Construção de Compiladores - INE5426

Relatório - AL 2

Construção de Analisador Sintático

Universidade Federal de Santa Catarina

Eduardo Gutterres [17200439]

Felipe de Campos Santos [17200441]

Ricardo Giuliani [17203922]

PROGRAM → (STATEMENT | FUNCLIST)?

FUNCLIST → FUNCDEF FUNCLIST | FUNCDEF

FUNCDEF → def ident(PARAMLIST){STATELIST}

PARAMLIST → (( int | float | string ) ident, PARAMLIST | ( int | float | string ) ident)?

STATEMENT → (VARDECL; | ATRIBST AT; | PRINTSTAT; | READSTAT; | RETURNSTAT; | IFSTAT | FORSTAT | {STATELIST} | break ; | ;)

VARDECL → ( int | float | string ) ident ([int constant])∗

ATRIBST AT → 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) ([ NUMEXP RESSION ])+

EXPRESSION → NUMEXPRESSION (( < | > | <= | >= | == | ! =) NUMEXPRESSION)?

NUMEXPRESSION → TERM ((+ |−) TERM)∗

TERM → UNARYEXPR(( ∗ | / | %) UNARYEXPR)∗

UNARYEX R → ((+ |−))? FACTOR

FACTOR → (int constant | float constant | string constant | null || LVALUE |(NUMEXPRESSION))

LVALUE → ident([NUMEXPRESSION])∗

Lembrando que:

o símbolo \* significa zero ou mais ocorrências;

o símbolo + significa uma ou mais ocorências;

o símbolo ? significa zero ou uma ocorrência;

# Colocando a gramática na forma convencional

PROGRAM → STATEMENT | FUNCLIST | &

FUNCLIST → FUNCDEF FUNCLIST | FUNCDEF

FUNCDEF → def ident(PARAMLIST){STATELIST}

PARAMLIST → int PARAMLIST’ | float PARAMLIST’ | string PARAMLIST’ | &

PARAMLIST’ → ident, PARAMLIST | ident

STATEMENT → VARDECL; | ATRIBSTAT; | PRINTSTAT; | READSTAT; |

RETURSTAT; | IFSTAT; | FORSTAT; | {STATELIST} | break; | ;

VARDECL → int VARDECL’ | float VARDECL’ | string VARDECL’

VARDECL’ → ident VARDECL’’

VARDECL’’ → [int\_constant]VARDECL’’ | &

ATRIBSTAT → LVALUE = RIGHT\_ATRIB

RIGHT\_ATRIB → EXPRESSION | ALLOCEXPRESSION | FUNCCALL

FUNCCALL → ident(PARAMLISTCALL)

PARAMLISTCALL → ident, PARAMLISTCALL | ident | &

PRINTSTAT → print EXPRESSION

READSTAT → read LVALUE

RETURNSTAT → return

IFSTAT → if(EXPRESSION) STATEMENT ELSESTAT

ELSESTAT → else STATEMENT | &

FORSTAT → for(ATRIBSTAT; EXPRESSION; ATRIBSTAT) STATEMENT

STATELIST → STATEMENT | STATEMENT STATELIST

ALLOCEXPRESSION → new ALLOCEXPRESSION’

ALLOCEXPRESSION’ → int OPT\_ALLOC\_EXPR | float OPT\_ALLOC\_EXPR |

string OPT\_ALLOC\_EXPR

OPT\_ALLOC\_EXPR → [NUMEXPRESSION] |

[NUMEXPRESSION] OPT\_ALLOC\_EXPR

EXPRESSION → NUMEXPRESSION | NUMEXPRESSION CMP NUMEXPRESSION

CMP → < | > | <= | >= | == | !=

NUMEXPRESSION → TERM NUMXPRESSION’

NUMEXPRESSION’ → + TERM NUMEXPRESSION’ | - TERM NUMEXPRESSION’ |

NUMEXPRESSION’ | &

TERM → UNARYEXPR OPT\_UNARY\_TERM

OPT\_UNARY\_TERM → \* UNARYEXPR | / UNARYEXPR | % UNARYEXPR | OPT\_UNARY\_TERM | &

UNARYEXPR → + FACTOR | - FACTOR

FACTOR → int\_constant | float\_constant | string\_constant | null | LVALUE |

(NUMEXPRESSION)

LVALUE → ident LVALUE’

LVALUE’ → [NUMEXPRESSION] | LVALUE’ | &

obs: grifado em amarelo, vemos os terminais dessa gramática. Esses destaques foram omitidos no resto do relatório.

