Compiladores - 2021.1

Marcos Gabriel Pereira Paulo - 117031087

23 de julho de 2021

Mostre na prática que a tabela de parsing SLR é ambígua e que a LALR não é utilizando o gerador de analisador sintático PLY,
uma implementação em Python da ferramenta Yacc.

BNF

S -> L = R | R
L -> * R | id
R -> L

Para realizar a implementação da tabela de parsing para essa BNF acima, é necessário primeiro implementar um lexer e para isso, uma biblioteca Python chamada PLY foi utilizada. Abaixo está descrita a implementação do lexer.

```
from ply.lex import lex

tokens = ('ID', 'TIMES', 'EQUALS')

t_EQUALS = r'='
t_TIMES = r'\*'
t_ID = r'id'
t_ignore = ' \t'


def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)


lexer = lex()
```

Os tokens são definidos utilizando a notação t_nome contendo a regex que define o token e a função t_error para definir o comportamento quando nenhum dos tokens é identificado e então instancia-se o lexer.

Agora, são definidas as regras da BNF com funções com o padrão p_nome e contendo apenas um comentário com a definição da regra, seguindo a mesma lógica para a definição de erro. Abaixo está descrita a implementação da BNF.

```
def p_stmt_L_EQ_R(p):
    """
    S : L EQUALS R
    """

def p_stmt_R(p):
    """
    S : R
    """

def p_L_TIMES_R(p):
    """
    L : TIMES R
    """

def p_L_ID(p):
    """
    L : ID
    """

def p_R_L(p):
    """
    R : L
    """

def p_error(p):
    print(p)
```

E então instanciam-se os parsers, um que utiliza o método SLR e outro que utiliza o método LALR. Também são utilizados alguns parâmetros adicionais para gerar os arquivos de output separadamente e facilitar a análise de ambos. Abaixo está descrita a implementação dos parsers.

```
from ply.yacc import yacc

parser_slr = yacc(method='SLR', debugfile='parser_slr.out',
tabmodule='parsetab_slr')

parser_lalr = yacc(method='LALR', debugfile='parser_lalr.out',
tabmodule='parsetab_lalr')
```

Quando o script python é executado, são gerados 4 arquivos:

- parsetab_slr.py
- parsetab_lalr.py
- parser_slr.py
- parser_lalr.py

Porém, apenas os dois últimos são interessantes, já que contém informações para nossa análise das tabelas de parsing geradas.

Ao executar o script, uma primeira informação relevante é mostrada.

```
$python slrvslalr.py
Generating SLR tables
WARNING: 1 shift/reduce conflict
Generating LALR tables
```

A linha de warning já indica que existe um conflito de shift/reduce na tabela SLR. Seguindo a análise, existem mais informações sobre o conflito nos arquivos *.out gerados na primeira execução do script.

Segue abaixo o state 2 com o método SLR

```
state 2

    (1) S -> L . EQUALS R
    (5) R -> L .

  ! shift/reduce conflict for EQUALS resolved as shift
    EQUALS          shift and go to state 6
    $end            reduce using rule 5 (R -> L .)

  ! EQUALS          [ reduce using rule 5 (R -> L .) ]
```

Ocorre um conflito, pois nesse estado, ele pode deslocar e ir para o estado 6 e reduzir utilizando a regra 5 ou apenas reduzir utilizando a regra 5. Em seguida, o state 2 com o método LALR

```
state 2

    (1) S -> L . EQUALS R
    (5) R -> L .

    EQUALS          shift and go to state 6
    $end            reduce using rule 5 (R -> L .)
```

Aqui não há ambiguidade, já que existe apenas uma action para EQUALS no estado 2.

Sendo assim, observa-se que, apesar de a gramática não ser ambígua, a construção da tabela de parsing pode trazer ambiguidade a depender do método utilizado.