GERAÇÃO DE CÓDIGO INTERMEDIÁRIO

- Transformação da árvore de derivação em uma seqüência de código mais próximo do código objeto.

- As seguintes vantagens estão associadas à geração de código intermediário:
  - maior portabilidade, já que apenas a geração de código objeto deve ser específica para a máquina;
  - facilidade de otimização do código.

- As representações mais utilizadas de código intermediário são:
  - Árvores de sintaxe;
  - Notação pós-fixada;
  - Código de três endereços.

Código de Três Endereços

- Seqüência de comandos com o seguinte formato geral:

$$X = y\ op\ z,$$

onde *x*, *y* e *z* são nomes, constantes ou variáveis temporárias geradas pelo compilador e *op* é qualquer operador.

- Uma expressão do tipo *x + y * z* é traduzida para o seguinte código:

*t1 = y * z;*
t2 = x + t1;

onde *t1* e *t2* são variáveis temporárias geradas pelo compilador.

- O CTE utiliza, para cada comando, no máximo três posições de memória: duas para operandos e uma para o resultado.

- Os seguintes tipos de comandos são utilizados no CTE:
  1. Comandos de atribuição do tipo *x = y op z*, onde *op* é um operador binário ou um operador lógico.
  2. Comandos de atribuição do tipo *x = op y*, onde *op* é um operador unário.
  3. Comandos de cópia do tipo *x = y*, onde o valor de *y* é atribuído a *x*.
  4. Desvio incondicional do tipo *goto L*, onde L é o label do comando a ser executado.
  5. Desvios condicionais, tais como *if x relop y goto L*, onde *relop* é um operador relacional.
  6. Chamadas de rotinas *param x* e *call p,n* e retornos de valores de rotinas, do tipo *return y*. Um exemplo de uso é apresentado a seguir, com o CTE gerador a partir de uma chamada de rotina *p(x1,x2,...,xn)*. *n* representa o número de parâmetros reais em *call p,n*.

     param x1
     param x2
     ...
     param xn
     call p,n

  7. Comandos de atribuição com variáveis indexadas, do tipo *x = y[i]* e *x[i] = y*.

8. Comandos de atribuição com variáveis como endereços e ponteiros, do tipo *x = &y*, *x = *y* e **x = y*.

- Questões relacionadas à escolha de um conjunto de comandos.
- Exemplo de geração de CTE's para comandos de atribuição:

PRODUÇÃO	REGRA SEMÂNTICA
S → id = E	S.code = E.code || gen(id.place "=" E.place);
E → E1 + E2	E.place = newtemp; E.code = E1.code || E2.code || gen(E.place "=" E1.place "+" E2.place); E.place = newtemp;
E → E1 * E2	E.code = E1.code || E2.code || gen(E.place "=" E1.place "*" E2.place); E.place = newtemp;
E → -E1	E.code = E1.code || gen(E.place "=" "-" E1.place); E.place = E1.place;
E → (E1)	E.code = E1.code; E.place = id.place;
E → id	E.code = "";

Código de Três Endereços para Declarações

- Para cada variável local, uma entrada na tabela de símbolos é criada, com o tipo e seu endereço relativo de armazenamento.

- O endereço relativo é calculado através de um deslocamento a partir da área de dados estáticos ou dos dados no registro de ativação.

- A rotina *enter(name, type, offset*) cria uma entrada na tabela de símbolos para *name*, com o tipo *type* e o endereço relativo *offset*.

- Na Figura .2, os inteiros têm tamanho de 4 bytes e os reais de 8 bytes. O tamanho de um *array* é calculado multiplicando-se o tamanho de cada elemento pelo número de elementos. O tamanho de cada ponteiro é assumido como sendo de 4 bytes.

```
P → { offset = 0; } D
D → D ; D
D → id : T              { enter(id.name,T.type,offset);
                          ofset = offset + T.width; }
T → integer                   { T.type = integer;
                          T.width = 4; }
T → real                { T.type = real;
                          T.width = 8; }
T → array [num] of T1   { T.type = array(num.val,T1.type);
                          T.width = num.val * T1.width; }
T → ^T1                 { T.type = pointer(T1.type);
                          T.width = 4;  }
```

Figura .2. Esquema de tradução para geração de CTE's para declarações.

