UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE CIÊNCIAS DA COMPUTAÇÃO

Trabalho 2 - Analisador Sintático

Matheus Schaly (18200436)
Ramna Sidharta (16100742)
Letícia do Nascimento (16104595)

Florianópolis, 26 de março de 2021

Introdução

A análise sintática, ou "parsing", é a segunda etapa de um processo de compilação de um código fonte, escrito em linguagem de alto nível, para linguagem de montagem ou linguagem de máquina. Nesta etapa faz-se uso do que foi produzido pela etapa anterior (análise léxica) para garantir que ou verificar se o programa de entrada é bem formado, isto é, as estruturas de suas declarações de funções, blocos iterativos, blocos lógicos, operações e afins está correta. O parsing produz uma árvore de derivação, que representa a relação sintática entre os tokens do programa.

Este documento descreve como foi implementado um parser para a linguagem AL.

Questões

1) CC-2020-1 está na forma BNF. Coloque-a na forma convencional de gramática. Chame tal gramática de ConvCC-2020-1.

```
PROGRAM -> STATEMENT | FUNCLIST | epsi
FUNCLIST -> FUNCDEF FUNCLIST | FUNCDEF
FUNCDEF -> def ident(PARAMLIST) {STATELIST}
PARAMLIST -> int ident, PARAMLIST | float ident, PARAMLIST | string ident,
PARAMLIST | int ident | float ident | string ident | epsi
STATEMENT -> VARDECL; | ATRIBSTAT; | PRINTSTAT; | READSTAT; | RETURNSTAT; | IFSTAT
| FORSTAT | {STATELIST} | break; | ;
VARDECL -> int ident INT_CONST_LIST | float ident INT_CONST_LIST | string ident
INT_CONST_LIST
INT_CONST_LIST -> [int_constant]INT_CONST_LIST | epsi
ATRIBSTAT -> LVALUE = EXPRESSION | LVALUE = ALLOCEXPRESSION | LVALUE = FUNCCALL
FUNCCALL -> ident(PARAMLISTCALL)
PARAMLISTCALL -> ident, PARAMLISTCALL | ident | epsi
PRINTSTAT -> print EXPRESSION
READSTAT -> read LVALUE
RETURNSTAT -> return
IFSTAT -> if(EXPRESSION) STATEMENT | if(EXPRESSION) STATEMENT else STATEMENT
FORSTAT -> for(ATRIBSTAT; EXPRESSION; ATRIBSTAT) STATEMENT
STATELIST -> STATEMENT | STATEMENT STATELIST
ALLOCEXPRESSION -> new int NUM_EXPR_LIST | new float NUM_EXPR_LIST | new string
NUM_EXPR_LIST
NUM_EXPR_LIST -> [NUMEXPRESSION] NUM_EXPR_LIST | [NUMEXPRESSION]
EXPRESSION -> NUMEXPRESSION | NUMEXPRESSION < NUMEXPRESSION | NUMEXPRESSION >
```

```
NUMEXPRESSION | NUMEXPRESSION <= NUMEXPRESSION | NUMEXPRESSION >= NUMEXPRESSION |
NUMEXPRESSION == NUMEXPRESSION | NUMEXPRESSION != NUMEXPRESSION
NUMEXPRESSION -> TERM | NUMEXPRESSION + TERM | NUMEXPRESSION - TERM | epsi
TERM -> UNARYEXPR | UNARYEXPR * TERM | UNARYEXPR / TERM | UNARYEXPR % TERM
UNARYEXPR -> + FACTOR | - FACTOR | FACTOR
FACTOR -> int_constant | float_constant | string_constant | null | LVALUE |
(NUMEXPRESSION)
LVALUE -> ident NUM_EXPR_LIST | ident
```

2) A sua ConvCC-2020-1 possui recursão à esquerda? Justifique detalhadamente sua resposta. Se ela tiver recursão à esquerda, então remova tal recursão.

