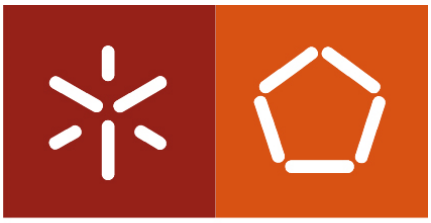

Escola de Engenharia
Universidade do Minho

Departamento de Informática
Mestrado Integrado em Engenharia Informática

Processamento de Linguagens

Trabalho Prático nº 2

Implementação de compilador para uma linguagem de programação imperativa simples em YACC

Bruno Pereira
Aluno nº 72628

Ricardo Oliveira
Aluno nº 58657

Braga, 2 de Junho de 2016

Resumo

Conteúdo

Introdução

Metas e objetivos

Estrutura do Relatório

Capítulo 1

Análise do Problema

1.1 Especificação dos requisitos

O desafio deste projeto consiste na criação de uma linguagem de programação imperativa simples (LPIS) e respetivo compilador. Para tal é necessário criar uma gramática em *Bauk-Naur Form*–BNF–, definir símbolos terminais e não terminais, e desenvolver o analisador léxico, seguido do desenvolvimento do analisador sintático, com base nas regras da gramática, tendo, de igual modo, em conta a análise léxica. O compilador da linguagem irá incorporar ambas as análises supramencionadas, e procederá a uma análise de ações semânticas e á geração do código. O código gerado será pseudo-código da maquina virtual VM, o analisador léxico será elaborado no *Flex*, e o YACC será usado para a geração do código e análises sintática e semântica.

1.2 Pedidos

Para a linguagem alvo deste projeto, pede-se que, no mínimo, permita:

- Declaração e manuseamento de variáveis atómicas do tipo inteiro que permitam a realização de operações aritméticas, relacionais e lógicas;
- Declaração e manuseamento variáveis estruturadas do tipo *array* de inteiros, com 1 ou 2 dimensões, que permitam apenas operações de indexação;
- Realização de instruções algorítmicas básicas como a atribuição de expressões a variáveis;
- Leitura do *stdin* e escrita no *stdout*;
- Realização de instruções de controlo do fluxo de execução que permitam aninhamento;

Opcionalmente, a linguagem definida deve ser capaz de definir e invocar subprogramas sem parâmetros, mas que possam retornar um resultado atómico.

Pede-se também que, qualquer variável não pode ser redeclarada.

1.3 Dados

Uma linguagem imperativa completa deve de permitir pelo menos duas estruturas de controlo:

- Estruturas de fluxo condicional (*if then else*);

- Estruturas cíclicas (*while*);

Por uma questão de simplificação do código, poderá adicionalmente permitir a estrutura *if then* e a estrutura *do while*.

De igual modo, permitindo a linguagem acesso a estruturas do tipo *array*, é necessário ter em conta que qualquer *array*, independentemente da dimensão, é representado em memória como um único *array* de uma dimensão. Contudo, é necessário estabelecer regras para o acesso a array*s* uni e bidimensionais. Para garantir a eficácia da linguagem, considerar-se-á que o acesso será feito em *row major* para array*s* bidimensionais.

Genericamente, o acesso a um *array* pode ser representado por a fórmula $A[i] = b + w * (i - lb)$, sendo:

- `b` o endereço base;
- `w` o tamanho do elemento;
- `i` o índice do elemento;
- `lb` o limite inferior na memória;

No caso do acesso em questão, $w = 1$ e $lb = 0$, logo $A[i] = b + i$.
Para um *array* bidimensional temos $A[i][j] = b + w[N(i - lr) + (j - lc)]$, onde:

- `b` é o endereço base;
- `i` é o índice da linha do elemento;
- `j` é o índice da coluna do elemento;
- `w` é o tamanho do elemento em bytes;
- `lr` é o limite inferior da linha;
- `lc` é o limite inferior da coluna;
- `N` é o número máximo de linhas;

Assumindo $w = 1$ e $lr = lc = 0$, temos $A[i][j] = b + N * i + j$.

