

# Universidade Federal de Santa Catarina

## Relatório T2 Construção de Compiladores

Isac de Souza Campos (17200449)

Arthur Philippi Bianco (17203358)

Enzo Coelho Albornoz (18100527)

### Forma BNF

A forma Backus-Naur serve para representação de gramáticas livres de contexto. Devemos eliminar esta notação e converter para forma convencional, a seguir a gramática **antes** de realizar esta conversão:

**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|FOR  
STAT|{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** → UNARY EXPR((\*|/|%)UNARY EXPR)\*

**UNARYEXPR** → ((+|-))?FACTOR

**FACTOR**  $\rightarrow$  (**int\_constant**|**float\_constant**|**string\_constant**|**null**|**LVALUE**|(**NUMEXPRESSION**))  
**LVALUE**  $\rightarrow$  **ident**( [**NUMEXPRESSION**] )\*

O processo de conversão da gramática em notação BNF para notação convencional foi feito de forma a não produzir recursão à esquerda. Vejamos um exemplo:

**NUMEXPRESSION**  $\rightarrow$  **TERM** ((+|-) **TERM**)\*

Esta expressão pode ser convertida para o formato convencional, ingenuamente, para a seguinte expressão:

**NUMEXPRESSION**  $\rightarrow$  **NUMEXPRESSION** (+|-) **TERM** | **TERM**

Esta expressão é equivalente à expressão anterior, porém, o problema deste tipo de conversão é a introdução da recursão direta à esquerda. Outra alternativa seria criar um novo não-terminal:

**NUMEXPRESSION**  $\rightarrow$  **NUMEXPRESSION2** | **TERM**  
**NUMEXPRESSION2**  $\rightarrow$  **NUMEXPRESSION** ((+|-) **TERM**)

Contudo, esta nova expressão possui também uma recursão, porém indireta à esquerda. Portanto, a melhor solução é a seguinte, que está na forma convencional e sem recursão à esquerda:

**NUMEXPRESSION**  $\rightarrow$  (**TERM**(**ADDSUBTERM**)) | **TERM**  
**ADDSUBTERM**  $\rightarrow$  ((+|-)**TERM**)**ADDSUBTERM** | ((+|-)**TERM**)

Estando consciente destes problemas de introdução de recursão à esquerda à gramática ao converter para a forma convencional, foram removidas as recursões já durante o processo de conversão. A seguir a gramática resultante:

**\*Programa ConvCC-2021-1 já sem recursão à esquerda:**

**PROGRAM**  $\rightarrow$  **STATEMENT** | **FUNCLIST** |  $\epsilon$   
**FUNCLIST**  $\rightarrow$  **FUNCDEF** **FUNCLIST** | **FUNCDEF**

**FUNCDDEF** -> def ident(PARAMLIST){STATELIST} | def  
 ident(){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(VARDECLARRAY)) |  
 ((int|float|string)ident)  
**VARDECLARRAY** -> ([int\_constant]VARDECLARRAY) | [int\_constant]  
 ATRIBSTAT -> LVALUE=(EXPRESSION|ALLOCEXPRESSION|FUNCCALL)  
 FUNCCALL -> ident() | ident(PARAMLISTCALL)  
**PARAMLISTCALL** -> (ident, PARAMLISTCALL) | ident  
 PRINTSTAT -> print EXPRESSION  
 READSTAT -> read LVALUE  
 RETURNSTAT -> return  
**IFSTAT** -> if(EXPRESSION)STATEMENT else STATEMENT |  
 if(EXPRESSION)STATEMENT  
 FORSTAT -> for(ATRIBSTAT;EXPRESSION;ATRIBSTAT) STATEMENT  
**STATELIST** -> STATEMENT STATELIST | STATEMENT  
 ALLOCEXPRESSION -> new(int|float|string)NUMEXPRESSIONARRAY  
**NUMEXPRESSIONARRAY** -> ([NUMEXPRESSION]NUMEXPRESSIONARRAY) |  
 [NUMEXPRESSION]  
**EXPRESSION** -> (NUMEXPRESSION((< | > | <= | >= | == |  
 !=)NUMEXPRESSION)) | NUMEXPRESSION  
**NUMEXPRESSION** -> (TERM(ADDSUBTERM)) | TERM  
**ADDSUBTERM** -> ((+|-)TERM)ADDSUBTERM | ((+|-)TERM)  
**TERM** -> UNARYEXPR TERMULTDIVMOD | UNARYEXPR  
**TERMULTDIVMOD** -> (((\* | / | %)UNARYEXPR)TERMULTDIVMOD) | ((\* | /  
 | %)UNARYEXPR)  
 UNARYEXPR -> ((+|-)) FACTOR | FACTOR  
 FACTOR -> (int\_constant|float\_constant|string\_constant|null||LV  
 ALUE| (NUMEXPRESSION))  
**LVALUE** -> ident[NUMEXPRESSIONARRAY] | ident

