Universidade do Minho

Mestrado Integrado em Engenharia Informática

Processamento de Linguagens TP3
Linguagem para definição de dados genealógicos

Bruno Martins (a80410)
Filipe Monteiro (a80229)

10 de Junho de 2019

Resumo

Este trabalho tem como foco a utilização das ferramentas *Flex* e *Yacc* para o desnvolvimento de uma linguagem para definir dados genealógicos. Como auxílio tabém foram utilizadas estruturas de dados na linguagem C, nomeadamente a biblioteca *GLib* no caso de Listas Ligadas e *Hash Tables*. Para isso foi desenvolvida uma gramática que define formalmente esta notação de forma a ser possível a geração automática do processador pretendido através do *Yacc*.

Conteúdo

1 Introdução

Este trabalho tem como objetivo a utilização do analisador sintático *Yacc* em conjunto com o analizador léxico *Flex* para a criação de uma linguagem capaz de representar dados genealógicos. O *Yacc* é o que vai definir as regras gramaticas que a notação tem de seguir obrigatóriamente. O *Flex* é utilizado pois o analisador sintático não consegue ler diretamente a partir de um input de dados, ele necessita de uma série de *tokens* que, neste caso, irão ser fornecidos pelo *Flex*. Para satisfazer as necessidades de output propostas são ainda utilizadas estruturas de dados na linguagem de programação C para armazenar a informação lida pelos analizadores.
Nos seguintes capítulos irá ser apresntado a descrição do problema, a implementação da solução, em que se irá entrar na explicação mais detalhada de como foi utilizado o *Yacc e Flex*, as Estruturas de dados que guardam a informação processada, os resultados obtidos de todo este programa e de seguida as respetivas conclusões a retirar.

2 Descrição do problema

O problema proposto foi implementar uma notação genealógica para definir relacionamentos familiares. Esta notação não se restringe apenas a relacionamentos incluindo também fotografias, eventos relevantes tais como datas de nascimento, casamentos ou outros à escolha. No seguinte exemplo podemos ver alguns desses casos:

```
Filipe/Monteiro *1998 +2019 EV 2000:batismo [3]
CC 2020 Ines Martins
FOTO pi.png
HIST pi.txt
P Teodosio Monteiro
M Raquel Monteiro
M M Eugenia Carvalho [4]
F Alexandre/Rodrigues [5]
```

É de salientar os graus de parentesco, podendo estes serem representados com as segintes marcas **P, M, -P, -M, PM, MM**, entre outras. Todas as relações de parentesco têm um sentido inverso em relação ao indivíduo que estamos a descrever. Também é importante retirar que todos as Pessoas possuem um identificador.

3 Implementação

3.1 Yacc

```
0 $accept: Ngen $end

    1 Ngen: Facto
    2     | Ngen Facto

    3 Facto: Pessoa '\n'
    4      | Pessoa '\n' Acoes

    5 Acoes: Acao '\n'
    6      | Acoes Acao '\n'

    7 Acao: Parentesco
    8     | Dados_Extra
    9     | Casamento
   10     | Evento

   11 Pessoa: Nomes Eventos ID
   12       | Nomes Eventos
   13       | Nomes ID
   14       | Nomes

   15 Parentesco: 'P' Info
   16           | '-' 'P' Info
   17           | 'P' Parentesco
   18           | '-' 'P' Parentesco
   19           | 'M' Info
   20           | '-' 'M' Info
   21           | 'M' Parentesco
   22           | '-' 'M' Parentesco
   23           | Filho

   24 Filho: 'F' Identificacao Eventos '{' Dados_Extra '}'
   25      | 'F' Identificacao Eventos
   26      | 'F' Identificacao

   27 Info: Identificacao Eventos
   28     | Identificacao

   29 Eventos: Evento
   30        | Eventos Evento
```

```
31 Nomes: Texto
32      | Texto '/' STRING
33      | Texto '%' NUM

34 Texto: STRING
35      | Texto STRING

36 Evento: NASCEU NUM
37       | NASCEUAPROX NUM
38       | MORREU NUM
39       | MORREUAPROX NUM
40       | EVENTO NUM ':' Texto

41 Casamento: CASAMENTO NUM Identificacao

42 Identificacao: ID
43              | Nomes ID
44              | Nomes

45 Dados_Extra: FOTO STRING
46            | HIST STRING

47 ID: '[' NUM ']'
```