# Recursão à esquerda

A gramática descrita acima, que será chamada de **ConvCC-2021-2** não possui recursão a esquerda, visto que para nenhum não-terminal A existe

A =\*> Ac

sendo c qualquer outro terminal ou não terminal.

# Fatoração à esquerda

A falta de fatoração à esquerda ocorre quando existe duas ou mais produções possíveis, que tem um começo em comum, para um mesmo não terminal A:

A => cB | cC

Nesse caso não, nossa gramática não está fatorada (ou seja, existem produções do tipo acima).

Nossa gramática fatorada:

PROGRAM → STATEMENT | FUNCLIST | &

FUNCLIST → FUNCDEF FUNCLIST2

FUNCLIST2 → FUNCLIST | &

FUNCDEF → def ident(PARAMLIST){STATELIST}

PARAMLIST → int PARAMLIST’ | float PARAMLIST’ | string PARAMLIST’ | &

PARAMLIST’ → ident PARAMLIST2

PARAMLIST2 → , PARAMLIST | &

STATEMENT → VARDECL; | ATRIBSTAT; | PRINTSTAT; | READSTAT; | RETURSTAT; | IFSTAT; | FORSTAT; | {STATELIST} | break; | ;

VARDECL → int VARDECL’ | float VARDECL’ | string VARDECL’

VARDECL’ → ident ARRAY\_OPT

ARRAY\_OPT → [int\_constant]ARRAY\_OPT | &

ATRIBSTAT → LVALUE = RIGHT\_ATRIB

RIGHT\_ATRIB → EXPRESSION | ALLOCEXPRESSION | FUNCCALL

FUNCCALL → ident(PARAMLISTCALL)

PARAMLISTCALL → ident PARAMLISTCALL2

PARAMLISTCALL2 → ,PARAMLISTCALL | &

PRINTSTAT → print EXPRESSION

READSTAT → read LVALUE

RETURNSTAT → return

IFSTAT → if(EXPRESSION) STATEMENT ELSESTAT

ELSESTAT → else STATEMENT | &

FORSTAT → for(ATRIBSTAT; EXPRESSION; ATRIBSTAT) STATEMENT

STATELIST → STATEMENT OPT\_STATELIST

OPT\_STATELIST → STATELIST | &

ALLOCEXPRESSION → new ALLOCEXPRESSION’

ALLOCEXPRESSION’ → int OPT\_ALLOC\_EXPR | float OPT\_ALLOC\_EXPR | string OPT\_ALLOC\_EXPR

OPT\_ALLOC\_EXPR → [NUMEXPRESSION] ALLOCEXPRESSION2

ALLOCEXPRESSION2 → OPT\_ALLOC\_EXPR | &

EXPRESSION → NUMEXPRESSION EXPRESSION2

EXPRESSION2 → CMP NUMEXPRESSION | &

CMP → < | > | <= | >= | == | !=

NUMEXPRESSION → TERM NUMEXPRESSION’

NUMEXPRESSION’ → + TERM NUMEXPRESSION’ | - TERM NUMEXPRESSION’ | &

TERM → UNARYEXPR OPT\_UNARY\_TERM

OPT\_UNARY\_TERM → \* UNARYEXPR | / UNARYEXPR | % UNARYEXPR | &

UNARYEXPR → + FACTOR | - FACTOR | FACTOR

FACTOR → int\_constant | float\_constant | string\_constant | null | LVALUE | (NUMEXPRESSION)

