Construção de Compiladores - INE5426

Relatório - AS 3

Construção de Analisador Semântico e Gerador de Código Intermediário

Universidade Federal de Santa Catarina

Eduardo Gutterres [17200439]

Felipe de Campos Santos [17200441]

Ricardo Giuliani [17203922]

Gramática de estudo

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.

1. Construção da árvore de expressão

Como pedido no enunciado, separamos as produções que derivam expressões aritméticas de **CC-2021-2**, e criamos EXPA:

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' | &

Com isso, pudemos criar a SDD L-Atribuída para possibilitar a criação da nossa árvore de expressões

SDDs são L-Atribuídas?

A garantia de que uma SDD seja L-Atribuída vem da confirmação de que em suas produções não hajam *ciclos de dependência*, ou seja, apenas os atributos *sintetizados ou herdados* da produção a esquerda podem ser usados, assim garantimos que esses atributos herdados vem sempre de uma "direção" (assim evitando ciclos). Já o uso dos atributos *sintetizados* evita que hajam ciclos entre pais e filhos.

Definicao da SDD EXPA

Aqui mostraremos apenas uma parte da definicao, pois ela completa é muito extensa e se encontra no arquivo yacc_builder.py (src>CC2021>semantic) no formato utilizado pelo yacc.

```
numexp:
  producao:
    NUMEXPRESSION : TERM OPT_ARITHM

  regras:
    se:
      OPT_ARITHM é vazio, entao NUMEXPRESSION.value = TERM.value

    se nao:
        NUMEXPRESSION.tipo = checkIfIsValid(TERM.value, OPT_ARITHM.value,
OPT_ARITHM.operacao)
```

NUMEXPRESSION.value = novo_nodo(OPT_ARITHM.operacao, NUMEXPRESSION.tipo, TERM.value, OPT_ARITHM.value)

rec_plus_minus:
producao:
OPT_ARITHM[1] : ARITHM TERM OPT_ARITHM[2] | empty

regras:
se a producao tem menos de 3 items:
OPT_ARITHM[1].value = none

se nao:
se OPT_ARITHM[2] != vazio:
OPT_ARITHM[1].tipo = checkIfIsValid(TERM.value,OPT_ARITHM[2].value,OPT_ARITHM[2].operacao)
OPT_ARITHM[1].operacao = ARITHM.operacao
OPT_ARITHM[1].value = novo_nodo(OPT_ARITHM[2].operacao, OPT_ARITHM[1].tipo, TERM.value, OPT_ARITHM.value)

se nao: (OPT_ARITHM[2] == vazio, nao tem mais recursao)
OPT_ARITHM[1].value = novo_nodo(TERM.value)
OPT_ARITHM[1].operacao = ARITHM.operacao

plus:
producao:
ARITHM : PLUS | MINUS

regras:
ARITHM.operacao = p[1].valor (p[1] sendo o PLUS ou o MINUS)

Demonstração do formato yacc

```
def p_numexp(p: yacc.YaccProduction):                  -> define a funcao que checa numexp
    """NUMEXPRESSION : TERM OPT_ARITHM"""              -> define a producao
    if p[2] is None:                                   -> p[2] -> OPT_ARITHM
        p[0] = p[1]                                    -> p[0] -> NUMEXPRESSION ; p[1] -> TERM

    else:                                              -> caso tenha OPT_ARITHM
        result_type = checkIfIsValid(p[1]['node'],     -> checa se a operacao é valida
                            p[2]['node'],              -> -
                            p[2]['operation'],         -> -
                            p.lineno(1))               -> -
        p[0] = {                                       -> atribui a NUMEXPRESSION:
            'node': Node(                              -> um novo nodo com:
                        p[2]['operation'],             -> a operacao de OPT_ARITHM
                         result_type,                  -> o tipo do resultado da expressao
                        p[1]['node'],                  -> o valor de TERM
                         p[2]['node'],                 -> e o valor de OPT_ARITHM (resultado)
                         )
        }
```

2. Inserção de tipos na tabela de símbolos

Separadas asproduções que derivam declarações de variáveis de **CC-2021-2**, e criamos DEC:

FUNCDEF → def ident(PARAMLIST){STATELIST}
PARAMLIST → int PARAMLIST' | float PARAMLIST' | string PARAMLIST' | &
PARAMLIST' → ident, PARAMLIST | ident
VARDECL → int VARDECL' | float VARDECL' | string VARDECL'
VARDECL' → ident VARDECL''
VARDECL'' → [int_constant]VARDECL'' | &
ARRAY_OPT → [NUMEXPRESSION] |
[NUMEXPRESSION] ARRAY_OPT
TYPE : INT | FLOAT | STRING

Definicao da SDD DEC

Aqui, vale a pena ressaltar o uso da struc ScopeEntry, que define (cria e insere) uma nova entrada na tabela de escopos. Sua criação é feita passando o nome do novo escopo (nome da função, por exemplo), seu tipo (function, loop, etc), tamanho (no caso de array ou matriz) e a linha que foi declarada.

