UNIVERSIDADE FEDERAL DE SANTA CATARINA – UFSC

CENTRO TECNOLÓGICO

DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

CURSO DE GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO

Relatório - ASem e GCI

Construção de Analisador Semântico e Gerador de Código Intermediário

Artur Ribeiro Alfa [17103919]

Augusto Vieira Coelho Rodrigues [19100517]

Leonardo Vieira Nunes [19102923]

Thainan Vieira Junckes [19100545]

FLORIANÓPOLIS, 25 DE JULHO DE 2022

Gramática ConvCC-2022-1 com pequenas mudanças

Nesta implementação, realizamos duas pequenas modificações na gramática ConvCC-2022-1. Na produção de *if* realizamos a troca de STATEMENT para {STATELIST}. No comando *for*, adicionamos "{ }" para padronizar a gramática conforme algumas linguagens mais conhecidas.

1. Construção da árvore de expressão (EXPA)

Para construir a EXPA, separamos as produções da gramática ConvCC-2022-1 que derivam expressões aritméticas. A gramática EXPA pode ser visualizada no arquivo grammar/expa.csf. A partir dessa gramática, criamos a SDD L-Atribuída para realizar a construção da árvore de expressão.

Para garantir que a SDD é realmente L-Atribuída seguimos alguns conceitos: apenas podemos utilizar os atributos sintetizados ou herdados da produção a esquerda da qual está sendo derivada. Ao usar atributos sintetizados, conseguimos evitar que haja ciclos entre pais e filhos diretamente, isso também evita que toda a SDD contenha ciclos.

A SDD e a SDT de EXPA podem ser encontradas nos arquivos grammar/expa_sdd.txt e grammar/expa_sdt.txt respectivamente.

2. Inserção do tipo na tabela de símbolos (DEC)

Para construir DEC, separamos as produções da gramática ConvCC-2022-1 que derivam declarações de variáveis. A gramática pode ser visualizada no arquivo grammar/dec.csf. Foi construída a partir dessa gramática a SDD, que utiliza uma tabela de símbolos com a função new_table_entry, que recebe o identificador, seu tipo, o número de dimensões com o tamanho e a linha na qual foi encontrado. Verifique a seção anterior para a prova de que a SDD é L-atribuída. Sua implementação é feita em src/compiler/semantic.py.

A SDD e a SDT de DEC podem ser encontradas nos arquivos grammar/dec_sdd.txt e grammar/dec_sdt.txt respectivamente.

3. Verificação de tipos

Nesta etapa, fazemos a validação de que as entradas de um expressão aritmética respeitam as regras definidas para as opereções realizadas entre tipos diferentes. No nosso trabalho, criamos as seguintes regras válidas:

a. int e int -> +, -, *, /, %
b. float e float -> +, -, *, /
c. float e int -> +, -, *, /
d. string e string -> concatenação(+)

Para verificar isso, implementamos a função check_type. Ela possui um dicionário valids com as regras listadas anteriormente. A função recebe como entrada dois nodos de uma árvore de exressão(left e right), a operação aritmética a ser realizada entre os dosi nodos e a linha onde está localizada esta operação. Após, um teste é feito para verificar se

a tupla com o atributo result_type de cada nodo e o símbolo da operação possuem uma entrada válida no nosso dicionário. Em caso de retorno None, essa operação é inválida em nossa gramática e uma mensagem de erro é printada no terminal.

4. Verificação de identificadores por escopo

Nesta etapa verificamos se as variáveis do programa não possuem duas declarações com tipos diferentes dentro de um mesmo escopo. Por exemplo, se criamos uma variável string x, não podemos declarar uma variável float x dentro desse escopo.

Na implementação, criamos uma estrutura de dados especificamente para isso, através da classe Scope presente no arquivo src/utils/data_structures.py. Essa classe possui uma tabela de símbolos, uma lista de escopos internos, uma variável do tipo bool para saber se é um escopo de repetição, e os escopos externos.

Através da função add_entry(), adicionamos novas entradas na tabela de símbolos e verificamos se a variável já está presente no escopo. Caso sim, retornamos um erro ao usuário indicando a linha e a variável com problema.

5. Comandos dentro de escopos

Este ponto refere-se à verificação do comando break no escopo de um comando de repetição. Se esse comando não estiver no escopo, ocorre um erro semântico.

Em nossa implementação, essa verificação ocorre na função p_statement_break(p) do arquivo src/compiler/semantic.py. Cada escopo possui um booleano indicando se é um comando de repetição ou não, e além guardar referência ao seu escopo externo. Usamos estes dois fatores para identificar se o break está dentro de um comando de repetição ao percorrer a cadeia de escopos e verificando se ao menos um deles é de repetição. Se um break não estiver dentro de um laço de repetição, a variável current_scope recebe None, o que eventualmente causa uma exceção indicando que há erro semântico com essa causa.

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

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

        current_scope = current_scope.upper_scope

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

6. Tarefa GCI

Para a implementação do gerador de código intermediário utilizamos o módulo YACC da ferramenta PLY do Python. Na próxima seção iremos descrever ela.

Nossa ideia foi fazer com que as produções mais próximas das folhas gerassem os códigos iniciais, e os nodos acima fazem a concatenação nestes códigos, assim representamo a opereaçã tual e temos a raiz da árvore com o código final como atributo.

