Faculdade Impacta de Tecnologia
Compiladores

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.

1 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.

A verificação semântica para variáveis ocorre em dois momentos:

- **Declaração:** Na seção de declaração de variável dada pelo não terminal <declaracão de variáveis>, 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 $false$ e $true$ às variáveis, por exemplo $var := false$.

2 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). De maneira simplificada, será necessário inserir ações semânticas para a geração das instruções da MEPA nas funções recursivas que implementam a gramática do analisador sintático. Tais ações semânticas vão 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>.

<comando while> ::= **while** <expressao> **do** <comando>

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

```
def comando_while():
  L1 = proximo_rotulo()
  L2 = proximo_rotulo()
  consome(WHILE)
  print(f'L{L1}:\tNADA')
  expressao()
  print(f'\tDSVF L{L2}')
  consome(DO)
  comando()
  print(f'\tDSVS L{L1}')
  print(f'L{L2}:\tNADA')
```

Suponha que a função proximo_rotulo() retorna o próximo rótulo consecutivo positivo (por exemplo L1, L2, L3, $\cdots$). 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:

```
def fator():
  if lookahead.tipo == IDENTIFICADOR:
    endereco = busca_tabela_simbolo(lookahead.lexema)
    print(f'\tCRVL {endereco}')
    consome(IDENTIFICADOR)
  elif lookahead.tipo == NUMERO:
    print(f'\tCRCT {lookeahead.lexema}')
    consome(NUMERO)
  else:
    consome(ABRE_PAR)
    expressao()
    consome(FECHA_PAR)
```

A função `busca_tabela_simbolos()` recebe como parâmetro o atributo lexema do átomo corrente e retorna o endereço da variável armazenado na tabela de símbolos. Caso o identifi-

cador não conste da tabela de símbolos a função gera um erro semântico e para a execução do compilador.

3 Execução do compilador - fase 2

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

```
(*
programa que calcula o fatorial de um numero lido
*)
program exemplo1;
   var fat, num, cont: integer;
begin
   read(num)
   fat := 1
   cont := 2
   while cont <= num do
   begin
      fat := fat * num
      cont := cont + 1
   end;
   write(fat)  // imprime fatorial calculado
end.
```

Saída do compilador:

```
INPP
AMEM 3 # declaração de variáveis fat (end=0), num (end=1) e cont (end=3)
LEIT
ARMZ 1 # leia(num)
CRCT 1
ARMZ 0 # fat := 1
CRCT 2
ARMZ 2 # cont := 2
L1: NADA
CRVL 2 # tradução da expressao condicional do while
CRVL 1
CMEG   # cont <= num
DVSF L2
CRVL 0
CRVL 2
MULT
ARMZ 0 # fat := fat * num
CRVL 2
CRCT 1
SOMA
```

```
ARMZ 2 # cont := cont + 1
DSVS L1
L2: NADA
CRVL 0
IMPR
PARA
```

Observações importantes: O programa deve estar bem documentado e pode ser feito em grupo de até 5 alunos, não esqueçam de colocar o nome dos integrantes do grupo no arquivo fonte do trabalho.

O trabalho será avaliado de acordo com os seguintes critérios:

1. Funcionamento do programa.
2. O trabalho deve ser implementado utilizando a linguagem python.
3. O quão fiel é o programa quanto à descrição do enunciado, principalmente ao formato do arquivo de entrada;
4. Clareza e organização, programas com código confuso (linhas longas, variáveis com nomes não-significativos, ...) e desorganizado (sem comentários, ...) também serão penalizados.