LVALUE → ident LVALUE’

LVALUE’ → [NUMEXPRESSION] LVALUE' | &

# LL(1)

Pelo teorema, temos que uma Gramática está em LL(1) se e somente se, para toda regra de produção, neste caso determinada por A -> B | C, temos:

* First(B) ∩ First(C) = Ø
* Se C -> Ɛ, então First(B) ∩ Follow(A) = Ø
* Se B -> Ɛ, então First(C) ∩ Follow(A) = Ø

Analisando a tabela de Firsts e Follows abaixo, podemos observar que todas estas condições são satisfeitas para a gramática descrita na seção anterior. Portanto, a gramática está em LL(1).

|  | **First** | **Follow** |
| --- | --- | --- |
| PROGRAM | Ɛ, break, def, return, float, if, ident, int, for, print, read, string, {, ; | $ |
| FUNCLIST | def | $ |
| FUNCLIST2 | Ɛ, def | $ |
| FUNCDEF | def | $, def |
| PARAMLIST | Ɛ, string, float, int | ) |
| PARAMLIST’ | ident | ) |
| PARAMLIST2 | Ɛ, , | ) |
| STATEMENT | print, break, read, string, return, float, if, ident, {, ;, int, for | $, break, return, float, if, else, ident, int, for, print, read, string, ;, {, } |
| VARDECL | string, float, int | ; |
| VARDECL’ | ident | ; |
| ARRAY\_OPT | Ɛ, [ | ; |
| ATRIBSTAT | ident | ), ; |
| RIGHT\_ATRIB | int\_constant, new, float\_constant, (, ident, string\_constant, +, null, - | ), ; |
| FUNCCALL | ident | ), ; |
| PARAMLISTCALL | ident | ) |
| PARAMLISTCALL2 | Ɛ, , | ) |
| PRINTSTAT | print | ; |
| READSTAT | read | ; |
| RETURNSTAT | return | ; |
| IFSTAT | if | ; |
| ELSESTAT | Ɛ, else | ; |
| FORSTAT | for | ; |
| STATELIST | print, break, read, string, return, float, if, ident, {, ;, int, for | } |
| OPT\_STATELIST | Ɛ, break, return, float, if, ident, int, for, print, read, string, {, ; | } |
| ALLOCEXPRESSION | new | ), ; |
| ALLOCEXPRESSION’ | string, float, int | ), ; |
| OPT\_ALLOC\_EXPR | [ | ), ; |
| ALLOCEXPRESSION2 | Ɛ, [ | ), ; |
| EXPRESSION | int\_constant, float\_constant, (, string\_constant, ident, +, null, - | ), ; |
| EXPRESSION2 | Ɛ, >=, !=, <, <=, ==, > | ), ; |
| CMP | >=, !=, <, <=, ==, > | int\_constant, float\_constant, (, string\_constant, ident, +, null, - |
| NUMEXPRESSION | int\_constant, float\_constant, (, string\_constant, ident, +, null, - | >=, !=, <, ), <=, ;, ==, ], > |
| NUMEXPRESSION’ | Ɛ, +, - | >=, !=, <, ), <=, ;, ==, ], > |
| TERM | int\_constant, float\_constant, (, string\_constant, ident, +, null, - | >=, !=, <, ), <=, ;, ==, +, ], >, - |
| OPT\_UNARY\_TERM | Ɛ, %, \*, / | >=, !=, <, ), <=, ;, ==, +, ], >, - |
| UNARYEXPR | int\_constant, float\_constant, (, string\_constant, ident, +, null, - | >=, %, ), <=, \*, ==, +, >, -, /, !=, <, ;, ] |
| FACTOR | int\_constant, float\_constant, (, string\_constant, ident, null | >=, %, ), <=, \*, ==, +, >, -, /, !=, <, ;, ] |
| LVALUE | ident | >=, %, ), <=, \*, ==, +, >, -, /, !=, <, ;, ], = |
| LVALUE’ | Ɛ, [ | >=, %, ), <=, \*, ==, +, >, -, /, !=, <, ;, ], = |

# Descrição da ferramenta

Assim como na primeira entrega, nessa usamos a ferramenta PLY, porém extendida para permitir que a análise sintática seja feita.