## Recursão à esquerda:

A linguagem resultante não possui recursão direta à esquerda, eliminando a necessidade de tratamento desta propriedade. Estes casos são indesejáveis pois podem acabar gerando recursão infinita se o analisador sintático utilizado na compilação for recursivamente descendente, por exemplo, quebrando o algoritmo de compilação. A propriedade é caracterizada pela sua forma direta e indireta. Um exemplo de caso direto é visto a seguir:

- $S \rightarrow S\alpha \mid \beta$

Nota-se que, o símbolo não terminal  $S$ , que é cabeça de produção, também é encontrado à esquerda de um dos possíveis caminhos a serem tomados. Porém, apesar da facilidade de se identificar o problema neste primeiro exemplo, em alguns casos a recursão à esquerda pode não ser tão facilmente encontrada, pois não se faz presente diretamente na produção de origem do símbolo não terminal analisado. A seguir vemos um exemplo de recursão indireta:

- $S \rightarrow A\alpha \mid \beta$
- $A \rightarrow A\alpha \mid S\alpha \mid \beta$

Percebemos que, apesar de não existir recursão diretamente na produção de  $S$ , se o caminho tomado pela gramática seguir por  $A\alpha$  e, então, chegar a  $S\alpha$ , isso pode gerar o mesmo problema de recursão infinita visto no exemplo anterior.

Para provarmos que não há recursão à esquerda na gramática ConvCC-2021-1 obtida, utilizaremos a própria ferramenta *pest* que escolhemos para realizar a parte prática deste projeto. O *pest* é uma biblioteca de parsing de propósito geral que implementa um Parsing Expression Grammar [1]. Esta gramática é bastante parecida com gramáticas livre de contexto (CFG), porém os PEGs não são capazes de reconhecer gramáticas ambíguas, pois eles escolhem a primeira opção de derivação sempre, de modo que para cada string válida existe apenas uma árvore de análise sintática que a-descreva. Mais importante do que isso, PEGs são incapazes de expressar recursão à esquerda, ficando para sempre presos em loop nestas produções recursivas. Portanto, para poder utilizar este tipo de analisador, deve-se remover a recursão à esquerda. Reescrevemos a gramática usada em código para ser idêntica à BNF que produzimos no item anterior, e, caso haja recursão à esquerda, existirão

erros. Como o *pest* conseguiu montar a PEG a partir da descrição BNF, ela necessariamente não possui recursão à esquerda.

## Fatoração à esquerda

A fatoração à esquerda é importante pois garante o funcionamento eficiente de analisadores sintáticos preditivos que tenham uma gramática com essa característica como entrada. Isso ocorre pois a fatoração garante que nenhuma produção tenha prefixos iguais, eliminando os não-determinismos e permitindo que o analisador não cometa erros ao escolher o caminho a ser tomado.

A gramática ConvCC-2021-1 vista anteriormente possui algumas produções que necessitam de tratamento para se encaixarem neste perfil. A seguir, temos o resultado obtido após a fatoração ser aplicada.

ConvCC-2021-1 fatorada à esquerda:

**PROGRAM** -> STATEMENT | FUNCLIST\_

**FUNCLIST** -> FUNCDEF FUNCLIST\_

**FUNCLIST\_** -> FUNCLIST |  $\epsilon$

**FUNCDEF** -> def ident(FUNCDEF\_

**FUNCDEF\_** -> PARAMLIST){STATELIST} | ){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 VARDECL\_

**VARDECL\_** -> VARDECLARRAY |  $\epsilon$

**VARDECLARRAY** -> [int\_constant] VARDECL\_

**ATRIBSTAT** -> LVALUE=(EXPRESSION | ALLOCEXPRESSION | FUNCCALL)

***FUNCCALL*** -> ident(***FUNCCALL\_***  
***FUNCCALL\_*** -> ) | ***PARAMLISTCALL***)  
***PARAMLISTCALL*** -> ident ***PARAMLISTCALL\_***  
***PARAMLISTCALL\_*** -> , ***PARAMLISTCALL*** |  $\epsilon$   
***PRINTSTAT*** -> print ***EXPRESSION***  
***READSTAT*** -> read ***LVALUE***  
***RETURNSTAT*** -> return  
***IFSTAT*** -> if(***EXPRESSION***)***STATEMENT*** ***IFSTAT\_***  
***IFSTAT\_*** -> else ***STATEMENT*** |  $\epsilon$   
***FORSTAT*** -> for(***ATRIBSTAT***;***EXPRESSION***;***ATRIBSTAT***) ***STATEMENT***  
***STATELIST*** -> ***STATEMENT*** ***STATELIST\_***  
***STATELIST\_*** -> ***STATELIST*** |  $\epsilon$   
***ALLOCEXPRESSION*** -> new(int | float | string) ***NUMEXPRESSIONARRAY***  
***NUMEXPRESSIONARRAY*** -> [***NUMEXPRESSION***] ***NUMEXPRESSIONARRAY\_***  
***NUMEXPRESSIONARRAY\_*** -> ***NUMEXPRESSIONARRAY*** |  $\epsilon$   
***EXPRESSION*** -> ***NUMEXPRESSION*** ***EXPRESSION\_***  
***EXPRESSION\_*** -> (< | > | <= | >= | == | !=) ***NUMEXPRESSION*** |  $\epsilon$   
***NUMEXPRESSION*** -> ***TERM***(***NUMEXPRESSION\_***)  
***NUMEXPRESSION\_*** -> ***ADDSUBTERM*** |  $\epsilon$   
***ADDSUBTERM*** -> ((+|-)***TERM***) ***NUMEXPRESSION\_***  
***TERM*** -> ***UNARYEXPR*** ***TERM\_***

**TERM\_ -> TERMULTDIVMOD |  $\epsilon$**

**TERMULTDIVMOD -> (\* | / | %)UNARYEXPR TERM\_**

**UNARYEXPR -> ((+|-)) FACTOR | FACTOR**

**FACTOR -> int\_constant | float\_constant | string\_constant | null | LVALUE |  
(NUMEXPRESSION)**

**LVALUE -> ident LVALUE\_**

**LVALUE\_ -> [NUMEXPRESSIONARRAY] |  $\epsilon$**

Algumas alterações além da fatoração foram feitas para fins de otimização da gramática. Após a fatoração, um símbolo não-terminal TERMULTDIVMOD\_ foi criado. Porém, este não-terminal possuía as mesmas configurações do símbolo TERM\_. Por conta disso e para diminuir o número de símbolos da gramática, os TERMULTDIVMOD\_ contidos à direita das produções foram substituídos por TERM\_. Outros dois casos como este foram os de substituição de ADDSUBTERM\_ por NUMEXPRESSION\_ e VARDECLARRAY\_ por VARDECL\_.

Além dessas substituições, uma pequena alteração foi feita na produção de PROGRAM, transformando o trecho “FUNCLIST |  $\epsilon$ ” em FUNCLIST\_, uma vez que a produção de FUNCLIST\_ já possuía como produção o trecho adaptado.

Após obtermos a gramática fatorada, passamos para o próximo passo, que consiste em provar que a gramática está em LL(1).

### **Gramática em LL(1):**

O último passo a ser dado no tratamento da gramática deste trabalho é provar que ela está em LL(1). Para isso, devemos primeiro calcular os conjuntos *First* e *Follow* dos símbolos pertencentes à ela. Visando uma maior objetividade do relatório, apenas os conjuntos referentes aos símbolos não-terminais serão demonstrados. O resultado da obtenção de ambos os conjuntos pode ser visto na tabela 1.

FIRST	FOLLOW
int, float, string, ident, print, read, return, if, for, {, break, ';'def, $\epsilon$	\$
def	\$
def, $\epsilon$	\$
def	def, \$
int, float, string, )	def, \$
int, float, string	)
int, float, string, ident, print, read, return, if, for, {, break, ';'	\$, int, float, string, ident, print, read, return, if, for, '{, break, ';'
int, float, string	"."
[, $\epsilon$	"."
[	"."
ident	"", )
(, ident	"", )
ident	)
",", $\epsilon$	)
print	"."
read	"."
return	"."
if	\$, int, float, string, ident, print, read, return, if, for, {, break, ';'
else, $\epsilon$	\$, int, float, string, ident, print, read, return, if, for, {, break, ';'
for	\$, int, float, string, ident, print, read, return, if, for, {, break, ';'
int, float, string, ident, print, read, return, if, for, {, break, ';'	}
int, float, string, ident, print, read, return, if, for, {, break, ';', $\epsilon$	}
new	"", )
[	"", ), ]



