ESPECIFICAÇÃO DO 2º TRABALHO PRÁTICO DE COMPILADORES

O objetivo da fase de análise sintática é verificar se a estrutura do código fonte de entrada é uma estrutura válida de acordo com as regras sintáticas da gramática da linguagem de programação.

Neste 2º trabalho, vocês devem construir um analisador sintático que recebe como entrada o vetor de tokens preenchido pelo analisador léxico. O analisador sintático implementará um dos métodos de análise sintática vistos em sala de aula: análise sintática descendente baseada em tabela, análise sintática descendente recursiva ou análise sintática ascendente.

Neste trabalho, também deve ser implementada a tabela de símbolos do compilador. A tabela de símbolos armazenará todos os identificadores declarados no código. Para cada identificador declarado no código, é criada uma entrada na tabela de símbolos. A entrada é criada apenas para declarações e não para os usos dos identificadores.

A tabela de símbolos é implementada como uma tabela hash. A chave da tabela de símbolos é uma cadeia de caracteres representando um lexema e o valor é uma referência para uma struct ou para um objeto. Esse objeto que representa a entrada da tabela de símbolos possui diversos atributos dentre os quais destaca-se:

- O lexema do identificador (obtido consultando-se o objeto token do identificador em questão).
- O tipo (INT, FLOAT, etc) do identificador.
- O número da linha em que o identificador foi declarado (obtido consultando-se o objeto token do identificador em questão).
- Uma referência para o valor que será armazenado no identificador. O valor poderia ser implementado como um objeto genérico, como o Python faz.

Ao final da análise sintática, deve ser impresso o conteúdo de cada linha da tabela de símbolos.

Durante a análise sintática, quando o analisador sintático encontrar algum erro, ele deve imprimir o tipo de erro e o número da linha onde ocorreu o erro. Por exemplo, caso era esperado um ID na linha 2 e o ID não foi encontrado imprima “ID esperado na linha 2”. A rotina que imprime o erro pode ser implementada como a seguir:

```
printSyntacticError(Token tokenEsperado){
        if(tokenEsperado.getNome() == “ID” || tokenEsperado.getNome() == “NUMBER”)
                printf(“%s esperado na linha %d”, tokenEsperado.getNome(), tokenEsperado.getNumLinha());
        else{
                printf(“%s esperado na linha %d”, tokenEsperado.getLexema(), tokenEsperado.getNumLinha());
        }
}
```

Além de imprimir o erro, o analisador deve chamar uma rotina de recuperação de erros que tenta sincronizar o token da entrada com um dos tokens do conjunto de sincronismo. O conjunto de sincronismo de um não-terminal A será formado por todos os tokens que estão em Follow(A).

Ao final da análise sintática, vocês devem verificar se ocorreu pelo menos um erro sintático e, nesse caso, o compilador não pode continuar e a compilação é terminada (vocês podem, por exemplo, utilizar uma flag que indica se ocorreu um erro ou não). Caso não tenha ocorrido erro, o compilador anuncia sucesso no reconhecimento sintático e a compilação pode continuar com as próximas fases.

Árvore de sintaxe abstrata

Além da análise sintática, vocês devem implementar a criação da árvore de sintaxe abstrata. A árvore de sintaxe abstrata (ASA) para nossa gramática é definida a seguir.

À esquerda dos dois pontos (:) temos o nome da classe que representa um dos tipos de nó da ASA e à direita temos uma lista dos nós filhos do nó definido no lado esquerdo. Cada nó filho de uma classe é representado por um atributo nessa classe.

Por exemplo, em

Attr: Identificador id, Expr e

definimos que a classe Attr representa um dos nós da árvore e que essa classe tem dois atributos:

- Um nó do tipo Identificador (implementado pela classe Identificador)
- Um nó do tipo Expr (implementado pela classe Expr)

Cada atributo desse representa um tipo de filho que o nó Attr tem.

Seja outro exemplo Or: Expr e_1, Expr e_2 .