Devido à extensão e do nível de complexidade da gramática gerada, não foi possível gerar o diagrama LR desta, apesar do YACC gerar automaticamente.
Esta gramática é caracterizada por:

- 46 regras;
- 24 Símbolos Terminais:
  $end, ’;’ ’%’, ’-’, ’/’, ’:’, ’F’, ’M’, ’P’, ’[’, ’]’, ’’, ’’, erro, FOTO, HIST, NASCEU, NASCEUAPROX, MORREU, MORREUAPROX, CASAMENTO, EVENTO, NUM, STRING ;
- 16 Símbolos Não-Terminais:
  $accept, Facto, Acoes, Acao, Pessoa, Parentesco, Filho, Info, Eventos, Nomes, Texto, Evento, Casamento, Identificacao, Dados_Extra, ID;
- O diagrama de estados (LR) possui um total de 75 diferentes estados.

Como podemos visualizar no código em cima, o input pode possuir 1 ou vários **Facto**. Este **Facto** pode ser composto por apenas uma **Pessoa** ou uma **Pessoa** e vários tipos de **Acao** (**Acoes**). As **Acao** podem ser

relações de parentesco - **Parentesco** -, dados extras da pessoa em questão, como foto ou biografia - **Dados_Extra** -, um casamento - **Casamento** -, e eventos, desde nascimento, morte entre outros especificados pelo utilizador - **Evento**.
Uma **Pessoa**, para além de um Nome (composto por nome e apelido, com número de indicação de repetição) - **Nomes** -, pode possuir um **ID** retirado do input e/ou **Eventos** como por exemplo nascimento.
Para os relacionamentos, este pode ser 'P' (pai), 'M' ou a junção de várias deste para representar vários graus de relação, com a informação da pessoa - **Info** - a que a "pessoa principal"está ligada. Pode também ter 'F' para indicar filhos, onde estes são representados usando **Identificacao** e **Eventos**, de modo semelhante à gramática da **Pessoa**, podendo também conter informação extra.
Como dito anteriormente, um **Evento** consiste em nascimento, morte, a aproximada de ambas e um evento personalizado pelo utilizador, caracterizado por um **Texto** e uma data - **NUM**.
Por fim, um **Texto** é constituido por 1 ou várias **STRING**.

Por fim, esta gramática não possui nenhum problema de **shift/reduce** nem **reduce/reduce**, tendo sido garantido, uma vez mais, pelo ficheiro de descrição da gramática gerado pelo YACC.

Para concretização do problema apresentado, foi necessário criar algumas estruturas de dados e variáveis globais dentro do .y de forma a guardarmos o interpretado pelo *parser*. Entre estas, temos uma HashTable, que contém todas as pessoas lidas até ao momento (sendo cada uma da estrutura *Pessoa*) e dois *char ** que contém possivel informaçao extra de um filho (juntamente com uma flag).

Para o *union*, estrutura que contém os parâmetros onde o Flex devolve o que lê e os tipos usados pelo *parser* para a atribuição de valores próprios dos diferentes símbolos não-terminais, foi criada a seguinte:

```
%union {
    int num;
    char* str;
    Pessoa p;
    Evento ev;
    GList* list;
    struct parenting rela;
}
```

Em termos de *token e types*, foram introduzidas:

```
%token FOTO HIST NASCEU NASCEUAPROX MORREU MORREUAPROX CASAMENTO
    EVENTO
%token <num> NUM
%token <str> STRING

%type <str> Dados_Extra Texto
%type <num> ID
%type <p> Pessoa Nomes Identificacao Info
%type <list> Eventos
%type <ev> Evento
%type <rela> Parentesco
```

3.2 Flex

Para a leitura do ficheiro, foram implementas 10 diferentes regras:

```
(FOTO)                { return FOTO; }
(HIST)                { return HIST; }
(\*)                  { return NASCEU; }
(\*C)                 { return NASCEUAPROX; }
(\+)                  { return MORREU; }
(\+C)                 { return MORREUAPROX; }
(CC)                  { return CASAMENTO; }
(EV)                  { return EVENTO; }
(F|M|P)               { return yytext[0]; }
[a-zA-Z]+(\.[a-zA-Z]+)? {yylval.str=strdup(yytext);return STRING;}
[\[\]:{}/%\n-]        { return yytext[0]; }
[0-9]+                { yylval.num = atoi(yytext); return NUM; }
```