1.4 Relações

Para o cálculo das expressões é necessário ter em conta algumas propriedades de cada tipo de expressão. Ou seja, é necessário verificar os tipos atómicos (variáveis, constantes, elementos de *arrays*), que assumimos como inteiros, e os resultados das expressões por inferência. Assim:

- Para as expressões aritméticas (soma, subtração, multiplicação, divisão inteira e módulo), bem como os elementos constituintes da expressão, o tipo deverá ser um inteiro;
- Para as expressões relacionais (maior, menor, maior ou igual, menor ou igual, igual e diferente), os elementos da expressão deverão ser do tipo inteiro e o resultado um valor booleano;
- Para as expressões lógicas, os elementos deverão ser booleanos e o resultado deverá também ser booleano;

De notar que várias expressões podem ser compostas, cuja verificação estará descrita na análise semântica. Adicionalmente, existe uma relação de precedência das operações, bem como de fatores. Um fator é uma expressão aninhada, um // ou uma variável. De igual modo, subprogramas e operações unárias? (uma vez que são funções) fazem parte deste conjunto. Consequentemente, um fator é prioritário em relação a todas as operações, uma vez que é o elemento atómico de uma expressão.

Em seguida, temos os termos, que são compostos por operações multiplicativas entre fatores, sendo estas a multiplicação, a divisão inteira, e módulo. Poderá também ser incluída a operação *E LÓGICO*, por razões que posteriormente serão explicadas. As expressões seguintes na escala de prioridades são as expressões aditivas (soma e subtração). Poderá ser incluída a operação *OU LÓGICO*, por razões que tal como a inclusão de *E LÓGICO*, que serão posteriormente explicitadas. As expressões de menor prioridade são as expressões relacionais (*menor*, *maior*, *menor ou igual*, *maior ou igual*, *igualdade* e *desigualdade*).

Para a execução do programa é necessário definir as instruções e operações que o definem. O programa pode efetuar cálculos utilizando as expressões previamente mencionadas. No entanto, é necessário guardar o resultado. Nas linguagem imperativas, a interação base é a atribuição, que pode ser atribuição do valor de uma expressão a uma variável. Esta poderá ser uma variável atómica ou a posição de um *array*. As restantes instruções terão por base estes cálculos de expressões e atribuições, usando restrições de controlo de fluxo, tais como *if then else*, *while* e *do while*.

Devido à necessidade de armazenar o valor de expressões, torna-se necessário declarar as variáveis, alocando espaço em memória para as mesmas. As declarações podem designar o nome da variável, o seu tipo, e o seu valor, não // redeclarações da mesma variável. Neste projeto, considerar-se-á que todas as variáveis são globais, ou seja, declaradas antes da execução do programa, e assumem número inteiro.

Para subprogramas, há que ter em conta que podem ter ou não parâmetros, e devolver ou não um valor. Visto que a alocação de memória é efetuada numa *stack* virtual, é necessário, no momento da sua //, criar uma *frame* no topo da *stack* com todas as variáveis locais declaradas, e parâmetros alocados em memória. Para identificar o programa principal, a linguagem usa um sistema de níveis, em que um nível assume o valor 0 para o programa principal ou o valor 1 para subprogramas. Finalmente, o início e fim de cada programa ou subprograma deve estar devidamente assinalado.

Capítulo 2

Desenho e implementação da solução

2.1 Desenho

2.1.1 Gramática Independente de Contexto

2.1.1.1 Axioma

⟨*Program*⟩ ::= ⟨*Declarations*⟩ ⟨*Body*⟩

⟨*Body*⟩ ::= '`BEGIN`' ⟨*InstructionList*⟩ '`END`'

⟨*Declaration*⟩ ::= ⟨*id*⟩
| ⟨*id*⟩ '`[`' ⟨*num*⟩ '`]`'
| ⟨*id*⟩ '`[`' ⟨*num*⟩ '`]`' '`[`' ⟨*num*⟩ '`]`'

⟨*Declarations*⟩ ::= 'VAR' ⟨*DeclarationsList*⟩ '`;`'

⟨*DeclarationsList*⟩ ::= ⟨*Declaration*⟩
| ⟨*DeclarationsList*⟩ '`,`' ⟨*Declaration*⟩

⟨*Term*⟩ ::= ⟨*id*⟩
| ⟨*num*⟩
| ⟨*id*⟩ '`[`' ⟨*ExpAdditiv*⟩ '`]`'
| ⟨*id*⟩ '`[`' ⟨*ExpAdditiv*⟩ '`]`' '`[`' ⟨*ExpAdditiv*⟩ '`]`'
| '`(`' ⟨*Exp*⟩ '`)`'
| '`NOT`' ⟨*Exp*⟩

2.1.1.2 Declarações de variáveis

2.1.1.3 Expressões

2.1.1.4 Instruções

2.1.2 Análise Semântica Estática

A análise semântica estática complementa a análise lexical e sintática, pois existem situações em que a apesar de a análise léxica e sintática estarem corretas, as sequências de símbolos não têm sentido. A título de exemplo, a data `2016-45-00` está sintaticamente correta pois segue o formato `aaaa-mm-dd`, mas não tem sentido do ponto de vista semântico.

