagem iremos utilizar uma Gramática Independente de Contexto ou (GIC).

O alfabeto da nossa linguagem é composto pelos símbolos não terminais: `expression`, `varDecl`, `whileStatement`, `doUntil`, `forStatement`, `letStatement`, `statement`, `statements`, `main`, `ifThenElseStmt`, `ifThenStatement`, `condition`, `print`, `funcao`, `functionCall`, e pelos símbolos terminais que estão descritos na secção Análise léxica (estes são os TOKENS retornados pelo flex).

O símbolo inicial da nossa gramática chama-se `program` e é definido da seguinte forma:

```
program         : decls main
```

Como já foi mencionado em Declarações, o programa está dividido. No inicio são efetuadas as declarações e só depois a função main.

```
decls           :
                | decls varDecl
                | decls funcao

varDecl         : INT IDENTIFIER ';'
                | INT IDENTIFIER '=' expression ';'
                | INT IDENTIFIER '[' INTEGER ']' ';'

funcao          : INT IDENTIFIER '(' ')' '{' statements RETURN expression ';' '}'
```

As `decls` são compostas por `0` ou mais declarações do tipo `varDecl` ou `funcao`.
A segunda parte do programa é composto unicamente pela função main com a seguinte produção:

```
main            : INT MAIN '(' ')' '{' statements '}'
```

O único símbolo não terminal da produção main é `statements` que pode ter 0 ou mais `statement`:

```
statements      :
                | statements statement

statement       : ifThenElseStmt
                | ifThenStatement
                | whileStatement
                | doUntil
                | forStatement
                | letStatement
                | print
                | functionCall

ifThenElseStmt  : IF '(' condition ')' '{' statements '}' ELSE '{' statements '}'
ifThenStatement : IF '(' condition ')' '{' statements '}'
whileStatement  : WHILE '(' condition ')' '{' statements '}'
doUntil         : UNTIL '(' condition ')' '{' statements '}'
forStatement    : FOR '(' IDENTIFIER '=' expression ';' expression ')' '{' statements '}'
letStatement    : IDENTIFIER '=' expression ';'
                | IDENTIFIER '[' expression ']' '=' expression ';'
                | IDENTIFIER '=' functionCall
print           : PRINT '(' expression ')' ';'
functionCall    : IDENTIFIER '(' ')' ';'
```

A produção `statement` e as suas sub produções correspondem com o que foi descrito em Instruções. Por fim temos as produções `expression` e `condition` que também correspondem com o que foi mencionado em Expressões:

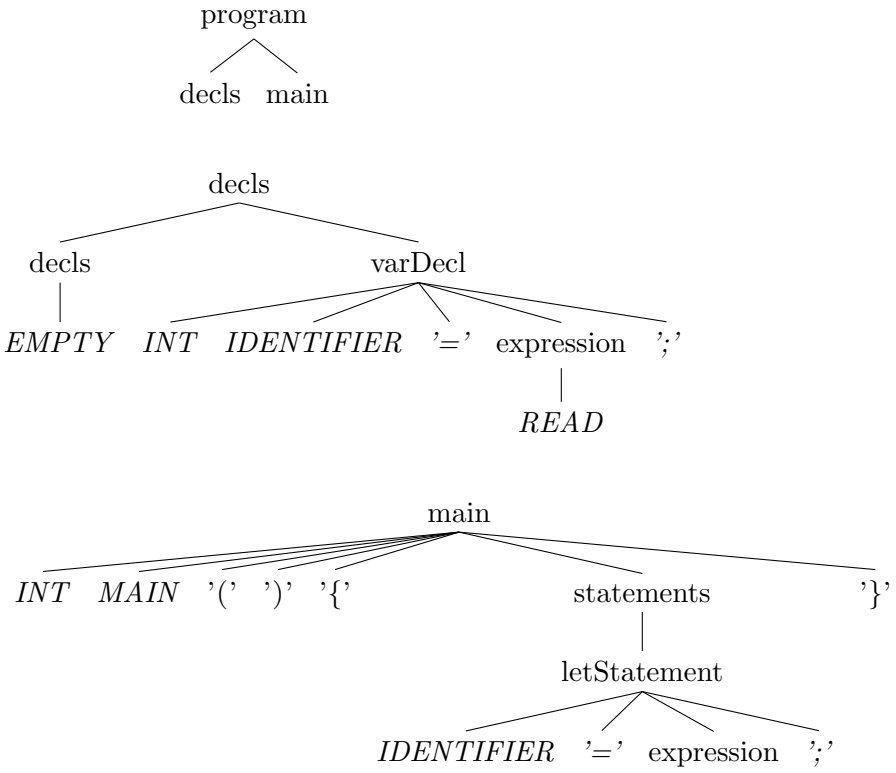
```
expression      : INTEGER
                | IDENTIFIER
                | IDENTIFIER '[' expression ']'
                | READ
                | '(' expression ')'
                | expression '+' expression
                | expression '-' expression
                | expression '*' expression
                | expression '/' expression
                | expression '%' expression

condition       : INTEGER
                | IDENTIFIER
                | '(' condition ')'
                | condition '>' condition
```

```
| condition '<' condition
| condition SUPEQ condition
| condition INFEQ condition
| condition EQ condition
| condition NEQ condition
| NOT condition
| condition AND condition
| condition OR condition
```

De seguida está um programa muito simples na nossa linguagem e a respetiva árvore de Parsing.

```
int a = read();
int main(){
  a = 5;
}
```



2.4 Semântica

Ao nível da semântica, o compilador cria uma tabela de símbolos com a estrutura de dados *Hash Table* para que a informação das variáveis declaradas seja guardada. Esta tabela de símbolos armazena o identificador, a posição da variável na stack e o tipo da variável. O valor da variável nunca é armazenado porque não é da responsabilidade do compilador, mas sim da máquina virtual. O compilador apenas precisa da localização para que possa ser feita referência à variável sempre que necessário.