funcdef:
produção:
FUNCDEF : DEF IDENT new_scope LPARENTHESES PARAMLIST RPARENTHESES LEFTBRACE STATELIST RIGHTBRACE
regras:
scope_list.getLastScope() -> retorna ao escopo "pai"
scope = scope_list.getLastScopeOrNoneIfEmpty() -> pega o ultimo escopo declarado (ou vazio)
scopeEntry = ScopeEntry(IDENT.value, 'function', [], p.lineno(2)) -> cria um novo escopo chamado de "IDENT.value", que é uma função e esta na linha declarada
scope.addToScopeTable(scopeEntry) -> adiciona esse escopo criado ao ultimo escopo declarado

paralist_param:
produção:
PARAMLIST : TYPE IDENT PARAMLIST2
| empty
regras:

se existem mais de dois elementos na produção:
scope = scope_list.getLastScopeOrNoneIfEmpty()
scopeEntry = ScopeEntry(IDENT.value, TYPE.value, [], p.lineno(2))
PARAMLIST.sin = scopeEntry
scope.addToScopeTable(scopeEntry)

type:
produção:
TYPE : INT
| FLOAT
| STRING
regras:
TYPE.value = (...).value (seja int, float ou string)

vardecl:
produção:
VARDECL : TYPE IDENT ARRAY_OPT
regras:
scopeEntry = ScopeEntry(IDENT.value, TYPE.value, ARRAY_OPT.value, p.lineno(2))
VARDECL.sin = scopeEntry
scope = scope_list.getLastScopeOrNoneIfEmpty()
scope.addToScopeTable(scopeEntry)

opt_vector:
produção:
ARRAY_OPT : LBRACKET INTCONSTANT RBRACKET ARRAY_OPT
| empty
regras:
se tem mais de 2 elementos na produção:
ARRAY_OPT.sin = nova_lista(INTCONSTANT.value, ARRAY_OPT.value) ->
nova_lista(dimensão1, outras_dimensões [caso de matriz])
else:
ARRAY_OPT.sin = vazio

3. Verificação de tipos

Para a verificação de tipos, criamos uma função e uma estrutura auxiliares.
A estrutura se encontra em src>utils, e se chama *validOperationResults.* Nela, temos um dicionario de tuplas, no formato:

(*tipo1, operacao, tipo2): tipo3*

Significando que, uma operação *operação* entre *tipo1* e *tipo2* retorna um *tipo3.*
Assim podemos não só pegar o tipo-resultado de uma operação entre dois outros tipos, como saber se uma operação entre dois tipos é valida (caso ela não exista no dicionário, não é válida).

```
validOperationResults = {
    ('int', '+', 'int'): 'int',
    ('int', '-', 'int'): 'int',
    ('int', '*', 'int'): 'int',
    ('int', '/', 'int'): 'float',
    ('int', '%', 'int'): 'int',
    ('int', '+', 'float'): 'float',
    ('int', '-', 'float'): 'float',
    ('int', '*', 'float'): 'float',
    ('int', '/', 'float'): 'float',
    ('float', '+', 'float'): 'float',
    ('float', '-', 'float'): 'float',
    ('float', '*', 'float'): 'float',
    ('float', '/', 'float'): 'float',
    ('float', '+', 'int'): 'float',
    ('float', '-', 'int'): 'float',
    ('float', '*', 'int'): 'float',
    ('float', '/', 'int'): 'float',
    ('string', '+', 'string'): 'string',
}
```

A função que criamos usa a estrutura acima, e basicamente ela recebe dois tipos e checa na lista se uma operação entre eles existe. Se existir, retorna o tipo-resultado, se não existir retorna um erro.
A função está em src>CC2021>semantic>helper.py

```
def checkIfIsValid(left: Node, right: Node, op, lineNumber):
    opResult = validOperationResults.get((left.type, op,
right.type), None)

    if opResult is None:
        raise
ExceptionAsInvalidOperation(f'{left.type},{right.type},{lineNumbe
r}')

    return opResult
```

4. Verificação de identificadores por escopo

Para evitar que sejam declaradas variaveis com os mesmos identificadores dentro de um mesmo escopo, usamos a struc *Scope*, um objeto que contem uma tabela de simbolos própria (como indicado no enunciado) e também uma lista de escopos “filho” (escopos que sejam declarados dentro desse, como por exemplo um loop numa função).
Para possibilitar isso, sempre que uma nova variável é criada, ao tentarmos adicionar ela à tabela do escopo, checamos se uma variável igual já não existe:

```
def doesVarAlreadyExists(self, identificator):
  for l in self.table:
    if l.label == identificator:
      return True, l.line

  return False, -1

def addToScopeTable(self, entryToAdd):
  exists, line = self.doesVarAlreadyExists(entryToAdd.label)

  if exists:
    raise ExceptionAsInvalidIdentifierDeclaration(line)

  self.table.append(entryToAdd)
  # return 1 indicating succes, and empty string
  return 1, ''
```

Assim, caso uma variável seja declarada com o mesmo nome de outra no mesmo escopo, é avisado o erro.

