Gerador de codigo para Compilador Pascal

Akira Oyama

Este exemplar corresponde à redação da tese defendida pelo Sr. Akira Oyama e aprovada pela Comissão Julgadora.

Orientador: Prof. Dr. Celio Cardoso Guimaraes

Dissertacao apresentada ao Instituto de Matematica, Estatistica e Ciencia da Computacao como requisito parcial para a obtencao do titulo de Mestre em Ciencia da Computacao.

Agosto - 1984

AGRADECIMENTOS

Ao Prof. Celio Cardoso Guimaraes pelo trabalho de orientacao.

Aos colegas e amigos que direta ou indiretamente contribuiram para a realizacao deste trabalho.

'A Solange que apoiou o meu estudo com bastante sacrificio e paciencia.

'A minha filhinha Mayara...

SUMMARY

We present in this thesis a code generator for a Pascal Compiler. The code genarator was written as the second pass of an existent Pascal Compiler, Pascal NBS for PDP 11. The target machine is the INTeL 8088 microprocessor. Our objective is to obtain an efficient code generator in terms of memory.

The results shown in the chapter 6 demonstrate that our objective was reasonably achieved.

RESUMO

O trabalho aqui apresentado e' um gerador de codigo para um Compilador Pascal. O gerador de codigo foi escrito como o segundo passo de um compilador Pascal existente, o compilador NBS para o PDP 11. A maquina objeto e' o microprocessador 8088 da INTEL. O nosso objetivo foi obter um gerador de codigo eficiente em termos de memoria.

Os resultados descritos no capitulo 6 mostram que o nosso objetivo foi razoavelmente atingido.

GERADOR DE CODIGO PARA COMPILADOR PASCAL

SUMARIO

CAPITULO 1

INTRODUCAO

1.1 OBJETIVO DA TESE

O objetivo principal da tese foi desenvolver um eficiente gerador de codigo para um compilador Pascal. O objetivo a longo prazo e' desenvolver um compilador Pascal "residente" num sistema baseado em microprocessador de 16 bits. O microprocessador escolhido para este projeto foi o INTEL 8088. Varios objetivos foram perseguidos neste projeto.

1. O tamanho do codigo do compilador deve ser razoavelmente pequeno. Para o compilador ser residente, o codigo do compilador deve ser compacto. O objetivo deste projeto nesse aspecto e' conseguir um compilador Pascal de dois passos que se auto-compila em cerca de 64 K bytes de memoria.

2. O compilador deve gerar codigo eficiente e compacto. Esta condicao e' relacionada a condicao citada acima, pois o compilador e' auto-compilavel.

3. O compilador deve implementar quase todos os recursos definidos no Pascal por Niklaus Wirth. (Chamaremos de Pascal original, no que se segue, o Pascal definido por Niklaus Wirth no "Revised Report"[Wirt76].) Esta condicao tem uma relacao de compromisso com a condicao (1). A linguagem Pascal-S definida por Niklaus Wirth,por exemplo, e' um subconjunto do Pascal. Pascal-S possui apenas 20 por cento a menos das construcoes do Pascal original e, em consequencia desta simplificacao, consegue 70 % de reducao na complexidade de implementacao[wils79]).

4. A execucao do codigo gerado pelo compilador deve ser rapida. A primeira implementacao do Pascal em micro processador de 8 bits foi o compilador de San Diego, para o INTEL 8080. O codigo gerado por este compilador e' interpretado, e isto substancialmente diminui a velocidade de execucao. Esta condicao esta' relacionada com a condicao (2).

5. O compilador deveria ser desenvolvido em tempo relativamente curto.

6. O codigo gerado deve ser confiavel

Considerando estas condicoes, resolvemos adaptar um compilador ja' existente (compilador Pascal NBS), ao inves de desenvolver um original. O compilador Pascal NBS(National Bureau of Standards) foi desenvolvido por Brian G. Lucas e Justin C. Walker para o National Bureau of Standards, o que de agora por diante chamaremos de compilador Pascal NBS.

1.2 DESCRICAO GERAL DO COMPILADOR PASCAL N+S

O compilador Pascal N+S se destina ao minicomputador PDP 11 sob os sistemas UNIX, RSX11-D, RSX11-M, e RT-11 e apresenta duas caracteristicas muito importantes para o projeto;

1. E' feito em dois passos independentes;

   1. Passo 1
      Faz a analise sintatica descendente recursiva e gera codigo intermediario denominado codigo-P. a ideia basica do codigo-P foi derivada do "P-compiler" desenvolvido por Urs Ammann[Amman78].
      O codigo intermediario e' gerado em arquivos temporarios. Este codigo e' quase independente da maquina.

   2. Passo 2
      A partir do codigo intermediario, constroi a arvore de sintaxe de cada procedimento e gera codigo de maquina para o PDP 11, com algumas otimizacoes locais.

   O compilador N+S requer aproximadamente 59 K bytes de memoria para o Passo 1 e 62 K bytes para o Passo 2, para auto-compilacao no PDP 11.

2. E' auto compilavel no PDP 11. Os dois passos sao escritos em Pascal N+S, exceto a biblioteca de suporte em tempo de execucao. Ele e' capaz de auto-compilar-se em cerca de cinco minutos de CPU do PDP-11/45.

1.3 ESQUEMA GERAL DO TRABALHO

O projeto foi desenvolvido com duas ferramentas fisicas principais:

1. Computador DEC 10 com Pascal de Hamburgo, Montador e Ligador cruzado para o INTEL 8088.

2. Sistema de desenvolvimento do INTEL 8088 do laboratorio de Microprocessadores do centro de computacao(CUECC) e instituto de matematica(IMECC) da UNICAMP. Atualmente o sistema possui 32K bytes de memoria RAM e esta' ligado ao DEC 10 atraves de uma linha serial de 1200 bauds. Um carregador especial permite a carga atraves da linha serial de codigo objeto montado no DEC 10(DOWN LINE LOADING). O microcomputador possui recursos simples para depuracao de programas.

O passo 1 do Compilador Pascal N&S foi modificado para funcionar no PDP-10 com o compilador Pascal de Hamburgo. O Passo 2 do compilador original foi dispensado inteiramente e decidimos desenvolver um outro Passo 2, pelas seguintes razoes:

1. O Passo 2 original e' bastante complicado e as arquiteturas do PDP 11 e INTEL 8088 sao bem diferentes.

2. Como entendemos o funcionamento do Passo 1 muito bem, tinnamos condicao de desenvolver o Passo 2 original.

O passo 2 gera codigo em linguagem de montagem e codigo da maquina. Apenas o codigo em linguagem de montagem foi testado.

A adaptacao do Passo 1 para o sistema DEC 10 foi bastante trabalhosa. Alem disso, a falta de acesso ao PDP 11, dificultou o entendimento do Compilador. As conclusoes citadas posteriormente mostram que conseguimos desenvolver gerador de codigo equivalente, em eficiencia de memoria, ao original.

1.4 MODIFICACOES E ADICOES NO COMPILADOR NBS

Foram feitas algumas modificacoes no Compilador:

1. Adicao de verificacao em tempo de execucao de:

   1. Disponibilidade de memoria na alocacao dinamica e estatica

   2. Limites de indices de vetores

   3. Overflow e Underflow em operacoes aritmeticas

   4. Validade de referencia a area de memoria HEAP

   5. Validade de valor de seletor num comando CASE

2. Adicao de comando HALT e seletor OTHERS no comando CASE. Na execucao do comando HALT, o sistema para a execucao e informa o numero da linha do programa fonte em que este comando foi executado. A partir deste ponto, podemos tambem continuar a execucao com um comando do sistema MONITOR do INTEL 8088.

3. A interface de sistema de exeucao foi modificada. O compilador original usa uma cadeia estatica e uma cadeia dinamica. A cadeia estatica nao e' utilizada porque a passagem de funcao com parametros foi excluida no Pascal NBS. Um vetor de display e' guardado na memoria. O acesso 'a uma variavel global do programa e' feito com enderecamento direto. Para aumentar a eficiencia do codigo gerado, utilizamos o registrador BP como apontador do nivel da rotina corrente na pilha. O acesso a uma variavel ou parametro de nivel lexico intermediario e' feita indiretamente atraves de vetor de DISPLAY na memoria.

   Este metodo e' razoavel em termos de execucao. De acordo com [Wils79] 96 por centos de todos acessos a variaveis e parametros, sao acessos ao nivel global ou nivel corrente.

4. Pascal NBS implementa dois tipos de reais :REAL e LONGREAL. O tipo REAL e' implementado em 32 bits, o tipo LONGREAL e implementado em 64 bits. O nosso compilador implementa apenas um tipo REAL em 16 bits sob forma de logaritmo por razoes praticas. O metodo adotado e' o sistema FOCUS que garante precisao de tres digitos significativos. Apesar desta precisao ser

pobre, esta implementacao tem utilidade em alguns casos especificos. A execucao do calculo de numeros reais com este metodo e' extremamente rapida em comparacao aos metodos convencionais. Alem disso, a implementacao do sistema FOCUS e' muito simples.

5. O compilador aceita 12 opcoes de compilacao. A lista completa de opcoes estao no Apendice C. Algumas opcoes, e.g. TRACE, SELECT, DUMPT, PRINT, sao importantes para verificar o codigo gerado pelo compilador.

CAPITULO 2

DESCRICAO DO PASCAL NBS

2.1 A LINGUAGEM PASCAL

O primeiro Compilador Pascal foi implementado para o computador CDC 6400 em ETH(Zurique) em 1968. Varios outros compiladores Pascal proliferaram rapidamente nos ultimos dez anos. No ano 1978 existiam mais do que 2000 membros dos usuarios de compiladores Pascal em 31 paises.

O sucesso do Pascal e' atribuido a quatro caracteristicas principais da linguagem;

1. E' elaborada para programacao estruturada.

2. A sintaxe e a semantica do Pascal e' simples . Este fato induziu a grande facilidade em implementacao e maior confiabilidade do compilador. Os compiladores Pascal sao bem menores do que os compiladores das linguagens ALGOL 68, PL/I, COBOL, etc; e geram codigo mais compacto.

3. Permite declaracoes dos tipos de variavel, parametro e arquivo.

4. Possui alocacao dinamica de memoria.

Durante este periodo, varios dialetos do Pascal surgiram. Como a linguagem nao foi oficialmente padronizada, varios compiladores foram implementados para dialetos de Pascal. Ha' um movimento para padronizar o Pascal a nivel internacional[Reve79].

Surgiram varios sugestoes de adicionar mais recursos para o Pascal;

1. Seletor "OTHERS" no comando CASE.

2. Omissao do comando GOTO e rotulos.

3. Declaracao de constante estruturada.

4. Melhor formatacao dos arquivos INPUT e OUTPUT.

5. Passagem de parametro do tipo vetor com tamanho variavel(matrizes dinamicas).

6. Declaracao CWN como definida em ALGOL.

7. Extensao de funcoes para leitura e escritura.

8. Acesso aleatorio a arquivos.

9. Tipo "Cadeia de Caracteres" e operacoes correspondentes.

[Rich77].

2.2 DIFERENCA ENTRE PASCAL NBS E PASCAL ORIGINAL(ETH)

Ha' tres diferencas principais entre Pascal NBS e Pascal original. Estas diferencas sao derivadas de restricoes na arquitetura da maquina objeto PDP11. O PDP 11 possui apenas 64 K bytes de espaco logico de memoria. Quase todas as implementacoes do Pascal para microprocessadores enfrentam esta barreira e a solucionam de varias maneiras. Uma das solucoes mais comuns e' "OVERLAY". As tres diferencas principais sao as seguintes;

1. Constantes Estruturadas
   Compiladores usam varias tabelas relativamente grandes. Estas tabelas sao, de modo geral, do tipo estruturado, isto e', tipo registro ou tipo vetor de registros. A unica maneira de inicializar este tipo de constante de sintaxe do Pascal original e' fazer a atribuicao de cada elemento da constante em tempo de execucao. Esta maneira seria muito ineficiente em termos de memoria porque a memoria e' gasta para gerar codigo de atribuicao para inicializacao da Constante Estruturada. A ideia de Constante Estruturada no Pascal NBS e' tratar esta inicializacao em tempo de compilacao, de maneira mais elegante do que, por exemplo, o artificio de INITIPROCEDURE adotado no Pascal do DEC 10. O exemplo abaixo mostra a maneira de declarar Constantes Estruturadas no Pascal NBS.

Exemplo:

```
TYPE    elem = RECORD
                 n : ARRAY[ 1..2 ] OF char;
                 b : boolean;
                 i : integer
               END;

TYPE    cons = ARRAY[ 1 .. 2 ] OF elem;

VAR     vet : cons;

CONST
  tabela =
cons(('AB',true,1),('CD',false,2));

BEGIN
```

```
    vet:=(('XL',false,3),('XY',true,4));
END;
```

Em consequencia desta modificacao, a regra de ordem de declaracoes foi violada. O Pascal N&S permite numero variavel de declaracoes, em qualquer ordem, sendo os identificadores usados numa declaracao definidos anteriormente, exceto o identificador de tipo usado para declaracao de apontadores.

Esta sintaxe possui outras vantagens: alem de economizar a memoria para codigo, podemos agrupar declaracoes de identificadores de maneira mais conveniente, o que seria util para o melhor entendimento do programa.

Alem das Constantes Estruturadas, Pascal N&S permite declaracao de constante cujo valor e' uma expressao constante.

2. Comando LOOP

O comando GOTO e' omitido no Pascal N&S. Esta decisao foi tomada elaboradamente para simplificar o compilador. De acordo com Wilson[Wils79] , a linguagem Pascal-S possui 20 por cento menos de recursos da linguagem Pascal original e conseguiu retirar 70 por cento de complexidade em implementacao. O comando GOTO nao e' essencial do ponto de vista de programacao sistematica. O comando LOOP foi adicionado para enriquecer os recursos de programacao sistematica e, em certo sentido, para compensar a ausencia do comando GOTO.
A sintaxe do comando LOOP e' seguinte;

```
<comando LOOP> ::= LOOP
                   <seqcomando>
                   <seqexit>
                   END

<seqcomando> ::= <comando> { ;  < comando > }

<seqexit> ::= <partexit> !  <partexit> <seqexit>

<partexit> ::= <exit> !  <exit> { ;  <seqcomando> }

<exit> ::= EXIT IF <expbool> !
```

EXIT IF <expbool> THEN <comando>

O comando LOOP e' um comando repetitivo similar aos comandos REPEAT e WHILE, exceto que a condicao de saida pode estar em qualquer lugar do laco de repeticao. Note que pode existir mais do que um "EXIT" dentro do "LOOP", o que e' diferente da implementacao do comando LOOP no Compilador Pascal de Hamburgo, implementado no sistema DEC 10, e existem duas formas de saida do "LOOP". A semantica da segunda forma de saida do LOOP e', se a expressao booleana for verdade entao executar o comando que segue o simbolo "THEN" primeiro e sair do LOOP. O comando LOOP resolve um problema tipico das operacoes OR e AND em Pascal. Nas operacoes OR e AND, os dois operandos sao calculados incondicionalmente. Existem muitos casos em que nos nao queremos calcular o valor do segundo operando dependendo do valor do primeiro operando.