Para implementar a *Hash Table* foi criado um programa à parte `hashTable.c`. Um programa já existente que foi modificado para implementar a tabela de símbolos.

Utilizando esta tabela, são feitas verificações de tipos. Estas verificações são feitas em várias produções da gramática. De seguida está um excerto do código que verifica se a variável já está declarada e se os tipos estão corretos:

```
letStatement : IDENTIFIER '=' expression ';' {
  if (hasDuplicates(symbolTable, $1) == 0)
  return fprintf(stderr, "%d: error: `%s' undeclared (first use in this program)\n",
      yylineno, $1);
  if (strcmp("int", ((ht_search(symbolTable, $1))->type)) != 0 )
  return fprintf(stderr, "%d: error: types don't match\n", yylineno);
  ...
}
```

No exemplo seguinte, como se trata de uma declaração, apenas é verificado se a variável já existe:

```
varDecl : INT IDENTIFIER ';' {
  if (hasDuplicates(symbolTable, $2))
  return fprintf(stderr, "%d: error: redeclaration of `%s'\n",
      yylineno, $2);
  ...
}
```

Capítulo 3

Compilação

Na linguagem da máquina virtual VM as operações seguem a notação *postfix*. Ou seja, se na nossa linguagem tivermos a expressão:

```
a + b
```

Esta será traduzida para o formato:

```
a b +
```

3.1 Empilhar Expressões e Condições

Quando um inteiro ou um identificador é encontrado a tradução é direta e ficará no seguinte formato:

- Para um inteiro:

  ```
  pushi | n inteiro | empilha n
  ```

- Para um identificador é utilizada a tabela de símbolos para recolher a localização da variável.

  ```
  pushg | n inteiro | empilha o valor localizado em gp[n]
  ```

No caso dos arrays já será efetuado um conjunto de instruções por esta ordem:

- `pushgp` empilha o valor do registo sp. Este valor vai ser sempre 0 na nossa linguagem e com o tipo endereço.
- `pushi` empilha a localização da variável recorrendo à tabela de símbolos.
- `padd` retira da pilha um inteiro e um endereço e empilha a soma como endereço
- conjunto de instruções que representam a expressão do índice do array
- `loadn` retira da stack o inteiro `n` resultante do conjunto de instruções da expressão anterior e o endereço `a` da variável que empilhamos anteriormente. Empilha o valor de `a[n]`.

De seguida encontra-se representada na estrutura de stack a expressão a[2] seguindo as instruções descritas acima. Assumimos que a está na posição 5 e que no índice 2 do array temos o valor 10:

...	...	...
0	0	5
...	5	...
...	...	...
...	...	...
5	5	10
2	2	...
...	...	...

Para a função pré definida `read()` temos a instrução `read` que faz com que a máquina virtual aguarde por input do utilizador e temos a instrução `atoi` que converte a instrução introduzida para o tipo inteiro.
Na expressão ou condição entre parêntesis não é feita qualquer modificação nem adicionada nenhuma instrução. É da responsabilidade da gramática e das regras de precedência definidas no Yacc/Bison decidir a ordem final das instruções.

Na compilação, algumas expressões estão no formato:

```
expressão 'op' expressão      condição 'op' condição
```

em que 'op' é um dos seguintes operadores lógicos ou aritméticos:

```
+, -, *, /, %, >, <, >=, <=, ==
```

Estes operadores já possuem as funções pré-definidas: `add`, `sub`, `mul`, `div`, `mod`, `sup`, `inf`, `supeq`, `infeq`, `equal`, respetivamente.

Assim, basta-nos colocar a primeira expressão e, posteriormente, a segunda expressão aplicando a referente função da operação.
Para os restantes tipos de expressão a tradução deixa de ser tão direta uma vez que não existem funções pré-definidas para as mesmas na máquina virtual. Seguem de seguida as expressões `!=`, `not`, `and` e `or`, respetivamente. Nestas instruções são utilizadas regras da lógica matemática para que seja calculado o valor booleano correto:

- condição != condição:

  ```
  [conjunto de instruções que representam a 1ª condição]
  [conjunto de instruções que representam a 2ª condição]
  ```

```
equal
pushi 1
add
pushi 2
mod
```

- not condição:

```
[conjunto de instruções que representam a condição]
pushi 1
add
pushi 2
mod
```

- condição and condição:

```
condition AND condition
[conjunto de instruções que representam a 1ª condição]
[conjunto de instruções que representam a 2ª condição]
mul
```

- condição or condição:

```
condition OR condition
[conjunto de instruções que representam a 1ª condição]
[conjunto de instruções que representam a 2ª condição]
add
pushi 2
mod
[conjunto de instruções que representam a 1ª condição]
[conjunto de instruções que representam a 2ª condição]
mul
add
```

3.2 Declarações

- Declaração sem atribuição de valor:
  Quando temos a declaração de uma variável sem atribuição de valor apenas é feito um `pushi 0`. Isto faz com que o próximo lugar na stack fique com o valor zero guardado, que é o valor com que queremos que a variável se inicialize. Este lugar na stack coincide com o valor do contador de variáveis `labelCount` e é o mesmo valor que é armazenado na tabela de símbolos.

