Relatório do Trabalho 2 - Construção de Compiladores - INE5426
Grupo: David Grunheidt Vilela Ordine - 16202253

Modificações na linguagem:

Foram alteradas as seguintes produções:

PARAMLIST -> ((int | float | string) ident ([])?, PARAMLIST | ((int | float | string) ident ([])?
RETURNSTAT -> return (ident)?

1. Forma convencional da gramática CC-2020-1 (ConvCC-2020-1):

PROGRAM -> STATEMENT | FUNCLIST | ϵ
FUNCLIST -> FUNCDEF FUNCLIST | FUNCDEF
FUNCDEF -> def ident(PARAMLIST){STATELIST}
PARAMLIST -> VARTYPE ident, PARAMLIST | VARTYPE ident
PARAMLIST -> VARTYPE ident [], PARAMLIST | VARTYPE ident []
STATEMENT -> VARDECL; | ATRIBSTAT; | PRINTSTAT; | READSTAT; | RETURNSTAT; | IFSTAT | FORSTAT
| {STATELIST} | break ; | ;
VARTYPE -> int | float | string
ARRAYSIZE -> [int_constant] ARRAYSIZE | [int_constant]
ARRAYSIZEEXP -> [NUMEXPRESSION] ARRAYSIZEEXP | [NUMEXPRESSION]
VARDECL -> VARTYPE ident ARRAYSIZE | VARTYPE ident
ATRIBSTAT -> LVALUE = EXPRESSION | LVALUE = ALLOCEXPRESSION | LVALUE = FUNCCALL
FUNCCALL -> ident(PARAMLISTCALL)
PARAMLISTCALL -> ident, PARAMLISTCALL | ident | ϵ
PRINTSTAT -> print EXPRESSION
READSTAT -> read LVALUE
RETURNSTAT -> return ident | return
IFSTAT -> if(EXPRESSION) STATEMENT | if(EXPRESSION) STATEMENT else STATEMENT
FORSTAT -> for(ATRIBSTAT; EXPRESSION; ATRIBSTAT) STATEMENT
STATELIST -> STATEMENT STATELIST | STATEMENT
ALLOCEXPRESSION -> new VARTYPE ARRAYSIZEEXP
LOGICALCOMP -> < | > | <= | >= | == | !=
EXPRESSION -> NUMEXPRESSION LOGICALCOMP NUMEXPRESSION | NUMEXPRESSION
NUMOPSUM -> + | -
NUMOPMULT -> * | / | %
NUMEXPRESSION -> TERM NUMEXPRESSION'
NUMEXPRESSION' -> NUMOPSUM TERM NUMEXPRESSION' | ϵ
TERM -> UNARYEXPR TERM'
TERM' -> NUMOPMULT UNARYEXPR TERM' | ϵ
UNARYEXPR -> FACTOR | NUMOPSUM FACTOR
FACTOR -> int_constant | float_constant | string_constant | null | LVALUE | (NUMEXPRESSION)
LVALUE -> ident ARRAYSIZEEXP | ident

Para melhor entendimento, aqui estão as produções adicionadas e alteradas para transformar a gramática de BNF para a forma convencional:

Foram adicionadas as seguintes produções:
VARTYPE -> int | float | string
ARRAYSIZE -> [int_constant] ARRAYSIZE | [int_constant]
ARRAYSIZEEXP -> [NUMEXPRESSION] ARRAYSIZEEXP | [NUMEXPRESSION]
LOGICALCOMP -> < | > | <= | >= | == | !=
NUMOPSUM -> + | -
NUMOPMULT -> * | / | %

A criação das produções acima resultou na alteração das seguintes produções:
PARAMLIST -> (VARTYPE ident ([])?, PARAMLIST | VARYPE ident ([])?)
VARDECL -> VARTYPE ident ARRAYSIZE | VARTYPE ident
ALLOCEXPRESSION -> new VARTYPE ARRAYSIZEEXP
EXPRESSION -> NUMEXPRESSION LOGICALCOMP NUMEXPRESSION | NUMEXPRESSION
NUMEXPRESSION -> TERM (NUMOPSUM TERM)*
TERM -> UNARYEXPR(NUMOPMULT UNARYEXPR)*
UNARYEXPR -> FACTOR | NUMOPSUM FACTOR
LVALUE -> ident ARRAYSIZEEXP | ident