Possui apenas uma recursão à esquerda, a qual é uma recursão direta em duas produções partindo do símbolo NUMEXPRESSION:

```
NUMEXPRESSION -> NUMEXPRESSION + TERM | NUMEXPRESSION - TERM
```

Para remover estas recursões, modificamos a regra acima e adicionamos um novo símbolo NUMEXPRESSION_LINE:

```
NUMEXPRESSION -> TERM NUMEXPRESSION_LINE
NUMEXPRESSION_LINE -> + TERM NUMEXPRESSION_LINE | - TERM
NUMEXPRESSION_LINE | epsi
```

Há diversas vezes em que um símbolo não terminal X aparece sendo o resultado de uma derivação, sendo uma possível recursão indireta, contudo, em todos os casos a sequência β de símbolos que aparece antes de X (e.g. A' -> βXα) são derivadas para ao menos um símbolo terminal. A exemplo, temos

```
STATELIST -> STATEMENT | STATEMENT STATELIST
STATEMENT -> VARDECL; | ATRIBSTAT; | PRINTSTAT; | READSTAT; |
RETURNSTAT; | IFSTAT | FORSTAT | {STATELIST} | break; | ;
```

Nota-se que STATELIST é produzido indiretamente, mas junto a um símbolo terminal {.

3) A sua ConvCC-2020-1 está fatorada à esquerda? Justifique detalhadamente sua resposta. Se ela não estiver fatorada à esquerda, então fatore.

A ConvCC-2020-1 não estava fatorada à esquerda. Nota-se vários não-determinismos diretos e um não-determinismo indireto. A única produção que apresentou não-determinismo indireto foi a ATRIBSTAT_. Para tal produção, foi primeiramente realizado a transformação, através de derivações sucessivas, do não-determinismo indireto para o não-determinismo direto e posteriormente eliminou-se o não-determinismo direto. As derivações sucessivas consistem em substituir os não-terminais por suas produções.

```
FUNCLIST -> FUNCDEF FUNCLIST_
FUNCLIST_ -> FUNCLIST | epsi

PARAMLIST -> int ident PARAMLIST_ | float ident PARAMLIST_ | string
ident PARAMLIST_ | epsi
PARAMLIST_ -> epsi | , PARAMLIST

ATRIBSTAT -> LVALUE = ATRIBSTAT_
ATRIBSTAT_ -> ALLOCEXPRESSION | int_constant TERM_ NUMEXPRESSION_
EXPRESSION_ | float_constant TERM_ NUMEXPRESSION_ EXPRESSION_ |
string_constant TERM_ NUMEXPRESSION_ EXPRESSION_ | null TERM_
NUMEXPRESSION_ EXPRESSION_ | ( NUMEXPRESSION ) TERM_ NUMEXPRESSION_
EXPRESSION_ | + FACTOR TERM_ NUMEXPRESSION_ EXPRESSION_ | - FACTOR
TERM_ NUMEXPRESSION_ EXPRESSION_ | ident ATRIBSTAT__
ATRIBSTAT__ -> ( PARAMLISTCALL ) | LVALUE_ TERM_ NUMEXPRESSION_
EXPRESSION_

PARAMLISTCALL -> ident PARAMLISTCALL_ | epsi
PARAMLISTCALL_ -> , PARAMLISTCALL | epsi

IFSTAT -> if (EXPRESSION) STATEMENT IFSTAT_
IFSTAT_ -> epsi | else STATEMENT

STATELIST -> STATEMENT STATELIST_
STATELIST_ -> epsi | STATELIST
```

```
ALLOCEXPRESSION -> new ALLOCEXPRESSION_ ALLOCEXPRESSION_ -> int
NUM_EXPR_LIST | float NUM_EXPR_LIST | string NUM_EXPR_LIST

NUM_EXPR_LIST -> [NUMEXPRESSION] NUM_EXPR_LIST_
NUM_EXPR_LIST_ -> NUM_EXPR_LIST | epsi

EXPRESSION -> NUMEXPRESSION EXPRESSION_
EXPRESSION_ -> < EXPRESSION_LTE_GTE_ | > EXPRESSION_LTE_GTE_ | ==
NUMEXPRESSION | != NUMEXPRESSION
EXPRESSION_LTE_GTE -> NUMEXPRESSION | = NUMEXPRESSION

TERM -> UNARYEXPR TERM_
TERM_ -> epsi | * TERM | / TERM | % TERM

LVALUE -> ident LVALUE_
LVALUE_ -> epsi | NUM_EXPR_LIST
```