- Declaração com atribuição de valor:
  Quando há a atribuição de valor fazemos na mesma o `pushi 0`[1] para guardar a posição na stack. De seguida, estará a expressão que representa o valor que queremos atribuir e, por fim, a instrução `storeg n`. Esta última instrução guarda o valor representado pela expressão na posição `n` da stack.

[1] o valor zero em específico não é importante e apenas é usado como um *placeholder*.

- Declaração de arrays de inteiros:
  Esta declaração é feita apenas com a instrução `pushn n` em que `n` é o tamanho do array. Esta instrução reserva as próximas `n` posições na stack, todas inicializadas com o valor zero. Na tabela de símbolos, a posição do array é representada com a posição do primeiro elemento e a `labelCount` é incrementada `n` valores.

- Declaração de uma função:
  As funções são colocadas depois do `stop` pois só queremos que sejam processadas quando são chamadas. O nome da função que está a ser declarada irá ser colocado no código da máquina virtual `nome:` para servir de identificador. De seguida são colocadas as instruções que a compõem seguidas da expressão a ser retornada. A instrução `storel -1` fará com que o valor da expressão a ser retornada seja colocada na posição anterior na stack. A instrução `return` faz com que o programa saia da função e continue para a próxima instrução.

3.3 Instruções

3.3.1 Atribuição

Para a atribuição de um inteiro, é colocada a expressão do valor a atribuir e o `storeg n` para guardar o valor na posição `n`.

Para a atribuição do valor de uma expressão a um dado índice de um array, será efetuado um conjunto de instruções por esta ordem:

- `pushgp`, `pushi` e `padd` seguem a mesma ideia da expressão do array falado anteriormente, definindo o endereço `a`.

- De seguida vem o conjunto de instruções que representam o índice `n` do array seguidas de um outro conjunto de instruções que representam o valor `v` a colocar nesse índice.

- Por fim, existe uma função pré definida `storen` que retira da stack o valor `v`, o inteiro `n` e um endereço `a` e arquiva `v` no endereço `a[n]` na pilha ou na heap

Também considerada uma atribuição, temos a functionCall. Será utilizado o `pushi 0` para reservar a localização do valor que será retornado. De seguida teremos `pusha f` em que `f` é o nome da função a ser chamada seguido da instrução `call`, a primeira empilha o endereço da função e a segunda retira o endereço da pilha e corre a função. Terminando de a correr guarda o valor retornado na localização que reservamos no inicio, utilizando a instrução `storeg`.

3.3.2 Condições

- Condição if then else:
  Serão utilizadas duas labels: uma das labels aponta para o final da instrução e a outra para o inicio do else. Se a condição for falsa, a instrução `jz ELSEn` sendo n o identificador da `label`, faz com que o programa salte para essa mesma `label`. Caso contrario o programa avança para a próxima instrução. Encontrando a instrução `jump ENDIFn` o programa salta para o fim, ignorando o else.

- Condição if then:
  Aqui só existe uma `label`. Sendo a condição falsa a instrução `jz ELSEn` faz com que o programa saia da instrução. Caso contrário, avança para a próxima instrução.

3.3.3 Ciclos

O procedimento nos ciclos passa por definir uma Label inicial e um Label final. O Program Counter, posteriormente a percorrer o código, dará um salto para a Label inicial, se a condição for verdadeira ou falsa, conforme o ciclo em questão. A compilação dos três ciclos são variações deste método.

- Ciclo Do While

```
WHILEn:
[conjunto de instruções que representam a condição]
jz ENDWHILEn
[conjunto de instruções a serem executadas]
jump WHILEn
ENDWHILEn
```

- Ciclo Do Until

```
UNTILn:
[conjunto de instruções a serem executadas]
[conjunto de instruções que representam a condição]
jz UNTILn
```

- Ciclo For

```
[conjunto de instruções que representam o valor inicial]
storeg %d         localização identifier
FORn:
pushg %d          localização identifier
[conjunto de instruções que representam o valor final]
equal
pushi 1
add
pushi 2
mod
jz ENDFORn
[conjunto de instruções a serem executadas]
pushg %d          localização identifier
pushi 1
add
storeg %d         localização identifier
jump FORn
ENDFORn:
```

3.3.4 Escrita

`writei`:

É utilizada a instrução `writei` para escrever no standard output, é retirado da pilha um inteiro e imprimido o seu valor na saída standard.

Capítulo 4

Conclusão

Com a realização deste projeto, aprendemos a utilizar duas ferramentas muito importantes, Flex e bison/Yacc, e a junção das mesmas.

Para além da aprendizagem adquirida sobre as ferramentas em si, ganhámos capacidades na produção de linguagens e compiladores e mostrou-nos a importância das GIC- gramáticas Independentes de Contexto.

Tendo finalizado este projeto mais cedo que o previsto, para além do pedido pelo enunciado que seria escolher entre implementar `arrays` de inteiros e respetiva operação de indexação ou implementar a invocação de subprogramas sem parâmetros que retornam inteiros, decidimos concretizar ambas.

