Universidade Federal de Santa Catarina - UFSC
Departamento de Informática e Estatística
Curso de Ciências da Computação
INE5426 - Construção de Compiladores

Matheus Leonel Balduino
Marcelo Pietro Grützmacher Contin

# Relatório EP2 - Analisador Sintático

Florianópolis, 19 de Janeiro de 2022

# 1 Introdução

Este relatório tem por finalidade descrever a forma como foi implementado um Analisador Sintático para a disciplina de Compiladores, no curso de Ciências da Computação da UFSC. O analisador implementado tem como alvo a gramática disponibilizada pelo professor, chamada ConvCC-2021-2, conforme enunciado do EP2.

Durante a etapa de desenvolvimento, o grupo optou por utilizar uma ferramenta na construção e geração de alguns trechos de código, facilitando e acelerando o processo. A ferramenta selecionada foi o ANTLR, versão 4.

Para realizar a implementação, JAVA foi escolhido como a linguagem de programação.

# 2 Conversão de BNF para a forma convencional

A seguir, é apresentada a gramática ConvCC-2021-2, que foi obtida a partir da conversão da gramática CC-2021-2 da forma BNF para a forma convencional. Os tokens referentes a símbolos e pontuação, tipos, keywords, expressões aritméticas e operadores relacionais estão identificados entre ‘ ‘. Logo após a gramática, é explicado o modo como foi feita a transformação do mesmo.

PROGRAM → STATEMENT | FUNCLIST | &
FUNCLIST → FUNCDEF FUNCLIST | FUNCDEF
FUNDEF → ‘def’ ‘ident’ ‘(‘ PARAMLIST ‘)’ ‘{‘ STATELIST ‘}’
PARAMLIST → ‘ident’ ( ‘,’ PARAMLISTALL | & ) | &
STATEMENT → VARDECL ‘;’ | ATRIBSTAT ‘;’ | PRINTSTAT ‘;‘ | READSTAT ‘;’ |
RETURNSTAT ‘;’ | IFSTAT | FORSTAT | ‘{‘ STATELIST ’}’ | ‘break’ ‘;’ | ‘;’
VARDECL → ( ‘int’ | ‘float’ | ‘string’ ) ‘ident’ A
A → T1 A | &
T1 → ‘[‘ ‘int_constant’ ‘]’
ATRIBSTAT → LVALUE ‘=’ (EXPRESSION | ALLOCEXPRESSION | FUNCCALL )
FUNCCALL → ‘ident’ ‘(‘ PARAMLISTCALL ‘)’
PARAMLISTCALL →’ident’ ( , PARAMLISTCALL | & ) | &
PRINTSTAT → ‘print’ EXPRESSIONREADSTAT → ‘read’ LVALUE
RETURNSTAT → ‘return’
IFSTAT → ‘if’ (EXPRESSION) STATEMENT (‘else’ STATEMENT | &)

FORSTAT → ‘for’ ‘(‘ ATRIBSTAT; EXPRESSION; ATRIBSTAT ‘)’ STATEMENT

STATELIST → STATEMENT (STATELIST | &)

ALLOCEXPRESSION → ‘new’ (‘int’ | ‘float’ | ‘string’ ) T2 B

B → T2 B | &

T2 → ‘[‘ NUMEXPRESSION ‘]’

EXPRESSION → NUMEXPRESSION ( ( ( ‘<’ | ’ >’ | ‘ <=’ | ‘ >=’ | ‘==‘ | ‘!=’ )

NUMBEXPRESSION) | & )

NUMEXPRESSION → TERM C

C → T3 C | &

T3 → ( ‘+’ | ‘-’ ) TERM

TERM → UNARYEXPR D

D → T4 D | &

T4 → ( ‘*’ | ‘ /’ | ‘%’ ) UNARYEXPR

UNARYEXPR → ( ‘+’ | ‘ -’ | &) FACTOR

FACTOR → ( ‘int_constant’ | ‘float_constant’ | ‘string_constant’ | ‘null’ | LVALUE | ‘(‘

NUMEXPRESSION ‘)’ )

LVALUE → ‘ident’ B

Para chegar nessa gramática, o grupo fez substituições referentes às produções que na gramática BNF constavam como: ***(expressão)****, ***(expressão1 | expressão2)?*** e ***(expressão)+***.