Além disso, implementamos também ferramentas de geração dos conjuntos first e follow, e um verificador LL(1), que serão descritos neste documento.

### src/main.py

Principal arquivo do programa, nele que são rodadas as análises

### src/CC2021

Pasta que contém os arquivos que gerenciam as análises necessárias

### src/CC2021/lexer

Criação e definição da classe Lexer, que usa do lex.py para criar o analisador léxico da gramática (assim como feito na última entrega)

### src/CC2021/parser

Criação e definição da classe Parser, que também usa da biblioteca da ply (lex) para fazer o parsing do cédigo passado, usando a gramática como base.

Usa da tabela de análise sintatica criada pelo Processor de LLC para aplicar as produções

### src/CC2021/ply

Biblioteca disponibilizada por Dabeaz (<https://www.dabeaz.com/ply/>)

### src/CC2021/LLC

Pasta que gerencia, cria e organiza o processador e o parser de gramáticas LLC (como a passada na pasta src/utils/grammar/cc2021.grammar , que é resultado das operações feitas acima no relatório com base na gramética de estudo passada pelo professor)

É na classe Processor (src/CC2021/LLC/processor.py) que é realizada a checagem para ver se a gramatica base passada é LL(1):

A classe é criada e chamada em read\_llc, que recebe o caminho do arquivo .grammar. Esse caminho é passado para o Parser (src/CC2021/LLC/parser.py) que lê o arquivo .grammar linha por linha criando a gramática LLC definida nele.

O resultado do parse (que é uma LLC) é passado para a função create\_llc da classe Processor. Nela, são calculados os conjuntos first e follow usando como base o pseudocódigo visto em aula.

Feito isso, precisamos da tabela de analise sintática para realizar a analise sintática. Quando a função create\_table é chamada no parser de LLC, antes de iniciar a geração da tabela, o programa usa do teorema passado em aula, aqui divido em suas duas ‘checagens’ (ll\_first\_condition e ll\_second\_condition), para garantir de que a gramatica LLC é LL(1)

### Criação da tabela

A função create\_table usa da estrutura de dados TableSyntaticAnalyser, definida em src/CC2021/strucs, para criar a tabela de analise sintatica. Em sua função \_\_init\_\_, que recebe os terminais e não terminais da gramatica, a classe faz a criação da estrutura que virá a ser a tabela, inicializando um dict com espaços vazios.

Feito isso, ainda na função create\_table, percorremos as produções da LLC adicionando essas produções à tabela no formato ‘esta cabeça, por esta produção, gera este produto’.

# Alterações

Abaixo, as alterações feitas na gramática:

* Foi adicionado valores TRUE e FALSE como terminais
* Foram adicionados tokens de comentário (ignorados na análise)
* No statement FOR, foi adicionado a necessidade de chaves na declaração do escopo (passa a ser for{...}). Por esse motivo, o exemplo1.lcc disponibilizado pelo professor falha na análise sintática. Este mesmo exemplo está corrigido na entrega.
* Em return, foi adicionado a possibilidade de ser seguido por um nove de variavel, possibilitando o programa retornar essa variavel.

# Link para o repositório do Github:

<https://github.com/felipecampossantos/AL1_INE5426>

## Links Úteis

<https://www.dabeaz.com/ply/>

<https://johnidm.gitbooks.io/compiladores-para-humanos/content/part1/lexical-analysis.html>

<https://www.dabeaz.com/ply/PLYTalk.pdf>

<https://sites.google.com/site/2012pcs25086482782/home/o-analisador-lexico>

<https://earthly.dev/blog/python-makefile/>

<https://www.dfki.de/compling/pdfs/cfg-slides.pdf>