Existe sempre espaço para melhorias futuras para este compilador que incluem a implementação de novos tipos como por exemplo `floats` e `strings`.

Assim, sentimo-nos satisfeitos ao realizar este trabalho porque acreditamos que foram desempenhadas todas as metas pedidas pelo professor.

Apêndice A

Código do Programa scanner.l

```
%{
#include <stdio.h>
#include <stdlib.h>
#include "grammar.tab.h" %}
%option noyywrap
%option yylineno
INTEGER             \-?[0-9]+
IDENTIFIER          [_a-zA-Z][_a-zA-Z0-9]*
%%
"main"                      { return MAIN; }
"return"                    { return RETURN; }
"read()"                    { return READ; }
"print"                     { return PRINT; }
"int"                       { return INT; }
"if"                        { return IF; }
"else"                      { return ELSE; }
"while"                     { return WHILE; }
"for"                       { return FOR; }
"until"                     { return UNTIL; }
"or"                        { return OR; }
"and"                       { return AND; }
"=="                        { return EQ; }
"!="                        { return NEQ; }
">="                        { return SUPEQ; }
"<="                        { return INFEQ; }
"not"                       { return NOT; }
[\[\]{}();+\-*/<>=%]        { return yytext[0]; }
{INTEGER}                   { yylval.num = atoi(yytext); return INTEGER; }
{IDENTIFIER}                { yylval.str = strdup(yytext); return IDENTIFIER; }
"//".*                      { ; }
"/*"([^*]|\*+[^*/])*\*+"/"  { ; }
[ \t\n]                     { ; }
.                           { return yytext[0]; }
%%
```