Nesse caso, o nó Or tem dois atributos que são referências para seus filhos Expr e_1 (que é a expressão à esquerda do operador ou) e Expr e_2 (que é a expressão à direita do operador ou).

Nós da ASA

ASTNode: List<ASTNode> filhos, string nome
*Onde filhos é uma lista de objetos do tipo ASTNode. Todas as outras classes herdam de ASTNode, como mostrado na figura a seguir.
Bloco: possui uma lista de filhos ASTNode herdada de ASTNode
Expr: int type, Temp temporario, Token token, int valorInteiro, float valorPontoFlutuante
Attr: Identificador id, Expr e
If: Expr e, List<ASTNode> c_true, List<ASTNode> c_false
While: Expr e, List<ASTNode> c_true
Read: Identificador id
Print: Expr e
LogicalOp: Expr e_1, Expr e_2
RelOp: Expr e_1, Expr e_2
ArithOp: Expr e_1, Expr e_2

Identificador: TableEntry* refEntradaTabSimbolos
Numero: TableEntry* refEntradaTabSimbolos

Obs: A raiz da árvore é um nó ASTNode que representa o método main.
Obs: List<ASTNode> filhos guarda os filhos de um nó. No caso de um comando (como bloco) que pode ter vários comandos dentro, essa lista é usada para guardar as referências para os filhos.

Essas listas podem ser implementadas da forma que preferirem. É importante que essas listas sejam atribuídas corretamente ao nó pai delas.

Obs: Todas as classes herdam de ASTNode, então todas as classes vão possuir um atributo filhos, mesmo que ele não seja usado, e um atributo nome que irá armazenar o nome do nó. Por exemplo, o atributo nome da classe Attr é inicializado com a string “Attr”, o atributo nome da classe If é inicializado com a string “If”, o atributo nome da classe Identificador é inicializado com a string “Identificador” e assim por diante.
A respeito da criação da ASA, uma forma de implementar a criação dos nós filhos de um determinado nó da ASA é através da utilização do atributo filhos, definido na classe ASTNode. Esse atributo é herdado por todas as outras classes da ASA. Dessa forma, todos os nós filhos de um determinado nó podem ser adicionados nessa lista à medida que vão sendo reconhecidos na análise.

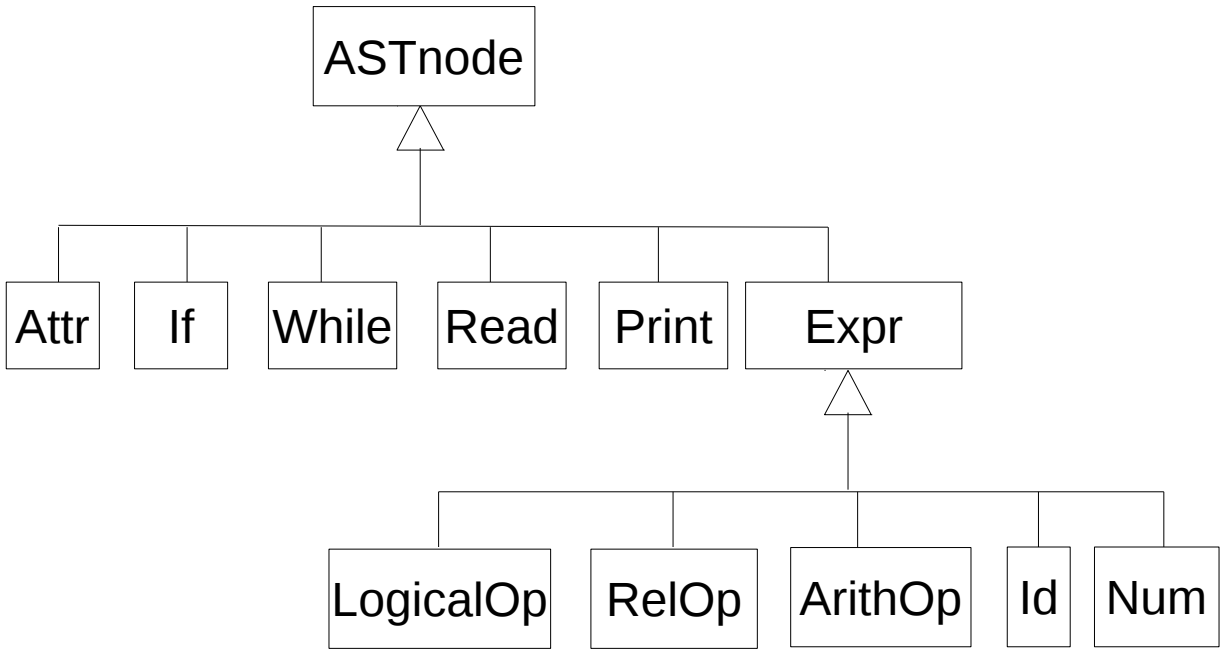
LogicalOp é a classe que gera código para os operadores lógicos and (&&) e or (||), RelOp é a classe que gera código para os operadores relacionais (==, !=, <, <=, > e >=) e ArithOp é a classe que gera código para os operadores aritméticos (+, -, * e /).

