Instituto Politécnico de Bragança

Engenharia Informática

Processamento de Linguagem

Trabalho nº 15

Trabalho Prático 1

Aluno 1: Gaofeng Yin a36233

Aluno 2: Hugo Gázio a36481

Professor: Paulo Matos & Célia Ramos

Maio

2020

Resumo

Uma linguagem para ser valida tem que cumprir as regras sintácticas e semânticas, porém nem todas as frases sintacticamente validas são semanticamente corretas. A gramática é um modo de descrever sintaxe de linguagens. As expressões regulares são utilizadas para descrever os símbolos válidos de uma dada linguagem.
Neste trabalho prático vamos construir um parser que faz o processamento da estrutura de dados da notação JSON que posteriormente serão necessários para próximo trabalho.
O LEX é o analisador léxico e a sua execução é chamado dentro do próprio ficheiro YACC. O YACC por sua vez valida a sequência de tokens detectados. Com YACC e LEX devemos produzir um analisador léxico e sintáctico que deve ser capaz de processar os dados de um ficheiro como input em formato JSON do Google Calendar.

Desenvolvimento

LEX

Iniciamos com o ficheiro.l, fazemos o include do y.tab.h em que vai estabelecer a comunicação entre YACC e LEX. Criamos as expressões regulares para identificar os valores de cada chave, vou apenas abordar algumas ER mais complexas:
A expressão regular date é capaz de conhecer datas em formato yyyy-mm-

```
3     //data em formato 1996-07-13
4   date (1|2)[0-9]+\-((0)[1-9]|1[012])\-(0[1-9]|[12][0-9]|3[01])
5     //hora em formato 23:50:46 / 06:00:00pm
6   time ([0-1]?[0-9]|2[0-3])\:[0-5][0-9]\:([0-5][0-9]|60)(pm|am)?
7     //data e hora em formato 2003-05-17 15:45:23
8   datetime {date}" "{time}
9     //expressão regular para websites
10  url (http:|https:)?(\/\/)?(www\.)?[a-zA-Z0-9\/\.\#]+
11    //expressão regular que identifica email
12  email [a-zA-Z@\.\-_0-9]+
```

Figura 1: Expressões regulares pré definidas.

dd. O ano tem que começar com o digito 1 ou 2 para ser valido, seguido de um ou mais numero de 0-9, o mês vai de 01 até 09, mas também pode ser 10, 11 ou 12 e aplicando a mesma lógica obtemos o dia. Os respectivos yyyy mm dd é separados pelo carácter '-'. O time pode ser em 00-23h ou AM/PM.

Depois unimos o date e time para formar apenas uma expreção regular de datetime que serve para identificar a data e hora ao mesmo tempo.

A seguir definimos as regras para reconhecer as chaves e o respectivo valor:

O timeZone, hidden, date, createdon , url e entre outros chaves são palavras

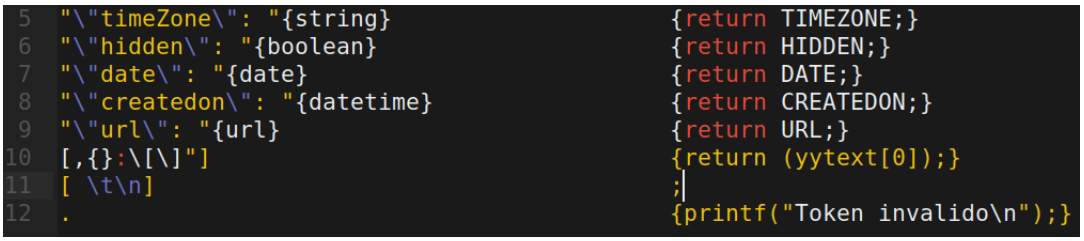

Figura 2: Identificadores de tokens.

reservadas ao qual está associado um valor dinâmico e por isso usamos a expressão regular pré definida entre chavetas ({string}) capaz validar o valor. Os simbolos da lina 10 da figura 2 são retornados como tokens. O programa não faz nada quando é encontrado uma quebra de linha, espaço e tab como está definido na linha 11 da figura 2. E por fim tudo o resto são tokens inválidos.

YACC

No YACC definimos a função main e a sequências de gramáticas validas. Os tokens que declaramos no YACC são reconhecidos no LEX e são usados para retornar o significado do input encontrado. É de denotar que os simbolos terminais estão em maiúsculas e os não terminais em minúsculas.

O start indica o ponto da partida da árvore de derivação que por derivações

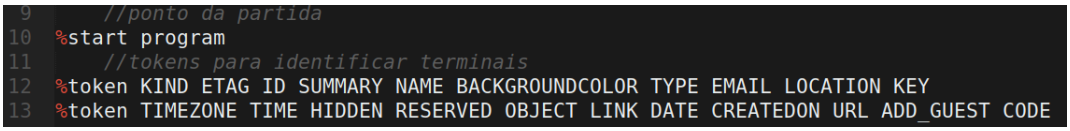

Figura 3: Tokens.

vai chegar até um terminal. Ao longo deste processo temos que analisar com atenção o documento de input, a forma como os dados estão agrupados e organizados.

```
            /*programa começa com duas chavetas que abrange todo ficheiro json,
            dentro desses chavetas estão os dados.*/
program     :    '{' datas '}'
        ;
            /*recursiva a direita.Existe pelo menos 1 ou mais objetos.*/
datas       :    data datas
            |
        ;
            /*os dados podem ser key , array , object ou array_object comforme o dado processado*/
data        :     key comma
            |     array comma
            |     object comma
            |     array_object comma
        ;
```

Figura 4: Gramátcia.

