Universidade Federal de Santa Catarina
Departamento de Informática e Estatística
INE5426: Construção de Compiladores

Relatório do Exercício Programa 2: Analisador Sintático

Grupo:
Artur Barichello (16200636)
Lucas Verdade (17104409)
Lucas Zacchi (16104597)

Professor:
Álvaro Junio Pereira Franco

Agosto
2021

Sumário

1 Introdução **1**

2 Gramática CC-2021-1 **2**
2.1 Gramática CC-2021-1 2
2.2 Reescritura em forma convencional 3
2.3 Recursão à esquerda 4
2.4 Fatoração à esquerda 4

3 Implementação **6**
3.1 Analisador Sintático 6
3.2 Símbolo Inicial 6
3.3 Regras da gramática 6
3.4 Produções Vazias 7
3.5 Declaração de Variáveis 8
3.6 Tratamento de Erros 9

4 Entradas e Saídas **11**
4.1 Modificações da Entrega 1 11
4.2 Output PLY 12
4.3 Códigos de exemplo 13

1 Introdução

Este relatório descreve a implementação de um Analisador Léxico da gramática CC-2021-1.

Inicialmente foi feita a conversão da gramática que estava na forma BNF para a sua forma convencional intitulada ConvCC-2021-1, como pode ser vista na seção 2.2. Logo após foi verificado se na gramática resultante havia produções com recursão a esquerda ou produções que precisassem ser fatoradas a esquerda, respectivamente nas seções 2.3 e 2.4.

Na seção 3 foi descrita a implementação do exercício-programa, bem como a ferramenta utilizada e as entradas e saídas obtidas durante a execução.

A descrição e implementação do novo programa, escrito na linguagem CC-2021-1, assim como a descrição dos outros três programas escritos anteriormente, se encontra na seção 4.3.

Em relação ao Analisador Léxico, foram feitas alterações na tabela de símbolos. Este processo é descrito na seção 4.1.

2 Gramática CC-2021-1

2.1 Gramática CC-2021-1

PROGRAM	→ (STATEMENT | FUNCLIST)?
FUNCLIST	→ FUNCDEF FUNCLIST | FUNCDEF
FUNCDEF	→ def ident(PARAMLIST) {STATELIST}
PARAMLIST	→ ((int | float | string) ident, PARAMLIST | (int | float | string) ident)?
STATEMENT	→ (VARDECL; | ATRIBSTAT; | PRINTSTAT; | READSTAT; | RETURNSTAT; | IFSTAT | FORSTAT | STATELIST | break; | ;)
VARDECL	→ (int | float | string) ident ([int constant])*
ATRIBSTAT	→ LVALUE= (EXPRESSION | ALLOCEXPRESSION | FUNCCALL)
FUNCCALL	→ ident(PARAMLISTCALL)
PARAMLISTCALL	→ (ident, PARAMLISTCALL | ident)?
PRINTSTAT	→ print EXPRESSION
READSTAT	→ read LVALUE
RETURNSTAT	→ return
IFSTAT	→ if(EXPRESSION) STATEMENT (else STATEMENT)?
FORSTAT	→ for(ATRIBSTAT; EXPRESSION; ATRIBSTAT) STATEMENT
STATELIST	→ STATEMENT(STATELIST)?
ALLOCEXPRESSION	→ new(int | float | string) ([NUMEXPRESSION])$^+$
EXPRESSION	→ NUMEXPRESSION((< | > | <= | >= | == | ~=) NUMEXPRESSION)?
NUMEXPRESSION	→ TERM((+ | -) TERM)*
TERM	→ UNARYEXPR((*| |%) UNARYEXPR)*
UNARYEXPR	→ ((+ | -))? FACTOR
FACTOR	→ (int_constant | float_constant | string_constant | nil | LVALUE | (NUMEXPRESSION))
LVALUE	→ ident([NUMEXPRESSION])*