Na *LPIS* existem situações em que a gramática e análise léxica não são suficientes, nomeadamente na verificação de tipos, ou seja, se estes são consistentes na sua definição. Anteriormente, é mencionada a inferência de tipos em fatores, variáveis, arrays, aditivos, termos, expressões e expressões relacionais. Assim é necessário não só verificar os elementos de cada operação binária, como inferir o tipo do seu resultado. Como já foi mencionado, esta *LPIS* apenas permite valores inteiros, mas no entanto o resultado das expressões lógicas e relacionais são do tipo booleano. Por uma questão de consistência, considera-se que as instruções terão o tipo `Any`.

Assim, as instruções assumem o tipo `Any`, variáveis atómicas e arrays o tipo `Integer` (exceto subprogramas sem valor de retorno), e o resultado de uma operação aditiva entre fatores como variáveis e arrays assumirá o tipo `Integer`, bem como o tipo de cada de cada membro da operação binária aditiva. O resultado de uma operação multiplicativa entre termos (resultantes da operação aditiva) também tomará o tipo `Integer`, bem como ambos os membros da operação binária em questão. Seguidamente, o resultado de expressões relacionais deverá ser do tipo `Boolean`. No entanto, os membros desta operação binária assumirão o tipo `Integer`. Para concluir, o resultado de uma expressão lógica assumirá o tipo `Boolean` e ambos os membros da operação binária assumem também o tipo `Boolean`.

Para além da verificação de tipos, a análise semântica deve assegurar a existência de etiquetas, ou `labels`, de referência no resultado da geração de código. Dado que a linguagem em causa tem quatro tipos de estruturas de controlo de fluxo, existirá um mecanismo que cria as `labels` para cada tipo de estrutura de controlo e que permita o aninhamento das mesmas. Ou seja, as etiquetas devem ter a referência do nível em que estão, e devem ser colocadas no seu devido lugar. Posteriormente, será descrito o algoritmo e a implementação do mecanismo de criação de `labels`.

Adicionalmente, as estruturas de controlo devem ser verificadas para confirmar a sua devida utilização. Por exemplo, se existem `breaks` fora de um `loop` ou `switch`. Na gramática desta linguagem não existem formas de //..

Finalmente, é necessário considerar outro tipo de análise semântica: a análise semântica dinâmica. Ao contrário da anterior, este tipo de análise não é efetuada em tempo de compilação, mas sim em tempo de execução. Existem exemplos retirados da gramática da linguagem alvo deste projeto, que não serão considerados, visto que a máquina virtual VM já os inclui, como é o caso da divisão por zero, em que os valores retirados da pilha não têm o tipo esperado, ou acessos indefinidos a uma //? de código. Na especificação léxica da linguagem, definiu-se que todos os números tomarão valores não negativos, e previu-se a existência de números negativos na gramática // a uma operação unária?. Deste modo não é necessária a reespecificação? semântica estática de declarações de arrays com tamanho zero, resultados negativos em cálculos de índices e divisões por zero, uma vez que estes casos fazem parte da análise semântica dinâmica.

2.2 Implementação

2.2.1 Analisador léxico

Na fase de construir o analisador léxico, tomou-se em conta a análise anterior da gramática e definiu-se para cada simbolo terminal uma expressão regular, e respetiva ação associada.

2.2.1.1 Expressões Regulares

Em primeira instância definiu-se o que era uma letra, um digito e, tudo o que será para ser rejeitado da seguinte forma:

```
letra [A-Za-z]
digito [0\-9]
lixo \.|\\n
```

Em seguida definiram-se as expressões para captura dos símbolos terminais, para serem usadas como *tokens* no ficheiro *YACC*, definiram-se da seguinte forma:

- Captura de operadores de apenas um caractere;

  ```
  [-/;,\[\]+\(\)\{\}\%=]
  ```

  Com esta expressão regular são capturados os símbolos referentes às operações aditivas e multiplicativas, bem como caracteres usados para delimitar expressões e na construção de estruturas de controlo (chavetas e parêntesis). De igual modo, são capturados o símbolo unário de um número negativo, delimitadores de final de instruções (ponto e vírgula), separadores de declarações (vírgula), elementos de declaração de *arrays* (parêntesis retos) e o sinal de atribuição, que nesta linguagem é o sinal de igual.