EXEMPLO :

```
i:=1;
WHILE (i <= 10) AND (v[i]<>n) DO
i:=i+1;
```

Repare que quando o valor de i for 11, o valor da subexpressao (i <= 10) fica falso e refere-se a um elemento de vetor que esta' fora do limite de indice declarado(Supondo que o vetor v esteja declarado como ARRAY[1..10] OF INTEGER). Este trecho do programa pode ser reescrito elegantemente usando o comando LOOP.

```
i:=1;
LOOP
  EXIT IF  i> 10 THEN writeln('Nao existe');
  EXIT IF v[i]=n THEN writeln('Existe');
  i:=i+1
END;
```

3. Extensao de comandos para alocacao dinamica de memoria

Pascal NBS possui os comandos MARK e RELEASE que permitem fazer a coleta de lixo de memoria(Garbage Collection) simples. O comando DISPOSE no Pascal de Hamburgo e' equivalente ao comando RELEASE no Pascal NBS.

Exemplo:

```
VAR     apont : @integer;
BEGIN                       BEGIN
  .                           .
  .                           .
  NEW(apont);                 MARK;
  .                           .
  .                           .
  DISPOSE(apont);             RELEASE;
  .                           .
  .                           .
END;                        END;
```

Alem dos comandos MARK e RELEASE,o Pascal NoS possui uma terceira forma do comando NEW. As outras duas formas sao iguais as formas definidas no Pascal original. A terceira forma e' utilizada para economizar a memoria do tipo vetor de registros alocada de forma que somente os N primeiros elementos do vetor declarado neste tipo registro sao necessarios na alocacao.

Exemplo:

```
TYPE
  id = RECORD
         l : integer;
         str : ARRAY[1..80] OF char
       END;

VAR
  p : @id;       i, n : integer;

BEGIN
  n:=15;
  NEW(p,n);
  WITH p@ DO
  BEGIN
    l:=n;
    FOR ii TO n DO str[i]:=' '
  END
END;
```

Na chamada do comando NEW(p,n) do exemplo, apenas 15 posicoes do vetor str sao alocados. O seguinte parametros da chamada pode ser qualquer expressao. Este recurso e' indispensavel para montagem de arvore de sintaxe de cada procedimento a partir do Codigo-P feita no Passo 2 do Compilador NBS.

Existem algumas diferencas de sintaxe miscelaneas entre o Pascal NBS e o Pascal original:

1. Numa declaracao de um vetor, os valores inferior e superior de indice podem ser declarados com expressao constante.

2. Os tipos dos parametros de procedimento ou funcao podem ser declarados na propria declaracao de procedimento ou funcao.

3. Constante estruturada pode ser utilizada numa atribuicao.

Exemplo:

```
TYPE    t = RECORD
              a : char;
              b : boolean
            END;

VAR     x : t;

BEGIN
  x:=t('P',true)
END;
```

2.3 Limitacoes na implementacao do Compilador Pascal NBS

O compilador Pascal NBS possui uma serie de limitacoes como outros Compiladores:

1. Os primeiros 15 caracteres num identificador sao reconhecidos.

2. O numero maximo de procedimentos declarados num programa e' 150.

3. A profundidade maxima de encaixamento de procedimento e' 15.

4. Os valores de seletores num comando CASE podem estar somente no intervalo de 0 .. 255.

5. A palavra "Packed" e' ignorada.

6. A declaracao de parametros do programa principal(tipo arquivo) e' ignorada.

7. O tipo CHAR e' implementado em 8 bits e cobre o conjunto completo de ASCII que e' diferente da implementacao do Compilador Pascal no DEC 10.

8. O numero maximo de elementos num conjunto e' 16. Esta limitacao causa um problema serio do compilador NBS. Na maioria dos casos, recuperacoes de erros do Compiladores sao implemantadas usando os recursos de conjunto cuja cardinalidade e' cerca de 60 a 70. Portanto, recuperacao de erros do Compilador NBS e' muito pobre , isto dificulta muito o uso do compilador na pratica.

CAPITULO 3

DESCRICAO DO CODIGO-P

O Codigo-P definido no compilador Pascal NBS e' um tipo de codigo intermediario utilizado em compilacao. Existem varios tipos de codigos intermediarios. Quanto mais o codigo intermediario estiver proximo do codigo da maquina, a geracao de codigo final, a partir desse codigo, ficara mais facil. Por outro lado, este tipo de codigo possui apenas informacoes locais e isto dificulta a otimizacao do codigo objeto final.

O metodo mais convencionalmente adotado por compiladores de dois passos e' definir cada codigo intermediario como uma instrucao de uma maquina ficticia, orientada a pilha. [Amma79] As vantagens deste metodo sao: a facilidade na geracao de codigo e confiabilidade do codigo gerado.

O Codigo-P do Compilador Pascal NBS nao e' um codigo interemediario deste tipo, mas e' uma imagem da arvore de sintaxe do programa fonte, que o Passo 1 analizou e codificada linearmente na memoria. O Passo 2 monta a arvore de sintaxe explicitamente de cada procedimento a partir da sequencia do Codigo-P antes de gerar o codigo final. Esta arvore de sintaxe facilita a otimizacao de codigo. A relacao entre o Codigo-P e a arvore de sintaxe e' explicada posteriormente. Teoricamente a otimizacao de codigo a partir do Codigo-P convencional e' possivel [Coli], [Corn].

Alem do mais, a exploracao das caracteristicas de operacao, por exemplo, as associativas, causam problemas de "overflow", "underflow" e precisao .
O Codigo-P convencional de uma expressao seria a cadeia de operandos e operacoes em Pos-ordem da arvore equivalente, porque o numero de operandos de cada operacao aritmetica e logica e' sempre fixo. Existem alguns estudos muito

interessantes feitos sobre otimizacao do Codigo-P convencional.

3.1 CODIGO-P E ARVORE DE SINTAXE

Normalmente arvore de sintaxe nao e' uma arvore binaria mas arvore livre. Arvore livre e' uma arvore em que cada no' da arvore possui numero variavel de subarvores. Numa arvore que representa uma expressao aritmetica e logica, o numero de subarvores de cada no' e' constante. Porem existem varios tipos de arvores em que este numero de subarvore e' variavel. Um comando composto, por exemplo, pode ter numero nao negativo qualquer de sub-comandos.
A partir do Codigo-P convencional, i.e. a sequencia de nos da arvore em Pos-ordem, nao podemos reconstruir a arvore original. Precisamos acresentar mais informacao ao Codigo-P para reconstruir a arvore. Esta informacao seria o numero de subarvores de cada no'. A cada Codigo-P esta' associado o numero de subarvores do no'.

No inicio do Passo 2, e' executado um algoritmo de reconstruir a arvore de sintaxe a partir de Codigo-P, que e' relativamente simples. Uma pilha de apontadores e' usada neste algoritmo.

Algoritmo :

1) Ler um codigo-P e alocar memoria para o codigo

2) Se o numero de subarvore deste codigo for zero empilha-lo no topo da pilha.
Se nao, retirar o numero de subarvores deste codigo do topo da pilha, coloca-los como subarvores do codigo atualmente lido e colocar este codigo na pilha.

3.2 IMPLEMENTACAO DE CODIGO-P NO PASCAL NBS

Cada no' da arvore de sintaxe montada pelo Passo 2 e' definido da seguinte maneira:

```
CONST

  maxarg = 255;

TYPE

  byte = 0 .. 255;

  ptr = ^ node;

  node = RECORD
           code : byte;
           size : byte;
           disp : RECORD
                  CASE boolean OF
                    false : (disp : integer);
                    true  : (xval : fvalptr)
                  END;
           segnr : byte;
           nrarg : byte;
           arg   : ARRAY[1..maxarg] OF ptr
         END;
```

Cada campo de tipo NODE tem seguinte sentido:

1. code
   a numeracao do Codigo-P. A tabela completa de codigos implementados no compilador esta' no apendice A.

2. size
   Para variaveis ou parametros, o tamanho de cada elemento indicado pelo no' e' indicado neste campo. Para operacoes aritmeticacas e logicas, o tamanho do resultado calculado e' guardado. Para o Codigo-P INDEX que indica um endereco indexado, tamanho de cada elemento de vetor e' indicado.

   disp
   Para variavel ou parametro do tipo scalar, este campo indica o deslocamento. Para valor literal o valor mesmo e' armazenado. Este campo esta' declarado como RECORD do tipo campo variante por uma razao. Numeros reais estao representados em 32 ou 64 bits. Para economizar o espaco de memoria o subcampo fvalptr aponta o endereco na area de HEAP onde este valor esta' armazenado.

3. segnr
Para variavel ou parametro, este campo indica o nivel lexico.

4. nrarg
Este campo indica o numero de subarvores este no' possui. Para codigos comuns isto nao e' necessario. Porem para os codigos SEG(Comando composto), LOOP etc, precisamos indicar o numero de subarvores.

5. arg
Os apontadores de raiz de subarvores do codigo estao armazenados neste campo. Repare que o tamanho de vetor declarado de 255. Para poder alocar apenas o numero de bytes necessarios para cada codigo, uma modificacao de sintaxe do comando NEW foi feita. Quando um comando composto tiver subcomandos maior do que de 255, o Passo 1 forca fechar os primeiros 255 subcomandos como um comando composto ficticio.

A tabela completa de codigo-P esta' no apendice A.

CAPITULO 4

ADAPTACAO DO COMPILADOR PARA A MAQUINA OBJETO

4.1 CARACTERISTICAS RELEVANTES DO INTEL 8088

O microprocessador INTEL 8088 possui algumas caracteristicas favoraveis para geracao de codigo para a linguagem Pascal.

1. E' uma maquina hibrida com operandos de um e dois bytes, isto e', possui instrucoes com operandos de tamanho de um e dois bytes. Esta caracteristica e' extremamente importante, porque o proprio Pascal define tipos de tamanho de um e dois bytes. Em comparacao a microprocessadores de 8 bits, os microprocessadores hibridos de um e dois bytes possuem instrucoes mais poderosas. O tamanho de codigo gerado para o microprocessador MOTOROLA 6800, que e' maquina de 8 bits para o comando, por exemplo, A[i]=B[i], fica 52 bytes, enquanto o codigo gerado para o microprocessador MOTOROLA 6809, que e' maquina hibrida, para o mesmo comando fica apenas 20 bytes[Fors79]. No caso do nosso compilador o tamanho do codigo gerado para este comando e' 20 bytes. Implementacao de Compilador Pascal para uma maquina de 8 bits ficaria bastante limitada especialmente em termos de espaco de memoria. Esta e' a razao principal porque o primeiro Compilador Pascal para o microprocessador INTEL 8088(Compilador de San Diego) gera codigo interpretado.

2. Possui o espaco de memoria suficientemente amplo para trabalhar com o Compilador Pascal; permite a configuracao de um megabyte de memoria.

3. Possui varias instrucoes poderosas em relacao a maquina de um byte.
   Ha' tres grupos de instrucoes muito uteis na geracao de codigo para Compilador Pascal.

1. Grupo 1 : Instrucoes aritmetica e logica
   INTEL 8088 possui instrucoes de divisao e multiplicacao com sinal e sem sinal. Os operandos podem ser de tamanho de um byte e de dois bytes. Para ajustar o tamanho dos operandos, existem duas instrucoes: CBW e CWD.

2. Grupo 2 : Instrucoes para controle de lacos.
   Existem duas instrucoes que fazem controle de laco explicitamente por codigo: LOOP e REP. Estas instrucoes sao equivalentes aos comandos FOR, REPEAT e WHILE em Pascal. Porem nao usamos estas instrucoes na geracao de codigo em geral, porque estas instrucoes requerem o uso especifico do registrador CX. Normalmente o registardor CX e' utilizado na execucao dos comandos dentro dos comandos FOR, REPEAT, e WHILE. Alem disso, encaixamento dos comandos repetidos na linguagem Pascal nao podem ser implementados atraves destas instrucoes repetitivas do INTEL 8088. As instrucoes repetitivas sao usadas nas subrotinas de suporte de execucao.

3. Grupo 3 : Instrucoes para manipulacao de cadeia de caracteres.
   Estas instrucoes sao extremamente uteis para gerar codigo de comandos do tipo A=B onde A e B sao vetores ou do tipo registro. Uma comparacao de vetores, tambem, pode ser implementada por este tipo de instrucao. Outro exemplo e' passagem de vetor como parametro com valor.

4. Nao ha' problemas de alinhamento de palavra como no caso do PDP 11. O compilador NBS original e' obrigado fazer alinhamento. Como INTEL 8088 nao tem este problema, o nosso compilador pode aproveitar a memoria melhor.

5. Possui instrucoes aritmeticas com valores literais operando diretamente sobre a memoria.
   Estas instrucoes sao:
   ADD, SUB, INC, DEC, NEG, OR, AND, NOT, etc.
   Elas podem ser usadas para geracao de codigo de comandos como os seguintes:

```
i:=succ(i);
A[i,j]:=A[i,j] - 10;
p@.x:=-p@.x;
```

   Todos esses comandos sao comandos de atribuicao. Este tipo de atribuicao e' especifico porem a frequencia de atribuicoes deste tipo em programas escritos em Pascal nao e' insignificante. Descricao detalhada do uso de instrucoes deste tipo e' feita na parte de geracao de codigo.

O INTEL 8088 apresenta tambem varios problemas:

1. Os registradores nao sao de proposito geral

2. As instrucoes de salto condicional tem apenas um byte para representar o deslocamento relativo de salto.

3. Nao existe instrucoes de PUSH de valores literais. Este tipo de instrucoes sao desejaveis para gerar o codigo de chamada de procedimento ou funcao com parametros passados por valor do tipo literais.

Observamos que o Compilador nao utiliza todas as instrucoes do INTEL 8088. Apenas 80 % delas sao utilizadas. As instrucoes nao utilizadas sao principalmente as instrucoes relacionadas com funcoes do sistema operacional, e.g. HLT, WAIT, ESC, PUSHF, POPF,etc[Fors79], [Hart80].

4.2 ALOCACAO DE REGISTRADORES

O INTEL 8088 possui 12 registradores, usados da seguinte forma:

1. AX (AH + AL) :
   Usado para calculo de expressoes.

2. CX (CH + CL) :
   Idem ao AX. Mas usado para armazenar os valores temporarios de calculo de operacoes binarias. Tambem e' usado para os comandos repetitivos.