- O esquema de tradução da Figura 3 constrói uma árvore de tabelas de símbolos, sendo uma tabela para cada rotina, dada a possibilidade de geração de subrotinas embutidas.

- São usadas duas pilhas, *tblPtr* e *offset*, que contêm, respecitavemente, ponteiros para as tabelas de símbolos e os próximos endereços de memória local disponíveis, relativos às áreas de dados das subrotinas em análise.

- Durante a compilação, o ponteiro do topo da pilha *tblPtr* aponta para a tabela de símbolos da subrotina em análise. Cada tabela tem um apontador para a tabela da subrotina envolvente.

- Os símbolos não-terminais M e N servem, respectivamente, para gerar a tabela raiz (tabela de símbolos relativa ao programa principal, que contém as variáveis globais), e as tabelas referentes às demais subrotinas. As seguintes funções constituem parte do esquema abaixo:
  - *mktable(ptr)*: gera uma tabela de símbolos (filha da tabela apontada por ptr) e retorna um ponteiro para a tabela gerada;
  - *addWidth(ptr,width)*: armazena na tabela de símbolos apontada por ptr, o tamanho da área de dados local da subrotina correspondente;
  - *enterProc(p1,name,p2)*: armazena na tabela de símbolos apontada por p1 o nome da subrotina e o ponteiro para a tabela correspondente;
  - *enter(ptr,name,type,offset)*: insere na tabela de símbolos apontada por ptr, um novo símbolo, seu tipo e endereço relativo à área de dados local correspondente.

P → M D	{ addwidth(top(tblptr),top(offset)); pop(tblptr); pop(offset); }
M → ε	{ t = mktable(nil); push(t,tblptr); push(0,offset); }
D → D1 ; D2	
D → proc id; N D1; S	{ t = top(tblptr); addwidth(t,top(offset));

	pop(tblptr); pop(offset); enterproc(top(tblptr),id.name,t); }
D → id: T	{ enter(top(tblptr), id.name, T.type, top(offset)); top(offset) = top(offset) + T.width; }
N → ε	{ t = mktable(top(tblptr)); push(t,tblptr); push(0,offset); }

Código de Três Endereços para Atribuições

- A Figura 4 apresenta um esquema de tradução para geração de CTE's para atribuições.

- A função *lookup(id.name)* verifica se existe uma entrada para o identificador na tabela de símbolos. Caso exista, retorna um ponteiro para esta entrada; caso contrário, retorna nil, indicando que o identificador não foi encontrado. A ação semântica *emit* grava CTE's em um arquivo de saída.

S → id = E	{ p = lookup(id.name); if p <> nil then emit(p "=" E.place) else error; }
E → E1 + E2	{ E.place = newtemp; emit(E.place "=" E1.place "+" E2.place); }
E → E1 * E2	{ E.place = newtemp; emit(E.place "=" E1.place "*" E2.place); }
E → -E1	{ E.place = newtemp; emit(E.place "=" "-" E1.place); }
E → {E1}	{ E.place = E1.place; }
E → id	{ p = lookup(id.name); if p <> nil then E.place = p else error; }

Figura 4. Esquema de tradução para geração de CTE's de atribuições.

Conversão de Tipos para Expressões Aritméticas

- Na geração de código intermediário para expressões aritméticas, o tipo dos operandos determina a natureza da operação (por exemplo, adição de inteiros, ou de ponto flutuante) que deve ser aplicada.

- As ações semânticas da Figura 5, associadas à produção soma, fazem verificação de tipo de operandos para determinar o tipo de operação a ser aplicado.

```
E → E1 + E2
{ E.place = newtemp;
if E1.type = integer and E2.type = integer
        then begin
                emit(E.place "=" E1.place "int+" E1.place);
                E.type = integer;
                end
        else if E1.type = real and E2.type = real
                then begin
                        emit(E.place "=" E1.place "real+" E2.place);
                        E.type = real;
                       end
                else if E1.type = integer and E2.type = real
                        then begin
                                u = newtemp;
                                emit(u "=" "inttoreal" E1.place);
                                emit(E.place "=" u "real+" E2.place);
                                E.type = real;
                               end
```

```
else if E1.type = real and E2.type = integer
        then begin
                u = newtemp;
                emit(u "=" "inttoreal" E2.place);
                emit(E.place "=" E1.place  "real+" u);
                E.type = real;
               end
        else E.type = typeError;}
```

Figura 5 Ações semânticas para E → E1 + E2.