Apêndice B

Código do Programa grammar.y

```
%{
#define _GNU_SOURCE
#include <stdio.h>
#include <string.h>
#include "hashTable.h"
#define CAPACITY 50000
int yydebug=1;
int yylex();
int yyerror();
extern int yylineno;
int globalCount = 0;
int localCount = 0;
int functionCount = 0;
int labelCount;
HashTable *symbolTable;

typedef struct {
    char *str;
    int num;
    char *funcs;
} info;

%}
%define parse.error verbose

%union { char* str; int num; info info; }
%token INT MAIN IF ELSE WHILE UNTIL FOR READ PRINT RETURN
%token <num> INTEGER
%token <str> IDENTIFIER

%type <str> expression varDecl whileStatement doUntil forStatement
%type <str> letStatement statement statements main
%type <str> ifThenElseStmt ifThenStatement condition print funcao functionCall
%type <info> decls
```

```
%right  '='
%left   OR
%left   AND
%left   EQ NEQ
%left   '>' '<' SUPEQ INFEQ
%left   '+' '-'
%left   '*' '/'
%left   NOT

%%

program : decls main {
  printf("// Declarations\n"
         "%s\n"
         "// Program\n"
         "start\n"
         "%s"
         "stop\n\n", $1.str,$2);
  if (strcmp($1.funcs, "") != 0) printf("// Functions\n%s", $1.funcs);
  print_table(symbolTable);
}

decls : {
  asprintf(&$$.str, "%s", ""); asprintf(&$$.funcs, "%s", "");
}

      | decls varDecl {
  asprintf(&$$.str, "%s"
                    "%s", $1.str, $2);
}

      | decls funcao {
  asprintf(&$$.str, "%s\n", $1.str);
  asprintf(&$$.funcs, "%s\n", $2);
}

varDecl : INT IDENTIFIER ';' {
  if (hasDuplicates(symbolTable, $2))
  return fprintf(stderr, "%d: error: redeclaration of `%s'\n", yylineno, $2);
  asprintf(&$$, "pushi 0\n");
  ht_insert(symbolTable, $2, globalCount++, "int"); }

      | INT IDENTIFIER '=' expression ';' {
  if (hasDuplicates(symbolTable, $2))
  return fprintf(stderr, "%d: error: redeclaration of `%s'\n", yylineno, $2);
  asprintf(&$$, "pushi 0\n"
                "%s"
```

```
                "storeg %d\n", $4, globalCount);
    ht_insert(symbolTable, $2, globalCount++, "int");
  }

        | INT IDENTIFIER '[' INTEGER ']' ';' {
    if (hasDuplicates(symbolTable, $2))
    return fprintf(stderr, "%d: error: redeclaration of `%s'\n", yylineno, $2);
    asprintf(&$$, "pushn %d\n", $4);
    ht_insert(symbolTable, $2, globalCount, "intArray");
    globalCount += $4;
  }

funcao : INT IDENTIFIER '(' ')' '{'
         statements
         RETURN expression ';' '}' {
    if (hasDuplicates(symbolTable, $2))
    return fprintf(stderr, "%d: error: redeclaration of `%s'\n", yylineno, $2);
    ht_insert(symbolTable, $2, -1, "function");
    asprintf(&$$, "%s:\n"
                "%s"
                "%s"
                "storel -1\n"
                "return\n", $2, $6, $8);
  }

main : INT MAIN '(' ')' '{' statements '}' {
    asprintf(&$$, "%s", $6);
  }

statements : {
    asprintf(&$$, "%s", "");
  }

        | statements statement {
    asprintf(&$$, "%s"
                "%s", $1, $2);
  }

statement : ifThenElseStmt { asprintf(&$$, "%s", $1); }
        | ifThenStatement { asprintf(&$$, "%s", $1); }
        | whileStatement { asprintf(&$$, "%s", $1); }
        | doUntil { asprintf(&$$, "%s", $1); }
        | forStatement { asprintf(&$$, "%s", $1); }
        | letStatement { asprintf(&$$, "%s", $1); }
        | varDecl {
    return fprintf(stderr, "%d: error: variables must be declared before any function\n",
                   yylineno);
  }
```

```
        | print { asprintf(&$$, "%s", $1); }
        | functionCall { asprintf(&$$, "%s", $1); }

ifThenElseStmt : IF '(' condition ')'
                 '{' statements '}'
                 ELSE '{' statements '}' {
    asprintf(&$$, "%s"
                 "jz ELSE%d\n"
                 "%s"
                 "jump ENDIF%d\n"
                 "ELSE%d:\n"
                 "%s"
                 "ENDIF%d:\n", $3, labelCount, $6, labelCount, labelCount,
                           $10, labelCount);
    labelCount++;
  }

ifThenStatement : IF '(' condition ')'
                  '{' statements '}' {
    asprintf(&$$, "%s"
                  "jz L%d\n"
                  "%s"
                  "L%d:\n", $3, labelCount, $6, labelCount);
    labelCount++;
  }

whileStatement : WHILE '(' condition ')'
                 '{' statements '}' {
    asprintf(&$$, "WHILE%d:\n"
                  "%s"
                  "jz ENDWHILE%d\n"
                  "%s"
                  "jump WHILE%d\n"
                  "ENDWHILE%d:\n", labelCount, $3, labelCount,
                                   $6, labelCount, labelCount);
    labelCount++;
  }

doUntil : UNTIL '(' condition ')'
            '{' statements '}' {
    asprintf(&$$, "UNTIL%d:\n"
                  "%s"
                  "%s"
                  "jz UNTIL%d\n", labelCount, $6, $3, labelCount);
    labelCount++;
  }

forStatement : FOR '(' IDENTIFIER '=' expression ';' expression ')'
```

```
                ’{’ statements ’}’ {
    if (strcmp("int", ((ht_search(symbolTable, $3))->type)) != 0 )
    return fprintf(stderr, "%d: error: types don’t match\n", yylineno);
    int varPos = ((ht_search(symbolTable, $3))->varPos);
    asprintf(&$$, "%s\n"
                "storeg %d\n"
                "FOR%d:\n"
                "pushg %d\n"
                "%s"
                "equal\n"
                "pushi 1\n"
                "add\n"
                "pushi 2\n"
                "mod\n"
                "jz ENDFOR%d\n"
                "%s"
                "pushg %d\n"
                "pushi 1\n"
                "add\n"
                "storeg %d\n"
                "jump FOR%d\n"
                "ENDFOR%d:\n", $5, varPos, labelCount, varPos, $7,
                      labelCount, $10, varPos, varPos, labelCount, labelCount);
    labelCount++;
  }

letStatement : IDENTIFIER ’=’ expression ’;’ {
    if (hasDuplicates(symbolTable, $1) == 0)
    return fprintf(stderr, "%d: error: ‘%s’ undeclared (first use in this program)\n",
                        yylineno, $1);
    if (strcmp("int", ((ht_search(symbolTable, $1))->type)) != 0 )
    return fprintf(stderr, "%d: error: types don’t match\n", yylineno);