3. BX (BH + BL) :
   Usado para carregar o endereco efetivo de variaveis ou parametros.

4. DX (DH + DL) :
   Usado apenas para divisao e multiplicacao.

5. SP :
   Apontador de pilha.

6. BP :
   Usado para o apontador de nivel do procedimento ou funcao corrente na pilha(LOCAL POINTER).

7. DI :
   Registrador de uso auxiliar. Usado tambem para operacao do tipo transferencia de cadeias.

8. SI :
   Idem ao DI.

4.3 ENDERECAMENTO E ALOCACAO DE MEMORIA

4.3.1 Alocacao De Memoria Para Cada Tipo

Alocacao de memoria para tipo e' feita em bytes no Compilador Pascal NBS. Como o microprocessador INTEL 8088 e' maquina que endereca a nivel de byte, compactacao em alocacao de memoria nao teria sentido. Portanto todas as palavras PACKED no programa fonte sao reconhecidas, mas ignoradas pelo compilador. Existe um detalhe neste caso, o tipo CHAR e' representado por um byte no Compilador NBS enquanto um caracter no vetor compactado ou arquivo compactado do tipo char do Compilador Pascal de Hamburgo e' representado por 7 bits. A definicao do tipo CHAR no Pascal NBS contem o conjunto ASCII enquanto a definicao do Pascal de Hamburgo e' um subconjunto de ASCII. Em muitos casos, o tipo CHAR do Pascal NBS e' utilizado como se fosse um tipo INTEGER de um byte(e.g. SHORTINTEGER do MODULA). A seguinte tabela mostra alocacao de memoria para cada tipo:

SCALAR.....	1 byte
BOOLEAN.....	1 byte
CHAR.....	1 byte
INTEGER.....	2 bytes
LONGINTEGER...	4 bytes
REAL.....	2 bytes
LONGREAL.....	2 bytes
SET.....	1 ou 2 bytes
POINTER.....	2 bytes
FILE.....	522 bytes

O tipo LONGINTEGER definido no Pascal NBS nao foi implementado. O tipo SET pode ser representado por um byte ou dois bytes, dependendo a cardinalidade do conjunto. A cardinalidade maxima de um conjunto e' 16 no Pascal NBS. Todas as entradas e saidas do programa sao feitas atraves do terminal, apenas. Os 512 bytes do tipo FILE sao utilizados como BUFFER de entrada ou saida e 10 bytes sao utilizados para representar o estado do BUFFER do arquivo,e.g. apontador no buffer, estado de ECLN e EOF, tamanho de cada registro do arquivo, etc. O tipo definido pelo RECORD depende dos tipos de subcampos. Como o Compilador NBS faz alinhamento de enderecos devido ao problema do PDP 11, a orderm de declaracao de subcampo influencia o espaco de memoria alocado para este tipo RECORD, mesmo definindo os mesmos subcampos de tipos iguais. O tipo REAL e' tratado de maneira bem diferente aos outros tipos. Uma variavel do tipo REAL, por exemplo, nao representa seu valor, mas aponta o endereco onde este valor esta' representado na area de

HEAP por 4 bytes. O tipo LONGINTEGER e' representado por 8 bytes na area de HEAP.

4.3.2 Enderecos De Constantes, Variaveis E Parametros.

1. Constantes

As constantes simples, e.g. integer, boolean, char, real, sao guardados na tabela de simbolos do Passo 1 do compilador. Todas as constantes do tipo estruturado (incluindo o tipo cadeia) sao agrupadas e guardados numa area reservada na memoria, e eles sao associados a deslocamento relativo a partir do inicio desta area. Portanto o acesso a uma constante e' sempre direto. O compilador Pascal NBS original gasta um pouco mais memoria para guardar as constantes, porque o PDP 11 tem problema de alinhamento.

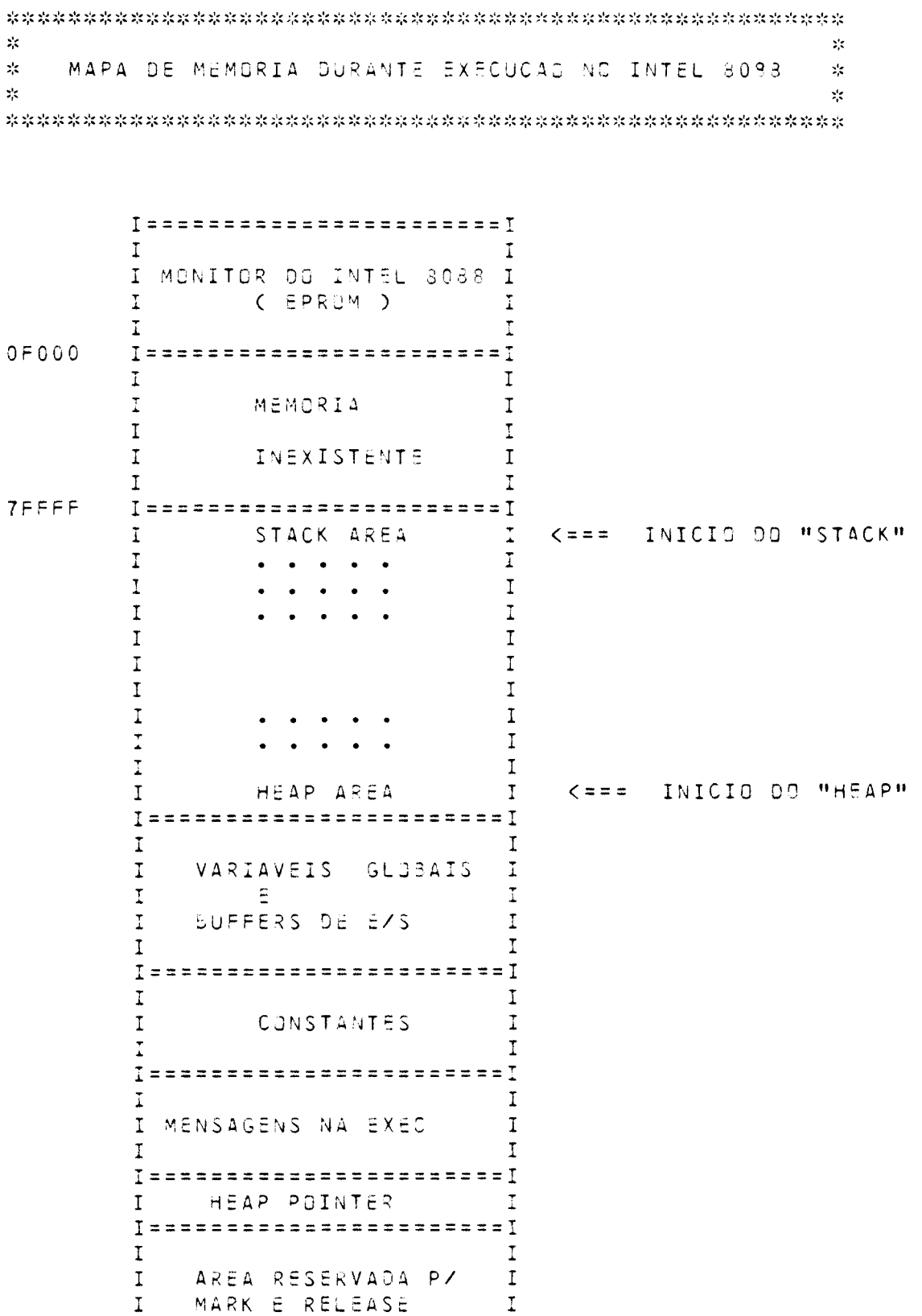
2. Variaveis

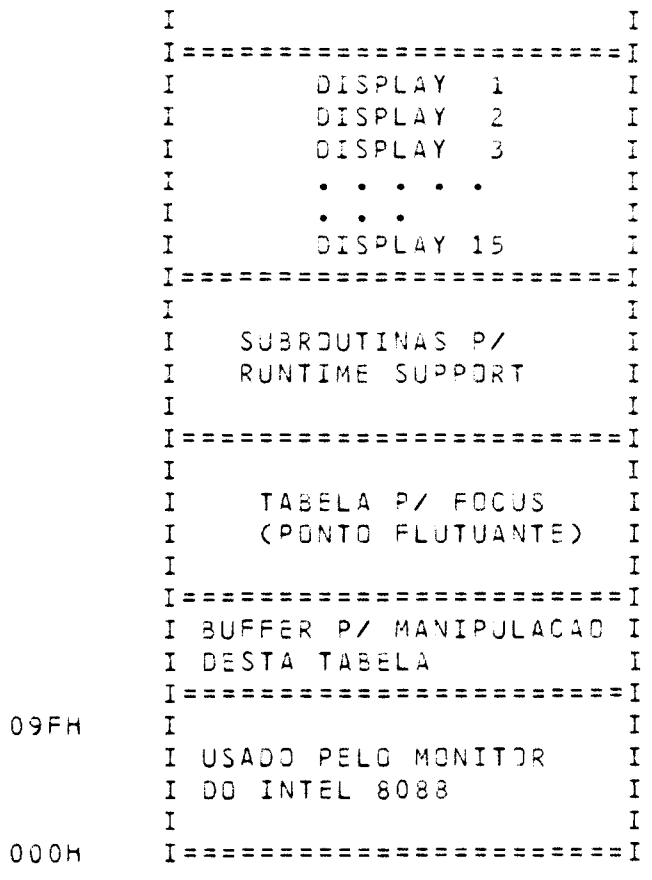
Variaveis e parametros sao enderecados por dois argumentos; nivel corrente da rotina e deslocamento relativo do nivel corrente da pilha. Para aumentar a eficiencia na geracao de codigo, variaveis globais do programa(nivel lexico 1) sao acessadas diretamente. A memoria para variaveis globais e' estaticamente alocada e desalocada apenas no momento em que o programa termina. Como a frequencia com que variaveis globais sao referenciadas e' bastante alta em relacao as variaveis locais de rotinas, este tipo de otimizacao e' muito importante. As variaveis globais sao enderecadas com deslocamento positivo e as variaveis locais de rotinas sao enderecadas com deslocamentos negativos enquanto todos os parametros, a menos os parametros do programa fonte do tipo arquivo, que sao ignorados pelo Compilador, sao enderecados com deslocamentos positivos.

3. Parametros

Os parametros sao enderecados com deslocamentos positivos. Como no caso de variaveis, os enderecos sao alinhados. Este alinhamento foi intencionalmente deixado porque o INTEL 8088 nao possui a instrucao PUSH de um byte. Se compactar o enderecamento de parametros, a geracao de codigo para passagem de parametros ficaria muito cara. O parametro do tipo vetor passado por valor nao e' empilhado inteiramente na pilha na chamada, mas e' passado apenas o endereco inicial do vetor. Este tipo de parametro e' tratado como se fosse uma variavel local da rotina que esta sendo chamada e seus valores de vetor sao copiados na entrada da rotina.

4.4 MAPA DE MEMORIA




DISPLAY 1
DISPLAY 2
DISPLAY 3
DISPLAY 15
SUBROUTINAS P/
RUNTIME SUPPORT
TABELA P/ FOCUS
(PONTO FLUTUANTE)
BUFFER P/ MANIPULACAO
DESTA TABELA
09FH
USADO PELO MONITOR
DO INTEL 8088
000H

4.5 ROTINAS DE SUPORTE DURANTE EXECUCAO

Existem 38 rotinas INTRINSECAS:

Tabela de rotinas INTRINSECAS

1 .. Get	2 .. Put	3 .. Break	4 .. Position
5 .. Reset	6 .. Rewrite	7 .. Update	8 .. Read
9 .. Readln	10 .. Write	11 .. Writeln	12 .. Eof
13 .. Eoln	14 .. New	15 .. Free	16 .. Mark
17 .. Release	18 .. Pred	19 .. Succ	20 .. Any
21 .. All	22 .. Odd	23 .. Ord	24 .. Chr
25 .. Float	26 .. Round	27 .. Max	28 .. Min
29 .. Ceil	30 .. Floor	31 .. Abs	32 .. Sqr
33 .. Sqrt	34 .. Ln	35 .. Exp	36 .. Sin
37 .. Cos	38 .. Arctan		

O numero de cada rotina coresponde 'a sua numeracao interna no compilador NBS. As rotinas POSITION e UPDATE sao usados para implementar arquivos de acesso randomico(RANDOM ACCESS FILES). Este recurso nao esta' implementado.
A funcao FLOOR(m,n:integer):integer e' definida da seguinte maneira:

```
FLOOR:=m DIV n * n
```

A funcao CEIL(m,n:integer):integer e' definida da seguinte maneira:

```
CEIL :=(m + (n-1)) DIV n * n
```

As funcoes FLOOR e CEIL sao utilizadas para o calculo de numero de bytes alocados para cada tipo(especialmente tipo indexado).

No momento, entrada e saida de arquivos nao estao bufferizadas, portanto a rotina BREAK, GET, PUT, RESET, REWRITE, EOF nao estao implmentadas.
A funcao ANY(s:tipo conjunto):tipo scalar, devolve o primeiro elemento do tipo scalar que pertence ao parametro do tipo conjunto.
Todas as funcoes do tipo real nao estao implementadas tambem.

As rotinas implementadas estao detalhadas no Apendice B. As rotinas de suporte durante execucao estao escritas em linguagem de montagem do INTEL 8088. Uma maneira de carregar as rotinas e' montar as rotinas independentemente e pedir ao carregador para carregar no endereco predefinido.

Como o sistema INTEL 8088 esta' monoprogramado por enquanto, esta maneira funcionaria bem. Porem decidimos copiar as rotinas que estao escritas em linguagem de montagem para a saida do Passo 2 no inicio da execucao do Passo 2 para evitar erros causados pelo carregador. Uma maneira mais eficiente de copiar as rotinas seria copiar apenas as rotinas que sao necessarias para execucao do programa. Porem o Passo 2 copia todas as rotinas no momento.

CAPITULO 5

OTIMIZACAO E GERACAO DE CODIGO

5.1 EXPRESSOES

Uma maneira natural de representar uma expressao e' em notacao polonesa. A notacao polonesa possui recursividade implicita. O mecanismo para calcular uma expressao em notacao polonesa usa implicitamente uma pilha. Existem alguns computadores orientados a pilha como Burroughs 6800. Ha' tres metodos popularmente adotados para gerar codigo a partir de uma expressao polonesa. A ideia basica dos tres metodos e' implementar em software as operacoes de pilha que o hardware do computador nao possui.

Metodo 1 : Gerar codigo que e' interpretado numa linguagem implementada no computador. Este metodo e'mais facil e adotado em varios compiladores com o Compilador Pascal de San Diego(UCSD). A vantagem deste metodo e' economia de memoria especialmente quando implementado para microprocessadores de 8 bits. Geracao de codigo de maquina para operacoes com operandos de 16 bits para este tipo de processador ficaria cara em termos de memoria[Ullm70]. A desvantegem principal deste metodo e' o tempo de execucao bastante demorado. Normalmente, execucao de um programa interpretado leva tempo na faixa de dez a cinquenta vezes maior do que o programa equivalente executado por codigo compilado. Portanto, se um compilador for implementado com este metodo ele levaria muito tempo para se auto-compilar.

Metodo 2 : Implementar as operacoes de pilha atraves de macros em linguagem de montagem. Este mtodo e' superior ao Metodo 1 em termos de tempo de execucao. Porem o codigo gerado fica muito grande. Este metodo e' impraticavel quando implementado num microprocessador com espaco de memoria restrito.

Metodo 3 : Implementar as operacoes de pilha atraves de chamadas de subrotinas. Esta tecnica e' chamada THREADED CODE. Este metodo e' vantajoso em relacao ao Metodo 2 em termos de memoria. Por outro lado, o calculo de expressao leva mais tempo do que o Metodo 2 devido a passagem de parametros, desvio para chamada e retorno de cada subrotina[Bell73].