Primeiro indicamos o regex mais específico que possuiamos, como certas palavras chave, para que estas não fossem confundidas com outras palavras ditas "normais". Por isso, retornavamos então os símbolos Terminais (palavras chave).
Necessitamos depois de ler vários símbolos, como por exemplo, '/' para separar o Nome e Apelido de uma pessoa, retornando assim o símbolo diretamente para o parser. Por fim, criamos uma regra para ler números, para datas e ID's, assim como uma regra para ler palavras, que podem conter ou não '.', devido a ficheiros mencionados no input. Estes dois, antes de retornarem o símbolo Terminal, atribuem o valor lido ao respetivo campo da estrutura "union"do YACC para este ter acesso ao valor interpretado.

4 Estruturas de Dados

As estruturas de dados utilizadas tiveram em contas a necessidade de armazenar a iniformação de forma adequado para que os eu acesso fosse rápido e sem grande complexidade. Foi criada uma estrutura Pessoa, que armazena toda a informação relativa a essa pessoa como nome, eventos na vida e identificadores de pai e de mãe quando estes se emcontram presentes nas declarações. Também foi necessário uma estrutura Evento para guardar os eventos genéricos que poderão ocorer.

```
typedef struct pessoa
{
    int id;
    int repetido;
    char* nome;
    char* apelido;
    int nasceu; // data de nascimento
    int morte; //data da morte
    char* foto;
    int idPai;
    int idMae;
    int idCasado; //id do seu conjuge
    int dataCasado;
    GList* filhos; //id dos filhos
    GList* eventos;
    char* historia;
    int flagMorteAprox;
    int flagNasceAprox;

}*Pessoa;

typedef struct evento
{
    int data;
    char* descricao;
}*Evento;
```

Para armazenar todas as pessoas que são lidas é utilizada uma *Hash Table* em que o ID de cada Pessoa é a chave utilizada nessa tabela. Esta *Hash Table* utilizada é proveniente da biblioteca *GLib* para facilitar toda a gestão de inserção e consultas à mesma. Também foi utilizada a lista ligada desta biblioteca.

5 Resultados

Exemplo gerado:

```
Filipe/Monteiro *1998 +2019 EV 2000:batismo [3]
CC 2020 Ines Martins
FOTO pi.png
HIST pi.txt
P Teodosio Monteiro
M Raquel Monteiro
M M Eugenia Carvalho [4]
F Alexandre/Rodrigues [5]
Rodrigo Monteiro *1993 +2100 [10]
```

Tendo gerados dois ficheiros, um prolog e outro semelhante ao pedido no problema, mais textual. Prolog:

```
pai('Teodosio','Filipe').
filho('Filipe','Teodosio').
mae('Raquel','Filipe').
filho('Filipe','Raquel').
avo('Eugenia','Filipe').
neto('Filipe','Eugenia').
casado('Filipe', 'Ines').
filho('Alexandre', 'Filipe').
pai('Filipe', 'Alexandre').
filho('Alexandre','Ines').
mae('Ines', 'Alexandre').
```

Este ficheiro Prolog apenas cria factos em relação ao parentesco das Pessoas. Todas estes factos podem ser testados num interpretador da liguagem. Textual:

```
#I3 nome Filipe Monteiro
#I3 data-nascimento 1998
#I3 data-falecimento 2019
#I3 FOTO pi.png
#I3 HISTORIA pi.txt
#I3 tem-como-mae #aut2
#I2 nome Raquel Monteiro
#I3 tem-como-pai #aut1
#I1 nome Teodosio Monteiro
#I3 tem-como-MM #aut4
#I4 nome Eugenia Carvalho
#F1 = #I3 #I0
#F1 data-casamento 2020
```

```
#I0 nome Ines Martins
#I5 nome Alexandre Rodrigues
#I3 batismo em 2000
```

Como é possível observar o output é igual ao esperado com ligeiras alterações de estética.

6 Conclusão

Com a realização deste trabalho podemos concluir que o *Yacc*, juntamente com o *Flex*, é uma ferramenta poderosa no que toca à criação de novas linguagens com uma notação à escolha de quem as desenvolve. No entanto, quando a linguagem necessita de uma gramática extensa o nível de complexidade dela pode ficar um pouco alto tanto ao nível de regras, símbolos terminais, simbolos não-terminais e totalidade de estados, dificultando a realização do trabalho aquando da interpretação do *input*.
Dito isto, todos os requisitos apresentados foram cumpridos.