Compilador fase 2: Análise Semântica e Geração de Código

O objetivo dessa fase da implementação do Compilador é implementar as fases de **Análise Semântica** e **Geração de Código Intermediário**, a implementação dessa fase será baseada na implementação realizada na **fase 1**, caso você não tenha implementado a **fase 1**, para essa fase **terá que implementar tanto a fase 1 quanto a fase 2**.

Análise Semântica

Na Análise Semântica o seu Compilador deverá verificar se as construções sintáticas da fase anterior estão coerentes, o Compilador implementado na fase anterior deve manter as funcionalidades de identificação de erros **léxicos** e **sintáticos,** e adicionalmente, emitir as mensagens de erros semânticos, caso ocorram.

Basicamente o Compilador fará somente a verificação semântica para variáveis em dois momentos:

- **Declaração**: Na seção de declaração de variável **`<declaracao_de_variaveis>`** o Compilador deve garantir que os identificadores usados no nome de variável sejam únicos, ou seja, não podemos ter duas variáveis declaradas com o mesmo identificador, caso aconteça uma repetição de identificador o Compilador deve ser finalizado informando que ocorreu um erro semântico. Para isso deverá ser implementado uma **minitabela de símbolos** que armazenará as variáveis declaradas (**identificador**, **endereço** e **tipo**), O endereço da variável seria a ordem em que a variável foi declarada, dessa forma a primeira variável tem endereço **`0`**, a segunda endereço **`1`** e assim sucessivamente.

- **Corpo do programa:** As variáveis declaradas na seção de declaração podem ser referenciadas nos comandos de atribuição, nas expressões e nas chamadas das funções de entrada e saída. Assim toda vez que uma variável for referenciada no corpo de programa, o Compilador deve verificar se a variável foi declarada corretamente na seção de declaração de variáveis, caso não tenha sido declarada é gerado um erro semântico explicativo e compilador é finalizado.

Para simplificar a análise semântica e geração de código intermediário o Compilador não fará distinção entre expressões inteiras e lógicas, ou seja, para esse trabalho o Compilador só terá variáveis e expressões do tipo **`inteiro`**, portanto não teremos construções do tipo **`25+(x>y)`** e nem atribuição das constantes **`falso`** e **`verdadeiro`** às variáveis, por exemplo **`var:=falso`**.

Geração de Código Intermediário

A Geração de Código Intermediário será baseada na proposta do livro do professor **Tomasz Kowaltowiski** Implementação de Linguagem de Programação (**Seção 10.3 Análise Sintática e Geração de Código**), basicamente será necessário inserir a geração das instruções da MEPA nas funções mutuamente recursivas que implementam a gramática do analisador sintático, para tanto basta **imprimir as instruções da MEPA** nas mesmas funções que fazem análise sintática e semântica do Compilador. Por exemplo, considere a produção abaixo para o comando **`<comando_enquanto>`**, conforme visto na gramática da fase 1 do Compilador.

`<comando_enquanto> ::= enquanto “(” <expressao> “)” faca <comando>`

A implementação da função correspondente que gera código intermediário para produção do **<comando_enquanto>** seria:

```
void comando_enquanto(){
   int L1 = proximo_rotulo();
   int L2 = proximo_rotulo();
   consome(ENQUANTO);
   printf("L%d:\tNADA\n",L1);
   consome(ABRE_PAR);
   expressao();
   consome(FECHA_PAR);
   printf("\tDSVF L%d\n",L2);
   consome(FACA);
   comando();
   printf("\tDSVS L%d\n",L1);
   printf("L%d:\tNADA\n",L2);
}
```

Suponha que a função **proximo_rotulo()** retorna o próximo rótulo consecutivo positivo (por exemplo L1, L2, L3, ...). **Importante:** Como todas as funções são recursivas, deve-se tomar o cuidado na atribuição das variáveis que vão receber o retorno da função e a ordem de chamadas da função **proximo_rotulo()**.

Como explicado acima, vamos considerar somente variáveis do tipo **inteiro**, dessa forma a produção **<fator>** precisa ser modificada para:

<fator> ::= **identificador** | **numero** | “(” <expressao> “)”

E a sua implementação seria:

```
void fator(){
    if(lookahead == IDENTIFICADOR){
        int endereco = busca_tabela_simbolos(InfoAtomo.atributo_ID);
        printf("\tCRVL %d\n",endereco);
        consome(lookahead);
    }
    else if(lookahead == NUMERO){
        printf("\tCRCT %d\n", InfoAtomo.atributo_numero);
        consome(lookahead);
    }else{
        consome('(');
        E();
        consome(')');
    }
}
```

A função **busca_tabela_simbolos()** recebe como parâmetro o atributo atributo_ID do átomo corrente (um vetor de caracteres) e retorna o endereço da variável armazenado na minitabela de símbolos, caso o identificador não conste da tabela de símbolos a função gera um erro semântico e para a execução do Compilador. Lembre-se que a variável **InfoAtomo** é uma variável global do tipo **TInfoAtomo** e é atualizada na função **consome()** e armazena os atributos do **átomo** reconhecido no **analisador léxico**.

Execução do Compilador – fase 2

A seguir temos um outro programa em **Portugal** que calcula o fatorial de um número informado ao programa, considere que o programa **`exemplo1`** não possui erros léxicos e sintáticos.

```
 1 /*
 2 programa calcula o fatorial de um numero lido
 3 */
 4 algoritmo exemplo1;
 5     variavel fat,num,cont:inteiro;
 6 inicio
 7      leia(num);
 8      fat := 1;
 9      cont := 2;
10      while( cont <= num ) faca
11      inicio
12          fat := fat*cont;
13          cont := cont + 1
14      fim;
15      escreva(fat) // imprime o fatorial calculado
16 fim.
```

Saída do compilador:

```
    INPP
    AMEM 3    # declaração das variáveis fat (end=0), num (end=1) e cont (end=2)
    LEIT
    ARMZ 1    # leia(num)
    CRCT 1
    ARMZ 0    # fat := 1;
    CRCT 2
    ARMZ 2    # cont := 2;
L1: NADA
    CRVL 2
    CRVL 1
    CMEG      # tradução da expressão condicional do enquanto
    DVSF L1   # cont <= num
    CRVL 0
    CRVL 2
    MULT
    ARMZ 0    # fat := fat*cont;
    CRVL 2
    CRCT 1
    SOMA
    ARMZ 2    # cont := cont + 1
    DSVS L1
L2: NADA
    CRVL 0
    IMPR      # escreva(fat)
    PARA
```

Observações importantes:

O programa deve estar bem documentado e pode ser feito em grupo de até **2 alunos**, não esqueçam de colocar o **nome dos integrantes** do grupo no arquivo fonte do trabalho e sigam as **Orientações para Desenvolvimento de Trabalhos Práticos** disponível no **Moodle**.

O trabalho será avaliado de acordo com os seguintes critérios:

- Funcionamento do programa, caso programa apresentarem ***warning*** ao serem compilados serão penalizados. Após a execução o programa deve finalizar com **retorno igual a 0**;
- O trabalho deve ser desenvolvido na **linguagem C** e será testado usando o compilador do **CodeBlocks 17.12**.
- O quão fiel é o programa quanto à descrição do enunciado, principalmente ao formato de do **arquivo de entrada**;
- Clareza e organização, programas com código confuso (linhas longas, variáveis com nomes não-significativos, ...) e desorganizado (sem indentação, sem comentários, ...) também serão penalizados.