O metodo adotado e' diferente dos tres, porque todos eles nao resolvem o problema basico de que a expressao polonesa e' destinada a maquina orientada a pilha e a nossa maquina e' uma maquina de um endereco(ONE-ADDRESS-MACHINE). O problema que queremos resolver e' gerar codigo de maquina eficiente para expressoes para uma maquina de um endereco. A nossa solucao e' deixar os valores do topo da pilha em registradores em vez de na propria pilha.

Geracao de codigo e' feita atraves da arvore de sintaxe explicitamente montada. Os valores do topo da pilha representam os valores das subexpressoes correntemente calculadas. Sendo estes valores guardados em registradores, operacoes sobre os valores do topo da pilha podem ser feitas diretamente.

O microporcessador INTEL 8088 possui cinco formas de operacoes basicas:

A - op [reg1 , reg2] => reg1
B - op [reg1 , memoria] => reg1
C - op [reg1 , literal] => reg1
D - op [memoria , reg] => memoria
E - op [memoria , literal] => memoria

As tres primeiras formas sao importantes na geracao de codigo para expressoes. As outras formas sao utilizadas na geracao de codigo para atribuicoes, o que sera explicado posteriormente. Nem todas as formas sao permitidas para todas as operacoes no INTEL 8088. Este problema sera detalhadamente discutido no fim da descricao de geracao de codigo para expressoes.

Sendo os valores de TOPO - 1 e TOPO da pilha guardados nos registradores reg1 e reg2 respectivamente, uma operacao binaria da forma A emula uma operacao binaria na pilha. As operacoes de formas B e C sao equivalentes a' forma A quando o valor de reg2 contem um valor de variavel simples ou valor literal. Uma operacao binaria da forma B, por exemplo, poderia ser feita com a sequencia de operacoes:

reg2 <= memoria

op[reg1 , reg2] => reg1

A vantagem de operacoes das formas B e C sobre a sequencia acima e' nao precisar empilhar o segundo operando, i.e. carregar o operando em registrador.

Apenas dois registradores AX e CX sao usados para o calculo de expressoes pelas seguintes razoes:
Os registradores BX, BP, SP sao reservados para calculo de endereco, apontador de nivel da pilha da rotina corrente, e topo da pilha respectivamente. O registrador DX nao e' apropriado para armazenar valores porque o seu valor e' alterado depois de uma operacao de multiplicacao ou divisao quando os operandos forer de dois bytes. Os registradores DI e SI nao permitem operacoes sobre byte e sao usados para operacoes de vetores ou cadeia de caracteres; alem disso eles sao usados para fins auxiliares. Como multiplicacao de AX por um valor literal nao e' permitido no INTEL 8088, o valor literal e' carregado primeiramente no registrador DI ou SI o que acontece conummente no calculo de endereco de matrizes.

Em muitos compiladores, alocacao de registradores para cada operacao e' feita atrazes de analise da expressao antes de gerecao de codigo final. Este metodo nao e' facilmente aplicavel pelas razoes citadas anteriormente. Portanto adotamos alocacao de registradores da maneira mais simples para obter maior confiabilidade no codigo gerado.

5.1.1 Algoritmo Geral De Otimização E Geração De Código

[Algoritmo para geração de código para uma árvore de expressão usando N registradores de propósito geral]

Primeiramente o algoritmo para N registradores "gerais" é citado e explicamos a adaptação deste algoritmo para o nosso caso particular.
A palavra "geral" possui os seguintes sentidos:

1. Qualquer registrador pode ser usado para todas as operações das formas A, B, e C.

2. Todas as operações binárias permitem as três formas A, B, e C.

3. Todos os registradores permitem as operações binárias sobre um e dois bytes.

Em primeiro lugar, precisamos introduzir o algoritmo descendente recursivo de rotulação dos nós da árvore de expressão. O valor do rótulo do nó significa o número de registradores necessários para gerar código da subárvore cuja raiz é esse nó[Ullm70].
Como o caso de operações unárias é bastante simples, discutimos apenas os casos em que todas as operações da expressão são binárias.

<u>Algoritmo 1</u>

[Algoritmo para rotular os nós da árvore de expressão]

Notação : l(P) é o valor atribuído ao nó P.

1. Quando o nó P for folha, i.e. variável simples ou valor literal , existem dois casos:

   1. P é descendente 'a esquerda
      l(P) = 1

   2. P é descendente 'a direita
      l(P) = 0

2. Quando P tiver duas subárvores com rótulos l1 e l2, existem dois casos:

1. l1 <> l2
   l(P) = max(l1,l2)

2. l1 = l2
   l(P) = l1 + 1

Exemplo: a/(b+c)-d*(e+f)

Aproveitando a propriedade de operações comutativas, podemos diminuir o valor de rotulo do no' em certos casos. Quando a operação do no' for comutativa, podemos trocar as subarvores esquerda e direita do no' mantendo o mesmo resultado da expressao representada pelo no'.

Exemplo: (A+(B+C)) ((B+C)+A)

O algoritmo descendente recursivo de modificar arvore de sintaxe para diminuir o valor de rotulo de expressao em termos de operações comutativas e' seguinte.

Algoritmo 2

1. Se o no' P for uma folha retorna l(P)

2. Aplicar este algoritmo para subarvore esquerda do no'.

3. Aplicar este algoritmo para subarvore direita do no'.

4. Se a raiz da arvore for operacao binaria, e se a operacao for comutativa e o valor do rotulo da raiz da subarvore esquerda for menor do que o valor do rotulo da raiz da subarvore direita, entao trocar as duas subarvores do no' da expressao corrente.

Aplicando este algoritmo para a expressao do exemplo anterior;

a/(b+c)-d*(e+f)

obtemos a seguinte expressao equivalente.

a/(b+c) - (e+f)*d

Este algoritmo pode diminuir o valor do rotulo da expressao, porem nao necessariamente e' otimo porque o algoritmo e' aplicado para operacoes comutativas apenas. Existem outras maneiras, por exemplo modificacao de arvore sobre operacoes associativas, que diminuim o valor de rotulo da raiz da arvore. O exemplo a seguir mostra um caso em que o rotulo da arvore pode ser diminuido atraves de outro tipo de modificacar da arvore.

Exemplo: a-(b-c) (a+c)-b

Neste exemplo, otimizacao de rotulacao da arvore e' feita sobre operacoes nao-comutativas e nao-associativas. Matematicamente, as duas expressoes a-(b-c) e (a+c)-b sao equivalentes, porem quando os valores a,b e c forem suficientemente grandes, o calculo de (a+c) pode causar overflow. De modo geral, este tipo de otimizacao de rotulo da arvore nao e' preferivel em termos de precisao especialmente quando os operandos da arvore forem do tipo real.

[Otimizacao do rotulo da arvore de expressao atrves de operacoes associativas]

Exemplo: (a+b)+(c+d) (((a+b)+c)+d)

Neste exemplo, troca de subarvores esquerda e doreita nao consegue diminuir o rotulo da arvore. Aproveitando a propriedade associativa de operacoes, podemos diminuir o valor do rotulo da arvore. Primeiramente precisamos definir a palavra "cluster". O cluster e' a subarvore maximal da arvore de expressao em que cada no' interno possui a mesma operacao associativa. Observe que os nos externos podem ser

expressões. Supondo, por exemplo, um cluster C tem 5 subarvores e1, e2, e3, e3, e4, e5 sobre uma operação OP, C pode ser modficado numa forma de lista L de subarvores sobre OP equivalente a C da seguinte forma:

C = ((e1 OP (e2 OP e3)) OP (e4 OP e5))

L = ((((e1 OP e2) OP e3) OP e4) OP e5)

Repare que o valor de rotulo da lista L de subarvores sobre OP associada ao conjunto de subexpressões {e1, e2, e3, e4, e5} e' max(l(e1), l(e2), l(e3), l(e4), l(e5)) ou max(l(e1), l(e2), l(e3), l(e4), l(e5)) + 1 de acordo com o algoritmo de rotulação citado anteriormente enquanto que valor do rotulo da arvore original pode ser maior do que o valor maximo da lista L.

O algoritmo de otimização de rotulo de arvore em termos de operações comutativas e associativas e' descendente recursivo:

Algoritmo 3

1. Obter o cluster da raiz da arvore correntemente aplicada. (O algoritmo e' bastante simples e sera omitido).

2. Para cada no' externo do cluster,aplicar este algoritmo recursivamente e obter o valor do rotulo da cada subexpressao.

3. Modificar este cluster para a forma de lista de subarvores como no exemplo citado anteriormente.

4. Buscar a subexpressao cujo valor de rotulo e' maximo no cluster obtido e trocar esta subexpressao com a subexpressao mais esquerda da lista.

Este tipo de otimizacao nao foi feita no nosso caso. A razao e' bastante simples. A frequencia que este tipo de otimizacao ocorre seria muito pequena na pratica e nao compensaria implementa-la especialmente quando escrevemos para uma maquina que possui espaco de memoria bem restrito.

Depois da aplicacao do algoritmo para diminuir o valor de rotulo da arvore atraves de operacoes comutativas, a arvore modificada possui a seguinte caracteristica. Para qualquer no' da arvore cuja operacao for comutativa, o valor do rotulo da subarvore direita do no' e' menor ou igual ao valor do rotulo da subarvore esquerda. O algoritmo para gerar codigo a partir da arvore otimizada e' descendente recursivo. Para melhor entendimento, suponhamos que o numero de registradores e' ilimitado primeiro.

Algoritmo 4

1. Quando l(P) do no' da arvore P sendo analizada e' 0, P so pode ser uma variavel ou parametro simples ou valor literal e tambem subarvore esquerda de uma expressao maior.

   Alocar um registrador r1 e gerar o codigo do tipo
   MOV r1 , [P]

2. Quando l(P) > 0
   O no' P possui uma operacao binaria.
   Alocar um registrador r1
   Aplicar este algoritmo para subarvore esquerda do no' P e carregar o valor no registrador r1.

1. Quando l(dir(P)) = 0
   Neste caso nao precisamos alocar registrador para subarvore direita do no' P.
   Gerar o codigo do tipo
   OP r1 , dir(P)

2. Quando l(dir(P)) > 0
   Alocar um registrador r2
   Aplicar este algoritmo para subarvore direita do no' P e carregar no registrador r2.
   Gerar o codigo do tipo
   OP r1 , r2
   Dealocar o registrador r2.

Na maioria dos casos, a modificacao da arvore e' feita implicitamente. Nao precisamos trocar as subarvores esquerda e direita para otimizar o valor do rotulo da arvore, so basta decidir a ordem de gerar codigo para subarvores. Porem, no nosso caso, o compilador modifica a arvore explicitamente antes de geracao de codigo por dois motivos:

1. Para facilitar outros tipos de otimizacoes miscelaneas.

2. Para facilitar verificar o processo de otimizacao da arvore(depuracao do compilador).

Este algoritmo garante que o numero de registradores necessarios para gerar codigo a partir de arvore e' otimo em termos de operacoes comutativas. A prova rigorosa da teoria e' escrita no livro "Compiler Techniques"[Ullm70].

O algoritmo de otimizacao com N registradores e' um pouco diferente dos algoritmos citados anteriormente. A ideia basica do algoritmo e' definir registradores ficticios de memoria para obter numero ilimitado de regristradores.

Algoritmo 5

Ao gerar o codigo para uma arvore apontada por P, se os dois rotulos das subarvores esquerda e direita de P estiverem

maiores do que N, alocar um registrador ficticio e aplicar o algoritmo 4[Ano77].

De acordo com Ullman[Ullm70], a troca de subarvores de P nao otimiza o codigo mesmo que l(esq(P)) seja menor do que l(dir(P)) se os dois rotulos forem maiores do que N, o que faria diferenca com o algoritmo 2. Ullman sugere gerar o codigo para a subarvore direita primeiro.

5.1.2 Otimizacao E Geracao De Codigo Adotadas No Passo 2

Apenas o registrador AX e' usado para calcular expressoes. Neste sentido, o numero de registradores utilizados e' um. Porem o registrador CX e' usado para armazenar o valor temporario de subexpressores tambem.

A diferenca entre algoritmo 5 e o algoritmo adotado e' que as instrucoes PUSH e POP sao utilizadas ao inves da instrucao MOV. Como o algoritmo de gerar codigo e' descendente recursivo, os registradores ficticios alocados seguem uma pilha. As instrucoes PUSH e POP ocupam apenas 1 byte enquanto a instrucao MOV ocupa 3 bytes e sao mais rapidas do que MOV.

[Algoritmo para otimizar arvores de expressoes]

Algoritmo 6

Para uma arvore apontada por P,
Se operacao de P for

1. Operacao unaria
   Otimizar a subarvore de P

2. Operacao binaria

   1. Aplicar este algoritmo para subarvore esquerda

   2. Aplicar este algoritmo para subarvore direita

   3. Se operacao e' comutativa e
      l(esq(P)) < l(dir(P)) e
      l(esq(P)) < 2
      Entao trocar subarvores esquerda e direita de P

A definicao de operacoes comutativas do Passo 2 e' diferente da definicao convencional. O Passo 2 reconhece operacoes semi-comutativas; e.g. <, <=, >, >= etc, como se fosse comutativas trocando o codgc da operacao.

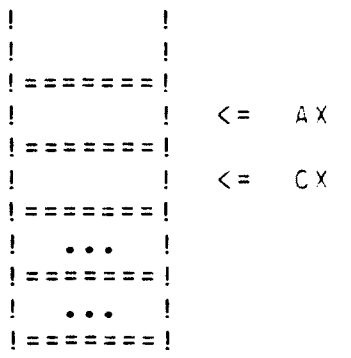
Exemplo :

EXP1 < EXP2 <=> EXP2 > EXP1

Existe um detalne ao trocar subarvores esqurda e direita do P no algoritmo 6, i.e. efeitos colatelais. Se a subarvore esquerda de P tiver chamadas de rotinas, entao as duas subarvores nao devem ser trocadas.

[Algorimo para gerar codigo a partir de arvore]

Existe um procedimento chamado LOADAX(P:PTR) que gera o codigo da arvore apontada pelo parametro P. Como todos os calculos sao feitos pelo registrador AX apenas, o valor da arvore de expressao sempre e' carregado no AX.
A ideia basica do algoritmo e' ao calcular uma operacao binaria de P, deixar sempre os valores de subarvores esquerda e direita nos registradores CX e AX respectivamente.



A geracao de codigo e' feita sempre para a subarvore esquerda primeiro. Para operacoes de subtracao e divisao, seria mais conveniente gerar codigo para subarvore direita primeiro. Porem se a subarvore esquerda tiver chamadas de rotinas, este metodo poderia causar efeitos colatelais.

[Geracao de codigo para arvore apontada por P]

Algoritmo 7

Existem tres casos:

1. P aponta para arvore de variavel ou valor literal

    1. Carregar o valor no registrador AX

2. P aponta para uma arvore unaria

1. Gerar código para subarvore

2. Gerar código UNIOP AX

3. P aponta para uma arvore binaria

   1. Aplicar este algoritmo para subarvore esquerda de P

   2. Existem dois casos em termos da subarvore direita do P

      1. Variavel ou valor literal
         Gerar o código BINOP AX , (dir(P))

      2. Não e' variavel ou valor literal.
         Existem dois casos em termos de rotulo da subarvore direita de P, l(dir(P)).