  A ação para esta expressão regular é devolver o valor do código ASCII-estendido de cada caractere capturado.

- Captura de operadores de operações lógicas e relacionais.

- ```
  (OR)                    {return OR;}
  (<)                     {return L;}
  (>)                     {return G;}
  (<=)                    {return LEQ;}
  (>=)                    {return GEQ;}
  (==)                    {return EQ;}
  (!=)                    {return NEQ;}
  ```

- Captura de palavra reserva para início e fim de programa.

  ```
  (BEGINNING)             {return BEGINNING;}
  (END)                   {return END;}
  ```

  A ação será retornar um valor numérico, que será atribuído aquando o *linking* do ficheiro *Flex* e do ficheiro *YACC*, no `yy.tab.c`.

- Captura das palavras reservadas para funções de leitura e escrita.

  ```
  (READ)                  {return READ;}
  (WRITE)                 {return WRITE;}
  ```

  Ação a mesma que a anterior.

- Captura das palavras reservadas usadas para declarações e estruturas de controlo.

  ```
  (VAR)                   {return VAR;}
  (WHILE)                 {return WHILE;}
  (IF)                    {return IF;}
  (ELSE)                  {return ELSE;}
  (DO)                    {return DO;}
  ```
  ```

Ação: *ibidem*.

- Captura de uma *string*

  `\"[^"]+\"`

  Neste caso é atribuído ao valor da variável global `yylval` no elemento da união `char * val_string` uma cópia da *string*, sendo retornado uma valor que será atribuído pelo *YACC*.

- Captura de um dígito

  ```
  1  {digito}+
  ```

  A ação é atribuir ao valor da variável global `yylval` no elemento da união `int val_nro` o valor do digito capturado., sendo retornado uma valor que será atribuído pelo *YACC*.

- Captura de um identificador.

  ```
  1  {letra}+
  ```

  Neste caso é atribuído ao valor da variável global `yylval` no elemento da união `char * val_id` uma cópia da palavra capturada, sendo retornado uma valor que será atribuído pelo *YACC*.

- Outros

  ```
  1  {lixo}                    {;}
  ```

  Ação: ignorar.

2.2.2 Analisador sintático e tradutor

Para o analisador sintático, criaram-se algumas estruturas de suporte ao *parser*, nomeadamente uma *hashtable* para a tabela de identificadores, uma biblioteca para os dados referentes a cada identificador, uma biblioteca para suporte ao *parser*, com toda a informação relativa ao estado do *parser*– apontador de endereços na *stack* virtual, *stacks* para calculo do nível das *etiquetas*– e, adicionalmente definiram-se tipos enumerados para serem usados em transversalmente na aplicação.

2.2.2.1 Estruturas de dados

A biblioteca `entry` possui uma estrutura composta pelos campos mencionados em secções anteriores: tipo, classe, nível, que são tipos enumerados, e, endereço base, número de linhas máximo (caso seja uma matriz) e tamanho máximo (para *arrays* unidimensionais e bidimensionais). Adicionalmente, considerou-se a criação de uma lista de argumentos, no entanto, por razões que serão posteriormente explicitadas, decidiu-se não se incluir.

As funções referentes à esta biblioteca, inicializam e desalocam memória e vão buscar os dados da estrutura, ou atualizam os dados desta estrutura. Para facilitar a criação de entradas na tabela, com diferentes tipos de classes (*array*, *matriz* e variável), criaram-se métodos que providenciam a criação da entrada por classe.

A biblioteca `program_status` que guarda informação sobre o *parsing* tem o formato que se segue;

