Universidade do Sul de Santa Catarina - UNISUL
Curso de Ciência da Computação **Campus de Tubarão**
Disciplina: Tradução de Linguagens de Programação
Professor: Charbel Szymanski **E-mail:** charbel@evoluma.com

Análise semântica e geração de código intermediário

Descrição das ações semânticas

Neste documento são descritas as ações do compilador para verificação semântica e geração de código para a máquina hipotética da linguagem LMS. São também apresentadas recomendações para a implementação e recuperação de erros. Este documento é apenas um guia para servir de base durante a implementação da análise semântica e geração de código intermediário para a LMS. Portanto, algumas verificações adicionais poderão ser necessárias durante a implementação e caberá aos alunos desenvolvê-las.

Observação: Dependendo da versão da linguagem LMS utilizada, algumas ações semânticas descritas neste documento poderão não ser utilizadas. Cabe aos alunos a análise das ações semânticas necessárias, a partir da gramática contendo as ações semânticas dispostas nas produções.

Módulos: cada equipe poderá, a seu critério, estruturar os módulos de compilação e interpretação. Para efeito didático, sugere-se a seguinte estrutura geral (sob o ponto de vista de uma implementação em Java):

Classe Pilha – utilizada para o gerenciamento de diversas pilhas a serem utilizadas no processo de compilação. Principais métodos:

```
estaVazia(), estaCheia(), inicializa(), insereElemento(elem),
removeElemento(), veTopo(), listaPilha()
```

Pode-se também utilizar a classe *Stack* padrão do Java, que é de uso simples e já implementa uma pilha. Outra opção que acompanha a API padrão do Java é a interface *Deque*, que pode ser usada da seguinte forma:

```
Deque<Integer> stack = new ArrayDeque<Integer>();
```

Essa forma de uso utiliza um array de tamanho variável. Outra opção para a interface *Deque* é utilizar uma implementação baseada na classe *LinkedList* ao invés de *ArrayDeque*.

Classe Máquina – gerenciamento da máquina hipotética, com os seguintes métodos:

- inicializaAreaIntrucoes → inicializa área de instruções
- inicializaAreaLiterais → inicializa área de literais

- incluiAreaInstrucao → inclui instrução na área de instruções
- alteraAreaInstrucao → altera determinada instrução na área de instruções
- incluiAreaLiterais → inclui literal na área de literais
- interpreta

Classe TS - gerenciamento da Tabela de Símbolos, com os seguintes métodos:

- inicializaTs → inicializa a tabela de símbolos
- busca → busca nome na TS – se estiver devolve a sua posição na TS
- insere → busca nome na TS – se identificador está na tabela e já foi declarado no nível, então indica erro; senão insere.
- remove → deleta nomes de determinado nível da TS. Garante a alocação dinâmica de nomes na tabela – é acionada quando o compilador termina a compilação de um bloco.

Classe Sintático – gerenciamento da análise sintática comandando a semântica e geração de código, com os seguintes métodos:

- inicializaMatrizParsing
- inicializaRegrasProducao
- analiseSintatica
- trataAcao (faz verificações de ordem semântica e comanda a geração de código)

A seguir serão apresentadas as ações (fase de compilação), cabendo aos alunos a sua análise, interpretação e conveniente adaptação na construção do compilador.

#100 : Reconhecendo o nome do programa.

- Inicializa pilhas (ifs, while, repeat, procedures, case, for) –
- Inicializa tabela de símbolos (vetor tab_hash e a própria tabela tab_simb) –
- Inicializa área de instruções da máquina hipotética
- Inicializa área de literais da máquina hipotética
- Inicializa algumas variáveis :
  - nível_atual:=0 (faz o controle do nível atual),
  - Pt_livre:=1 , (aponta para a próxima posição livre da tabela de símbolos)
  - escopo[0]:=1 (usada juntamente com Tabela de símbolos) ,

    Observa-se que estas três variáveis, dependendo da implementação, podem ser inicializadas na Classe TS.
  - número de variáveis nv :=0 (número de variáveis num bloco)

- deslocamento=:3 (em relação a base),
- Lc:=1 (aponta para a próxima instrução a ser gerada)
- Lit := 1 { ponteiro auxiliar para área de literais – n. de ordem}

#101: Final de programa

- Gera instrução PARA
- Verifica utilização de rótulos através da tabela de símbolos

#102: Após declaração de variável

- Gera instrução AMEM utilizando como base o número de ações acumuladas na ação #104