. Endereçamento de Elementos de Matrizes

- O endereço do i-ésimo elemento de um vetor A é obtido por:

*base + (i-low) * w*

onde w = tamanho de cada elemento;
low = limite inferior;
base = endereço relativo da memória reservada para o vetor, ou seja, é o endereço A[low].

- A expressão acima pode ser parcialmente avaliada em tempo de compilação se reescrita como: i * w + (base-low*w). A subexpressão c = base-low*w pode ser avaliada quando a declaração do vetor é analisada. O valor dela pode ser armazenado na tabela de símbolos, na entrada para o vetor A, tal que o endereço relativo de A[i] é obtido somando-se (i*w) a c.

- O mesmo tipo de cálculo pode ser aplicado a matrizes bidimensionais, tridimensionais, etc. Uma matriz bidimensional é, normalmente, armazenada de uma das duas seguintes maneiras: por linha ou por coluna.

- Por exemplo, uma matriz A, 2x3, pode ser armazenada por linha como:

A[1,1]	A[1,2]	A[1,3]	A[2,1]	A[2,2]	A[2,3]

Ou por coluna:

A[1,1]	A[2,1]	A[1,2]	A[2,2]	A[1,3]	A[2,3]

- No caso de matriz armazenada por linha, o endereço relativo A[i1,i2] pode ser calculado pela fórmula:

base + ((i1-low1)*n2+i2-low2)*w,

onde low1 e low2 são os limites inferiores;
n2 é o número de elementos da segunda dimensão (n2=high2-low2+1).

- Assumindo que i1 e i2 são os únicos valores não conhecidos em tempo de compilação, a expressão acima pode ser reescrita da seguinte forma:

((i1*n2)+i2) * w + (base – ((low1 * n2)+low2)*w)

cujo último termo pode ser calculado em tempo de compilação.

Geração de CTE's para Expressões Lógicas

- Expressões lógicas são usadas, normalmente, em comandos de atribuição para avaliar variáveis lógicas e em expressões condicionais de comandos de controle de fluxo. Dois métodos de tradução de expressões lógicas são comumente utilizados:
    1. codificação númerica dos valores *true* e *false* (por exemplo, true<>1 e false=0) e avaliar uma expressão lógica tal como uma expressão aritmética;
    2. representação do valor de uma expressão lógica por um ponto a ser atingido no programa usando instruções *if-goto*. Tal método é mais conveniente para implementação de expressões lógicas em comandos de controle.

Representação Númerica de Expressões Lógicas

- Assumindo false=0 e true=1, as expressões são, integralmente, avaliadas da esquerda para a direita. Por exemplo, a expressão *A or B and not C* é traduzida para:

t1 = not C;
t2 = B and t1;
t3 = A or t2;

- A expressão relacional A < B é equivalente ao comando *if A<B then 1 else 0*, sendo traduzida para:

100: if A < B goto 103

101: t1 = 0
102: goto 104
103: t1 = 1
104:...

- O esquema de tradução a seguir inclui ações semânticas para geração de código para expressões lógicas conforme o exemplo acima:

```
E → E1 or E2            { E.place = newtemp;
                          emit(E.place "=" E1.place "or" E2.place);}
E → E1 and E2   { E.place = newtemp;
                          emit(E.place "=" E1.place "and" E2.place); }
E → not E1              { E.place = newtemp;
                          emit(E.place "=" "not" E1.place); }
E → (E1)          { E.place = E1.place; }
E → id1 relop id2 { E.place = newtemp;
                          emit("if" id1.place relop.op id2.place "goto" nextstat + 3);
                          emit(E.place "=" "0");
                          emit("goto" nextstat+2);
                          emit(E.place "=" "1");  }
E → true          { E.place = newtemp;
                          emit(E.place "=" "1"); }
E → false         { E.place = newtemp;
                          emit(E.place "=" "0"); }
```

Figura 6. Esquema de tradução de expressões lógicas usando representação numérica.