5. Comandos dentro de escopos

Para evitar que o comando ‘break’ seja utilizado fora de um laço de repetição, também usamos a struct Scope. Durante a etapa da análise semântica, ao realizar a análise de um ‘break’, é recuperado o escopo em que este ‘break’ está inserido, bem como todos os escopos a que este escopo pertence, através do laço ‘while’ como visto abaixo. Caso algum destes escopos seja identificado como um laço de repetição, a função é executada com sucesso. Caso após verificar toda a estrutura de escopos nenhum laço de repetição seja encontrado, é lançada uma exceção, indicando que existe um ‘break’ fora de um loop.

```
def p_statement_break(p: yacc.YaccProduction):
    """STATEMENT : BREAK SEMICOLON"""
    # If is not inside loop scope, consider semantic failure
    current_scope = scope_list.getLastScopeOrNoneIfEmpty()

    # Go into upper scopes trying to find a for loop
    while True:
        if current_scope.isLoop:
            break

        current_scope = current_scope.previousScope

        if current_scope is None:
            raise ExceptionAsBreakOutsideLoop(p.lineno(2))
```

6. GCI - Geração de Código Intermediário

Para a criação do GCI, utilizamos uma estrutura análoga à utilizada para a análise semântica, se apoiando nas ferramentes que o *yacc* nos disponibiliza (visto que a construção da árvoce para a GCI parte de um princípio em comum com a análise semântica) e nas video-aulas da materia. Sua implementação está em *src>CC2021>semantic>gci.py.*
A estrutura que usamos faz com que seja usada uma abordagem bottom-up, onde o código intermediário é concatenado dos nodos folha pra cima (facilitando o trabalho dos goto por conta dos escopos), assim à raiz dessa árvore é atribuido o GCI já finalizado.

Abaixo, um exemplo da nossa implementação pela produção FOR

```
1. def p_forstat(p: yacc.YaccProduction):
2.     """FORSTAT : create_for_loop_label FOR LPARENTHESES ATRIBSTAT
3.     SEMICOLON EXPRESSION SEMICOLON ATRIBSTAT RPARENTHESES LEFTBRACE
4.      STATELIST RIGHTBRACE"""
5.     starting_label = generate_new_label()
6.     next_label = gci.label
7.     code_conditionalBody = p[6]['code']
8.     temporary_variabel= p[6]['temp_var']

9.     code_loopFirstAtribution= p[4]['code'] + '\n'
10.    code_conditional= f'if False {cond_temp_var} goto {next_label}\n'
11.    code_body= p[11]['code']
12.    code_incrementVar= p[8]['code']
13.    code_gotoStart= f'goto {start_label}\n'

14.    code = code_loopFirstAtribution+\
15.         starting_label + ':\n' +\
16.         code_conditionalBody +\
17.         code_conditional+\
18.         code_body+\
19.         code_incrementVar+\
20.         code_gotoStart+\
21.         next_label + ':\n'

21.     p[0] = {
22.         'code': code
23.     }
```

Nas linhas 2, 3 e 4 temos a definição da produção FOR (chamada de FORSTAT)
Na linha 5, criamos um novo label (para possibilitar o loop)
Na linha 6, pegamos também o proximo label (para possibilitar sair do loop)
Nas linhas 7 e 8, pegamos de p[6] (EXPRESSION) o codigo já gerado por ele e também o nome da variável temporário atribuido
Na linha 9, pegamos o codigo da atribuição feita por ATRIBSTAT (p[4] - a primeira parte da definição da condição do loop)
Na linha 10, criamos o codigo da condição do loop
Na linha 11, pegamos o código já existente em STATELIST, que é o código de dentro do loop em si
Na linha 12, pegamos o código da definição do loop onde é feito o incremento da variável de controle
Na linha 13 fazemos o loop (retorno ao label do inicio)

Da linha 14 à linha 21, associamos à variável *code* a concatenação de tudo que foi criado até agora, ou seja, o código completo do loop
E nas linhas 21 a 23, associamos esse código à cabeça da produção para que isso possa ser retornado para o Nodo pai

Abaixo, um programa exemplo e sua respectiva saída (esse programa se encontra em *src>examples>exemplo_for.lcc*)

```
{
    int i;
    for (i = 1; i <= 10; i = i + 1){
    print i;
  }
}
```

Para esse código, foi gerado o seguinte código intermediário:

```
int i
t1 = 1
i = t1

LABEL1:
t3 = 10
t2 = i
t4 = t2 <= t3
if False t4 goto LABEL0
t8 = i
t9 = t8
print t9
t6 = 1
t7 = i + t6
i = t7
goto LABEL1
LABEL0:
```

Avisos:

- O enunciado pedia que os outputs fossem feitos diretamente no terminal, porem para facilitar o entendimento e correcao, optamos por faze-los em arquivos no diretorio raiz do programa.

Link para o repositório do Github:

https://github.com/felipecampossantos/INE5426

Nele se encontram as SDDs e SDTs:

https://github.com/felipecampossantos/INE5426/tree/main/docs/files

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