Sendo assim, a gramática completa é:

```
~~PROGRAM -> STATEMENT | FUNCLIST | epsi~~
~~FUNCLIST -> FUNCDEF FUNCLIST_~~
~~FUNCLIST_ -> FUNCLIST | epsi~~
FUNCDEF -> def ident(PARAMLIST) {STATELIST}
PARAMLIST -> int ident PARAMLIST_ | float ident PARAMLIST_ | string ident
PARAMLIST_ | epsi
PARAMLIST_ -> epsi | , PARAMLIST
STATEMENT -> VARDECL; | ATRIBSTAT; | ~~PRINTSTAT;~~ | ~~READSTAT;~~ | ~~RETURNSTAT;~~ |
IFSTAT | FORSTAT | {STATELIST} | break; | ;
VARDECL -> int ident INT_CONST_LIST | float ident FLOAT_CONST_LIST | string
ident string_constant
INT_CONST_LIST -> int_constant INT_CONST_LIST | epsi
FLOAT_CONST_LIST -> float_constant FLOAT_CONST_LIST | epsi
ATRIBSTAT -> LVALUE = ATRIBSTAT_
ATRIBSTAT_ -> ALLOCEXPRESSION ~~| int_constant TERM_ NUMEXPRESSION_~~
EXPRESSION_ | float_constant TERM_ NUMEXPRESSION_ EXPRESSION_ |
string_constant TERM_ NUMEXPRESSION_ EXPRESSION_ | null TERM_
NUMEXPRESSION_ EXPRESSION_ | ( NUMEXPRESSION ) TERM_ NUMEXPRESSION_
EXPRESSION_ | + FACTOR TERM_ NUMEXPRESSION_ EXPRESSION_ | - FACTOR TERM_
NUMEXPRESSION_ EXPRESSION_ | ident ATRIBSTAT__
```

```
ATRIBSTAT__ -> ( PARAMLISTCALL ) | LVALUE_ TERM_ NUMEXPRESSION_ EXPRESSION_
FUNCCALL -> ident(PARAMLISTCALL)
PARAMLISTCALL -> identPARAMLISTCALL_ | epsi
PARAMLISTCALL_ -> , PARAMLISTCALL | epsi
~~PRINTSTAT -> print EXPRESSION~~
~~READSTAT -> read LVALUE~~
RETURNSTAT -> return IDENT
IFSTAT -> if (EXPRESSION) STATEMENT IFSTAT_
IFSTAT_ -> epsi | else STATEMENT
FORSTAT -> for(ATRIBSTAT; EXPRESSION; ATRIBSTAT) STATEMENT
STATELIST -> STATEMENT STATELIST_
STATELIST_ -> epsi | STATELIST
ALLOCEXPRESSION -> new ALLOCEXPRESSION_
ALLOCEXPRESSION_ -> int NUM_EXPR_LIST | float NUM_EXPR_LIST | string
NUM_EXPR_LIST -> [NUMEXPRESSION] NUM_EXPR_LIST_
NUM_EXPR_LIST_ -> NUM_EXPR_LIST | epsi
EXPRESSION -> NUMEXPRESSION EXPRESSION_
EXPRESSION_ -> < EXPRESSION_LTE_GTE_ | > EXPRESSION_LTE_GTE_ | ==
NUMEXPRESSION | != NUMEXPRESSION | epsi
EXPRESSION_LTE_GTE -> NUMEXPRESSION | = NUMEXPRESSION
NUMEXPRESSION -> TERM NUMEXPRESSION_
NUMEXPRESSION_ -> + TERM NUMEXPRESSION_ | - TERM NUMEXPRESSION_ | epsi
~~TERM -> UNARYEXPR TERM_~~
~~TERM_ -> epsi | * TERM | / TERM | % TERM~~
UNARYEXPR -> + FACTOR | - FACTOR | ~~FACTOR~~
FACTOR -> ~~int_constant | float_constant | string_constant~~ | ~~null~~ | LVALUE |
(NUMEXPRESSION)
LVALUE -> ident LVALUE_
LVALUE_ -> ~~epsi~~ | NUM_EXPR_LIST
```

4) Transforme CF em uma gramática LL(1). É permitido adicionar novos terminais conforme julgar necessário. Depois disso, mostre que CF está em LL(1) (você pode usar o Teorema ou a tabela de reconhecimento sintático vistos em videoaula).

Para transformar a CF em uma gramática LL(1) removemos a recursão à esquerda e realizamos a fatoração à esquerda, como mostrado acima. Não foi necessário adicionar novos terminais.