2. Quanto a possível recursão a esquerda da gramática ConvCC-2020-1 :

A gramática ConvCC-2020-1 não é recursiva à esquerda, pois não possui recursão a esquerda direta ou indireta. Para todo não terminal A, não existe uma produção do tipo A -> Aα, ou seja, não existe recursão a esquerda direta e não existe também uma derivação A ->+ Aα, ou seja, não existe recursão a esquerda indireta do tipo A -> Sα e S -> Aβ.

3. Quanto a fatoração à esquerda da gramática ConvCC-2020-1:

A gramática ConvCC-2020-1 não está fatorada á esquerda, pois existem 2 ou mais alternativas de produção para um certo não terminal A (A -> αB1 | αB2). As produções da gramática ConvCC-2020-1 com mais de uma alternativa são:

FUNCLIST -> FUNCDEF FUNCLIST | FUNCDEF
PARAMLIST -> VARTYPE ident, PARAMLIST | VARTYPE ident
PARAMLIST -> VARTYPE ident [], PARAMLIST | VARTYPE ident []
ARRAYSIZE -> [int_constant] ARRAYSIZE | [int_constant]
ARRAYSIZEEXP -> [NUMEXPRESSION] ARRAYSIZEEXP | [NUMEXPRESSION]
VARDECL -> VARTYPE ident ARRAYSIZE | VARTYPE ident
ATRIBSTAT -> LVALUE = EXPRESSION | LVALUE = ALLOCEXPRESSION | LVALUE = FUNCCALL
PARAMLISTCALL -> ident, PARAMLISTCALL | ident | ϵ
RETURNSTAT -> return ident | return
IFSTAT -> if(EXPRESSION) STATEMENT | if(EXPRESSION) STATEMENT else STATEMENT
STATELIST -> STATEMENT STATELIST | STATEMENT
EXPRESSION -> NUMEXPRESSION LOGICALCOMP NUMEXPRESSION | NUMEXPRESSION
LVALUE -> ident ARRAYSIZEEXP | ident

Alterou-se as produções acima para produzir a nova gramática ConvCC-2020-1 fatorada à esquerda, a qual segue abaixo:

PROGRAM -> STATEMENT | FUNCLIST | ϵ
FUNCLIST -> FUNCDEF FUNCLIST'
FUNCLIST' -> FUNCLIST | ϵ
FUNCDEF -> def ident(PARAMLIST){STATELIST}
PARAMLIST -> VARTYPE PARAMLIST'
PARAMLIST' -> ident PARAMLIST''
PARAMLIST'' -> , PARAMLIST | [PARAMLIST''' | ϵ
PARAMLIST''' ->] PARAMLIST''''
PARAMLIST'''' -> , PARAMLIST | ϵ
STATEMENT -> VARDECL; | ATRIBSTAT; | PRINTSTAT; | READSTAT; | RETURNSTAT; | IFSTAT | FORSTAT | {STATELIST} | break ; | ;
VARTYPE -> int | float | string
ARRAYSIZE -> [int_constant] ARRAYSIZE'
ARRAYSIZE' -> ARRAYSIZE | ϵ
ARRAYSIZEEXP -> [NUMEXPRESSION] ARRAYSIZEEXP'