         1. l(dir(P)) < 2

            1. Gerar o código XCHG AX , CX

            2. Aplicar este algoritmo para subarvore direita de P

            3. Gerar o código BINOP AX , CX

         2. l(dir(P)) >= 2

            1. Gerar o código PUSH AX

            2. Aplicar este algoritmo para subarvore direita de P

            3. Gerar o código POP CX
               (Neste caso XCHG AX, CX e' feita implicitamente)

            4. Gerar o código BINOP AX , CX

Ao gerar o código BINOP AX , CX , se a operação binaria e' subtração ou divisão precisamos gerar o código XCHG AX , CX antes.

Exemplos :

a+b MOV AX , a

```
                                ADD  AX , b

a*(b+c)                         MOV  AX , b
                                ADD  AX , c
                                IMUL WORD a

(a+b)*(c*d)                     MOV  AX , a
                                ADD  AX , b
                                XCHG AX , CX
                                MOV  AX , c
                                IMUL WORD d
                                IMUL AX , CX

(a+b)+(c-d)                     MOV  AX , a
                                ADD  AX , b
                                XCHG AX , CX
                                MOV  AX , c
                                SUB  AX , d
                                ADD  AX , CX

((a+b)-(b-c))+(a DIV b)         MOV  AX , a
                                ADD  AX , b
                                XCHG AX , CX
                                MOV  AX , b
                                SUB  AX , c
                                XCHG AX , CX
                                SUB  AX , CX
                                XCHG CX , AX
                                MOV  AX , a
                                IDIV WORD b
                                ADD  AX , CX

((a+b)-(b-c))-((a div b)+(c*d))
                                MOV  AX , a
                                ADD  AX , b
                                XCHG AX , CX
                                MOV  AX , b
                                SUB  AX , c
                                XCHG AX , CX
                                SUB  AX , CX
                                PUSH AX
                                MOV  AX , a
                                IDIV WORD b
                                XCHG AX , CX
                                MOV  AX , c
                                IMUL WORD d
                                ADD  AX , CX
                                POP  CX
                                XCHG AX , CX
                                SUB  AX , CX
```

Um detalhe importante deste algoritmo e' o ajuste de tamanho de operandos. A linguagem Pascal permite operacoes com tipos diferentes como o exemplo a seguir.

Exemplo :

```
i + ORD(cn)              MOV  AX , i
                         XCHG AX , CX
                         MOV  AL , cn
                         XOR  AH , AH ; Zerar AH
                         ADD  AX , CX

ch:=CHR(i+ORD(ch))       MOV  AX , i
                         ADD  ch , AL
```

5.2 GERACAO DE CODIGO PARA ENDERECOS

Existem nove tipos de Codigo-P definidos para calculo de enderecos no Compilador Pascal NBS:

1. REFER :
   Pseudo codigo. Indica que o resultado nao e' o conteudo do endereco calculado mas o endereco mesmo.

2. OFSET :
   Codigo para calcular o deslocamento de um subcampo de uma variavel ou parametro do tipo registro indexado.

3. INDIR :
   Codigo para calcular o endereco indicado por apontadores.

4. INDEX :
   Codigo para calcular o endereco de um campo de vetores.

5. RTEMP :
   Codigo que indica o endereco que e' implicitamente indicado dentro do comando WITH.

6. RDATA :
   Codigo que indica o endereco inicial de constantes do tipo estruturado.

7. VARBL :
   Codigo que indica o endereco de uma variavel.

8. PARAM :
   Codigo que indica o endereco de um parametro.

5.2.1 VARBL , PARAM

Existem tres casos possiveis em termos de nivel lexico:

1. Nivel lexico = 1
   Como os parametros do programa i.e. os arquivos usados no programa, sao ignorados pelo compilador, apenas variaveis globais podem ser referidas com nivel lexico um. Este endereco e' referido diretamente com o deslocamento do endereco inicial da area de memoria alocada(DLX1BASE) para variaveis globais do programa.

2. Nivel lexico = Nivel corrente da rotina
   O registrador BP foi alocado como LOCAL POINTER i.e. o nivel da pilha da rotina correntemente compilada. Logo referencia neste caso e' feita atraves do registrador

BP.

3. Nivel lexico <> Nivel corrente da rotina
Como o DISPLAY do nivel lexico da rotina corrente nao esta' em nenhum registrador, o acesso deve ser feito em dois passos. Primeiramente carregar o DISPLAY do nivel no registrador BX e calcular o endereco atraves deste valor no BX e o deslocamento.

Exemplo :

```
PROGRAM test;
VAR     k : integer;

  PROCEDURE p( l : integer);

    PROCEDURE q;
    VAR m : integer;
    BEGIN
      k:=10;
      l:=20;
      m:=30
    END;

  BEGIN
  END;

BEGIN
END.
```

Codigo gerado para atribuicoes:

```
k:=10  => MOV WORD LXIBASE+26 , 10
          26 e' deslocamento de k

l:=20 =>  MOV BX , DSP3
          Carregar o display no BX
          MOV WORD 4[BX] , 20
          4 e deslocamento de l

m:=30  => MOV WORD -2[BP] , 30
```

5.2.2 RDATA

O Passo 1 calcula o espaco de memoria necessario para os constantes do programa inteiro. O endereco e' calculado atravez do endereco inicial da area reservada para os constantes(@DATABASE) e o deslocamento da constante.

Exemplo:

```
PROGRAM RDATA;
CONST   str = 'ABC';
VAR     v : ARRAY[1..3] OF char;
BEGIN
  v:=str
END.
```

Arvore montada para v:=str

```
MOVEM  size : 3
  VARBL v
  RDATA str
```

Codigo gerado

```
MOV DI , @ (Endereco de v)
; @ significa imediato
MOV SI , @ (endereco de str)
MOV CX , 3
REP
MOVB
```

5.2.3 OFSET

O codigo OFSET indica o deslocamento do um elemento do tipo registro(RECORD) a partir do endereco inicial do registro. O calculo de endereco do tipo OFSET e' feito em dois passos: Carregar o endereco inicial do registro no BX, e somar o deslocamento do subcampo para o BX. Se o deslocamento estiver menos do que 2, entao a instrucao INC BX e' usada o invez da instrucao ADD BX.

Exemplo:

```
PROGRAM OFSET;
TYPE    t = RECORD
              a : boolean;
```

```
                b : char
              END;
VAR       v : ARRAY[1..10] OF t;
          i : integer;
BEGIN
  v[i].b:='A'
END.
```

Arvore de sintaxe da tribuicao v[i].b:='A';

```
STOL
  OFSET  size : 1  dsp : 1
    INDEX  size : 2
      VARBL  v
      ISUB
        VARBL  i
        LITER  1
  LITER 1
```

Codigo gerado

```
MOV BX , @ ('Endereco de v' menos tres)
MOV AX , i
SHL AX , 1 ; AX <= AX * 2
ADD BX , AX
INC BX     ; BX <= BX + 1 deslocamento de b
MOV BYTE O[BX] , 'A'
```

5.2.4 INDEX

O codigo INDEX possui dois argumentos:

1. ARG[1] : Endereco inicial de indexacao.

2. ARG[2] : Expressao de deslocamento do inicio.

Alem disso, o codigo INDEX possui a informacao de tamanho de cada elemento (em numero de BYTE) de cada INDEX. A meneira de calcular o endereco de INDEX e' seguinte:

1. Calcular o endereco inicial de indexcacao.

2. Calcular o valor de expressao no ARG[2]

3. Multiplica o valor calculado por o tamanho de elemento e somar cujo valor para o endereco inicial de indecacao anteriormente calculado.

Existe um detalne neste caso. O endereco inicial de endxacao e' calculado e armazenado no registrador BX. Pode acontecer que o registrador BX e' usado durante o calculo de expressao no ARG[2]. O Passo 2 verifica se BX e' usado durante este calculo. Se BX e' usado, entao o Passo 2 salva o valor do BX e restaura depois de calculo de expressao.

Exemplo:

```
PROGRAM index;
VAR     i , j : integer;
        v : ARRAY[1..5,1..10] OF integer;
BEGIN
  v[i,j]:=10
END.
```

Arvore montada para a atribuicao v[i,j]:=10

```
STOL
  INDEX  size : 2  dsp : 2
    INDEX  size : 20  dsp : 10
      VARBL  v
      ISUB
        VARBL i
        LITER 1
    ISUB
      VARBL  j
      LITER  1
  LITER  10
```

Codigo gerado para a atrubuicao v[i,j]:=10

```
MOV BX , @( 'Endereo de v' menos 20)
          ; @ significa inmediato
MOV AX , i
MOV SI , 20
MUL AX , SI
ADD BX , AX
MOV AX , j
DEC Ax
SHL AX , 1          ; AX <= AX * 2
ADD BX , AX
MOV WORD O[BX] , 10
```

Note que os primeiros cinco instrucoes geradas sao equivalentes 'a
seguinte sequencia de instrucoes:

```
MOV BX , @ Endereco de v
```

5.2.2 RDATA

O Passo 1 calcula o espaco de memoria necessario para os constantes do programa inteiro. O endereco e' calculado atravez do endereco inicial da area reservada para os constantes(@DATABASE) e o deslocamento da constante.

Exemplo:

```
PROGRAM RDATA;
CONST   str = 'ABC';
VAR     v : ARRAY[1..3] OF char;
BEGIN
  v:=str
END.
```

Arvore montada para v:=str

```
MOVEM  size : 3
  VARBL v
  RDATA str
```

Codigo gerado

```
MOV DI , @ (Endereco de v)
; @ significa imediato
MOV SI , @ (endereco de str)
MOV CX , 3
REP
MOVB
```

5.2.3 OFSET

O codigo OFSET indica o deslocamento do um elemento do tipo registro(RECORD) a partir do endereco inicial do registro. O calculo de endereco do tipo OFSET e' feito em dois passos: Carregar o endereco inicial do registro no BX, e somar o deslocamento do subcampo para o BX. Se o deslocamento estiver menos do que 2, entao a instrucao INC BX e' usada o invez da instrucao ADD BX.

Exemplo:

```
PROGRAM OFSET;
TYPE    t = RECORD
              a : boolean;
```

```
                b : char
              END;
VAR       v : ARRAY[1..10] OF t;
          i : integer;
BEGIN
  v[i].b:='A'
END.
```

Arvore de sintaxe da tribuicao v[i].b:='A';

```
STOL
  OFSET  size : 1  dsp : 1
    INDEX  size : 2
      VARBL  v
      ISUB
        VARBL  i
        LITER  1
  LITER 1
```

Codigo gerado

```
MOV BX , @ ('Endereco de v' menos tres)
MOV AX , i
SHL AX , 1 ; AX <= AX * 2
ADD BX , AX
INC BX     ; BX <= BX + 1 deslocamento de b
MOV BYTE O[BX] , 'A'
```

5.2.4 INDEX

O codigo INDEX possui dois argumentos:

1. ARG[1] : Endereco inicial de indexacao.

2. ARG[2] : Expressao de deslocamento do inicio.

Alem disso, o codigo INDEX possui a informacao de tamanho de cada elemento (em numero de BYTE) de cada INDEX. A meneira de calcular o endereco de INDEX e' seguinte:

1. Calcular o endereco inicial de indexcacao.

2. Calcular o valor de expressao no ARG[2]

3. Multiplica o valor calculado por o tamanho de elemento e somar cujo valor para o endereco inicial de indecacao anteriormente calculado.

```
MOV AX , 1
DEC AX
MOV SI , 20
MUL AX , SI
ADD BX , AX
```

O Passo 2 obtem a parte calculavel em tempo de compilacao.

5.2.5 INDIR

O codigo INDIR possui apenas um argumento. O argumento indica o endereco de variavel que esta' apontando. Este codigo possui o deslocamento de campo que esta' indicado dentro de registro(RECORD).

Exemplo:

```
PROGRAM indir;
TYPE    t = RECORD
              a : boolean;
              b : char
            END;
VAR     p : ^t;
BEGIN
  p^.b:='A'
END.
```

Arvore de sintaxe equivalente

```
STOL
  INDIR  size : 1 dsp : 1
    VARBL  p
  LITER  'A'
```

Codigo gerado

```
MOV BX , p
MOV BX , 0[BX] ; Indirecao
INC BX         ; Deslocamento de b
MOV BYTE 0[BX] , 'A'
```

5.3 O PROBLEMA DA GERACAO DE CODIGO PARA DESVIO CONDICIONAL

Existem dois tipos de desvios: Um e' desvio para um endereco absoluto. Outro e' para um endereco relativo(deslocamento) ao endereco de instrucao de desvio. A vantagem de desvio com deslocamento e' que o codigo fica auto-relocavel. A maioria das instrucoes de desvio do INTEL 8088 e' deste tipo.

Todas as instrucoes de desvios condicionais tem apenas um byte de deslocamento. Quando o deslocamento de desvio for maior do que um byte nao podemos gerar o codigo de desvio condicional diretamente, precisamos gerar o codigo da seguinte maneira:

Exemplo :

```
      J<cond>  LA
      ...
      ...
   LA:
      (CASO 1 Codigo ideal)


      J<not cond>  L1
      JMP  LA
   L1: ...
      ...
   LA:
      (CASO 2)


      J<not cond>  $ + 5
      JMPL  LA
      ...
      ...
   LA:
      (CASO 3)
```

NO caso 1 do exemplo, o custo de memoria para desvio condicional e' apenas 2 bytes. Porem para poder gerar este codigo, precisamos ter certeza de que o deslocamento e' menor do que 2 bytes. Para saber o deslocamento de desvio condicional, precisamos calcular quantos bytes sao necessarios para gerar o trecho do codigo entre J<cond> LA e LA:. Alem do mais, se tiver varias instrucoes de desvio condicionais encaixados, precisamos calcular os deslocamentos recursivamente, de dentro para fora. Isto significa que precisamos gerar o codigo duas vezes, para gerar codigos de desvio condicionais otimizados. O "overhead" da otimizacao deste tipo seria muito grande. Decidimos desistir este tipo de otimizacao.

O Caso 2 tem um problema tambem. Para cada desvio condicional, precisamos de dois rotulos e isto daria uma carga pesada para o montador do INTEL 8088. A maneira de evitar o uso de rotulo adicional, como o Caso 2, seria gerar o codigo de desvio condicional de deslocamento de valor fixo como no Caso 4. A nova instrucao JMPL esta' introduzida no Caso 3 por seguinte razao: O INTEL 8088 possui dois tipos de desvios incondicionais; o deslocamento de um byte e o deslocamento de dois bytes. A decisao de deslocamento de um ou dois bytes e' tomada pelo montador. Quando o deslocamento for menor do que de 2 bytes, o montador gera o codigo para desvio incondicional de um byte apenas. Portanto, neste caso a sequencia de codigos no exemplo seguinte nao funcionaria corretamente.

```
J<not cond>  $+5
JMP LA
```

Como nao sabemos o tamanho de deslocamento no momento de geracao de codigo para desvio condicional, precisamos forcar a geracao de um desvio com deslocamento de 2 bytes. Isto e' o que faz a instrucao JMPL. Ao invez de gerar codigo para desvio condicional de dois bytes diretamente, gastamos 5 bytes para cada instrucao deste tipo. Como isto ocorre frequentemente,e.g. comandos IF, WHILE, REPEAT, LOOP, FOR, etc. A perda de memoria por causa deste problema da arquitetura do INTEL 8088 nao e' insignificante.