```
typedef struct stat
{
    char label              [MAX_CONDITION_ROW] [ MAX_LABEL ];
    int  label_stack        [MAX_CONDITION_ROW] [ MAX_LABEL_STACK ];
    int  label_number_size  [MAX_CONDITION_ROW] [ MAX_LABEL_STACK ];
    int spointer            [MAX_CONDITION_ROW] [1];
    int strpointer          [MAX_CONDITION_ROW] [1];
    int size_label_string   [MAX_CONDITION_ROW] [1];
    int addresspointer;
} Program_status;
```

Basicamente, a estrutura possui *arrays* bidimensionais para a representação das *stacks* das *labels*. Estas têm 4 linhas, ou seja, uma linha para cada tipo de estrutura de controlo. Deste modo, a variável `label` irá guardar a concatenação dos valores dos níveis de aninhamento, a variável `label_stack` possuirá os contadores para cada nível de aninhamento e a variável `label_number_size` irá guardar o tamanho da da *string* resultante do valor de cada nível concatenado. A biblioteca possui funções comuns às *stacks*, como `pop`, `push`, `top` e, adicionalmente possui funções específicas para a manipulação da informação das *stacks*, que usa o tipo enumerado `CompoundInstruction` para aceder por índice às *stacks*.

As funções específicas para o cálculo das etiquetas são:

- `reset_label_stack` Esta função tem por âmbito reinicializar o contador após sair de uma ação semântica, como se poderá ver na secção sobre algoritmos, na posição em `stack[ stack_pointer ]`. Note-se que o `top` é em `stack [stack_pointer-1 ]`

- `increment_top_label_stack` Esta função incrementa o valor do contador na posição `stack_pointer -1` da *stack* de contadores, ou seja, incrementa uma nova ocorrência no mesmo nível.

- `char *get_label`

  Esta função obtém a *string* criada até ao momento, com os valores das ocorrências dos níveis concatenados na *string* da *label*.

- `char *push_label`