ARRAYSIZEEXP' -> ARRAYSIZEEXP | ϵ
VARDECL -> VARTYPE ident VARDECL'
VARDECL' -> ARRAYSIZE | ϵ
ATRIBSTAT -> LVALUE = ATRIBSTAT'
ATRIBSTAT' -> EXPRESSION | ALLOCEXPRESSION | FUNCCALL
FUNCCALL -> ident(PARAMLISTCALL)
PARAMLISTCALL -> ident PARAMLISTCALL' | ϵ
PARAMLISTCALL' -> , PARAMLISTCALL | ϵ
PRINTSTAT -> print EXPRESSION
READSTAT -> read LVALUE
RETURNSTAT -> return RETURNSTAT'
RETURNSTAT' -> ident | ϵ
IFSTAT -> if(EXPRESSION) STATEMENT IFSTAT'
IFSTAT' -> else STATEMENT | ϵ
FORSTAT -> for(ATRIBSTAT; EXPRESSION; ATRIBSTAT) STATEMENT
STATELIST -> STATEMENT STATELIST'
STATELIST' -> STATELIST | ϵ
ALLOCEXPRESSION -> new VARTYPE ARRAYSIZEEXP
LOGICALCOMP -> < | > | <= | >= | == | !=
EXPRESSION -> NUMEXPRESSION EXPRESSION'
EXPRESSION' -> LOGICALCOMP NUMEXPRESSION | ϵ
NUMOPSUM -> + | -
NUMOPMULT -> * | / | %
NUMEXPRESSION -> TERM NUMEXPRESSION'
NUMEXPRESSION' -> NUMOPSUM TERM NUMEXPRESSION' | ϵ
TERM -> UNARYEXPR TERM'
TERM' -> NUMOPMULT UNARYEXPR TERM' | ϵ
UNARYEXPR -> FACTOR | NUMOPSUM FACTOR
FACTOR -> int_constant | float_constant | string_constant | null | LVALUE | (NUMEXPRESSION)
LVALUE -> ident LVALUE'
LVALUE' -> ARRAYSIZEEXP | ϵ

4. Conversão de ConvCC-2020-1 para gramatica LL(1):

Atualmente, a gramática fatorada a esquerda da seção 3 não é LL(1).

Na produção ATRIBSTAT' -> EXPRESSION | ALLOCEXPRESSION | FUNCCALL, temos que First(EXPRESSION) ∩ First(FUNCALL) != Ø, pois First(EXPRESSION) = {int_constant, float_constant, string_constant, null, (, +, -, ident} e First(FUNCALL) = {ident}, assim First(EXPRESSION) ∩ First(FUNCALL) = {ident}. Isso não satisfaz as condições 1) do teorema que prova que uma gramática está em LL(1) (Pra toda produção A -> α | β, First(α) ∩ First(β) = Ø).

Para solucionar esse problema, basta adicionar o novo terminal ident_func na gramática e alterar as seguintes produções:
FUNCDEF -> def ident_func(PARAMLIST){STATELIST}
FUNCCALL -> ident_func(PARAMLISTCALL)
RETURNSTAT' -> ident | ident_func | ϵ

Já as produções IFSTAT -> if(EXPRESSION) STATEMENT IFSTAT' e IFSTAT' -> else STATEMENT | ϵ resultam em ambiguidade na linguagem. Assim, amarramos um if sempre a um else com a seguinte produção:
IFSTAT -> if(EXPRESSION) STATEMENT else STATEMENT

A nova gramática ConvCC-2020-1 em LL(1) é a seguinte:

PROGRAM -> STATEMENT | FUNCLIST | ϵ
FUNCLIST -> FUNCDEF FUNCLIST'
FUNCLIST' -> FUNCLIST | ϵ
FUNCDEF -> def ident_func(PARAMLIST){STATELIST}
PARAMLIST -> VARTYPE PARAMLIST'
PARAMLIST' -> ident PARAMLIST''
PARAMLIST'' -> , PARAMLIST | [PARAMLIST''' | ϵ
PARAMLIST''' ->] PARAMLIST''''
PARAMLIST'''' -> , PARAMLIST | ϵ
STATEMENT -> VARDECL; | ATRIBSTAT; | PRINTSTAT; | READSTAT; | RETURNSTAT; | IFSTAT | FORSTAT
| {STATELIST} | break ; | ;
VARTYPE -> int | float | string
ARRAYSIZE -> [int_constant] ARRAYSIZE'
ARRAYSIZE' -> ARRAYSIZE | ϵ
ARRAYSIZEEXP -> [NUMEXPRESSION] ARRAYSIZEEXP'
ARRAYSIZEEXP' -> ARRAYSIZEEXP | ϵ
VARDECL -> VARTYPE ident VARDECL'
VARDECL' -> ARRAYSIZE | ϵ
ATRIBSTAT -> LVALUE = ATRIBSTAT'
ATRIBSTAT' -> EXPRESSION | ALLOCEXPRESSION | FUNCCALL
FUNCCALL -> ident_func(PARAMLISTCALL)
PARAMLISTCALL -> ident PARAMLISTCALL' | ϵ
PARAMLISTCALL' -> , PARAMLISTCALL | ϵ
PRINTSTAT -> print EXPRESSION
READSTAT -> read LVALUE
RETURNSTAT -> return RETURNSTAT'
RETURNSTAT' -> ident | ident_func | ϵ
IFSTAT -> if(EXPRESSION) STATEMENT else STATEMENT
FORSTAT -> for(ATRIBSTAT; EXPRESSION; ATRIBSTAT) STATEMENT
STATELIST -> STATEMENT STATELIST'
STATELIST' -> STATELIST | ϵ
ALLOCEXPRESSION -> new VARTYPE ARRAYSIZEEXP
LOGICALCOMP -> < | > | <= | >= | == | !=
EXPRESSION -> NUMEXPRESSION EXPRESSION'
EXPRESSION' -> LOGICALCOMP NUMEXPRESSION | ϵ
NUMOPSUM -> + | -
NUMOPMULT -> * | / | %
NUMEXPRESSION -> TERM NUMEXPRESSION'
NUMEXPRESSION' -> NUMOPSUM TERM NUMEXPRESSION' | ϵ
TERM -> UNARYEXPR TERM'
TERM' -> NUMOPMULT UNARYEXPR TERM' | ϵ
UNARYEXPR -> FACTOR | NUMOPSUM FACTOR
FACTOR -> int_constant | float_constant | string_constant | null | LVALUE | (NUMEXPRESSION)
LVALUE -> ident LVALUE'
LVALUE' -> ARRAYSIZEEXP | ϵ

A tabela de reconhecimento sintático segue em anexo com o nome "RelatorioA2TabelaReconhecimento" prova que a gramática ConvCC-2020-1 está em LL(1), pois temos no máximo uma produção para todas as suas entradas. Abrir a tabela com o excel ou planilhas do google (Alguns símbolos não terminais começam com o ' no título da coluna pois o google planilhas mostra um erro caso não coloque. Ex: = virou '= , mas na implementação do analisador está tudo certo…)

5. Descrição da implementação do analisador sintático:

O analisador sintático usa a bibliotéca PLY pra python, a qual é responsável por gerar os tokens. Essa mesma biblioteca foi usada no analisador léxico e detalhada no relatório anterior. A biblioteca só foi usada para a geração de tokens.

A implementação do analisador sintático foi feita em python.

Para isso, transformou-se a tabela de reconhecimento sintático em uma arquivo .csv. Após isso, na implementação em python3, abriu-se esse arquivo, transformando-o em um dicionário do tipo [string, [dict, string], ou seja, ao passar um símbolo não terminal como primeira chave e um símbolo terminal como segunda chave, temos a próxima produção a ser empilhada.

Ex: dict[PROGRAM, def] retorna FUNCLIST.
A disposição da tabela de reconhecimento sintático em forma de dicionário pode ser vista no arquivo json_parsin_table.

O algoritmo começa transformando a tabela de reconhecimento sintático em um dicionário. Após isso,

Após isso, ele utiliza a biblioteca PLY para gerar todos os tokens.

Na sequência, ele inicia a pilha com o símbolo $ na base e o símbolo PROGRAM no topo.

Até que o símbolo desempilhando seja igual a um símbolo terminal, o programa desempilha e empilha a produção que está na parsingTable[variável desempilhada][token atual]. Quando o que foi desempilhado é um símbolo terminal, o programa avança para o próximo token.

Infelizmente os programas escritos para a gramática estão dando erros sintáticos, e, mesmo debugando toda a semana, não consegui achar o motivo disso.