[, ε	";", ), ]
+', '-', int_constant, float_constant, string_constant, null, ident, '('	";", )
<, >, <=, >=, ==, !=, ε	";", )
+', '-', int_constant, float_constant, string_constant, null, ident, '('	], <, >, <=, >=, ==, !=, ";", )
"+", "-", ε	], <, >, <=, >=, ==, !=, ";", )
"+", "-"	], <, >, <=, >=, ==, !=, ";", )
+', '-', int_constant, float_constant, string_constant, null, ident, '('	+', '-', ']', '<', '>', '<=', '>=', '==', '!=', ';;', ')'
"*", "/", "%", ε	+', '-', ']', '<', '>', '<=', '>=', '==', '!=', ';;', ')'
"*", "/", "%"	+', '-', ']', '<', '>', '<=', '>=', '==', '!=', ';;', ')'
+', '-', int_constant, float_constant, string_constant, null, ident, '('	*, '/', '%', +, -, ], <, >, <=, >=, ==, !=, ';;', ')'
int_constant, float_constant, string_constant, null, ident, (	*, '/', '%', +, -, ], <, >, <=, >=, ==, !=, ';;', ')'
ident	=, '*', '/', '%', +, -, ], <, >, <=, >=, ==, !=, ';;', ')'
[, ε	=, '*', '/', '%', '+', '-', ']', '<', '>', '<=', '>=', '==', '!=', ';;', ')'

Tabela 1: Conjuntos *First* e *Follow* obtidos através da gramática ConvCC-2021-1.

Com base nos conjuntos gerados, uma Tabela de Análise Sintática pôde ser criada para observarmos se a gramática pertence ou não à LL(1). Lembrando que, para que uma gramática pertença à LL(1), cada espaço de relação entre um símbolo terminal e um não-terminal na tabela (que indica o caminho a ser tomado pelo algoritmo) deve contar com apenas uma produção. Isso ocorreu na tabela feita para a gramática ConvCC-2021-1 em estudo. Por conta de seu tamanho extenso, esta tabela não será mostrada no presente documento. Porém, pode-se ter acesso à mesma através do link permanente:

[https://docs.google.com/spreadsheets/d/e/2PACX-1vTm8uK7CX\\_v2Zc0gl97WdxfHh7iojtpPaPaZfcYF82GI96x8QhFbRdq3kGymWeDxoewujYBn1LYk7B3/pubhtml?gid=1892824522&single=true](https://docs.google.com/spreadsheets/d/e/2PACX-1vTm8uK7CX_v2Zc0gl97WdxfHh7iojtpPaPaZfcYF82GI96x8QhFbRdq3kGymWeDxoewujYBn1LYk7B3/pubhtml?gid=1892824522&single=true)

## Programa

Foi adicionado o subcomando **syntax** cuja execução irá fazer uma análise sintática do arquivo fonte dado. Em caso de sucesso, uma mensagem de sucesso irá aparecer.

```
> make run ARGS="syntax ./examples/T2/T2E3.ccc"
./target/release/sapo syntax ./examples/T2/T2E3.ccc
```

Statistics	
Status	Success
Elapsed Time:	0.001s

```
enzo@Nitro5 ~/Workspace/study/compilersConstruction/sapo master ● 1
```

Por uma questão de como o *parser* é implementado (como ele reconhece mais que uma LL(1), seu reconhecimento não possui o mesmo fluxo que execução, consumindo os tokens de maneira diferente e usando algumas estruturas diferentes) não temos como mostrar a entrada na tabela LL(1). Em outras palavras, não há tabela de análise sintática em LL(1) da qual tirar as informações requisitadas.

Contudo, temos os dados de token lido e tokens esperados. Caso haja um erro na leitura é mostrado o token lido e dito quais tokens seriam os corretos. Vejamos um exemplo abaixo:

```
make run ARGS="syntax ./examples/T2/T2E3.ccc"
./target/release/sapo syntax ./examples/T2/T2E3.ccc
--> 18:22
```

18		c[i][j]	sum;	␣
			^---	

```
= expected kw_bracket_open or kw_attrib
enzo@Nitro5 ~/Workspace/study/compilersConstruction/sapo master ● 1
```

Para validar o sistema (tanto para sucesso quanto para erro), executamos o build da aplicação (obtendo um tempo de 32.89 segundos) e verificamos os códigos fonte desenvolvidos para esta entrega. Os testes foram feitos em um I5 8300H (Notebook Acer).

## Referencias

[1] Pest The Elegant Parser. Disponível em: <https://github.com/pest-parser/pest>. Acesso em 13 de Agosto de 2021.