  A função `pushlabel` incrementa o valor do `top` da *stack* de contadores, converte o valor numérico do `top` para uma *string*, calcula o tamanho desta *string* e guarda na *stack* de tamanhos das *strings*. Em seguida, concatena a *string* convertida ‘a *string* em construção, sendo esta copiada para ser retornada pela função. Adicionalmente, guarda o tamanho da *string* convertida na devida *stack* e avança o apontador da *string* em construção por esse tamanho.

- `pop_label`

  Obtém o tamanho da *string* concatenada, guardada anteriormente na *stack* `label_number_size` e coloca caracteres nulos na pilha com a *string* para as *labels*, decrementando o apontador da pinha com a *string* no valor desse tamanho.

Note-se que, a descrição desta funções é fundamental para a compreensão do código, uma vez que serão utilizadas mais adiante no documento.

De igual modo, possui funções para o cálculo de endereços para cada classe de variável, onde o apontador para a *stack* virtual é incrementado pelo tamanho da variável. Outras funções são a inserção

de um identificador com os seu devidos valores na tabela de identificadores, bem como a procura de um identificador e remoção de todos os identificadores. Por último, a função `check_type` compara dois tipos.

Além destas bibliotecas, foram definidos novos tipos, enumerados, no cabeçalho `types.h`, que são os seguintes:

- `CompoundInstruction`; Este tipo carateriza tipos de estruturas de controlo diferentes para acesso por índice aos *arrays* multidimensionais, representam as diferentes *stacks* para as *labels*.
  - `if_inst`
  - `else_inst`
  - `while_inst`
  - `do_while_inst`
- `Class`;a O tipo `Class` serve para diferenciar as categorias, ou classes, de objetos que podem ser declarados e invocados.
  - `Variable`
  - `Array`
  - `Matrix`
  - `Function`
  - `Procedure`
  - `Nothing`
- `Level`; O nível tem o propósito de diferenciar programas e subprogramas.
  - `Program`
  - `Subprogram`
- `Type`; Os tipos das variáveis e expressões podem ser os seguintes:
  - `Any`
  - `Integer`
  - `Boolean` Embora se possa atribuir o tipo `Any` a instruções, não existe relevância para o fazer.