#103: Após palavra LABEL em declaração de rótulo

- Seta tipo_identificador = rótulo

#104: Encontrado o nome de rótulo, de variável, ou de parâmetro de procedure em declaração

```
caso tipo_identificador =  RÓTULO  {setado na ação #103}
        se nome está na tabela de símbolos no escopo do nível
        (*usar rotina de inserção na TS*)
                então  erro
                senão
                  - insere identificador na tabela de símbolos com os
                    atributos: categoria = rótulo, nível, endereço da instrução rotulada =0 e
                    cabeça de lista de referências futuras = 0
        fim se

caso tipo_identificador = VARIÁVEL {setado em #107}
        se nome está na tabela de símbolos no escopo do nível
                  (* usar rotina de inserção*)
                 então erro
                senão
                 - insere identificador na TS com os atributos: categoria =  variável, nível,
                deslocamento;
                - acumula número de variáveis (* nv:=nv +1 *)
        fim se

caso tipo_identificador = PARÂMETRO {setado em #111}
        se nome está na tabela de símbolos no escopo do nível
        (* usar rotina de inserção na TS*)
        então erro
        senão
                insere nome na tabela de símbolos preenchendo
                atributos: categoria = parâmetro, nível;
                acumula número de parâmetros (* np=np+1*)
        fim se
```

#105: Reconhecido nome de constante em declaração

. se nome já declarado no escopo do nível

então erro
senão
insere identificador na tabela de símbolos preenchendo
atributos : categoria = constante, nível
Salva endereço do identificador na TS
fim se

#106: Reconhecido valor de constante em declaração
. preenche atributo para constante na TS (valor base 10), utilizando endereço do identificador na TS salvo em ação #105

#107: Antes de lista de identificadores em declaração de variáveis
- seta tipo_identificador = variável

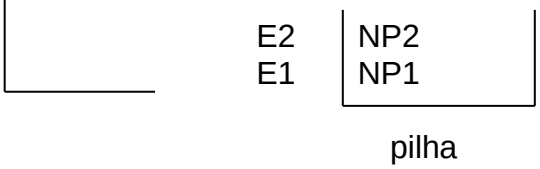
#108: Após nome de procedure, em declaração
Faz:
- categoria := proc
- inserção
- houve_parametros := false
- n_par := 0
- incrementa nível (Nível_atual:= nível_atual + 1)

#109: Após declaração de procedure

se houver parâmetro então

atualiza numero de parâmetros na TS para a procedure em questão
GeralB = np
preenche atributos dos parâmetros (deslocamento):
primeiro parâmetro –> deslocamento = - (np)
segundo parâmetro –> deslocamento = - (np – 1)

---
fim se
gera instrução DSVS com parâmetro zero, e salva na pilha de controle de desvios de procedure o endereço da instrução de desvio e o número de parâmetros.



pilha

#110: Fim de procedure
- retira da pilha de controle de procedures: número de parâmetros (np) , endereço da instrução de desvio
- gera instrução RETU
- verifica utilização de rótulos na TS

- completa instrução de desvio da procedure (aponta para LC)
- deleta nomes do escopo do nível na TS
- decrementa nível (Nível_atual:= nível_atual – 1)

#111: Antes de parâmetros formais de procedures
- seta tipo_identificador = parâmetro
- houve parâmetro = true

#112: Identificador de instrução rotulada ou comando de atribuição
- salva nome do identificador

#113: Instrução rotulada

```
se nome (salvo em #112) esta na TS e é nome de rótulo então
    se nível <> nível atual então
        erro
    senão
        marca o endereço da instrução rotulada na TS
        se lista de referências futuras não estiver vazia então
            percorre lista e preenche endereços
        fim se
    fim se
senão
    erro
fim se
```

#114: Atribuição parte esquerda

```
se nome está na tabela de símbolos então
        se nome <> nome de variável então erro
        senão salva  atributos do nome
        fim se
senão erro (“identificador não declarado”)
fim se
```

#115 : Após expressão em atribuição

. gera instrução armazena (ARMZ) para variável da esquerda (atributos salvos em #114)

#116 : Chamada de procedure

```
se nome esta  na TS e é nome de procedure
então  salva endereço do nome
senão   erro
fim se
```

#117: Após comando call

```
se num. de parâmetros <> num. de parâmetros efetivos
então  erro
senão     gera instrução CALL, utilizando informaçoes da procedure, contidas na TS (
endereço na TS salvo em ação #116)
 fim se
```

#118: Após expressão, em comando call

- acumula número de parâmetros efetivos

#119: Comando GOTO

```
se identificador está na TS e é nome de rótulo
  então     se nível <> nível atual
                então   erro
                senão  se endereço de instrução rotulada existe
                          então  gera DSVS para endereço
                          senão    gera DSVS e guarda seu endereço na lista de referências
                             futuras
                          fim se
            fim se
 senão   erro
 fim se
```

#120: Após expressão num comando IF

- gera DSVF com operando desconhecido
- empilha endereço da instrução (*para ser resolvido o endereço do operando futuramente *)

#121: Após instrução IF

- completa instrução DSVS gerada na ação #122
- operando recebe valor de LC

#122: Após domínio do THEN, antes do ELSE

- resolve DSVF da ação #120, colocando como operando o endereço (LC + 1)
- gera instrução DSVS, com operando desconhecido, salvando seu endereço na pilha dos IF's para posterior marcação

#123: Comando WHILE antes da expressão

- o valor de LC é armazenado na pilha dos WHILE's, este é o endereço de retorno do WHILE

#124: Comando WHILE depois da expressão

- gera DSVF com operando desconhecido. Como o operando não é conhecido no momento, o seu endereço (ou da instrução) é guardado na pilha dos WHILE's para posterior marcação

#125: Após comando WHILE

- resolve DSVF da ação #124 colocando como operando o endereço(LC + 1)
- gera DSVS com operando = endereço de retorno, salvo na pilha de ação #123

#126: Comando REPEAT – início

- o valor de LC é armazenado numa pilha (pilha dos repeat´s) - este é o endereço de retorno.