2.2 Reescritura em forma convencional

$PROGRAM$	→ STATEMENT | FUNCLIST | &
$FUNCLIST$	→ FUNCDEF FUNCLIST | FUNCDEF
$FUNCDEF$	→ def ident(PARAMLIST) {STATELIST}
$PARAMLIST$	→ DATATYPE ident, PARAMLIST | DATATYPE ident | &
$STATEMENT$	→ VARDECL; | ATRIBSTAT; | PRINTSTAT; | READSTAT; | RETURNSTAT; | IFSTAT | FORSTAT | STATELIST |break| ;
$VARDECL$	→ DATATYPE ident OPTIONAL_VECTOR
$OPTIONAL\_VECTOR$	→ [int_constant] OPTIONAL_VECTOR &
$ATRIBSTAT$	→ LVALUE=ATRIB_RIGHT
$ATRIB\_RIGHT$	→ EXPRESSION | ALLOCEXPRESSION | FUNCCALL
$FUNCCALL$	→ ident(PARAMLISTCALL)
$PARAMLISTCALL$	→ ident, PARAMLISTCALL | ident | &
$PRINTSTAT$	→ print EXPRESSION
$READSTAT$	→ read LVALUE
$RETURNSTAT$	→ return
$IFSTAT$	→ if(EXPRESSION) STATEMENT OPTIONAL_ELSE
$OPTIONAL\_ELSE$	→ else {STATEMENT} | &
$FORSTAT$	→ for(ATRIBSTAT; EXPRESSION; ATRIBSTAT) STATEMENT
$STATELIST$	→ STATEMENT OPTIONAL_STATELIST
$OPTIONAL\_STATELIST$	→ STATELIST | &
$ALLOCEXPRESSION$	→ new DATATYPE [NUMEXPRESSION] OPTIONAL_ALLOC_NUMEXPRESSION
$OPTIONAL\_ALLOC\_$ $\_NUMEXPRESSION$	→ [NUMEXPRESSION] OPTIONAL_ALLOC_NUMEXPRESSION | &
$EXPRESSION$	→ NUMEXPRESSION OPTIONAL_REL_OP_NUMEXPRESSION
$OPTIONAL\_REL\_OP\_$ $\_NUMEXPRESSION$	→ OPTIONAL_REL_OP_ NUMEXPRESSION | &
$NUMEXPRESSION$	→ TERM RECURSIVE_MINUS_OR_PLUS
$RECURSIVE\_$ $\_MINUS\_OR\_PLUS$	→ MINUS_OR_PLUS TERM RECURSIVE_MINUS_OR_PLUS | &
$MINUS\_OR\_PLUS$	→ + | −
$TERM$	→ UNARYEXPR RECURSIVE_UNARYEXPR
$RECURSIVE\_UNARYEXPR$	→ UNARYEXPR_OPERATOR RECURSIVE_UNARYEXPR | &
$UNARYEXPR\_OPERATOR$	→ ∗ | / | %
$UNARYEXPR$	→ MINUS_OR_PLUS FACTOR | FACTOR
$FACTOR$	→ int_constant | float_constant | string_constant | nil | LVALUE | (NUMEXPRESSION))
$LVALUE$	→ ident OPTIONAL_ALLOC_NUMEXPRESSION

2.3 Recursão à esquerda

Foi analisado se a gramática ConvCC-2021-1 possuía recursões a esquerda e pôde-se perceber que não há. Pois não existe nenhuma produção que seja do tipo:

$A \quad \rightarrow$ Ab | b

2.4 Fatoração à esquerda

Também foi solicitado para analisar se havia produções que precisassem ser fatoradas a esquerda e então fatorá-las. Notou-se que havia algumas produções que precisavavam ser fatoradas a esquerda. Abaixo estão as produções antes e depois de serem fatoradas:
Produção 1:

Antes **:**
$FUNCLIST$ → FUNCDEF FUNCLIST | FUNCDEF

Depois **:**
$FUNCLIST$ → FUNCDEF FUNCLISTTMP
$FUNCLISTTMP$ → FUNCLIST | &

Produção 2:

Antes **:**
$PARAMLISTCALL$ → LABEL COMMA PARAMLISTCALL | LABEL

Depois **:**
$PARAMLISTCALL$ → LABEL COMMA_PARAMLISTCALL_ OR_EMPTY | &

$COMMA\_PARAMLISTCALL\_OR\_EMPTY$ → COMMA PARAMLISTCALL | &

Produção 3:

***Antes* :**	
$PARAMLIST$	→ DATATYPE LABEL COMMA PARAMLIST | DATATYPE LABEL
***Depois* :**	
$PARAMLIST$	→ DATATYPE LABEL COMMA_PARAMLIST_ OR_EMPTY | &
$COMMA\_PARAMLIST\_$ OR_EMPTY	→ COMMA PARAMLIST | &

3 Implementação

3.1 Analisador Sintático

O Analisador Sintático e suas regras estão descritas no arquivo *src/syntax.py*.

Foi utilizada a ferramenta *PLY*[3] para a implementação do analisador sintático, especificamente o módulo *ply.yacc*. *Yacc* significa *"Yet Another Compiler Compiler"*, e é uma ferramenta comum para construção de analisadores sintáticos e compiladores, reescrita como uma biblioteca *Python*.

Através do uso desta biblioteca, foi definido um arquivo *src/syntax.py* onde são descritas as regras da gramática como funções *Python*, que são descritas e exemplificadas ao longo desta seção.

3.2 Símbolo Inicial

A primeira regra criada define a regra inicial da gramática, chamada de *PROGRAM* e demonstrada no trecho de código 1

Código Fonte 1: Função p_PROGRAM

```
def p_PROGRAM(p: LexToken) -> None:
    """
    PROGRAM : STATEMENT
            | FUNCLIST
            | empty
    """
    p[0] = p[1]
```

3.3 Regras da gramática

Em concordância com a documentação da biblioteca *PLY* cada regra da gramática foi implementada como uma função, como mostra o trecho de código 2:

Código Fonte 2: Função p_DATATYPE

```
def p_DATATYPE(p: LexToken) -> None:
    """
    DATATYPE : INTEGER
```

```
            | FLOATING_POINT
            | STRING
    """
    p[0] = p[1]
```

As *docstrings* de cada função descrevem a especificação das regras de produção da gramática, e os corpos das funções representam as ações realizadas pelas regras.

Cada função recebe um argumento p, que consiste em uma lista de símbolos e valores da regra correspondente. Utilizando o exemplo do trecho de código 2, a lista p e a declaração da função se dão como segue:

Código Fonte 3: Função p_DATATYPE Exemplificada

```
def p_DATATYPE(p: LexToken) -> None:
    """
    DATATYPE : INTEGER
             | FLOATING_POINT
             | STRING
    """
    #  ^           ^
    # p[0]       p[1]

    p[0] = p[1]
```

3.4 Produções Vazias

Definimos uma produção vazia da seguinte maneira:

Código Fonte 4: Função p_empty

```
def p_empty(p: LexToken) -> None:
    "empty :"
     pass
```

E regras de produção que contém a produção vazia utilizam o termo *empty* para denotá-las, como exemplificado:

Código Fonte 5: Regra com produção vazia

```
def p_PARAMLISTCALL(p: LexToken) -> None:
    """
    PARAMLISTCALL : LABEL COMMA PARAMLISTCALL
                  | LABEL
                  | empty
    """
    if len(p) > 2:
        p[0] = (p[1], p[3])
    else:
        p[0] = p[1]
```

Aproveitando o exemplo da função p_PARAMLISTCALL5, quando uma regra de produção possui quantidade de termos variável, essas situações foram testas analisando o tamanho da lista de Tokens, através de *len(p)*.

Além disso, quando houve a necessidade de retornar mais de um termo por função, como no caso já mencionado, onde *PARAMLISTCALL* pode produzir *LABEL COMMA PARAMLISTCALL*, foram utilizadas estruturas de dados para retornar os múltiplos termos.