2.2.2.2 Algoritmos

http://lh3lh3.users.sourceforge.net/udb.shtml.

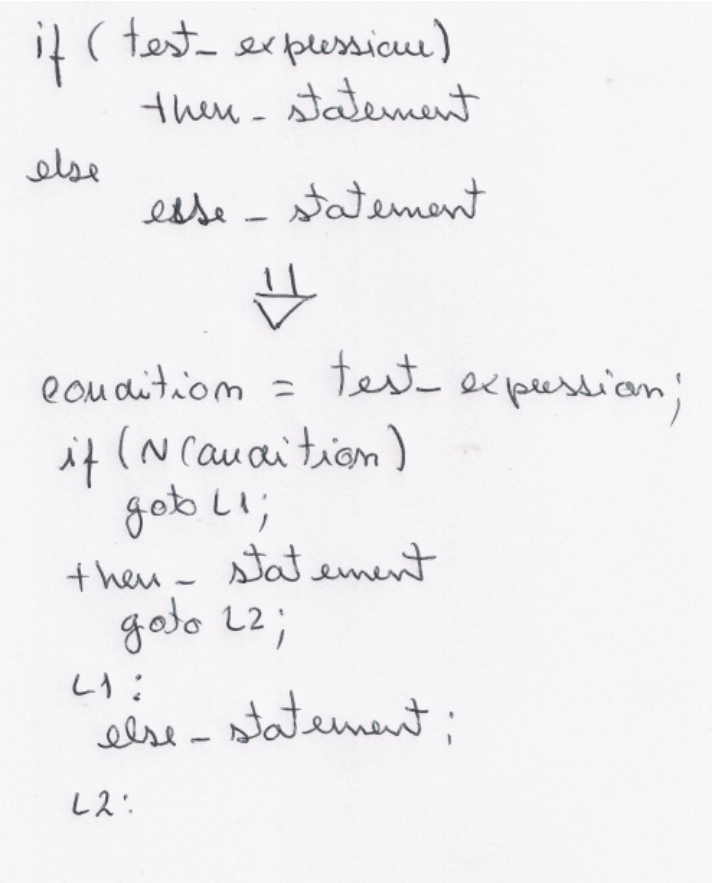

Figura 2.1: Algoritmo para `if else`

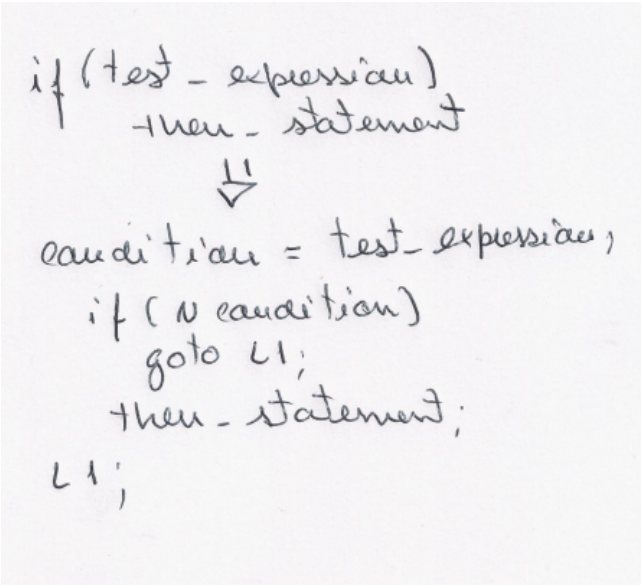

Figura 2.2: Algoritmo para `if`

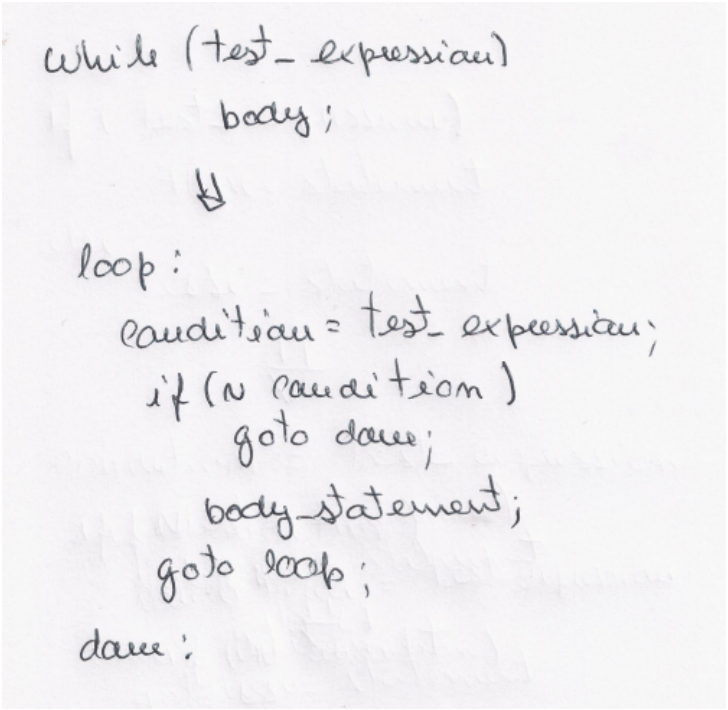

Figura 2.3: Algoritmo para ciclo `while`

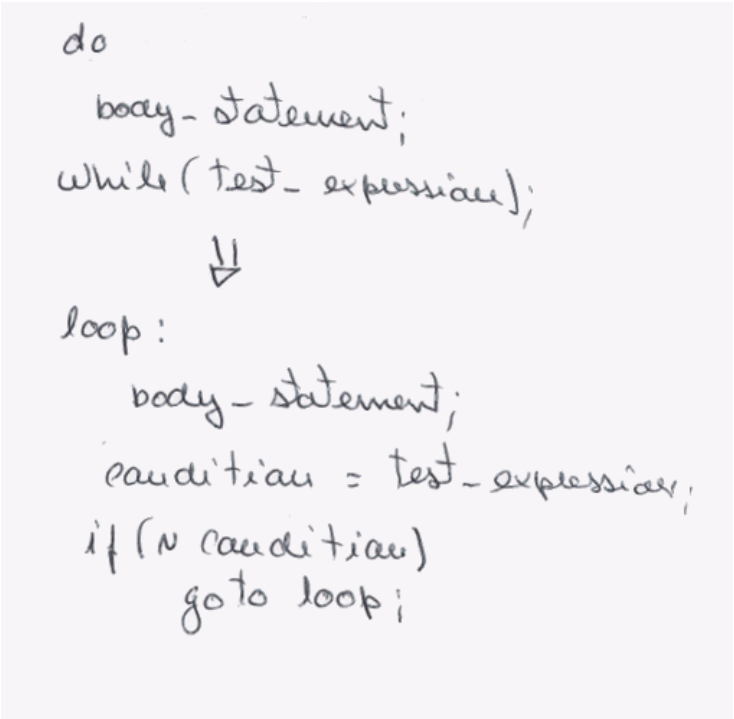

Figura 2.4: Algoritmo para ciclo `do while`

Capítulo 3

Testes e Resultados

3.1 Resultados

3.2 Alternativas, Decisões e Problemas de Implementação

Conclusão

Bibliografia

[1] T. H. Cormen. *Introduction to Algorithms*. The MIT Press, 2n edition edition, 2009. ISBN 0262533057.

[2] N. Drakos and R. Moore. Bibtex Entry Types, Field Types and Usage Hints, 1993. URL http://www.openoffice.org/bibliographic/bibtex-defs.pdf.

[3] K. Eleftherios and S. C. Nort. Editing graphs with dotty, 1996. URL http://www.graphviz.org/pdf/dottyguide.pdf.

[4] A. Feder. BibTeX, 2006. URL http://www.bibtex.org/.

[5] P. Lehman, P. Kime, A. Boruvka, and J. Wright. The BibLaTeX Package, 2016. URL http://ftp.eq.uc.pt/software/TeX/macros/latex/contrib/biblatex/doc/biblatex.pdf.

[6] M. E. Lesk and E. Schmidt. Lex: a lexical analyzer generator. 1975. URL http://dinosaur.compilertools.net/lex/index.html.

[7] J. Levine. *Flex & Bison*. 2009. ISBN 978-0-596-15597-1.

[8] J. R. Levine, T. Mason, D. Brown, and T. Niemann. *Lex And Yacc*. O'Reilly, 1992. ISBN 1-56592-000-7.

[9] T. Niemann. Lex & Yacc Tutorial. URL http://epaperpress.com/lexandyacc/.

[10] H. Refsnes, S. Refsnes, K. Jim Refsnes, and J. Egil Refsnes. *Learn HTML and CSS with W3Schools*. Wiley Publishing, Inc., Indianapolis, Indiana, 2010. ISBN 0470611952.

ANEXOS

Apêndice A

Gramática para linguagem criada

[1]

⟨*Program*⟩ ::= ⟨*Declarations*⟩ ⟨*Body*⟩

⟨*Body*⟩ ::= ‘`BEGIN`’ ⟨*InstructionList*⟩ ‘`END`’

⟨*Declaration*⟩ ::= ⟨*id*⟩
| ⟨*id*⟩ ‘`[`’ ⟨*num*⟩ ‘`]`’
| ⟨*id*⟩ ‘`[`’ ⟨*num*⟩ ‘`]`’ ‘`[`’ ⟨*num*⟩ ‘`]`’

⟨*Declarations*⟩ ::= ‘`VAR`’ ⟨*DeclarationsList*⟩ ‘`;`’

⟨*DeclarationsList*⟩ ::= ⟨*Declaration*⟩