Para transformar as produções do tipo ***(expressão)**** na forma convencional, criamos um loop da seguinte forma (exemplo com parte aplicada na gramática):

1. Considere como a produção original:

   ***VARDECL → (‘int’ | ‘float’ | ‘string’ ) ‘ident’ ( ‘[‘ ‘int_constant’ ‘]’)****

2. Criamos um novo não-terminal para substituir uma parte do corpo da produção que contém o operador *, nesse caso o não-terminal A, e outro não-terminal, T1, para representar a expressão dentro do operador:

   ***VARDECL → (‘int’ | ‘float’ | ‘string’ ) ‘ident’ A***

   ***T1 → ‘[‘ ‘int_constant’ ‘]’***

3. Criamos uma nova produção para o não-terminal A para representar um loop:

   ***A → T1 A | &***

   ***T1 → ‘[‘ ‘int_constant’ ‘]’***

4. Para o operador **+** também utilizamos essa mesma técnica, mas no corpo da produção original colocamos o não-terminal T1 para obrigar uma ocorrência dele antes do loop:

   ***VARDECL → (‘int’ | ‘float’ | ‘string’ ) ‘ident’ T1  A***

5. Para o operador **?** apenas adicionamos uma nova produção para o Épsilon (**&**), como por exemplo:

   ***PROGRAM → ( STATEMENT | FUNCLIST )? ⇒***

   ***PROGRAM → STATEMENT | FUNCLIST | &***

# 3 Recursão à Esquerda

Analisando a gramática ConvCC-2021-2 na forma convencional, não foi observada nenhuma produção do tipo A $\rightarrow^1$ Aa, o qual define que não há nenhuma **recursão direta** à esquerda.

Para analisar se a gramática possui **recursão indireta** à esquerda,foi feita a análise de cada não-terminal que estivesse no início de cada produção da expressão de maneira individual. Para isso, cada produção não-terminal que se encontra à esquerda foi substituída. Tendo como exemplo a produção a seguir, a verificação de existência ou não de recursão à esquerda deu-se da seguinte forma:

***LVALUE → ‘ident’ B***

Primeiramente, é possível ter certeza de que essa produção nunca apresentará uma recursão à esquerda visto que começa com um terminal. Com isso, é possível descartar algumas outras produções que não irão apresentar recursão a esquerda, tais como:

***UNARYEXPR , T4, T3, T2, ALLOCEXPRESSION, FORSTAT, IFSTAT, RETURNSTAT, READSTAT, PRINTSTAT, PARAMLISTCALL, FUNCCALL, T1, VARDECL, PARAMLIST e FUNDEF.***

Observando todas as outras produções que começam com os não-terminais que estão presentes na lista acima, visto que eles não possuem recursão à esquerda, também não irão apresentar a recursão à esquerda. São eles:

***FACTOR, D, TERM, C, B, ATRIBSTAT, A, RETURNSTAT, FUNCLIST*.**

Mais uma vez, realizando o mesmo procedimento, agora nos utilizando de ambas as listas, temos mais dois não-terminais:

***NUMEXPRESSION*** e ***STATEMENT.***

Finalizando, temos os últimos não-terminais:

***EXPRESSION, STATELIST e PROGRAM.***

Com isso, conseguimos mostrar que a gramática ConvCC-2021-2 **não possui recursão à esquerda direta e indireta**.

# 4 Fatoração à Esquerda

Analisando a gramática ConvCC-2021-2 na forma convencional, é possível observar que a gramática **não está fatorada à esquerda**. Isso se torna visível no produtor ***FUNCLIST***, que pode gerar duas produções que começam com o ***FUNCDEF***, gerando ambiguidade.

***FUNCLIST → FUNCDEF FUNCLIST | FUNCDEF***

Podemos fatorá-la facilmente criando uma nova produção **FUNCLIST2**:

***FUNCLIST → FUNCDEF FUNCLIST2***

***FUNCLIST2→ FUNCLIST | &***

Após a fatoração completa da gramática, foi observado que a seguinte produção ficou inalcançável:

***FUNCCALL → 'ident' '(' PARAMLISTCALL ')'***

Uma vez que o não-terminal ***ATRIBSTAT2***, que continha uma produção com o ***FUNCCALL***, precisou ser fatorado novamente devido a uma fatoração indireta com o terminal 'ident', ***FUNCCALL*** acabou sendo removido do corpo das produções de ***ATRIBSTAT2*** e não aparecendo nas produções de ***ATRIBSTAT3***. As produções antigas do ***ATRIBSTAT2*** e as resultantes da fatoração do ***ATRIBSTAT2*** encontram-se logo abaixo.