A princípio, para mostrar que a CF está em LL(1), implementamos um programa para provar através do teorema. Entretanto, optamos em provar utilizando a ferramenta http://jsmachines.sourceforge.net/machines/ll1.html pois tal ferramenta mostra, de forma organizada, a gramática final, os firsts dos não-terminais (sabe-se que o first de um terminal é o próprio terminal), os follows dos não-terminais e a tabela de reconhecimento. A tabela pode ser encontrada no arquivo *tabela_de_reconhecimento.png*. Percebe-se que a gramática está em LL(1) pois cada uma das células da tabela possui no máximo uma única produção.

5) Se não usou ferramenta, uma descrição da implementação do analisador sintático (usou uma tabela de reconhecimento sintático para gramáticas em LL(1)? Se não, então o que foi usado?)

Utilizou-se apenas uma biblioteca Python, o PLY, já apresentado e utilizado na entrega anterior. Anteriormente o projeto dependia apenas do módulo Lex (lex.py), desta vez adicionou-se o Yacc (yacc.py). O Yacc é a ferramenta de parsing do PLY, e seu nome é emprestado da ferramenta UNIX Yacc, que significa "yet another compiler compiler" ou, em português, algo como "mais um compilador de compilador". Seu uso é análogo ao do Lex, explicado no relatório anterior: para a utilização daquele componente, é necessário definir uma função (ou variável global para casos mais simples), para cada token da linguagem sendo definida; no caso do Yacc, é necessário haver uma função para cada regra de produção. A seguir isso é explicado detalhadamente.

6) Se usou ferramenta, uma descrição da entrada exigida pela ferramenta e da saída dada por ela.

Entrada:

```
def max(int a, int b) {
  if (a >= b) {
      return a;
  }
  return b;
}
```

Saída: (não está completa pois é muito grande)

```
PLY: PARSE DEBUG START

State  : 0
Stack  : . LexToken(DEF,'def',1,0)
Action : Shift and goto state 18

State  : 18
Stack  : DEF . LexToken(IDENT,'max',1,4)
Action : Shift and goto state 33

State  : 33
Stack  : DEF IDENT . LexToken(LPAREN,'(',1,8)
Action : Shift and goto state 53

State  : 53
Stack  : DEF IDENT LPAREN . LexToken(FLOAT,'float',1,10)
Action : Shift and goto state 65

State  : 65
Stack  : DEF IDENT LPAREN FLOAT . LexToken(IDENT,'a',1,16)
Action : Shift and goto state 76

State  : 76
Stack  : DEF IDENT LPAREN FLOAT IDENT . LexToken(COMMA,',',1,17)
Action : Shift and goto state 81

State  : 81
Stack  : DEF IDENT LPAREN FLOAT IDENT COMMA . LexToken(INT,'int',1,19)
Action : Shift and goto state 64

State  : 64
Stack  : DEF IDENT LPAREN FLOAT IDENT COMMA INT . LexToken(IDENT,'b',1,23)
Action : Shift and goto state 75
```

Regras da gramática (algumas de um total de 50)

Rule 0 S' -> program

Rule 1 program -> statement

Rule 2 program -> funclist

Rule 3 funclist -> funcdef funclist_

Rule 4 funclist_ -> funclist
Rule 5 funclist_ -> <empty>
Rule 6 funcdef -> DEF IDENT LPAREN paramlist RPAREN LCBRACKET statelist RCBRACKET
Rule 7 paramlist -> INT IDENT paramlist_
Rule 8 paramlist -> FLOAT IDENT paramlist_
Rule 9 paramlist -> STRING IDENT paramlist_
Rule 10 paramlist -> <empty>
Rule 11 paramlist_ -> COMMA paramlist
Rule 12 paramlist_ -> <empty>
Rule 13 statement -> vardecl SEMICOLON
Rule 14 statement -> atribstat SEMICOLON
Rule 15 statement -> returnstat SEMICOLON
Rule 16 statement -> ifstat
Rule 17 statement -> LCBRACKET statelist RCBRACKET
Rule 18 statement -> SEMICOLON
Rule 19 vardecl -> INT IDENT int_const_list
Rule 20 vardecl -> FLOAT IDENT float_const_list
Rule 21 vardecl -> STRING IDENT STRING_CONSTANT
Rule 22 int_const_list -> INT_CONSTANT int_const_list
Rule 23 int_const_list -> <empty>
Rule 24 float_const_list -> FLOAT_CONSTANT float_const_list
Rule 25 float_const_list -> <empty>
Rule 26 atribstat -> lvalue ASSIGN atribstat_
Rule 27 atribstat_ -> INT_CONSTANT
Rule 28 term -> unaryexpr term_
Rule 29 term_ -> MULTIPLY term
Rule 30 term_ -> DIVIDE term
Rule 31 term_ -> MOD term
Rule 32 term_ -> <empty>
Rule 33 unaryexpr -> factor
Rule 34 returnstat -> RETURN IDENT

