Laboratório 2:

Projeto Analisador Léxico C-

Equipe:

Rafael Cassol

Alexandre Bernat

Erick Coelho

Turma 23

Objetivo do Trabalho

O objetivo deste trabalho é implementar o módulo de análise sintática de um compilador para a linguagem C-. Para isso foi usado o bison para a construção do parser, com a análise dos tokens sendo feita pelo módulo léxico, desenvolvido previamente usando a ferramenta flex.

Descrição do Analisador

O analisador sintático constrói uma árvore sintática que segue as regras propostas pelo roteiro na árvore de exemplo. Um pequeno detalhe de diferença - cada função declarada no arquivo (no nível mais próximo da raiz), é um nó irmão do outro. Então, por exemplo, no caso de haver uma main e uma função auxiliar definida fora da main, os tipos dessas funções serão irmãos. A raiz é um ponteiro que aponta para cada uma dessas subárvores.

Então, seguiu-se a seguinte regra de construção:

1. Listagem de funções -> tipos das funções declaradas no mesmo escopo são filhos de um ponteiro raiz que aponta para cada um desses tipos.
    a. Cada função declarada gera uma subárvore em que o tipo da função é a raiz e o nome é o filho esquerdo.
        i. Os tipos dos argumentos são irmãos entre si, encadeadamente, da esquerda para a direita.
        ii. O tipo do argumento mais à esquerda é o filho esquerdo do nome da função.
        iii. O tipo do argumento mais à direita é irmão esquerdo das declarações no corpo da função.
2. Numa operação de atribuição, comparação ou conta, o operador é o pai dos operandos, e o operando à esquerda é filho esquerdo, operando à direita, filho direito.
3. Numa ativação de função, o nome da função é o pai, os argumentos passados, sejam eles constantes ou variáveis, são os filhos, onde o argumento mais à esquerda corresponde ao filho esquerdo, e assim por diante.
4. Declaração de variável - o tipo é o nó pai, o identificador é o filho esquerdo e caso seja um vetor o tamanho do vetor é colocado como filho do identificador.
5. Declarações de variáveis, expressões, statements ou chamadas de funções dentro de um mesmo escopo são irmãos, onde a precedência é dada pela ordem de aparição.
6. EMPTY foi tratado como $$ = NULL;
7. VOID como tipo de função foi tratado como tipo e criou-se um nó para tal; VOID como argumento tratou-se como $$ = NULL;
8. Terminais como, por exemplo, constantes, constituíram folhas.
9. A árvore não incluiu parêntesis e semelhantes, assim como não incluiu vírgula, ponto e vírgula.
10. RETURN foi tratado como folha no caso de não haver nada a ser retornado e como nó pai daquilo que estava a ser retornado quando era o caso.

11. Para statements do tipo IF-ELSE, cria-se um tipo de nó com o título de IF e atribui-se como filho esquerdo a condição do IF e os outros filhos são, sequencialmente o corpo do IF e por fim o statement que segue do ELSE, caso exista.
12. Para o WHILE, cria-se um nó do tipo RepeatK e atribui-se a condição de parada como filho esquerdo e o corpo do whille como filho direito.

Pontos importantes (mudanças principais com relação ao código fornecido)

1. Na construção da árvore, tomou-se o cuidado de criar uma variável global para os identificadores que era computados pelo analisador léxico, para o analisador sintático ser capaz de diferenciar duas regras gramaticais diferentes apenas nos últimos termos da produção (como A -> ABC | AB, por exemplo).
2. O YYSTYPE é treeNode *, de forma que o retorno do analisador é a raiz da árvore de análise sintática.
3. Utilizou-se para diferenciação de novos nós a estrutura já existente do TINY, adicionando-se alguns *kinds* de nodeExp e nodeStmt, declarados no globals.h: Retk, ActivK, DeclK.
4. Para acessar retorno de escopo intermediário na produção gramatical, tomou-se cuidado com as atribuições de cifrão (a numeração era alterada e o $$ representa coisas diferentes, se olhado no escopo intermediário ou no final).

Resultados Obtidos

Ao executar os comandos descritos no readme, temos como output os arquivos txt correspondente à saída do analisador sintático.

Output do programa mdc.c

```
Syntax tree:
  Id: int
    Id: gdc
      Id: int
        Id: u
      Id: int
        Id: v
      If
        Op: ==
          Id: v
          Const: 0
        Return
          Id: u
        Return
```

```
        Function: gdc
          Id: v
          Op: -
            Id: u
            Op: *
              Op: /
                Id: u
                Id: v
              Id: v
Id: void
  Id: main
    Id: int
      Id: x
    Id: int
      Id: y
    Assign to: x
      Id: x
      Function: input
    Assign to: y
      Id: y
      Function: input
    Function: output
      Function: gdc
        Id: x
        Id: y
```

Output do programa sort.c

```
Id: int
  Id: x
    Const: 10
Id: int
  Id: minloc
    Id: int
      Id: a
    Id: int
      Id: high
    Id: int
      Id: i
    Id: int
      Id: k
    Assign to: k
      Id: k
```

```
      Id: low
    Assign to: x
      Id: x
      Id: a
        Id: low
    Assign to: i
      Id: i
      Op: +
        Id: low
        Const: 1
    Repeat
      Op: <
        Id: i
        Id: high
      If
        Op: <
          Id: a
            Id: i
          Id: x
        Assign to: x
          Id: x
          Id: a
            Id: i
        Assign to: k
          Id: k
          Id: i
      Assign to: i
        Id: i
        Op: +
          Id: i
          Const: 1
    Return
      Id: k
Id: void
  Id: sort
    Id: int
      Id: a
    Id: int
      Id: high
    Id: int
      Id: i
    Id: int
      Id: k
    Assign to: i
```

```
      Id: i
      Id: low
    Repeat
      Op: <
        Id: i
        Op: -
          Id: high
          Const: 1
      Id: int
        Id: t
      Assign to: k
        Id: k
        Function: minloc
          Id: a
          Id: high
      Assign to: t
        Id: t
        Id: a
          Id: k
      Assign to: a
        Id: a
          Id: k
        Id: a
          Id: i
      Assign to: a
        Id: a
          Id: i
        Id: t
      Assign to: i
        Id: i
        Op: +
          Id: i
          Const: 1
Id: void
  Id: main
    Id: int
      Id: i
    Assign to: i
      Id: i
      Const: 0
    Repeat
      Op: <
        Id: i
        Const: 10
```

```
      Assign to: x
        Id: x
          Id: i
        Function: input
      Assign to: i
        Id: i
        Op: +
          Id: i
          Const: 1
    Function: sort
      Id: x
      Const: 10
    Assign to: i
      Id: i
      Const: 0
    Repeat
      Op: <
        Id: i
        Const: 10
      Function: output
        Id: x
          Id: i
      Assign to: i
        Id: i
        Op: +
          Id: i
          Const: 1
```

Output do programa simple.c

```
  Id: void
    Id: main
      Id: int
        Id: x
      Id: int
        Id: y
  Id: int
    Id: func
      Id: int
        Id: x

Syntax tree:
  Id: void
    Id: main
```

```
    Id: int
      Id: x
    Id: int
      Id: y
Id: int
  Id: func
    Id: int
      Id: x
```

Como podemos observar, ambos os resultados estão de acordo com as regras de construção da árvore sintática.