    asprintf(&$$, "%s"
                "storeg %d\n", $3, ((ht_search(symbolTable, $1))->varPos)); }

        | IDENTIFIER ’[’ expression ’]’ ’=’ expression ’;’ {
    if (hasDuplicates(symbolTable, $1) == 0)
    return fprintf(stderr, "%d: error: ‘%s’ undeclared (first use in this program)\n",
                        yylineno, $1);
    if (strcmp("intArray", ((ht_search(symbolTable, $1))->type)) != 0 )
    return fprintf(stderr, "%d: error: types don’t match\n", yylineno);
    asprintf(&$$, "pushgp\n"
                "pushi %d\n"
                "padd\n"
                "%s"
                "%s"
                "storen\n", (ht_search(symbolTable, $1)->varPos), $3, $6); }
```

```
        | IDENTIFIER '=' functionCall {
    if (hasDuplicates(symbolTable, $1) == 0)
    return fprintf(stderr, "%d: error: ‘%s’ undeclared (first use in this program)\n",
                        yylineno, $1);
    if (strcmp("function", ((ht_search(symbolTable, $1))->type)) != 0 )
    return fprintf(stderr, "%d: error: types don't match\n", yylineno);
    asprintf(&$$, "pushi 0\n"
                  "%s"
                  "storeg %d\n", $3, (ht_search(symbolTable, $1)->varPos)); }

print : PRINT '(' expression ')' ';' {
    asprintf(&$$, "%s"
                  "writei\n", $3); }

functionCall : IDENTIFIER '(' ')' ';' {
    if (hasDuplicates(symbolTable, $1) == 0)
    return fprintf(stderr, "%d: error: ‘%s’ undeclared (first use in this program)\n",
                        yylineno, $1);
    if (strcmp("function", ((ht_search(symbolTable, $1))->type)) != 0 )
    return fprintf(stderr, "%d: error: types don't match\n", yylineno);
    asprintf(&$$, "pusha %s\n"
                  "call\n", $1); }

expression : INTEGER {
    asprintf(&$$, "pushi %d\n", $1); }

        | IDENTIFIER {
    if (hasDuplicates(symbolTable, $1) == 0)
    return fprintf(stderr, "%d: error: ‘%s’ undeclared (first use in this program)\n",
                        yylineno, $1);
    if (strcmp("int", ((ht_search(symbolTable, $1))->type)) != 0 )
    return fprintf(stderr, "%d: error: types don't match\n", yylineno);
    asprintf(&$$, "pushg %d\n", ((ht_search(symbolTable, $1))->varPos)); }

        | IDENTIFIER '[' expression ']' {
    if (hasDuplicates(symbolTable, $1) == 0)
    return fprintf(stderr, "%d: error: ‘%s’ undeclared (first use in this program)\n",
                        yylineno, $1);
    if (strcmp("intArray", ((ht_search(symbolTable, $1))->type)) != 0 )
    return fprintf(stderr, "%d: error: types don't match\n", yylineno);
    asprintf(&$$, "pushgp\n"
                  "pushi %d\n"
                  "padd\n"
                  "%s"
                  "loadn\n", ((ht_search(symbolTable, $1))->varPos), $3); }

        | READ {
    asprintf(&$$, "read\n"
```

```
                "atoi\n");
  }

       | '(' expression ')' {
    asprintf(&$$, "%s", $2);
  }

       | expression '+' expression {
    asprintf(&$$, "%s"
                "%s"
                "add\n", $1, $3);
  }

       | expression '-' expression {
    asprintf(&$$, "%s"
                "%s"
                "sub\n", $1, $3);
  }

       | expression '*' expression {
    asprintf(&$$, "%s"
                "%s"
                "mul\n", $1, $3);
  }

       | expression '/' expression {
    asprintf(&$$, "%s"
                 "%s"
                "div\n", $1, $3);
  }

       | expression '%' expression {
    asprintf(&$$, "%s"
                "%s"
                "MOD\n", $1, $3);
  }

condition : INTEGER { asprintf(&$$, "pushi %d\n", $1); }

       | IDENTIFIER {
    if (hasDuplicates(symbolTable, $1) == 0)
    return fprintf(stderr, "%d: error: ‘%s’ undeclared (first use in this function)\n",
                      yylineno, $1);
    if (strcmp("int", ((ht_search(symbolTable, $1))->type)) != 0 )
    return fprintf(stderr, "%d: error: types don't match\n", yylineno);
    asprintf(&$$, "pushg %d\n", ((ht_search(symbolTable, $1))->varPos));
  }
```

```
      | IDENTIFIER '[' expression ']' {
  if (hasDuplicates(symbolTable, $1) == 0)
  return fprintf(stderr, "%d: error: ‘%s’ undeclared (first use in this program)\n",
                      yylineno, $1);
  if (strcmp("intArray", ((ht_search(symbolTable, $1))->type)) != 0 )
  return fprintf(stderr, "%d: error: types don't match\n", yylineno);
  asprintf(&$$, "pushgp\n"
                "pushi %d\n"
                "padd\n"
                "%s"
                "loadn\n", ((ht_search(symbolTable, $1))->varPos), $3);
}

      | '(' condition ')' { asprintf(&$$, "%s", $2); }

      | condition '>' condition {
  asprintf(&$$, "%s"
                "%s"
                "sup\n", $1, $3);
}

      | condition '<' condition {
      asprintf(&$$, "%s"
                    "%s"
                    "inf\n", $1, $3);
}

      | condition SUPEQ condition {
  asprintf(&$$, "%s"
                "%s"
                "supeq\n", $1, $3);
}

      | condition INFEQ condition {
  asprintf(&$$, "%s"
                "%s"
                "infeq\n", $1, $3);
}

      | condition EQ condition {
  asprintf(&$$, "%s%s"
                "equal\n", $1, $3);
}

      | condition NEQ condition {
  asprintf(&$$, "%s"
                "%s"
                "equal\n"
```

```
                "pushi 1\n"
                "add\n"
                "pushi 2\n"
                "mod\n", $1, $3);
}

      | NOT condition {
  asprintf(&$$, "%s"
                "pushi 1\n"
                "add\n"
                "pushi 2\n"
                "mod\n", $2);
}

      | condition AND condition {
  asprintf(&$$, "%s"
                "%s"
                "mul\n", $1, $3);
}

      | condition OR condition {
  asprintf(&$$, "%s"
                "%s"
                "add\n"
                "pushi 2\n"
                "mod\n"
                "%s%s"
                "mul\n"
                "add\n", $1, $3, $1, $3);
}

%%

#include "lex.yy.c"

int yyerror(const char *s) {
  fprintf(stderr, "%d: error: %s \n", yylineno, s);
  return 0;
}

int main() {
  symbolTable = create_table(CAPACITY); yyparse();
  return(0);
}
```