Produções antigas:

***ATRIBSTAT2 → EXPRESSION | ALLOCEXPRESSION |***

***FUNCCALL***

Novas produções devido à fatoração:

***ATRIBSTAT2 → ident ATRIBSTAT3 | ALLOCEXPRESION | ...***

***ATRIBSTAT3 → B D C EXPRESSION2 | '(' PARAMLISTCALL ')'***

# 5 Conversão para LL(1)

A transformação da ConvCC-2021-2 em uma gramática LL(1) apenas necessitou alterar a seguinte produção:

***IFSTAT → 'if' '(' EXPRESSION ')' STATEMENT IFSTAT2***

Para a seguinte:

***IFSTAT → 'if' '(' EXPRESSION ')' '{' STATELIST '}' IFSTAT2***

Essa mudança foi necessária porque a tabela de reconhecimento sintático estava apresentando um conflito na entrada ***[IFSTAT2, else]***, uma vez que a interseção de ***FOLLOW(IFSTAT2)*** com ***FIRST('else' STATEMENT)*** estava dando ***{'else'}***, sendo que o conjunto ***FIRST*** da segunda produção de ***IFSTAT2*** continha ε.

Para resolver esse problema, foi trocado o ***STATEMENT*** para ***'{' STATELIST '}'*** no corpo da produção para retirar o ***'else'*** do ***FOLLOW(IFSTAT2)*** e colocar o terminal ***'{'*** no lugar, tornando a interseção vazia. A tabela que abrange o ***FIRST*** e o ***FOLLOW*** se encontra a seguir.

| X | FIRST(X) | FOLLOW(X) |
|---|---|---|
| PROGRAM | {int, float, string, ident, print, read, return, if, for, '{', break, ';', def, &} | {$} |
| FUNCLIST | {def} | {$} |
| FUNCLIST2 | {def, &} | {$} |
| FUNCDEF | {def} | {def, $} |
| PARAMLIST | {int, float, string, &} | { ')' } |
| PARAMLIST2 | {',', &} | { ')' } |
| STATEMENT | {int, float, string, ident, print, read, return, if, for, '{', break, ';'} | {$, int, float, string, ident, print, read, return, if, for, '{', break, ';', '}'} |
| VARDECL | {int, float, string} | {';'} |
| A | {'[', &} | {';'} |
| T1 | {'['} | { '[', ';'} |
| ATRIBSTAT | {ident} | {';', ')'} |
| ATRIBSTAT2 | {ident, new, '+', '-', int_constant, float_constant, string_constant, null, '('} | {';', ')'} |
| ATRIBSTAT3 | {'[', '*', '/', '%', '+', '-', '<', '>', '<=', '>=', '==', '!=', '(', &} | {';', ')'} |
| PARAMLISTCALL | {ident, &} | { ')' } |
| PARAMLISTCALL2 | {',', &} | { ')' } |
| PRINTSTAT | {print} | {';'} |