Começamos com o program que vai derivar em datas entre as duas chavetas. considerando que um input vai ter pelo menos 1 ou mais dados, então recorremos a recursividade a direita para dizer que datas vai derivar em data datas ou nada.
Por sua vez data pode derivar em key, array , object ou array_object comforme a estrutura de dados. A comma deriva em virgula ou nada porque há

```
            // é uma virgula ou não é nada
comma       : ','
            |
        ;
```

Figura 5: Virgulas.

caso de dados que aparecem com virgula e outros não, e para poupar linhas de código optei por essa solução.

```
        /*o array podem ter varios valores dentro ou array multidimensional */
array       :     '"' OBJECT '"' ':' '[' datas ']'
        ;
        /*dentro de array pode ter objetos */
array_object:     '"' OBJECT '"' ':' '[' program ']'
        ;
        /*um objeto*/
object      :     '"' OBJECT '"' ':' '{' datas '}'
        ;
        //key
key         :     KIND
            |     ETAG
          (...)
            |     KEY
            |     RESERVED
        ;
```

Figura 6: Formato de dados.

A sintaxe do array é constituido por um OBJECT que identifica o array e dentro tem datas que podem derivar em key , um objecto ou em array outra

vez.
O array_object tem um OBJECT que o identifica e dentro é outro program que pode ter varias derivações e pode ser tudo que definimos anteriormente. O object tal como array tem a estrutura semelhante, apenas muda nas chavetas. E por fim temos o key que são as chaves e o valor associado.
A função yyerror é chamado quando algo de incorrecto é detectado e mostra

```
72  /*o yyerror para identificar a linha onde ocorreu o erro caso haja alguma */
73  int yyerror(char *msg){
74      fprintf(stderr, "ERRO(%d):%s\n",yylineno, msg);
75      variavel = 1;
76      return 0;
77  }
```

Figura 7: função yyerror.

no ecrã o tipo de erro e a linha onde foi detectado o que ajuda imenso na implementação do parser e é essencial para reconhecimento do ficheiro JSON.
Finalmente a função main, se estiver tudo correcto é mostrado na ecrã

```
78  /*caso esteja tudo bem é executado o comando de printf dentro do if*/
79  int main(){
80      yyparse();
81      if(variavel != 1){
82      printf("Processado com sucesso!\n");
83      }
84      return 0;
85  }
```

Figura 8: Função main.

"Processado com sucesso!". Para isso criei uma variável de verificação. Se a função yyerror é executado então a variavel passa ser 1 e o printf do main não vai ser executado.

Resultado

Como podemos constatar na figura 9 os comandos yacc e flex executou sem nenhum erro e geramos o executável com sucesso. Na pasta estão 4 ficheiros de input, o input, input2 e input3 são aqueles em que os dados estão em formato JSON e foram validados com sucesso. O input4 deu erro porque o seu conteúdo não é legível, apenas foi incluído para mostrar o output quando dá erro.

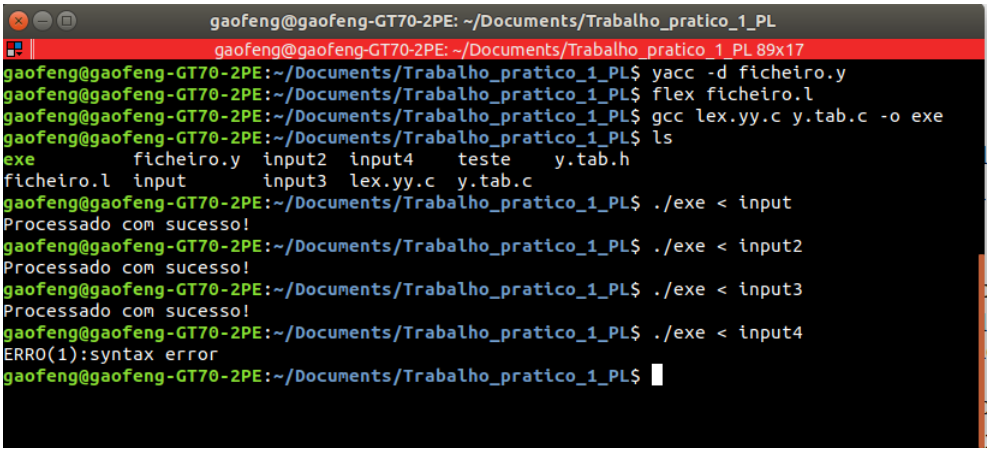

Figura 9: Resultado final.