| ⟨*DeclarationsList*⟩ ‘`,`’ ⟨*Declaration*⟩

⟨*Constant*⟩ ::= ⟨*num*⟩

⟨*Term*⟩ ::= ⟨*Constant*⟩>
| ⟨*Variable*⟩
| ⟨*id*⟩ ‘`[`’ ⟨*ExpAdditiv*⟩ ‘`]`’ ‘`[`’ ⟨*ExpAdditiv*⟩ ‘`]`’
| ‘`(`’ ⟨*Exp*⟩ ‘`)`’
| ‘`NOT`’ ⟨*Exp*⟩

⟨*Variable*⟩ ::= ⟨*id*⟩
| ⟨*id*⟩ ‘`[`’ ⟨*ExpAdditiv*⟩ ‘`]`’
| ⟨*id*⟩ ‘`[`’ ⟨*ExpAdditiv*⟩ ‘`]`’ ‘`[`’ ⟨*ExpAdditiv*⟩ ‘`]`’

⟨*ExMultipl*⟩ ::= ⟨*Term*⟩
| ⟨*ExMultipl*⟩ ‘`*`’ ⟨*Term*⟩
| ⟨*ExMultipl*⟩ ‘`/`’ ⟨*Term*⟩
| ⟨*ExMultipl*⟩ ‘`%`’ ⟨*Term*⟩
| ⟨*ExMultipl*⟩ ‘`AND`’ ⟨*Term*⟩

⟨*ExpAdditiv*⟩ ::= ⟨*ExMultipl*⟩
| ⟨*ExpAdditiv*⟩ ‘`+`’ ⟨*ExMultipl*⟩
| ⟨*ExpAdditiv*⟩ ‘`-`’ ⟨*ExMultipl*⟩
| ⟨*ExpAdditiv*⟩ ‘`OR`’ ⟨*ExMultipl*⟩

[1] A gramática está na notação *Bauk-Naur Form*

⟨*Exp*⟩ ::= ⟨*ExpAdditiv*⟩
  | ‘(’ ⟨*ExpAdditiv*⟩ ‘>’ ⟨*ExpAdditiv*⟩ ‘)’
  | ‘(’ ⟨*ExpAdditiv*⟩ ‘<’ ⟨*ExpAdditiv*⟩ ‘)’
  | ‘(’ ⟨*ExpAdditiv*⟩ ‘>=’ ⟨*ExpAdditiv*⟩ ‘)’
  | ‘(’ ⟨*ExpAdditiv*⟩ ‘<=’ ⟨*ExpAdditiv*⟩ ‘)’
  | ‘(’ ⟨*ExpAdditiv*⟩ ‘==’ ⟨*ExpAdditiv*⟩ ‘)’
  | ‘(’ ⟨*ExpAdditiv*⟩ ‘!=’ ⟨*ExpAdditiv*⟩ ‘)’

⟨*Else*⟩ ::= ⟨⟩
  | ‘ELSE’ ’’⟨*InstructionsList*⟩ ’’

⟨*Atribution*⟩ ::= ⟨*Variable*⟩ ‘=’ ⟨*ExpAdditiv*⟩

⟨*Instruction*⟩ ::= ⟨*Atribution*⟩ ‘;’
  | ‘READ’ ⟨*Variable*⟩ ‘;’
  | ‘WRITE’ ⟨*ExpAdditiv*⟩ ‘;’
  | ‘WRITE’ ⟨*string*⟩ ‘;’
  | ‘IF’ ‘(’ ⟨*Exp*⟩ ‘)’ ‘{’ ⟨*InstructionsList*⟩ ‘}’ ⟨*Else*⟩
  | ‘WHILE ‘(’ ⟨*Exp*⟩ ‘)’ ‘{’ ⟨*InstructionsList*⟩ ‘}’
  | ‘DO’‘{’ ⟨*InstructionsList*⟩ ‘}’‘WHILE ‘(’ ⟨*Exp*⟩ ‘)’ ‘;’

⟨*InstructionList*⟩ ::= ⟨*Instruction*⟩
  | ⟨*InstructionList*⟩ ⟨*Instruction*⟩

Apêndice B

Código do analisador léxico do *Flex*

Apêndice C

Código do analisador sintático e tradutor do *YACC*

Apêndice D

Código gerado a partir do tradutor do *YACC*, para os exemplos pedidos

D.1 Maior de três números

D.2 Somatório de `N` números

D.3 Sequência de pares de `N` números dados

D.4 Ordenação de *array* de tamanho `N`–*Insertion Sort*

D.5 Média e máximo de uma matriz `[N][M]`

D.6 Cálculo de uma potência com determinada base e expoente–função `potencia(Base,Exp)`