Estados: (apenas 3 para exemplo)

state 0

(0) S' -> . program
(1) program -> . statement
(2) program -> . funclist
(13) statement -> . vardecl SEMICOLON
(14) statement -> . atribstat SEMICOLON
(15) statement -> . returnstat SEMICOLON
(16) statement -> . ifstat
(17) statement -> . LCBRACKET statelist RCBRACKET
(18) statement -> . SEMICOLON
(3) funclist -> . funcdef funclist_
(19) vardecl -> . INT IDENT int_const_list
(20) vardecl -> . FLOAT IDENT float_const_list
(21) vardecl -> . STRING IDENT STRING_CONSTANT
(26) atribstat -> . lvalue ASSIGN atribstat_
(34) returnstat -> . RETURN IDENT
(35) returnstat -> . RETURN
(36) ifstat -> . IF LPAREN expression RPAREN statement
(6) funcdef -> . DEF IDENT LPAREN paramlist RPAREN LCBRACKET statelist RCBRACKET
(44) lvalue -> . IDENT lvalue_

LCBRACKET shift and go to state 9
SEMICOLON shift and go to state 5
INT shift and go to state 11
FLOAT shift and go to state 13
STRING shift and go to state 14
RETURN shift and go to state 16
IF shift and go to state 17
DEF shift and go to state 18
IDENT shift and go to state 12

program shift and go to state 1
statement shift and go to state 2
funclist shift and go to state 3
vardecl shift and go to state 4

```
  atribstat                shift and go to state 6
  returnstat               shift and go to state 7
  ifstat                   shift and go to state 8
  funcdef                  shift and go to state 10
  lvalue                   shift and go to state 15

state 1
  (0) S' -> program .

state 2
  (1) program -> statement .
  $end          reduce using rule 1 (program -> statement .)
```

Terminais, seguindo das regras que eles aparecem:

```
ASSIGN          : 26 41
COMMA           : 11
DEF             : 6
DIVIDE          : 30
FLOAT           : 8 20
FLOAT_CONSTANT      : 24 47
GT              : 40
IDENT           : 6 7 8 9 19 20 21 34 44
IF              : 36
INT             : 7 19
INT_CONSTANT        : 22 27 46
LCBRACKET       : 6 17
LPAREN          : 6 36
MOD             : 31
MULTIPLY        : 29
NULL            : 50
RCBRACKET       : 6 17
RETURN          : 34 35
RPAREN          : 6 36
```

SEMICOLON : 13 14 15 18
STRING : 9 21
STRING_CONSTANT : 21 48
error :

Não-terminais, com a regra da qual eles aparecem:

atribstat : 14
atribstat_ : 26
expression : 36
expression_ : 39
expression_lte_gte : 40
factor : 33
float_const_list : 20 24
funcdef : 3
funclist : 2 4
funclist_ : 3
ifstat : 16
int_const_list : 19 22
lvalue : 26 49
lvalue_ : 44
numexpression : 39 41 42
paramlist : 6 11
paramlist_ : 7 8 9
program : 0
returnstat : 15
statelist : 6 17 38
statement : 1 36 37 38
term : 29 30 31 43
term_ : 28
unaryexpr : 28
vardecl : 13

Observação

- A tabela de símbolos implementada no arquivo lexer.py (chamado “main.py” na entrega anterior) registrava apenas a última ocorrência de um identificador, e registrava outros tokens além de identificadores, como indicado pelo professor. Isso foi corrigido.
- A execução do projeto gera um arquivo “parser.out” (dentro da pasta “vlant”), que mostra organizadamente todas as regras da gramática, bem como todos os estados possíveis de parsing. As subseções “estado” e “regras” acima mostram parte do conteúdo deste arquivo.
- O PLY também gera as parsing tables do algoritmo, o resultado disso aparece no arquivo “vlant/parsetab.py”, também gerado com a execução do projeto.
- A saída da execução mostra todos os passos do parsing pois a opção debug está ativada. Ela pode ser desativada através do parâmetro booleano da chamada parser.parse(), em main.py.
- O parâmetro “tracking”, naquela mesma linha (parser.parse(..., tracking=True)) ativa o tracking de número de linha e coluna dos tokens sendo processados.