5.4 OTIMIZACOES MISCELANEAS

Todas as otimizacoes sao do tipo local. Algumas sao feitas no nivel da arvore de sintaxe construida pelo Passo 2 e algumas sao feitas utilizando as caracteristicas de hardware do INTeL 8088.

1. Modificacoes da arvore se sintaxe
   Exemplos

   1. <exp> + 1 => INC(<exp>)

   2. <exp> - 1 => DEC(<exp>)

   3. <exp> * <exp> = SQR(<exp>)
      quando as duas expressoes sao identicas.

   4. <exp> + 0 => <exp>

   5. <exp> * 1 => <exp>

   6. <exp> * 0 => 0

   7. NOT <cond> => <cond inversa>
      e.g. NOT (i < j) => i >= j
      Esta otimizacao e' bastante util especialmente para geracao de codigo para LOOP do tipo WHILE, porque a condicao de saida do LOOP deste tipo e' sempre contraria 'a condicao escrita em Pascal. O seguinte exemplo mostra um caso tipico deste. Note que a expressao NOT (NOT <cond>) e' modificada para <cond>.

```
        WHILE NOT ( i < j ) DO
        BEGIN
          ...
          ...
        END;

L1:
        MOV AX , i
        CMP AX , j
        JL L2
        < A parte do comando composto >
        JMP L1
L2:
```

   8. Calculo de partes fixas em indexacao
      Como a gerecao de codigo para indexacao fica bastante cara em termos de memoria, seria desejavel calcular as partes calculaveis de indexacao durante a compilacao. O Passo 2 verifica as partes constantes de indexacao e faz as partes calculaveis antes da geracao de codigo final. Neste momento, a

arvore de sintaxe e' modificada. Se todos os indices de indexacao forem constantes, seu endereco sao tratado como se fosse de uma variavel simples. Todos indices de matrizes com valor calculavel durante compilacao sao aproveitados na geracao de codigo. O seguinte exemplo mostra um caso de otimizacao de codigo deste tipo em combinacao a otimizacao de codigo para atribuicoes.

Exemplos :

```
a[2,5] := a[2,5] - 1


DEC  a[2,5]

a[2,5] := 10


MOV  a[2,5] , 10
Onde o endereco de a[2,5] e'
calculado pelo Passo 2
```

9. Calculo de endereco do tipo OFSET(RECORD)
Quanto ao endereco de um subcampo do tipo registro simples, podemos calcular seu endereco durante compilacao.

Exemplo :

```
x : RECORD
     a : integer;
     b : boolean;
     c : char
    END;
```

Note que os enderecos de subcampos a,b e c sao calculaveis durante compilacao. O passo 1 calcula os enderecos de cada subcampo e cada subcampo e' tratado como se fosse variavel simples.

2. Aproveitamento de hardware do INTEL 8088

1. Ao inves de multiplicar um valor que esta' num registrador por 2, utiliza a instrucao SHL para registrador. Seria muito util se este tipo de otimizacao fosse aplicavel para multiplicacoes por 4, 8, 16 etc. ou divisoes por 4, 8, etc. Infelizmente, quando este codigo SHL ou SHR estiver aplicado, a flag de overflow do INTEL nao indica o resultado corretamente, o que causaria problema de RUNTIME CHECK do tipo overflow. Por esta razao

desistimos de aplicar esta otimizacao para os casos extendidos. Sendo este tipo de otimizacao muito especifico, isto e' bastante util para geracao de codigo para indexacao. Variaveis ou tipos de matriz ou vetor do tipo inteiro sao comuns em Pascal. Neste caso a multiplicacao por 2 e' necessaria.

2. O INTEL 8088 possui instrucoes de PUSH e POP da seguinte forma:

```
PUSH  e.e (Endereco efeitivo)
POP   e.e
```

Ao gerar codigo para passagem de parametros, se os parametros estiverem variavel ou parametro simples as instrucoes PUSH e POP podem ser gerados diretamente. Quando tipo de parametros estiverem complexos, precisamos carregar seu enderecos ou valores de expressao nos registradores BX ou AX primeiro.

3. Ao inves de somar um valor que esta' num registrador por 1 ou 2, utiliza a instrucao INC. Este tipo de otimizacao tambem e' bastante especifico, porem, e' bastante util na pratica. Esta otimizacao nao e' aplicavel para apenas geracao de codigo para expressoes, mas muito util na geraco de codigo para enderecos.

```
Exemplo :
   x : ARRAY[1..10] OF RECORD
                         a : char;
                         b : boolean;
                         ...
                       END;

   ...
   x[i].b:=true;

   MOV  AX , i
   DEC  AX
   SHL  AX , 1
   LEA  BX , x
   ADD  BX , AX
   INC  BX       ; BX <= BX + 1
   MOV  0[BX] , 1  ; true
```

4. Ao inves de somar ou subtrair o valor do registrador SP para alocar ou desalocar a memoria(STACK) na entrada ou saida de uma rotina, cujo valor estiver menor ou igual a 6. Suponhamos que uma rotina tiver variaveis locais de 4 bytes em

total, precisamos alocar 4 bytes na pilha na entrada e desalocar 4 bytes na saida.

Exemplo :

```
SUB  SP , 4 ; <= alocacao
ADD  SP , 4 ; <= desalocacao

PUSH AX
PUSH AX     ; <= alocacao
POP  AX
POP  AX     ; <= desalocacao
```

A dealocacao de memoria na pilha nao e' feita na saida de rotina apenas, mas e' feita na saida dos comando FOR e WITH tambem. Apesar do numero de instrucoes geradas ser maior do que o primeiro caso, o tamanho de codigo gerado sera menor e a execucao sera mais rapida. Devido ao problema de alinhamento do PDP 11, o numero de bytes alocados para variaveis locais e' sempre ajustado para ser par.

5. Utilizacao da instrucao XLAT em indexacao. XLAT do INTEL 8088 e' uma instrucao muito interesante para geracao de codigo para indexacao. O funcionamento da XLAT e' o seguinte:

```
AL <= [ BX + AL ]
```

Note que esta instrucao e' aplicavel quando um valor indexado de vetor (apenas uma dimensao) cujo tipo de limites(escalar) for de um byte apenas.

```
ARRAY[ tipo1 ] OF tipo2
```

Os dois tipos tipo1 e tipo2 devem estar representado em um byte para aplicar esta instrucao.

Exemplo :

```
  x : ARRAY[ char ] OF boolean;
  ch : char;
  b : boolean;

b:=x[ch];


MOV  AL , ch
LEA  BX , x
XLAT
MOV  b  , AL
```

5.5 OTIMIZACAO PEEPHOLE

Otimizacao peephole e' um tipo de otimizacao local. A palavra peephole e' derivada do fato de que este tipo de otimizacao utiliza apenas uma visao restrita do codigo gerado. Um exemplo tipico desta otimizacao esta' no seguinte exemplo:

Exemplo :

```
x:=y;
z:=x+z
        MOV AX , y
        MOV x , AX
        MOV AX , x <= instrucao superflua
        ADD AX , z
        MOV z , AX
```

Para descobrir este tipo de instrucoes superfluas, basta verificar se uma instrucao antes da MOV reg , x e' do tipo MOV x , reg.

Como o nosso gerador de codigo trabalha com arvores de expressoes, podemos aplicar esta otimizacao no sentido mais geral do que esse. Na otimizacao peephole convencional, verificamos uma instrucao antes da instrucao atualmente sendo gerada. Isto significa que nos verificamos o valor sendo armazenado num registrador antes de gerar o codigo da instrucao do tipo MOV reg , x. Como o Passo 2 trabalha com arvore na geracao de codigo, podemos guardar a informacao de valores guardados nos registradores de uma forma de arvore tambem. Isto facilita a otiizacao de codigo de visao bem maior do que da otimizacao original.

No passo 2 existem duas variaveis globais AXpointer e BXpointer do tipo apontador de arvore de expressao que apontam a arvore de valores de cada registrador atualmente guardado durante geracao de codigo.

Exemplo :

```
i:=j + 1;
a[i]:=10 + a[i] * 5
        MOV AX , j
        INC  AX
        MOV i  , AX
        MOV BX , a
                ( MOV AX , i )
        SHL AX , 1        ; AX <= AX * 2
        ADD  BX , AX
                ( PUSH BX )
                ( LEA  BX , a )
                ( MOV AX , i )
```

```
            ( SHL  AX , 1 )
            ( ADD  BX , AX )
    MOV AX , 0[BX]
    MOV DX , 5
    IMUL DX
    ADD  AX , 10
            ( POP  BX )
    MOV 0[BX] , AX
```

NOTE : As instrucoes entre parenteses sao omitidas

A seguinte tabela mostra a frequencia das otimizacoes aplicadas do tipo peephole generalizado, para geracao de codigo de tres programas descritos secao 5.6.

Frequencia de otimizacoes de peephole

	programa 1	programa 2	programa 3
AX	63	25	3
BX	40	12	0

Um detalhe importante desta otimizacao e' que os valores carregados nos registradores podem ser destruidos se houver algum tipo de desvio envolvido antes da instrucao omitida. O seguinte exemplo mostra o caso em que esta otimizacao nao e' aplicael.

Exemplo :

```
          x:=y;
    10 :  z:=x + y;
```

A otimizacao peephole e' aplicavel dentro do bloco basico. Bloco Basico e' uma sequencia de instrucoes maximal em que para quaisquer nao existe instrucoes de desvio ou referencia de endereco de desvio. O passo 2 zera os valores de variaveis AXpointer e BXpointer quando instrucao de desvio e' gerada ou encontrou uma referencia de endereco de

desvio.

O passo 2 pode otimizar a expressao de atribuicao antes da geracao de codigo para verificar se este tipo de otimizacao pode ser feita.

Exemplo :

```
a[i]:=y;
z:=w + a[i]
```

A expressao w + a[i] e' modificada para a[i] + w antes da geracao de codigo.

Outro exemplo da otimizacao peephole e'otimizacao de codigo para uma expressao quando seu valor e' calculavel durante o tempo de compilacao.

Exemplo

```
x:=2.0 * 3.14159265/360.0 + x;
IF x < 0.0 THEN ...
```

Como o valor da subexpressao 2.0 * 3.149265/360.0 e' calculavel durante o tempo de compilacao, nao precisamos gerar o codigo para calcular o valor desta subespressao. Este tipo de otimizacao e' feita pelo proorio passo 1 do compilador nbs. Se uma expressao e' calculavel durante o tempo de compilacao, seu valor e' automaticamente calculado antes de gerar o codigo-p para o passo 2.

Outro tipo de otimizacao comum, considerada como otimizacao peephole, e' a otimizacao de multiplos desvios incondicionais.

Exemplo :

```
          GOTO L1
          ...
          ...
L1 :      GOTO L2
          ...
          ...
L2 :      ...
```

Este tipo de otimizacao apresenta mais vantagens em termos de tempo de execucao do que em termos de tamanho de codigo gerado e nao foi implemantado[Mcke65].

5.6 ATRIBUICAO

Ha' tres tipos de Codigo-P para atribuicoes no Pascal NBS:

1. STCL : Para atribuicao de uma variavel representada por um byte ou dois bytes. (e.g. integer, char, boolean, scalar)

2. STOR : Para atribuicao de valores do tipo real. Como o tipo real e' implementado por dois bytes no nosso compilador, o tratamento do STOR fica identico ao tratamento do Codigo-p STCL .

3. MOVEM : Para atribuicao de variavel cuja representacao interna e' maior do que 2 bytes. Este codigo e' usado para atribuicao de uma variavel ou um parametro do tipo vetor ou do tipo registro.

[STCL]

O Codigo-P STCL possui dois argumentos:

ARG[1] : Apontador da arvore de endereco
ARG[2] : Apontador da arvore de expressao

O funcionamento canonico deste Codigo seria o seguinte:

1. empilhar o endereco que esta' representado no ARG[1] para TOPO da pilha.

2. Empilhar o valor da expressao que esta' representada no ARG[2].

3. Executar a seguinte sequencia de operacao sobre pilha:

   1. [pilha[TOPO]] <= pilha[top] - 1

   2. TOPO <= TOPO - 1

Observe que a ordem de empilhar os dois operandos deste codigo nao e' irrelevante por causa de efeitos colaterais. Em certos casos a troca de ordem de calculo dos parametros e' interessante para otimizacao de codigo. O problema de efeitos colaterais ocorre quando o endereco de atribuicao e' representado por indice de vetores ou matrizes, ou aponta dores.

Como explicado anteriormente, os enderecos sao carregados no registrador BX e o valor de expressao e' carregado no registrador AX. Em certos casos, o uso de registrador BX e' requerido para o calculo de expressao. Quando o calculo de endereco deve ser feito antes do calculo de expressao e o calculo desta expressao pode usar o registrador BX, precisamos salvar o valor no registrador BX antes de calculo da expressao e restaula-lo depois.

No Passo 2, atribuiccao do tipo STOL sao divididas em 8 casos para aproveitar ao maximo os recursos do microprocessador INTEL 8088.

Na geracao de codigo para expressao, como uma operacao binaria, se o segundo operando for um valor literal por exemplo, este valor nao precisa ser carregado no registrador. Analogamente no caso de geracao de codigo para atribuicoes, se o valor de atribuicao estiver um valor literal, este valor nao precisa ser carregado no registrador AX.

Existem dois tipos de enderecos de atribuicao:

1. Endereco simples :
   Endereco de uma variavel simples ou parametro simples.
   Neste caso, nao precisamos carregar o endereco no registrador BX.

2. Endereco complexo :
   Endereco que nao e' simples.

Existem dois tipos de expressoes de atribuicao;

1. expressao simples :
   Expressao que e' valor literal.
   Neste caso, nao precisamos carregar o valor da expressao no registrador Ax..

2. Expressao complexa :
   Expressao que nao e' simples.

Essa combinacao de tipos de enderecos e expressoes, leva a 4 tipos diferentes de atribuicao.

	Expressao simples	Expressao complexa
Endereco Simples	ATR 1	ATR 2
Endereco complexo	ATR 3	ATR 4

Exemplo :
ATR 1 : i:=20 , b:=true etc.
ATR 2 : i:=j+k
ATR 3 : a[i]:=5
ATR 4 : a[i]:=j+k

Os tipos de atribuicoes ATR 2 e ATR 4 sao divididos em tres tipos cada.

AIR 2

1. AIR 21 :
<ende. simpl.> := <ende. simpl.> OP liter
Onde os dois enderecos sao identicos.
Exemplos : i:=i+10
i:=-i
ch:=succ(ch)

<ende. simpl.> := OP <ende. simpl.>
Exemplos : i:=pred(i)
i:= - i

2. AIR 22 :
<ende. simpl.> := <ende. simpl.> OP <expr. compl.>
Onde os dois endrecos sao identicos.
Exemplo : i:=i - a[i]*j

3. AIR 23 :
Caso do ATR 2 que nao pertence aos casos ATR 21 e ATR 22

Note que o compilador e' suficientemente esperto para detectar a atribuicao i:=10+i ,por exemplo, e' do tipo ATR 21. Otimizacao da expressao da atribuicao e' feita antes de determinacao do tipo de atribuicao tambem.