A seguir temos um exemplo com if/else:

```
def p_ifstat(p: yacc.YaccProduction):
    """IFSTAT : IF LEFTPARENTHESES EXPRESSION RIGHTPARENTHESES LEFTBRACE
STATELIST RIGHTBRACE OPT_ELSE"""
    cond_temp_var = p[3]['temp_var']
    next_label = new_label()

    else_start_label = p[8].get('start_label', None)
     cond_false_next_label = else_start_label if else_start_label else
next_label

     jump_over_else = f'goto {next_label}\n' if else_start_label is not
None else ''

        code = p[3]['code'] + f"if False {cond_temp_var} goto
{cond_false_next_label}\n" + \
          p[6]['code'] + jump_over_else + p[8]['code'] + next_label +
':\n'

    p[0] = {
        'code': code
    }
```

Podemos observar a criação de uma nova label em next_label = new_label(). Logo abaixo temos a verificação de possível existência de else. E depois, é realizada a concatenação do código gerado pelos símbolos no corpo da produção, e no final, associamos o atributo code à cabeça da produção.

7. Descrição da ferramenta PLY

Para implementar a parte semântica do trabalho (GCI, controle de escopos, verificação de tipos, etc), utilizou-se o módulo yacc da biblioteca PLY. Neste módulo, associamos ações às regras da gramática, como demonstrado abaixo.

```
def p_numexp(p: yacc.YaccProduction):
    """NUMEXPRESSION : TERM REC_PLUS_MINUS_TERM"""
    if p[2] is None:
        p[0] = p[1]

    else:
        result_type = check_type(p[1]['node'],
                                 p[2]['node'],
                                 p[2]['operation'],
                                 p.lineno(1))
        p[0] = {
            'node': Node(p[1]['node'],
                         p[2]['node'],
                         p[2]['operation'],
                         result_type)
        }
```

Primeiramente é nomeada uma função com prefixo p_* para identificar funções associadas ao módulo yacc. Logo após, definimos a produção em forma de comentário, e posteriormente, o código a ser executado. Por exemplo, na segunda e terceira linha da função estamos sinalizando que se não há REC_PLUS_MINUS_TERM (p[2]), NUMEXPRESSION(p[0]) recebe TERM(p[1]), caso contrário, executamos o else. A última atribuição do ramo *else* representa uma atribuição do tipo NUMEXPRESSION.node = Node(...) na sintaxe da SDD, que foi simulada por uma estrutura de dicionário na linguagem.

8. Saída e mensagens de erro

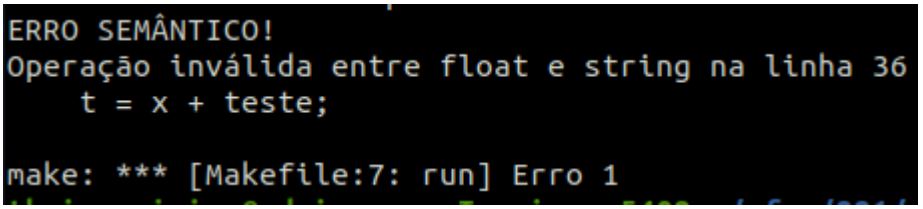
Em caso de sucesso em todos os processos, exportamos três arquivos e as seguintes mensagens aparecem no terminal:

```
As expressões aritméticas são válidas
As declarações das variáveis por escopo são válidas
Todo break está no escopo de um for
Análise semântica concluída com sucesso!!
Tabela de símbolos dos escopos exportada para output/scope_symbol_tables.json
Árvore de expressões exportada para output/expressions.json
Executando a geração de código intermediário...
Código intermediário exportado para output/gci.txt
```

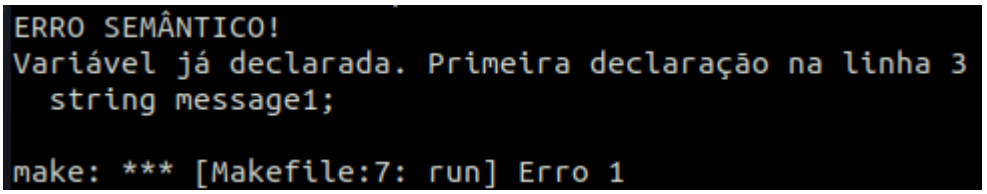
A tabela de símbolos dos escopos é exportada para output/scope_symbol_tables.json. A árvore de expressões está presente no arquivo

output/expressions.json. E por fim, o código intermediário gerado é exportado para output/gci.txt.
Em caso de erro semântico, temos as seguintes situações:

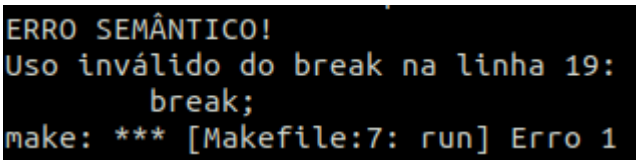
Operação inválida entre tipos diferentes.



Declaração de uma variável já declarada anteriormente no mesmo escopo.



Inserção de um comando break fora de um laço de repetição.



Utilização de uma variável sem declarar ou antes de sua declaração.