

## Compilador fase 2: Análise Semântica e Geração de Código<sup>1</sup>

Na **fase 1** do projeto foi implementado um Compilador que fazia a leitura de um **arquivo-fonte** na linguagem **CLite** e realizava a análise léxica e sintática do programa fonte. 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>** 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 precisará realizar a distinção entre expressões inteiras e lógicas, assim o Compilador só terá variáveis e expressões do tipo **int**, portanto não teremos construções do tipo **25+(x>y)** e nem atribuição das constantes **true** e **false** às variáveis, por exemplo **var=false**.

### Geração de Código Intermediário

A Geração de Código Intermediário será baseada na proposta do livro do professor **Tomasz Kowaltowski** 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\_while>**, conforme visto na gramática da fase 1 do Compilador.

```
<comando_while> ::= while "(" <expressao> ")" <comando>
```

Importante: A especificação desse trabalho pode sofrer modificações de acordo com discussões que tivermos em sala de aula.

---

<sup>1</sup> **Importante:** A especificação desse trabalho pode sofrer modificações de acordo com discussões que tivermos em sala de aula.

A implementação da função correspondente que gera código intermediário para produção do **<comando\_while>**, seria:

```
void comando_while(){
    int L1 = proximo_rotulo();
    int L2 = proximo_rotulo();
    consome(WHILE);
    printf("L%d:\tNADA\n",L1);
    consome(ABRE_PAR);
    expressao();
    consome(FECHA_PAR);
    printf("\tDSVF L%d\n",L2);
    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 **int**, dessa forma a produção **<operando>** precisa ser modificada para:

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

E a sua implementação seria:

```
void operando(){
    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**, que é informado na tela do computador, em seguida o processo de compilação é finalizado.

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**.

**Importante:** Com o compilador é *case sensitive*, ou seja, para **\_var1** e **\_Var1** são considerados identificadores distintos.

## Execução do Compilador – fase 2

A seguir temos um outro programa em **CLite** 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  int main(void){
5      int _fat,_Num,_cont;
6      scanf(_Num);
7      _fat = 0x1;
8      _cont = 0x2;
9      while(_cont <= _Num){
10         _fat = _fat*_cont;
11         _cont = _cont + 0x1;
12     }
13     printf(_fat); // imprime o fatorial calculado
14 }
```

### Saída do compilador:

```
INPP
AMEM 3    # declaração de variáveis:_fat (end=0), _Num (end=1) e _cont (end=2)
LEIT
ARMZ 1    # scanf(_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 while
DVSF L2   # _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      # printf(_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**;
- Caso o **programa não compile ou não execute** será penalizado com a **NOTA 0.0**;
- O trabalho deve ser desenvolvido na **linguagem C** e será testado usando o compilador do **MinGW** com **VSCode**, para configurar sua máquina no Windows acesse:  
<https://www.doug.dev.br/2022/Instalacoes-e-configuracoes-para-programar-em-C-usando-o-VS-Code/>
- 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 levados em consideração na correção.