Exemplo :

```
a[5]:=a[5]+1
```

Como o indice do vetor de a e' constante, o endereco de a[5] pode ser calculado em tempo de compilacao. Este tipo de otimizacao e' feita no Passo 2.Esta atribuicao nao e' do tipo ATR 42 mas ATR 21. Um caso mais interessante e' atribuicoes do tipo:

```
i:=i + 1
```

O compilador modifica a expressao i + 1 para succ(i) antes da determinacao do tipo e finalmente gera o codigo:

```
inc i
```

AIR 4

1. AIR 41 :
   <end. compl.> := <expr. compl.> OP liter
   Onde os dois enderecos sao identicos.
   Exemplo : a[i]:= not a[i]

2. AIR 42 :
   <ende. compl.> := <ende. compl.> OP <expr. compl>
   Onde os dois enderecos sao identicos.
   Exemplo : a[i] := a[i] + i + j

3. AIR 43 :
   Caso de ATR 4 que nao pertence aos casos ATR 41 e ATR 42. Este caso e' o caso pior porque precisamos carregar os enderecos e o valore da expressao de atribuicao nos registradores.

Os casos ATR 21, ATR 22, ATR 41, ATR 42 sao os casos em que podemos aproveitar os recursos do INTEL 8088. Para verificar a identidade de dois enderecos representados em forma de arvore, existe uma funcao booleana SUBTREEMATCH(r1,r2 : ptr):boolean que devolve o valor TRUE se as duas arvores representadas por r1 e r2 sao identicas.

Uma observacao importante na divisao de casos em atribuicoes e' que os casos ATR 1, ATR 2, e ATR 3 sao livres do problema de efeitos colaterais em relacoes aos endereco e valor da expressao de atribuicao. Especialmente quando o endereco de atribuicao for simples, nao precisamos carregar o endereco no registrador BX antes do calculo da expressao.

A geracao de codigo para cada tipo de atribuicao e' a seguinte:

1. ATR 1 : Usar a instrucao do tipo
MOV <ende.> , liter
Neste caso nem o endereco ou valor da expressao precisa ser carregado nos regidtradores AX e BX.
Exemplo : i:=10; MOV i , 10

2. ATR 21 : Usar instrucoes do tipo
OP <ende. simpl.> , liter
Onde OP pode ser ADD, SUB
ou
OP <ende. simpl.>
Onde OP pode ser INC, DEC, NEG
Um detalhe importante e' que operacoes de multiplicacao e divisao nao podem ser aplicadas para este tipo devido ao HARDWARE da maquina.

Exemplo :

```
i:= i-7          SUB i , 7
i:= - i          NEG i
```

3. ATR 22 :
Primeiramente carregar o valor da subexpressao da expressao da atribuicao no registrador AX e usar instrucoes do tipo:
OP <ende. simpl.> , AX
Onde OP podem ser ADD, SUB
Exemplo :

```
i:= i + i * j
        MOV AX , i
        IMUL AX , j
        ADD i , AX
```

4. ATR 41 :
A ideia basica de geracao de codigo e' semelhante aos casos de ATR 21, ATR 22, ATR 23. A unica diferenca entre os casos de ATR 21, ATR 22, ATR23 e os casos de ATR 41, ATR 42 e ATR 43 e' o endereco de atribuicao e' carregado no registrador BX primeiramente.
Carregar o endereco de atribuicao no BX primeiro.
Usar as instrucoes do seguinte tipo:
OP [BX] , liter
Onde OP podem ser ADD, SUB
ou
OP [BX]
Onde OP pode ser INC, DEC, NEG
Exemplo :

```
a[i]:=a[i] + 10;
        LEA BX , a
        MOV AX , i
        DEC AX
        LSH AX , 1
        ADD BX , AX
```

```
ADD [BX] , 10
```

5. ATR 42 :

Carregar o endereco comum da atribuicao no registrador BX primeiro e carregar o valor da subexpressao da atribuicao no registrador AX e gera o codigo do tipo :

```
OP [BX] , AX
```

Onde OP pode ser ADD ou SUB

Exemplo :

```
a[i]:=a[i] + i + j
        LEA BX , a
        MOV AX , i
        DEC AX
        LSH AX , 1
        ADD BX , AX
        MOV AX , i
        ADD AX , j
        ADD [BX} , AX
```

6. ATR 43:

Este e' o pior caso na geracao de codigo para atribuicoes. Precisamos carregar o endereco da atribuicao no registrador BX e carregar o valor de expressao no registrador AX e usar a instrucao:

```
MOV [BX] , AX
```

Exemplo :

```
a[i]:=i+j
        LEA BX , a
        MOV AX , i
        DEC AX
        SHL AX , 1
        ADD BX , AX
        MOV AX , i
        ADD AX , j
        MOV [BX] , AX
```

Um ultimo detalhe sobre geracao de codigo no caso da linguagem Pascal e' o ajuste do tamanho de valores na atribuicao.

Exemplo :

```
i:=ORD(ch - 40b)
```

Neste caso, a variavel ch esta' representada em um byte enquanto a variavel i esta' representada em dois bytes. Portanto precisamos ajustar o tamanho do dado antes de executar a atribuicao.

O codigo gerado pelo Passo 2 e' seguinte :

```
MOV AL , ch
SUB AL , 40h
XOR AH , Ah      ;  zerar o valor do AL
MOV WORD i , AX
```

O Passo 2 ajusta o tamanho de operandos na instrucao de MOV dependendo do tamanho de variavel a ser atribuida. Portanto quando o tamanho de variavel de destino da atribuicao for menor do que o tamanho de valor de expressao, nao ha' problema de reajuste de tamanho, pois o valor de expressao e' ajustado automaticamente.

A tabela abaixo mostra que os casos ATR 22, ATR 41, ATR 42 sao bem menos frequentes do que os outros. Observamos que o caso ATR 1 e' muito frequente apesar de que este tipo de atribuicao e' um caso especifico. A otimizacao feita sobre atribuicoes do tipo ATR 1 e' muito significante. Os programas 2 e 3 foram escritos pelo autor.

Programa 1 :
Passo 1 do compilador NBS original.(4500 linhas)

Programa 2 :
Interpretador LOGO (3000 linhas)
Esse programa e' basicamente um interpretador BASIC com chamada recursiva de procedimentos com parametros e recursos graficos.

Programa 3 :
Interpretador LISP (800 linhas)
Este programa implementa apenas recursos basicos da LISP.

Frequencia de casos em atribucoes

	Programa 1	Programa 2	Programa 3
ATR 1	306 (30.1 %)	173 (27.8 %)	46 (34.1 %)
ATR 21	77 (7.7 %)	44 (7.1 %)	5 (3.7 %)
ATR 22	15 (1.5 %)	1 (0.2 %)	0 (0.0 %)
ATR 23	335 (33.4 %)	229 (36.8 %)	69 (51.1 %)
ATR 3	126 (12.6 %)	48 (7.7 %)	2 (1.5 %)
ATR 41	1 (0.1 %)	0 (0.0 %)	0 (0.0%)
ATR 42	2 (0.2 %)	1 (0.2 %)	0 (0.0%)
ATR 43	140 (14.0 %)	126 (20.3 %)	13 (9.6 %)
TOTAL	1002	622	135

[MOVEM]

Este codigo-P possui tres informacoes:

1. DISP : Numero de bytes para transferir
2. ARG[1] : Apontador de endereco de fonte de transferencia
3. ARG[2] : Apontador de endereco de destino de transferencia

Existe uma instrucao muito conveniente para este codigo: MOVB. Esta instrucao transfere o numero de bytes que esta' carregado no registrador CX a partir do endereco que esta' carregado no registrador SI para o endereco que esta' carregado no registrador DI(Block Transfer).

Exemplo:
```
VAR     a,b :  ARRAY[1..10] OF integer;
a:=b;

 LEA DI , a
 LEA SI , b
 MOV CX , 20
 REP
 MOVB
```

O numero de bytes para transferencia que e' carregado no registrador CX deve ser logo antes da instrucao MOVB, porque o registrador CX pode ser usado durante calculos de enderecos para carregar no AX e CX. Neste caso particular o numero de byte para transferir e' sempre definido durante compilacao por virtude da definicao de Pascal. p;Analogamente no caso de geracao de codigo para o Codigo-P STOL, precisamos carregar o valor do registrador DI devido ao problema de efeitos colaterais e verificar se o valor carregado no DI e' destruido durante o calculo de endereco para carregar no registrador SI. Se o valor for destruido, precisamos salvar o conteudo no registrador DI antes de calcular o endereco para carregar no registrador DI e restaura-lo depois.

5.7 COMANDO IF

Nao na' muito para otimizar na geracao de codigo do comando IF. O comando possui duas formas :

1. IF <exp. booleana> THEN <comando>;

2. IF <exp. booleana> THEN <comando 1>
   ELSE <comando 2>

A geracao de codigo para cada forma seria seguinte :
Calcula em AX o valor da expressao booleana.

Existem dois tipos de otimizacoes feitas na geracao de codigo para este comando.

1. Omissao de carregamento de valor booleano no AX.
   A maneira mais simples de gerar codigo para o comando IF seria, primeiramente, calcular o valor da expressao booleana no registrador AX e testar a paridade do valor no AX(convencionalmente o valor FALSE esta' representado 0 e o valor TRUE esta' representado 1). Porem, esta maneira nao seria eficiente no seguinte caso:
   Exemplo :
   IF i < j THEN <comando>

   Codigo gerado da maneira usual

   ```
   ================================

   MOV  AX , i
   CMP  AX , j
   MOV  AX , 0
   JGE  $+3
   INC  AX
   JO   L1
   <codigo gerado para o comando>
    l1:
   ```

   Codigo gerado mais eficiente. Omitir o carregamento do valor booleano em AX.

   ```
   ========================================

   MOV  AX , i
   CMP  AX , j
   JL   $+5
   ```

```
JMPL L1
<codigo gerado para o comando>
 L1:
```

Note que este tipo de geracao nao e' sempre possivel. Quando a expressao booleana for mais complexa, como "b and (i < j)", precisamos carregar o valor da subexpressao (i < j) no AX para calcular o valor da expressao booleana inteira. O gerador de codigo detecta os casos de expressoes booleanas complexas e gera o codigo correspondente.

2. Geracao de codigo condicional quando o valor de expressao e' calculavel durante a compilacao.
As vezes o valor da expressao booleana e' calculavel durante a compilacao como no exemplo abaixo :

Exemplo :

```
CONST   debug = true;
IF debug THEN <comando 1>
         ELSE <comando 2>
```

Neste caso, o compilador gera codigo para o comando1 apenas.

5.8 COMANDO CASE

Existem dois metodos basicos para implementar o comando case:

5.8.1 Utilizar Uma Tabela De Saltos

A tecnica usada e' simples e a geracao de codigo e' feita em quatro partes:

1. Calcular o valor de expressao do comando case.

2. Verificar o valor calculado com os valores inferior e superior dos valores dos seletores, e subtrair do valor calculado o valor inferior de seletores. Como os valores de seletores do comando case sao constantes de acordo a definicao do Pascal, os valores inferior e superior de seletores sao calculaveis durante a compilacao. O endereco de salto na tabela de saltos tambem deve ser calculado.

3. Gerar a tabela de saltos

4. Gerar codigo para cada subcomando do comando CASE.

Exemplo :

```
CASE ch OF
  'a' , 'b' : <comando1>
  'd'       : <comando2>
END


;Calcular o valor da expressao
MOV  AL , ch

;Testar o valor com limites e
;Calcular o endereco de salto
CMP  AL , 'a'  ; limite inferior
JL   L3
CMP  AL , 'd'  ; limite superior
JG   L3
SUB  AX , 61   ;ASCII para 'a'
SHL  AX , 1    ;AX<=AX * 2
ADD  AX ,  TAB ;   imediato
XCHG AX , BX   ;para indirecao
MOV  AX , CS:[BX]
JMP  AX

;tabela de salto
TAB: DW  L1 ; 'a'
```

```
        DW  L1 ; 'b'
        DW  L3 ; 'c'
        DW  L2 ; 'd'

;subcomandos
L1:  <comando1>
     JMP L3
L2:  <comando2>
L3:
```

Certamente este metodo e' muito eficiente em termos de tempo de execucao. Independentemente do valor da expressao do comando case, o tempo de execucao na parte de selecao de casos e' constante. Porem o tamanho da tabela e' proporcional 'a diferenca entre os limites superior e inferior. O seguinte exemplo mostra um caso em que este metodo e' indesejavel em termos de espaco de memoria.

```
Exemplo :

CASE i OF
  0    : <comando1>
  1000 : <comando2>
END
```

5.8.2 Comparacao Linear Com Cada Seletor

Este metodo e' ineficiente em termos de tempo de execucao, que e' proporcional ao numero de seletores no comando case. Porem a geracao do codigo para o exemplo anterior seria bem melhor. O codigo gerado para o penultimo exemplo seria o seguinte:

```
MOV  AL , ch
CMP  AL , 'a'
JE   L1
CMP  AL , 'b'
JE   L1
CMP  AL , 'd'
JE   L2
JMP  L3
L1:  <comando1>
     JMP L3
L2:  <comando2>
l3:
```

O metodo ideal e' analizar cada caso do comando case e decidir qual metodo e' mais eficiente. Porem a decisao de metodo apropriado para cada caso tambem e' dificil. O seguinte exemplo mostra um caso em que esta decisao nao seria tao facil.

```
Exemplo :
CASE i OF
  1,2,3,4,5,
  6,7,8,9,10 : <comando1>
  50         : <comando2>
  61,62,63,64,65,
  66,67,68,69,70 : <comando3>
END
```

O metodo adotado no passo 2 e' o segundo, pois o objetivo principal do projeto nao e' eficiencia em tempo de execucao, mas eficiencia em tamanho de codigo gerado. Alem disso, implementacao deste metodo e' relativamente mais facil do que o primeiro metodo. Quando este metodo e' adotado, seria desejavel que a sintaxe do comando case fosse extendida como mostra o seguinte exemplo.

```
Exemplo :

CASE i OF
  1..10 : <comando1>
  50    : <comando2>
  61..70: <comando3>
END
```

[Sale81], [Atki82]

5.9 COMANDO WHILE, REPEAT E LOOP

No Compilador Pascal NBS, os comandos; WHILE, REPEAT, e LOOP sao representados internamente como um comando LOOP. A arvore de sintaxe do comando LOOP tem numero variavel de subarvores. Cada sequencia de comandos ate' um EXIT ou END(exclusiva) corresponde a uma subarvore. Cada EXIT tambem corresponde a uma subarvore. No exemplo abaixo o codigo-P LOOP possui 3 subarvores. Para a n-esma subarvore, temos:

1. n e' par
   O codigo da subarvore e' SEQ, i.e. codigo de comando composto. Se esta parte nao existir, entao o numero de subarvores do codigo SEQ e' zero.