Obs: A classe Temp é uma abstração para um registrador ou posição de memória e possui um único atributo, um nome criado para o temporário. A cada vez que o construtor de Temp é chamado, é criado um temporário com um novo nome.
Temp: string nome;

Obs: Os atributos type, temporario e token são definidos somente na classe abstrata Expr e são herdados por todas as subclasses de Expr.

Hierarquia de classes

A seguinte hierarquia de classes deve ser implementada para a ASA:



A classe ASTnode representa um nó genérico da ASA. Todos os outros nós da ASA herdam de ASTnode.

A ASA deverá ser criada ao longo do processo de análise sintática. Para a análise descendente recursiva, foi disponibilizado um pseudocódigo no moodle que realiza, ao longo da análise sintática, a criação da ASA. Para os outros métodos de análise, os nós da árvore devem ser criados à medida que determinados terminais e não-terminais são empilhados.

No caso do método baseado em tabela, podem ser inseridas regras semânticas nos corpos das produções. Quando essas regras estiverem no topo da pilha elas ativam funções correspondentes para a criação de um determinado nó da árvore.

Impressão da ASA ou AST

A ASA deve ser impressa em um arquivo seguindo o formato mostrado abaixo. Ela deve ser impressa do nó raiz para os nós folhas e da esquerda para direita durante o caminhamento.

```
<Astnode>
    <Attr>
        <Id  lexema=’a’ tipo=’integer’>
        <Num valor=’0’ tipo=’integer’>
    </Attr>
    <Attr>
        <Id  lexema=’b’ tipo=’float’>
        <Num valor=’1.1’ tipo=’float’>
    </Attr>
    <While>
        <LogicalOp op=’&&’>
```

```
                <RelOp op='<' >
                        <Id lexema='a' tipo='integer'>
                        <Id  lexema='b' tipo='float'>
                </RelOp>
                <RelOp op='>' >
                        <Id lexema='a' tipo='integer'>
                        <Id  lexema='b' tipo='float'>
                </RelOp>
        </LogicalOp>
        <Print>
                <ArithOp op='+'>
                        <Id lexema='a' tipo='integer'>
                        <Num valor='20' tipo='integer'>
                </ArithOp>
        </Print>
        <Attr>
                <Id  lexema='b' tipo='float'>
                <ArithO op='/'>
                        <Id  lexema='b' tipo='float'>
                        <Num valor='2' tipo='integer'>
                </ArithOp>
        </Attr>
    </While>
</Astnode>
```

Uma última tarefa do trabalho consiste em realizar um caminhamento na árvore de sintaxe abstrata interpretando/avaliando, durante o caminhamento, todas as expressões lógicas, aritméticas e relacionais. Deve ser impressa cada expressão avaliada juntamente com seu resultado.

O trabalho vale 20 pontos. O trabalho deverá ser entregue dia 16/10 via e-mail e deverá ser apresentado nos dias 17/10 e 19/10. Caso um aluno não apareça para apresentar sem a devida justificativa ele recebe nota 0 no trabalho.