Exercício: Gere o CTE para a expressão *A<B or C<D and E<F*, usando o esquema de tradução da Figura 6.

. Expressões Lógicas Traduzidas como Fluxo de Controle

- O esquema da Figura 7 inclui ações semânticas para traduzir expressões lógicas como instruções de fluxo de controle do tipo if-goto. No esquema, os atributos herdados *true* e *false* servem para armazenar rótulos que são gerados pela função *newlabel*. O atributo sintetizado *code* conterá o código gerado para a expressão analisada.

```
E → E1 or E2      { E1.true = E.true;
                    E1.false = newlabel;
                    E2.true = E.true;
                    E2.false = E.false;
                    E.code = E1.code || gen(E1.false ":") || E2.code; }
E → E1 and E2     { E1.true = newlabel;
                    E1.false = E.false;
                    E2.true = E.true;
                    E2.false = E.false;
                    E.code = E1.code || gen(E1.true ":") || E2.code; }
E → not E1              {  E1.true = E.false;
                    E1.false = E.true;
                    E.code = E1.code; }
E → (E1)          {  E1.true = E.true;
                    E1.false = E.false;
                    E.code = E1.code; }
E → id1 relop id2
                {  E.code = gen("if" id1.place relop.op id2.place "goto" E.true)
||
                  gen("goto" E.false);}
E → true          { E.code = gen("goto" E.true); }
E → false         { E.code = gen("goto" E.false); }
```

Figura 7. Esquema de tradução para CTE's de expressões booleanas.

Exercício: Gere o CTE para a expressão *A<B or C<D and E<F*, usando o esquema de tradução da Figura 7. Suponha que para os valores true e false da expressão tenham sido atribuídos os rótulos Ltrue e Lfalse.

Geração de CTE's para Comandos de Controle

- O esquema a seguir inclui ações semânticas para traduzir comandos de fluxo de controle do tipo if-then-else, if-then e while-do.

```
S → if E then S1          { E.true = newlabel;
                            E.false = S.next;
                            S1.next = S.next;
                            S.code = E.code || gen(E.true ":") || S1.code; }
S → if E then S1 else S2  { E.true = newlabel;
                            E.false = newlabel;
                            S1.next = S.next;
                            S2.next = S.next;
                            S.code = E.code || gen(E.true ":") || S1.code ||
                                     gen("goto" S.next) || gen(E.false ":") ||
                                     S2.code; }
S → while E do S1         { S.begin = newlabel;
                            E.true = newlabel;
                            E.false = S.next;
                            S1.next = S.begin;
                           S.code = gen(S.begin ":") || E.code || gen(E.true ":") ||
                                     S1.code || gen("goto" S.begin); }
```

- Para o comando

```
while A < B do
  if C < D
    then X = Y + Z
    else X = Y – Z;
```

a seguinte seqüência de comandos é gerada:

```
L1:   if A < B goto L2
      goto Lnext
L2:   if C < D goto L3
      goto L4
L3:   t1 = Y + Z
      X = t1
      goto L1
L4:   t2 = Y – Z
      X = t2
      goto L1
Lnext: ...
```

Backpatching

- São necessárias três funções para o processo de backpatching:
    - *makelist(i)*: cria uma lista contendo i, índice do vetor de quádruplas, retornado um ponteiro para a lista criada.
    - *merge(p1,p2)*: concatena as listas apontadas por p1 e p2, e retorna um ponteiro para a lista resultante.
    - *backpatch(p,i)*: insere i como rótulo destino para cada um dos comandos da lista apontada por p.

- O esquema de tradução da Figura .8 produz quádruplas a partir de expressões lógicas, durante a análise redutiva (*bottom-up*). Foi acrescentado à gramática de geração de expressões lógicas o símbolo não-terminal M, que tem como objetivo guardar o endereço da próxima quádrupla disponível no momento que M é empilhado.