| | | |
|---|---|---|
| READSTAT | {read} | {‘;’} |
| RETURNSTAT | {return} | {‘;’} |
| IFSTAT | {if} | {$, int, float, string, ident, print, read, return, if, for, ‘{‘, break, ‘;’, ‘}’} |
| IFSTAT2 | {else, &} | {$, int, float, string, ident, print, read, return, if, for, ‘{‘, break, ‘;’, ‘}’} |
| FORSTAT | {for} | {$, int, float, string, ident, print, read, return, if, for, ‘{‘, break, ‘;’,’}’} |
| STATELIST | {int, float, string, ident, print, read, return, if, for, ‘{‘, break, ‘;’} | { ‘}’ } |
| STATELIST2 | {int, float, string, ident, print, read, return, if, for, ‘{‘, break, ‘;’ , &} | { ‘}’ } |
| ALLOCEXPRESSION | {new} | {‘;’, ‘)’} |
| ALLOCEXPRESSION2 | {int, float, string} | {‘;’, ‘)’} |
| B | {‘[‘, &} | {‘=’, ‘*’, ‘/’, ‘%’, ‘+’, ‘-’, ‘<’, ‘>’, ‘<=’, ‘>=’, ‘==’, ‘!=’, ‘;’, ‘)’, ‘]’} |
| T2 | {‘[‘} | {‘=’, ‘*’, ‘/’, ‘%’, ‘+’, ‘-’, ‘<’, ‘>’, ‘<=’, ‘>=’, ‘==’, ‘!=’, ‘;’, ‘)’, ‘]’, ‘[‘} |
| EXPRESSION | {‘+’, ‘-’, int_constant, float_constant, string_constant, null, ident, ‘(‘} | {‘;’, ‘)’} |
| EXPRESSION2 | {‘<’, ‘>’, ‘<=’, ‘>=’, ‘==’, ‘!=’, &} | {‘;’, ‘)’} |
| NUMEXPRESSION | {‘+’, ‘-’, int_constant, float_constant, string_constant, null, ident, ‘(‘} | {‘;’, ‘)’, ‘]’, ‘<’, ‘>’, ‘<=’, ‘>’=, ‘==’, ‘!=’} |
| C | {‘+’, ‘-’, &} | {‘<’, ‘>’, ‘<=’, ‘>=’, ‘==’, ‘!=’, ‘;’, ‘)’, ‘]’} |
| T3 | {‘+’, ‘-’} | {‘+’, ‘-’, ‘<’, ‘>’, ‘<=’, ‘>=’, ‘==’, ‘!=’, ‘;’, ‘)’, ‘]’} |
| TERM | {‘+’, ‘-’, int_constant, | {‘+’, ‘-’, ‘<’, ‘>’, ‘<=’, ‘>=’, ‘==’, |

| | | |
|---|---|---|
| | float_constant, string_constant, null, ident, ‘(‘} | ‘!=’, ‘;’, ‘)’, ‘]’} |
| D | {‘*’, ‘/’, ‘%’, &} | {‘+’, ‘-’, ‘<’, ‘>’, ‘<=’, ‘>=’, ‘==’, ‘!=’, ‘;’, ‘)’, ‘]’} |
| T4 | {‘*’, ‘/’, ‘%’} | {‘*’, ‘/’, ‘%’, ‘+’, ‘-’, ‘<’, ‘>’, ‘<=’, ‘>=’, ‘==’, ‘!=’, ‘;’, ‘)’, ‘]’} |
| UNARYEXPR | {‘+’, ‘-’, int_constant, float_constant, string_constant, null, ident, ‘(‘} | {‘*’, ‘/’, ‘%’, ‘+’, ‘-’, ‘<’, ‘>’, ‘<=’, ‘>=’, ‘==’, ‘!=’, ‘;’, ‘)’, ‘]’} |
| FACTOR | {int_constant, float_constant, string_constant, null, ident, ‘(‘} | {‘*’, ‘/’, ‘%’, ‘+’, ‘-’, ‘<’, ‘>’, ‘<=’, ‘>=’, ‘==’, ‘!=’, ‘;’, ‘)’, ‘]’} |
| LVALUE | {ident} | {‘=’, ‘*’, ‘/’, ‘%’, ‘+’, ‘-’, ‘<’, ‘>’, ‘<=’, ‘>=’, ‘==’, ‘!=’, ‘;’, ‘)’, ‘]’} |

E a tabela de reconhecimento sintático encontra-se na Tabela.

## 6 Gramática final

PROGRAM → STATEMENT | FUNCLIST | &

FUNCLIST → FUNCDEF FUNCLIST2

FUNCLIST2 → FUNCLIST | &

FUNCDEF → ‘def’ ‘ident’ ‘(‘ PARAMLIST ‘)’ ‘{‘ STATELIST ‘}’

PARAMLIST → ‘int’ ‘ident’ PARAMLIST2 | ‘float’ ‘ident’ PARAMLIST2 | ‘string’ ‘ident’ PARAMLIST2 | &

PARAMLIST2 → ‘,’ PARAMLIST | &

STATEMENT → VARDECL ‘;’ | ATRIBSTAT ‘;’ | PRINTSTAT ‘;’ | READSTAT ‘;’ | RETURNSTAT ‘;’ | IFSTAT | FORSTAT | ‘{‘ STATELIST ‘}’ | break ‘;’ | ‘;’

VARDECL → ‘int’ ‘ident’ A | ‘float’ ‘ident’ A | ‘string’ ‘ident’ A

A → T1 A | &

T1 → ‘[‘ ‘int_constant’ ‘]’