Apêndice C

Exemplos propostos

C.1 ler 4 números e dizer se podem ser os lados de um quadrado

C.1.1 Código do programa:

```
int a = read();
int b = read();
int c = read();
int d = read();
int main() {
  if (a==b and b==c and c==d) {
    print(1);
  }
  else {
    print(0);
  }
}
```

C.1.2 Output do compilador:

```
// Declarations
pushi 0
read
atoi
storeg 0
pushi 0
read
atoi
storeg 1
pushi 0
read
atoi
storeg 2
pushi 0
read
```

```
atoi
storeg 3

// Program
start
pushg 0
pushg 1
equal
pushg 1
pushg 2
equal
mul
pushg 2
pushg 3
equal
mul
jz ELSE0
pushi 1
writei
jump ENDIF0
ELSE0:
pushi 0
writei
ENDIF0:
stop

//Tabela de Símbolos
//-------------------------------------
//Index:97, identifier:a, varPos:0, type:int
//Index:98, identifier:b, varPos:1, type:int
//Index:99, identifier:c, varPos:2, type:int
//Index:100, identifier:d, varPos:3, type:int
//-------------------------------------
```

C.2 ler um inteiro N, depois ler N números e escrever o menor deles

C.2.1 Código do programa:

```
int n = read();
int menor;
int temp;

int main() {
  menor = read();
  while (n > 1) {
    temp = read();
    if (temp < menor) {
      menor = temp;
```

```
    }
    n = n - 1;
  }
  print(menor);
}
```

C.2.2 Output do compilador:

```
// Declarations
pushi 0
read
atoi
storeg 0
pushi 0
pushi 0

// Program
start
read
atoi
storeg 1
WHILE1:
pushg 0
pushi 1
sup
jz ENDWHILE1
read
atoi
storeg 2
pushg 2
pushg 1
inf
jz L0
pushg 2
storeg 1
L0:
pushg 0
pushi 1
sub
storeg 0
jump WHILE1
ENDWHILE1:
pushg 1
writei
stop

//Tabela de Símbolos
//--------------------------------------
```

```
//Index:110, identifier:n, varPos:0, type:int
//Index:438, identifier:temp, varPos:2, type:int
//Index:545, identifier:menor, varPos:1, type:int
//-------------------------------------
```

C.3 ler N (constante do programa) números e calcular e imprimir o seu produtório.

C.3.1 Código do programa:

```
int n = 5;
int prod;

int main() {
  prod = read();
  while (n > 1) {
    prod = prod * read();
    n = n - 1;
  }
  print(prod);
}
```

C.3.2 Output do compilador:

```
// Declarations
pushi 0
pushi 5
storeg 0
pushi 0

// Program
start
read
atoi
storeg 1
WHILE0:
pushg 0
pushi 1
sup
jz ENDWHILE0
pushg 1
read
atoi
mul
storeg 1
pushg 0
pushi 1
```

```
sub
storeg 0
jump WHILE0
ENDWHILE0:
pushg 1
writei
stop

//Tabela de Símbolos
//--------------------------------------
//Index:110, identifier:n, varPos:0, type:int
//Index:437, identifier:prod, varPos:1, type:int
//--------------------------------------
```

C.4 contar e imprimir os números ímpares de uma sequência de números naturais

C.4.1 Código do programa:

```
int v[10];
int i;
int mod;
int count;
int main() {
  while(i < 10) {
    v[i] = read();
    i = i + 1;
  }
  i = i - 1;
  while (i > 0) {
    mod = v[i] % 2;
    if (mod == 1) {
      print(v[i]);
      count = count+1;
    }
    i = i - 1;
  }
}
```

C.4.2 Output do compilador:

```
// Declarations
pushn 10
pushi 0
pushi 0
pushi 0
```

```
// Program
start
WHILE0:
pushg 10
pushi 10
inf
jz ENDWHILE0
pushgp
pushi 0
padd
pushg 10
read
atoi
storen
pushg 10
pushi 1
add
storeg 10
jump WHILE0
ENDWHILE0:
pushg 10
pushi 1
sub
storeg 10
WHILE2:
pushg 10
pushi 0
sup
jz ENDWHILE2
pushgp
pushi 0
padd
pushg 10
loadn
pushi 2
MOD
storeg 11
pushg 11
pushi 1
equal
jz L1
pushgp
pushi 0
padd
pushg 10
loadn
writei
pushg 12
```

```
pushi 1
add
storeg 12
L1:
pushg 10
pushi 1
sub
storeg 10
jump WHILE2
ENDWHILE2:
stop

//Tabela de Símbolos
//-------------------------------------
//Index:105, identifier:i, varPos:10, type:int
//Index:118, identifier:v, varPos:0, type:intArray
//Index:320, identifier:mod, varPos:11, type:int
//Index:553, identifier:count, varPos:12, type:int
//-------------------------------------
```

C.5 ler e armazenar N números num array; imprimir os valores por ordem inversa

C.5.1 Código do programa:

```
int v[10];
int i;
int main() {
  while (i < 10) {
    v[i] = read();
    i = i + 1;
  }
  i = i - 1;
  while (i > 0) {
    print(v[i]);
    i = i - 1;
  }
}
```

C.5.2 Output do compilador:

```
// Declarations
pushn 10
pushi 0

// Program
start
```

```
WHILE0:
pushg 10
pushi 10
inf
jz ENDWHILE0
pushgp
pushi 0
padd
pushg 10
read
atoi
storen
pushg 10
pushi 1
add
storeg 10
jump WHILE0
ENDWHILE0:
pushg 10
pushi 1
sub
storeg 10
WHILE1:
pushg 10
pushi 0
sup
jz ENDWHILE1
pushgp
pushi 0
padd
pushg 10
loadn
writei
pushg 10
pushi 1
sub
storeg 10
jump WHILE1
ENDWHILE1:
stop

//Tabela de Símbolos
//--------------------------------------
//Index:105, identifier:i, varPos:10, type:int
//Index:118, identifier:v, varPos:0, type:intArray
//--------------------------------------
```