3.5 Declaração de Variáveis

A declaração de variáveis, definida pela regra de prodrução *VARDECL*, declarada no trecho de código 6, foi construída de maneira diferente do que foi mostrado até o momento. Para sua implementação, foi definida uma variável global do tipo *Dict* para armazenar variáveis:
`variables: Dict[str, Tuple[str, Any]] = {}`, que é estruturada da seguinte forma:

```
variables: Dict[value, Tuple[type, value]]
```

Código Fonte 6: Função p_VARDECL

```
def p_VARDECL(p: LexToken) -> None:
    """
    VARDECL : DATATYPE LABEL OPTIONAL_VECTOR
    """
    datatype = p[1]
    label = p[2]

    if label in variables.keys():
        raise VariableAlreadyDeclared

    variables[label] = (datatype, None)
```

Em sua execução, a cada variável encontrada no código é feito um teste para verificar se ela já foi declarada anteriormente. Caso ela já esteja presente no dicionário *variables*, é levantada uma exceção *VariableAlreadyDeclared*. Caso contrário, ela é adicionada ao dicionário.

3.6 Tratamento de Erros

É importante que o *parser* não interrompa sua execução ao encontrar um erro, pois seria pouco eficiente e prejudicial para o funcionamento da análise sintática. Ao invés disso, foi utilizado um método de tratamento de erros sintáticos para que o *parser* possa notificar o erro, e se possível continuar a execução, dem modo que possíveis outros erros sejam identificados.

Para isso, foi criada a função *p_error* que recebe um token *p* e imprime a linha e a posição em que o erro ocorreu, como declarado a seguir:

Código Fonte 7: Trecho de syntax.py

```
def p_error(p: LexToken) -> None:
    print(
        f"""Syntax error at token {p}
        Line:{p.lineno-1} | Column:{p.lexpos-2}"""
    )
```

Quando um erro de sintaxe ocorre, a biblioteca *yacc.py* chama a função *p_error* com o token problemático como argumento.

p_error também é chamada quando o *parser* encontra o final do arquivo. Neste caso, o argumento passado é *None*.

4 Entradas e Saídas

4.1 Modificações da Entrega 1

Em relação ao Analisador Léxico, foram feitas alterações no processo de geração de tabela de símbolos, como demonstrado a seguir.

A partir da entrada descrita no trecho de código 8, foi gerada a tabela de símbolos, exemplificada logo abaixo.

Código Fonte 8: Função de exemplo para tabela de símbolos

```
def print_all {
    string x;
    string y;
    int a;
    int b;
    a = 2;
    b = 3;
    if (a > b) {
        x = "yes";
        print x;
    } else {
        y = "no";
        print y;
    }
}
```

```
Value        Index    Declaration (line)  Referenced (lines)
---------  -------  --------------------  --------------------
print_all        4                     1  []
x               27                     2  [9, 10]
y               41                     3  [12, 13]
a               52                     4  [6, 8]
b               63                     5  [7, 8]
```

A nova tabela de símbolos foi organizada para representar apenas os identificadores (*labels*), o índice e linha onde são declarados, e as linhas subsequentes onde são referenciados.

No caso da entrada possuir um símbolo que não está definido na gramática o programa pula este símbolo e aborta a sua execução com uma mensagem de erro indicando o caracter desconhecido e a sua posição no formato [linha:coluna] de maneira similar como é feito em Lua.

Tal comportamento pode ser observado ao rodar o arquivo *./source_ code/error.lua* que computa o seguinte resultado:

```
def invalid_char_function(int A, int B) {
  int invalidchar!![2];
  return;
}

Input file(2:18) '!' is an invalid character
```

4.2 Output PLY

A análise sintática é feita através do módulo Yacc implementado pela biblioteca PLY, o código fonte em string é passado da seguinte maneira:

```
print("Executing yacc")
parser = yacc.yacc()
# result = parser.parse(src) # non-debug mode
result = parser.parse(src, debug=True)
pprint(result)
```

O argumento *debug* no comando *parser.parse(input_ text, debug=True)* nos permite visualizar mais detalhes sobre a operação sendo executada. No código dessa entrega foi atribuído o valor True ao argumento debug seguindo as orientações da documentação oficial da PLY. Essa informação foi omitida do relatório devido ao seu tamanho porém pode ser vista ao rodar o comando *make example* ou na seção respectiva da documentação da biblioteca. [2]