#127: Comando REPEAT – fim

- gera DSVF, utilizando como operando o valor de LC guardado na pilha dos repeat´s conforme ação # 126.

#128: Comando READLN início

- seta contexto = readln

#129: Identificador de variável

```
caso contexto = readln {setado em #128}
        se identificador é nome de variável e está na tabela de símbolos então
                gera LEIT
                gera ARMZ
        senão erro
        fim se

caso contexto = expressão {setado em #156}
        se nome não está na tabela de símbolos
           então erro
           senão       se nome é de procedure ou de rótulo
                        então erro
                        senão  se nome é de constante
                                  então   gera CRCT valor decimal
                                  senão gera CRVL  - , deslocamento
                               fim se
                        fim se
         fim se
```

#130: WRITELN - após literal na instrução WRITELN

- armazena cadeia literal na área de literais (pega o literal identificado pelo léxico e transposta para área de literais – área_literais)
- atualiza ponteiro de literal (pont_literal – vetor que aponta para o inicio do literal respectivo na área de literais) - aponta para o inicio do proximo literal.
- gera IMPRLIT tendo como parâmetro o numero de ordem do literal (literal 1, literal 2 ...)
- incrementa no. de ordem do literal

Nota : a área de literais (área_literais) e o ponteiro de literais (pont_literal) são gerados na fase de compilação e utilizados na fase de interpretação (execução) do programa.

#131: WRITELN após expressão

- gera IMPR

#132 : Após palavra reservada CASE

. Acopla mecanismo de controle de inicio de CASE junto à pilha de controle de CASE

#133: Após comando CASE

- completa instruções de desvio (DSVS), relativas ao CASE em questão, com LC, utilizando endereços salvos na pilha de controle
- gera instrução AMEN -, -1 (limpeza)

#134: Ramo do CASE após inteiro, último da lista

- gera instrução COPIA
- gera instrução CRCT inteiro
- gera instrução CMIG
- resolve, se houver pendência, instruções de desvio (DSVT) utilizando endereços salvos na pilha de controle, colocando como operando (LC+1)
- gera instrução DSVF, guardando endereço do operando ou da instrução na pilha de controle dos CASE´s.

#135: Após comando em CASE

- resolve ultima instrução de desvio (DSVF) gerada, utilizando endereços salvos na pilha de controle, colocando como operando (LC+1)
- gera instrução DSVS, guardando endereço da instrução na pilha de controle, para posterior marcação.

#136: Ramo do CASE: após inteiro

- gera instrução COPIA
- gera instrução CRCT inteiro
- gera instrução CMIG
- gera instrução DSVT – salvando endereço da instrução na pilha de controle para posterior marcação.

#137: Após variável controle comando FOR

se nome esta na TS e é nome da variável então
salva endereço do nome em relação a TS
senão erro
fim se

#138: Após expressão valor inicial

- gera instrução ARMZ – considerando variável de controle atributos salvos em #137)

#139: Após expressão – valor final

- armazena valor de LC na pilha de controle do FOR
- gera instrução COPIA
- gera instrução CRVL – atributos salvos em #137
- gera instrução CMAI
- gera instrução DSVF com parâmetro desconhecido, guardando na pilha de controle o endereço do operando (ou da instrução) para posterior marcação.
- armazena na pilha de controle o endereço do nome da variável de controle relativo à tabela de símbolos.

#140: Após comando em FOR

- gera instrução CRVL, utilizando endereço salvo em #139(@ da TS da variável de controle na pilha de controle)
- gera instrução CRCT (1) base 10
- gera instrução soma (até aqui incrementa variável de controle)
- gera instrução ARMZ variável controle
- completa instrução DSVF, gerada na ação #139, utilizando como operando (LC+1)
- gera instruçao DSVS, utilizando como operando o valor de LC salvo na ação #139 (retorno)

- gera instrução AMEN, -1 (limpeza)

#141 à #146: comparações
- gera instrução de comparação correspondente

#147: Expressão – operando com sinal unário
- gera INVR

#148: Expressão – soma
- gera SOMA

#149: Expressão – subtração
- gera SUBT

#150: Expressão – or
- gera DISJ

#151: Expressão – multiplicação
- gera MULT

#152: Expressão – divisão
- gera DIV

#153: Expressão – and
- gera CONJ

#154: Expressão – inteiro
- gera CRCT

#155: Expresso – not
- gera NEGA

#156: Expressão – variável
- seta contexto = expressão