C.6 invocar e usar num programa seu uma função 'potencia()', que começa por ler do input a base B e o expoente E e retorna o valor B^E

C.6.1 Código do programa:

```
int b;
int e;
int c;
int resultado;
int potencia(){
  b=read();
  e=read();
  c=b;
  while(e>1){
    b=c*b;
    e=e - 1;
  }
  return b;
}
int main(){
  resultado = potencia();
}
```

C.6.2 Output do compilador:

```
// Declarations
pushi 0
pushi 0
pushi 0
pushi 0


// Program
start
pushi 0
pusha potencia
call
storeg 3
stop

// Functions
potencia:
read
atoi
storeg 0
read
atoi
```

```
storeg 1
pushg 0
storeg 2
WHILE0:
pushg 1
pushi 1
sup
jz ENDWHILE0
pushg 2
pushg 0
mul
storeg 0
pushg 1
pushi 1
sub
storeg 1
jump WHILE0
ENDWHILE0:
pushg 0
storel -1
return

//Tabela de Símbolos
//--------------------------------------
//Index:98, identifier:b, varPos:0, type:int
//Index:99, identifier:c, varPos:2, type:int
//Index:101, identifier:e, varPos:1, type:int
//Index:851, identifier:potencia, varPos:-1, type:function
//Index:979, identifier:resultado, varPos:3, type:int
//--------------------------------------
```

Apêndice D

Outros exemplos

D.1 Cálculo MDC - Algoritmo de Euclides

D.1.1 Código do programa:

```
int a; int b;
int main() {
  a = read();
  b = read();
  while (a > 0 and b > 0) {
    if (b > a) {
      b = b - a;
    }
    else {
      a = a - b;
    }
  }
  if (a > b) {
    print(a);
  }
  else {
    print(b);
  }
}
```

D.1.2 Output do compilador:

```
// Declarations
pushi 0
pushi 0

// Program
start
read
atoi
```

```
storeg 0
read
atoi
storeg 1
WHILE1:
pushg 0
pushi 0
sup
pushg 1
pushi 0
sup
mul
jz ENDWHILE1
pushg 1
pushg 0
sup
jz ELSE0
pushg 1
pushg 0
sub
storeg 1
jump ENDIF0
ELSE0:
pushg 0
pushg 1
sub
storeg 0
ENDIF0:
jump WHILE1
ENDWHILE1:
pushg 0
pushg 1
sup
jz ELSE2
pushg 0
writei
jump ENDIF2
ELSE2:
pushg 1
writei
ENDIF2:
stop

//Tabela de Símbolos
//--------------------------------------
//Index:97, identifier:a, varPos:0, type:int
//Index:98, identifier:b, varPos:1, type:int
//--------------------------------------
```

D.2 Algoritmo de ordenação de um array

D.2.1 Código do programa:

```
int v[5];
int i;
int j;
int a;
int n = 5;
int main() {
  for (i = 0; n) {
    v[i] = read();
  }
  i = 0;
  while (i < n) {
    j = i + 1;
    while (j < n) {
      if (v[i] > v[j]) {
        a = v[i];
 v[i] = v[j];
 v[j] = a;
      }
      j = j + 1;
    }
    i = i + 1;
  }
  for (i = 0; n) {
    print(v[i]);
  }
}
```

D.2.2 Output do compilador:

```
// Declarations
pushn 5
pushi 0
pushi 0
pushi 0
pushi 0
pushi 5
storeg 8

// Program
start
pushi 0

storeg 5
FOR0:
```

```
pushg 5
pushg 8
equal
pushi 1
add
pushi 2
mod
jz ENDFOR0
pushgp
pushi 0
padd
pushg 5
read
atoi
storen
pushg 5
pushi 1
add
storeg 5
jump FOR0
ENDFOR0:
pushi 0
storeg 5
WHILE3:
pushg 5
pushg 8
inf
jz ENDWHILE3
pushg 5
pushi 1
add
storeg 6
WHILE2:
pushg 6
pushg 8
inf
jz ENDWHILE2
pushgp
pushi 0
padd
pushg 5
loadn
pushgp
pushi 0
padd
pushg 6
loadn
sup
```

```
jz L1
pushgp
pushi 0
padd
pushg 5
loadn
storeg 7
pushgp
pushi 0
padd
pushg 5
pushgp
pushi 0
padd
pushg 6
loadn
storen
pushgp
pushi 0
padd
pushg 6
pushg 7
storen
L1:
pushg 6
pushi 1
add
storeg 6
jump WHILE2
ENDWHILE2:
pushg 5
pushi 1
add
storeg 5
jump WHILE3
ENDWHILE3:
pushi 0

storeg 5
FOR4:
pushg 5
pushg 8
equal
pushi 1
add
pushi 2
mod
jz ENDFOR4
```

```
pushgp
pushi 0
padd
pushg 5
loadn
writei
pushg 5
pushi 1
add
storeg 5
jump FOR4
ENDFOR4:
stop

//Tabela de Símbolos
//--------------------------------------
//Index:97, identifier:a, varPos:7, type:int
//Index:105, identifier:i, varPos:5, type:int
//Index:106, identifier:j, varPos:6, type:int
//Index:110, identifier:n, varPos:8, type:int
//Index:118, identifier:v, varPos:0, type:intArray
//--------------------------------------
```