Junto com a ply tentamos utilizar algumas estruturas de dados para identificar e propagar os tokens que foram analisados, tais estruturas estão no topo do arquivo *syntax.py* e são identificados pelas classes com o decorator *@dataclass*. Vários tutoriais online que foram utilizados como referência também implementam o parser desta maneira. [1] [4]

O único fluxo de manipulação dos tokens gerados que foi completamente implementado é o da produção *PRINTSTAT* produzida através de vários *STATELIST* portanto o único exemplo que roda inteiramente são programas pequenos como o *printstat.lua*, outros fluxos também foram implementados porém não foram completamente integrados portando não foi possível fazer o

parse de um arquivo de exemplo inteiro.

4.3 Códigos de exemplo

Na entrega anterior foram implementados 3 arquivos (*program1.lua, program2.lua, program3.lua*) conforme especificado pelo enunciado. O primeiro algoritmo simula um sistema de notas de uma turma de alunos com as notas das provas geradas de acordo com a sua posição. O programa calcula as médias de cada aluno e salva numa lista de notas finais. Também é executada uma função que imprime o nome de cada aluno, suas notas, sua média final e diz se o aluno foi aprovado ou não.

O segundo programa (*program2.lua*) simula um sistema de controle de estoque de produtos genéricos. Cada produto tem nome, descrição e preço. É possível adicionar novos produtos, atualizar os dados de um produto e deletar um produto. Também trata erros para não permitir a adição de um produto quando o estoque estiver cheio e avisa quando o produto solicitado não existe. Ao final do programa a função *main()* é chamada e nela são feitas algumas chamadas das funções, definidas anteriormente. São adicionados 5 produtos diferentes seguido de uma tentativa de adicionar outro produto com o estoque cheio, logo após um produto é apagado e outro é atualizado.

O terceiro programa (*program3.lua*) simula um sistema de reserva de assentos de um cinema/teatro. Nele o usuário pode definir o nome do seu estabelecimento, que será exibido na tela de inicialização, e também o tamanho máximo de filas de assentos. Também pode consultar o status de quantos assentos estão ocupados, quantos estão livres e qual o total de assentos. Foram definidas funções para lidar com a reserva de assentos e também para liberar uma posição.

A função de reservar verifica se a posição escolhida é válida e se está livre, caso esteja então coloca o id do usuário na posição do assento. Caso não esteja livre o programa informa que está ocupado e então faz uma sugestão de assentos vizinhos livres. A função *main* inicializa um cinema novo com 25 assentos (5 filas e 5 assentos) com o nome de: 'Praia de Belas Cinema'. Então, imprime o status do cinema e tenta fazer duas reservas, a segunda reserva é apenas para demonstrar que não é possível reservar um lugar já ocupado.

O novo código implementado (*program4.lua*) para essa entrega se trata

de uma biblioteca simples de operações matemáticas, nela existem funções definidas: calcular potência entre 2 números, exponenciais com a constante de Euler, valor absoluto, arredondar o input para o menor valor inteiro, arredondar o input para o maior valor inteiro e arredondar o input para o mais próximo inteiro. Também inclui uma função main que roda testes básicos e imprime o resultado esperado e o resultado encontrado. A constante de Euler foi fixada em *2.718* arbitrariamente.

Referências Bibliográficas

Referências

[1] *AST Construction*. URL: `https://ply.readthedocs.io/en/latest/ply.html#ast-construction`. (Acessado em: 13/08/2021).

[2] *Miscellaneous Yacc Notes*. URL: `https://ply.readthedocs.io/en/latest/ply.html#miscellaneous-yacc-notes`. (Acessado em: 13/08/2021).

[3] *Ply (Python Lex-Yacc)*. URL: `https://ply.readthedocs.io/en/latest/ply.html`. (Acessado em: 03/07/2021).

[4] *Writing your own programming language and compiler using Python and PLY*. URL: `https://blog.usejournal.com/writing-your-own-programming-language-and-compiler-with-python-a468970ae6df`.