E → E1 or M E2	{ backpatch(E1.falselist,M.quad); E.truelist = merge(E1.truelist,E2.truelist); E.falselist = E2.falselist; }
E → E1 and M E2	{ backpatch(E1.truelist,M.quad); E.truelist = E2.truelist; E.falselist = merge(E1.falselist,E2.falselist); }
E → not E1	{ E.truelist = E1.falselist; E.falselist = E1.truelist; }
E → (E1)	{ E.truelist = E1.truelist; E.falselist = E1.falselist; }
E → id1 relop id2	{ E.true = makelist(nextquad); E.false = makelist(nextquad + 1); emit("if" id1.place relop.op id2.place "goto _"); emit("goto _"); }
E → true	{ E.truelist = makelist(nextquad); emit("goto _"); }
E → false	{ E.falselist = makelist(nextquad); emit("goto _"); }
M → ε	{ M.quad = nextquad; }

Figura 8. Backpatching para expressões lógicas.

- A seguir é apresentado um esquema de tradução com backpatching em comandos de controle. Dois símbolos não-

terminais M e N, que geram produções vazias, foram introduzidos a fim de gerar informação para a produção do código intermediário. M, através do atributo m.quad, registra o índice da próxima quádrupla disponível. N serve para marcar o fim do bloco *then* em comandos *if-then-else* e gera o *goto* sobre o comando *else*. O atributo *next*, associado a *S* e *L* é um ponteiro para uma lista de desvios para a quádrupla que segue o comando/lista de comandos representado por S ou L.

```
S → if E then M S1          { backpatch(E.truelist,M.quad);
                              S.nextlist = merge(E.falselist, S1.nextlist); }
S → if E then M1 S1 N else M2 S2
                            { backpatch(E.truelist,M1.quad);
                              backpatch(E.falselist,M2.quad);
                              S.nextlist = merge(S1.nextlist,
                              merge(N.nextlist,S2.nextlist)); }
N → ε                       { N.nextlist = makelist(nextquad);
                              emit("goto _"); }
M → ε                             { M.quad = nextquad; }
S → while M1 E do M2 S1     { backpatch(S1.nextlist,M1.quad);
                              backpatch(E.truelist,M2.quad);
                              S.nextlist = E.falselist;
                              emit("goto" M1.quad); }
S → begin L end             { S.nextlist = L.nextlist; }
S → A                       { S.nextlist = nil; }
L → L1; M S                 { backpatch(L1.nextlist,M.quad);
                              L.nextlist = S.nextlist; }
L → S                       { L.nextlist = S.nextlist; }
```

Geração de Código Intermediário para Expressões Booleanas – Fluxo de Controle

- De acordo com a estratégia de geração pelo fluxo de controle, cada expressão é traduzida em uma seqüência de comandos de três endereços, que se constituem em comandos de desvio condicional ou incondicional a um de dois rótulos: *E.true*, aonde a expressão deve levar, caso seja verdadeira e *E.fase*, caso a expressão seja falsa.

- A idéia básica é a seguinte: supondo que E seja *a<b*, o código gerado seria o seguinte:

if a<b goto E.true
goto E.false

- Supondo que E seja *E1 or E2*. Se *E1* é verdadeiro, então sabe-se imediatamente que E é verdadeiro, então *E1.true = E.true*. Se *E1* é falso, então *E2* deve ser avaliado, usando o label *E1.false* para o primeiro comando do código de *E2*. As saídas *true* e *false* de E2 podem ser apresentadas da mesma maneira das saídas *true* e *false* de E, respectivamente.

- De forma análoga, pode-se considerar a tradução da expressão *E1 and E2*. Nenhum código é necessário para a expressão *not E1*, bastando apenas trocar entre si as saídas *E1.false* e *E2.false*.

- Considerando a expressão *A<B or C<D and E<F*, e supondo que as saídas *true* e *false* da expressão estão identificadas pelos rótulos *Ltrue* e *Lfalse*. Utilizando a definição apresentada na apostila, tem-se o seguinte código:

```
        if a < b goto Ltrue
        goto L1
L1:     if c < d goto L2
        goto Lfalse
L2:     if e < f goto Ltrue
        goto Lfalse.
```

Para o seguinte código fonte:

```
while a < b do
  if c < d  then
    x = y + z
  else
   x = y – z
```

tem –se o seguinte código intermediário gerado:

```
L1:     if a < b goto L2
        goto Lnext
L2:     if c < d goto L3
        goto L4
L3:     t1 = y + z
        x = t1
L4:     t2 = y – z
        x = t2
        goto L1
Lnext:
```