

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; | 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 | null || LVALUE
 |(NUMEXPRESSION))
 LVALUE → ident([NUMEXPRESSION])*

Lembrando que:

- o símbolo * significa zero ou mais ocorrências;
- o símbolo + significa uma ou mais ocorrências;
- o símbolo ? significa zero ou uma ocorrência;

1. 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 NUMEXPRESSION'
NUMEXPRESSION' → + TERM NUMEXPRESSION' | - TERM NUMEXPRESSION' |
 NUMEXPRESSION' | &
TERM → UNARYEXPR OPT_UNARY_TERM
OPT_UNARY_TERM → * UNARYEXPR | / UNARYEXPR | % UNARYEXPR |
 &
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.

2. 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

$$A \Rightarrow^* Ac$$

sendo c qualquer outro terminal ou não terminal.

3. 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 :

$$A \Rightarrow cB \mid cC$$

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; | RETURNSTAT; |
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 \rightarrow NUMEXPRESSION EXPRESSION2
 EXPRESSION2 \rightarrow CMP NUMEXPRESSION | &
 CMP \rightarrow < | > | <= | >= | == | !=
 NUMEXPRESSION \rightarrow TERM NUMEXPRESSION'
 NUMEXPRESSION' \rightarrow + TERM NUMEXPRESSION' | - TERM NUMEXPRESSION' | &
 TERM \rightarrow UNARYEXPR OPT_UNARY_TERM
 OPT_UNARY_TERM \rightarrow * UNARYEXPR | / UNARYEXPR | % UNARYEXPR | &
 UNARYEXPR \rightarrow + FACTOR | - FACTOR | FACTOR
 FACTOR \rightarrow int_constant | float_constant | string_constant | null | LVALUE |
 (NUMEXPRESSION)
 LVALUE \rightarrow ident LVALUE'
 LVALUE' \rightarrow [NUMEXPRESSION] LVALUE' | &

4.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 $A \rightarrow B \mid C$, temos:

- $\text{First}(B) \cap \text{First}(C) = \emptyset$
- Se $C \rightarrow \epsilon$, então $\text{First}(B) \cap \text{Follow}(A) = \emptyset$
- Se $B \rightarrow \epsilon$, então $\text{First}(C) \cap \text{Follow}(A) = \emptyset$

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	ϵ , break, def, return, float, if, ident, int, for, print, read, string, {, ;	\$
FUNCLIST	def	\$
FUNCLIST2	ϵ , def	\$
FUNCDEF	def	\$, def
PARAMLIST	ϵ , string, float, int)
PARAMLIST'	ident)
PARAMLIST2	ϵ , ,)
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	ϵ , [;
ATRIIBSTAT	ident), ;
RIGHT_ATRIB	int_constant, new, float_constant, (, ident, string_constant, +, null, -), ;
FUNCCALL	ident), ;
PARAMLISTCALL	ident)
PARAMLISTCALL2	ϵ , ,)
PRINTSTAT	print	;
READSTAT	read	;
RETURNSTAT	return	;
IFSTAT	if	;
ELSESTAT	ϵ , else	;
FORSTAT	for	;
STATELIST	print, break, read, string, return, float, if, ident, {, ;, int, for	}
OPT_STATELIST	ϵ , break, return, float, if, ident, int, for, print, read, string, {, ;	}
ALLOCEXPRESSION	new), ;
ALLOCEXPRESSION'	string, float, int), ;
OPT_ALLOC_EXPR	[), ;
ALLOCEXPRESSION2	ϵ , [), ;
EXPRESSION	int_constant, float_constant, (, string_constant, ident, +, null, -), ;
EXPRESSION2	ϵ , >=, !=, <, <=, ==, >), ;
CMP	>=, !=, <, <=, ==, >	int_constant, float_constant, (, string_constant, ident, +, null, -

NUMEXPRESSION	int_constant, float_constant, (, string_constant, ident, +, null, -	>=, !=, <,), <=, :, ==,], >
NUMEXPRESSION'	ϵ , +, -	>=, !=, <,), <=, :, ==,], >
TERM	int_constant, float_constant, (, string_constant, ident, +, null, -	>=, !=, <,), <=, :, ==, +,], >, -
OPT_UNARY_TERM	ϵ , %, *, /	>=, !=, <,), <=, :, ==, +,], >, -
UNARYEXPR	int_constant, float_constant, (, string_constant, ident, +, null, -	>=, %,), <=, *, ==, +, >, -, /, !=, <, :,]
FACTOR	int_constant, float_constant, (, string_constant, ident, null	>=, %,), <=, *, ==, +, >, -, /, !=, <, :,]
LVALUE	ident	>=, %,), <=, *, ==, +, >, -, /, !=, <, :,], =
LVALUE'	ϵ , [>=, %,), <=, *, ==, +, >, -, /, !=, <, :,], =

5. Descrição da ferramenta

Assim como na primeira entrega, nessa usamos a ferramenta PLY, porém estendida 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 código passado, usando a gramática como base.

Usa da tabela de análise sintática criada pelo Processador 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/utlis/grammar/cc2021.grammar , que é resultado das operações feitas acima no relatório com base na gramática de estudo passada pelo professor)

É na classe Proc (src/CC2021/LLC/proc.py) que é realizada a checagem para ver se a gramática 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 Proc. Nela, são calculados os conjuntos first e follow usando como base o pseudocódigo visto em aula.

Feito isso, precisamos da tabela de análise sintática para realizar a análise 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 dividido em suas duas 'checagens' (ll_first_condition e ll_second_condition), para garantir de que a gramática 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 análise sintática. Em sua função __init__, que recebe os terminais e não terminais da gramática, 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'.

6. 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 variável, possibilitando o programa retornar essa variável.

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>