ATRIBSTAT → LVALUE ‘=’ ATRIBSTAT2

ATRIBSTAT2→ ‘ident’ ATRIBSTAT3 | ALLOCEXPRESSION | ‘+’ FACTOR | ‘-’ FACTOR | ‘int_constant’ | ‘float_constant’ | ‘string_constant’ | ‘null’ | ‘(‘ NUMEXPRESSION ‘)’

ATRIBSTAT3 → B D C EXPRESSION2 | ‘(‘ PARAMLISTCALL ‘)’

PARAMLISTCALL → ‘ident’ PARAMLISTCALL2 | &
PARAMLISTCALL2 → ‘,’ PARAMLISTCALL | &
PRINTSTAT → ‘print’ EXPRESSION
READSTAT → ‘read’ LVALUERETURNSTAT → ‘return’
IFSTAT → ‘if’ ‘(‘ EXPRESSION ‘)’ ‘{‘ STATELIST ‘}’ IFSTAT2
IFSTAT2→ ‘else’ STATEMENT | &
FORSTAT → ‘for’ ‘(‘ ATRIBSTAT ‘;’ EXPRESSION ‘;’ ATRIBSTAT ‘)’ STATEMENT
STATELIST → STATEMENT STATELIST2
STATELIST2→ STATELIST | &
ALLOCEXPRESSION → ‘new’ ALLOCEXPRESSION2
ALLOCEXPRESSION2 → ‘int’ T2 B | ‘float’ T2 B | ‘string’ T2 B
B → T2 B | &
T2 → ‘[‘ NUMEXPRESSION ‘]’
EXPRESSION → NUMEXPRESSION EXPRESSION2
EXPRESSION2 → ‘<’ NUMEXPRESSION | ‘>’ NUMEXPRESSION | ‘<=’
NUMEXPRESSION | ‘>=’ NUMEXPRESSION | ‘==’ NUMEXPRESSION | ‘!=’
NUMEXPRESSION | &
NUMEXPRESSION → TERM C
C → T3 C | &
T3 → ‘+’ TERM | ‘-’ TERM
TERM → UNARYEXPR D
D → T4 D | &
T4 → ‘*’ UNARYEXPR | ‘/’ UNARYEXPR | ‘%’ UNARYEXPR
UNARYEXPR → ‘+’ FACTOR | ‘-’ FACTOR | FACTOR
FACTOR → ‘int_constant’ | ‘float_constant’ | ‘string_constant’ | ‘null’ | LVALUE | ‘(‘
NUMEXPRESSION ‘)’
LVALUE → ‘ident’ B

# 7 A Ferramenta

A implementação do Analisador Léxico, como já comentado, foi feita através de um ferramenta de geração de código, chamada ANTLR (versão 4), o qual foi usado também no trabalho anterior.

ANTLR possui integração com JAVA, que foi a nossa linguagem de programação escolhida, portanto isso é um ponto que influenciou na nossa escolha. Outro fator, é a

familiaridade de alguns membros da equipe com a ferramenta e principalmente a linguagem Java.

## 7.1 Descrição da entrada da ferramenta

ANTLR espera que o usuário entregue um arquivo de extensão “.g4” como entrada. Esse arquivo é o responsável por definir todas as regras da gramática a ser utilizada. A sintaxe é bem simples, sendo basicamente a definição do nome da regra, seguida pela definição e um ponto-e-vírgula identificando o fim daquela regra. Exemplo:

NOMEREGRA: <definicao>;

Nos arquivos enviados junto com este relatório, é possível visualizar a aplicação da sintaxe na prática. O mesmo se encontra no arquivo src/main/antlr4/ConvCC20212.g4.

O mesmo após a criação do arquivo gera diversos outros arquivos dentro da pasta target, o qual são utilizados posteriormente na aplicação para fazer o parser, identificação, geração da lista de tokens e verificação de erro léxico.

## 7.2 Descrição da saída da ferramenta

Ao executar o ANTLR, com os arquivos exigidos (no caso, o arquivo .g4), temos como saída uma classe (gerada automaticamente) que irá se parecer com “<*NomeArquivo.g4*>*Lexer.java*”. Nesta classe, estão os métodos que definem o analisador sintático.

Com um arquivo de código-fonte da linguagem representada pelo arquivo .g4, é possível fazer a identificação dos tokens presentes nesse arquivo. É possível, também, a identificação de erros que podem existir no código-fonte lido como já mencionado anteriormente.