2. n e' impar
   O codigo da subarvore e' EXIT. Este codigo possui dois argumentos(subarvores); subarvore para condicao de saida do LOOP, e a subarvore para comando a executar na saida do LOOP.

Exemplo :

```
LOOP
  i:=i+1;
EXIT IF i=j THEN i:=10;
  j:=j-1
END;
```

Arvore sintatica equivalente

```
LOOP
   SEQ
      STOL
         i
         ADD
            i
            1
   EXIT
      ICEQ
        i
        j
      STOL
        i
        10
   SEQ
      STOL
        j
        SUB
           j
           1
```

O numero de subavores do codigo LOOP e' sempre maior ou igual a tres. Repare que os comandos WHILE e REPEAT sao representados atraves do LOOP como segue:

```
REPEAT                ==>     LOOP
  <comando>                       <comando>
UNTIL <cond>                      EXIT IF <cond>
                                  END
```

```
WHILE <cond> DO       ==>     LOOP
<comando>                       EXIT IF NOT <cond>
                                  <comando>
                                END
```

Note que nos casos dos comandos WHILE e REPEAT, o segundo
argumento do Codigo-P EXIT e' sempre NULL.

Ao gerar codigo para LOOP, dois rotulos sao alocados:
RETL : O rotulo que indica o inicio do LOOP
EXTL : O rotulo que indica a saida do LOOP

Ao gerar codigo para EXIT, um rotulo e' alocado se o segundo argumento deste codigo nao for NULL.

Codigo gerado para o exemplo anterior

```
RETL:
   INC  i
   MOV  AX , i
   CMP  AX , j
   JE   $ + 5
   JMPL SKIPL
   MOV  i  , 10
   JMP  EXTL
SKIPL:
   DEC  j
   JMP  RETL
EXTL:
```

5.10 COMANDO FOR

O comando FOR possui cinco argumentos:

1. Variavel simples de controle
2. Expressao inicial do FOR
3. Expressao final do FOR
4. Indicacao do tipo do FOR, TO(+1) ou DOWNTO(-1)
5. Subcomando do FOR

Implementacao do Comando FOR e' simples. Este comando possui duas expressoes; a expressao inferior e a expressao superior do comando FOR.

Pela definicao do Pascal no Revised Report, a expressao superior do comando e' calculada apenas uma vez na entrada do comando. O valor da expressao superior calculado e' armazenado numa posicao temporaria da memoria. A posicao da memoria mais natural para esta finalidade seria o topo da propria pilha. No sistema de execucao atraves de pilha em Pascal, o nivel da pilha antes e depois de um comando seria sempre igual. Neste sentido 2 bytes sao alocados na entrada do comando FOR e desalocado na saida do comando FOR cujo endereco de memoria alocado e' estatico,i.e. topo da pilha. Portanto quando este valor calculado e' referido durante a execucao do comando FOR, o nivel atual da pilha e' sempre igual, o que seria um pouco diferente do caso do comando WITH. O seguinte exemplo mostra o codigo gerado pelo Passo 2.

Exemplo :

```
FOR i:=1 TO 10 DO
v[i]:=i;
```

```
        MOV   i  , 1 ; i:= inicializacao
        MOV   AX , 10 ; Load exp. sup.
        PUSH  AX      : Salva o valor
L1:     POP   AX     ; Referir o valor sup.
        PUSH  AX      ; Adustar o nivel
        CMP   AX , i ; Testa a saida
        JL    L2
        MOV   BX , v
        MOV   AX , i
        DEC   AX
        SHL   AX , 1
        ADD   BX , AX
        MOV   AX , i
```

```
        MOV   0[BX] , AX
        INC   i         ; Aumentar o valor de i
        JMP   L1        ; Voltar ao inicio do FOR
L2:     POP   AX        ; Ajustar o nivel da pilha
```

5.11 COMANDO WITH

A virtude do comando WITH nao e' apenas legibilidade, mas e' eficiencia do codigo gerado. O endereco do comando WITH ,i.e. o endereco inicial do tipo registro(RECORD) specificado no inicio do comando WITH, e' calculado apenas uma vez, na entrada do comando, e seu valor e' armazenado numa posicao temporaria na memoria. O comando WITH possui dois argumentos no compilador MaS:

1. arg[1] : arvore de endereco

2. arg[2] : arvore de comando

Tres Codigos-P sao utilizados para implementar o comando WITH;

1. DIEMP
   O codigo que indica o comando WITH.

2. REFER
   O codigo que indica o endereco do comando WITH a ser armazenado na pilha.

3. RIEMP
   O codigo que indica a referencia ao endereco armazenado na entrada

O lugar mais natural de armazenar o endereco especificado neste comando, seria a propria pilha. Neste sentido, a solucao adotada neste caso, e' identica 'a solucao adotada no caso do comando FOR, porem, a locacao em que este endereco e' armazenado nao e' sempre o topo da pilha ao ser referido. Lembre-se se que no caso do comando FOR, o valor da expressao de condicao de saida do comando sempre esta' no topo da pilha quando este valor e' referido. No caso do comando WITH isto nao e' necessariamente verdade. Se uma referencia para este endereco e' feita dentro do subcomando do comando WITH e se a referencia e' feita durante execucao de um comando FOR, este endereco referido nao esta' no topo da pilha.

Porem o deslocamento da pilha onde o endereco do comando WITH esta' armazenado e' estatico, quer dizer que e' calculavel durante o tempo de compilacao. O calculo de deslocamento de variavel temporaria que armazena este endereco do comando WITH na pilha e' feito pelo Passo 2 com a ajuda de informacao forneceda pelo Passo 1. O Passo 1 fornece ao Passo 2 o numero de bytes necessarios para alocar o espaco de memoria para variaveis locais da rotina corrente. Isto significa o topo de pilha atual na entrada da rotina. O deslocamento da variavel temporaria e' calculado da seguinte maneira;

Implementacao do comando WITH

Inicializacao de variaveis na entrada de cada rotina

1. Inicializacao do deslocamento de variavel temporaria TEMPADDR <= topo da pilha
   Esta inicializacao e' feita uma vez ao montar a arvore de sintaxe de cada rotina. Esta informacao e' forneceda pelo Passo 1.

2. Inicializar o valor do contador do comando WITH(WITHCOUNT) com zero. Esta variavel indica o nivel do encaixamento do comando WITH atual na rotina.

Atualizacao de nivel da pilha para calcular o deslocamento de variavel temporaria

1. No momento de entrada do comando FOR decrementa TEMPADDR por 1 ou 2 comforme a tipo de variavel simples de controle do comando FOR. Note que a pilha cresce para baixo.

2. No momento de sair do comando FOR incrementa TEMPADDR por 1 ou 2 com comforme o tipo de variavel simple de comando FOR.

3. No momento de entrada do comando WITH tres coisas sao feitas;

   1. Decrementar o valor de TEMPADDR por 2(Alocacao de memoria).

   2. Incrementar o valor de WITHCOUNT por 1 e armazenar o valor de TEMPADDR na tabela WITHTABLE associado ao indice WITHCOUNT.

   3. Calcular o endereco do comando WITH que esta' representado no arg[1] e empilhar o endereco calculado na pilha.

4. No momento de saida do comando WITH tres coisas sao feitas;

   1. Incrementa o valor de TEMPADDR por 2.

   2. Decrementa o valor de WITHCOUNT.

   3. Desalocar a memoria alocada na entrada do comando WITH. Isto significa SP <= SP + 2. O Passo 2 gera o codigo POP BX pela mesma razao citada na geracao de codigo do comando FOR.

5. Cada referencia a' variavel temporaria do comando WITH e' associada com a identificacao do comando WITH. Logo cada referencia a subcampo no comando WITH atravez do cogido-P REFER e' feita atravez da tabela WITHTABLE.

O seguinte exemplo mostra um caso que o endereco do comando WITH nao esta' no topo da pilha ao ser referido.

```
exemplo :

TYPE
  t =    RECORD
           a:integer;
           b:ARRAY[1..10] OF integer
         END;

VAR

  p : ^t;


BEGIN
  WITH p^ do
  BEGIN
    a:=10;
    FOR i:=1 TO a DO
    b[i]:=i
  END
END
```

5.12 INTERFACE DE SISTEMA DE EXECUCAO

O esquema adotado para interface de execucao pelo Passo 2 e' simples. No compilador original Pascal MBS, as duas cadeias, cadeia estatica e a cadeia dinamica sao usadas. Porem no nosso caso, usamos apenas uma cadeia dinamica. O vetor de display esta' guardado na memoria porque o INTEL 8088 nao tem numero suficiente de registradores para esta finalidade.

5.12.1 CHAMADA DE ROTINA

Ao chamar uma rotina, precisamos primeiramente salvar o valor do LOCAL POINTER na pilha e empilhar os parametros da rotina antes de executar a instrucao CALL. Ao voltar para a instrucao depois de CALL, precisamos restaurar o valor de LOCAL POINTER da pilha.

Chamada de rotina

```
PUSH BP          ;Salvar BP
<Empilhar os parametros>
CALL ROTINA
POP  BP          ;Restaurar BP
```

5.12.2 ENTRADA DE ROTINA

Ao entrar numa rotina, primeiramente o valor do DISPLAY do nivel corrente e' salvo e o valor de LOCAL POINTER tambem e' atualizado. Depois para verificar se a pilha estourou ou nao, uma rotina CHKMEM e' chamada. Finalmente o espaco de memoria para variaveis locais e' alocado.

Entrada de rotina

```
PUSH DIPLAY[nivel]
MOV  BP , SP     ; Atualizar LOCAL POINTER
SUB  SP , n      ; Alocar variaveis locais
CALL CHKMEM      ; Verificar a pilha
```

Existe um detalhe na chamada e na entrada de rotina. Quando um parametro do tipo vetor e' passado por valor, apenas seu endereco e' passado na chamada e o parametro e' tratado como se fosse uma variavel local cujo valor e' copiado na entrada. Este ato de copia e' feito logo na entrada da rotina antes da execucao do primeiro comando da rotina. A rotina CHKMEM funciona da seguinte maneira: O espaco de memoria alocado para execucao do programa e' dividido em dois tipos, a area STACK e a area HEAP. A rotina verifica o valor atual do SP(STACK) e HEAPPTR(HEAP). Se SP for menor ou igual ao valor do HEAPPTR, entao a subrotina informa que o espaco estourou e para a execucao. Esta verificacao de memoria e' feita incondicionalmente.

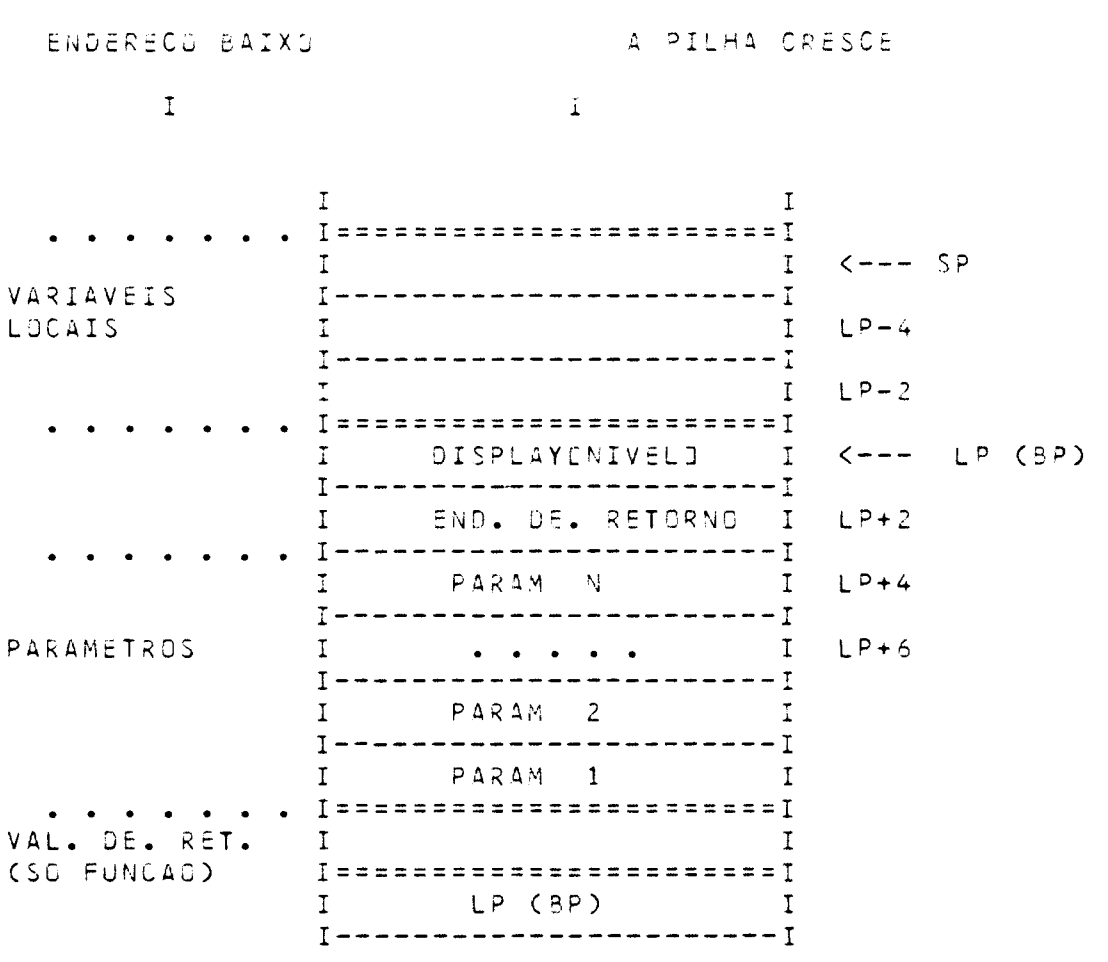
5.12.3 SAIDA DE ROTINA

Ao sair de rotina, primeiramente o espaco de memoria para variaveis locais e' dealocado. Depois o valor de DISPLAY do nivel corrente e' atualizado. Finalmente volta para a locacao onde esta rotina foi chamada. Como o INTEL 8088 possui a instrucao RET conveniente, podemos dealocar o espaco de memoria alocado para parametros ao retornar da rotina.

Saida de rotina

```
ADD  SP , m        ; Desalocar variaveis locais
POP  DISPLAY[nivel]
RET  n             ; Retornar e desalocar parametros
```

A INTERFACE DE ENTRADA E SAIDA DE PROCEDIMENTO



CAPITULO 6

TESTES DO COMPILADOR

6.1 TESTES DE CONFIABILIDADE DO CODIGO GERADO

As rotinas de suporte de execucao foram testadas primeiro. Todas as operacoes aritmeticas e logicas foram testadas tambem. O primeiro passo do teste do compilador foi testar cada comando com programas bastante simples. Para cada comando, varios programas simples foram compilados e executados no sistema 9098. Todos os casos de cada comando foram verificados.

O segundo passo do teste foi testar programas completos razoavelmente simples. A maioria dos programas que estao nos livros "DATA STRUCTURE + ALGORITHM = PROGRAM" e "SYSTEMATIC PROGRAMING" do Niklaus Wirth[Wirt76],[Wirt73] que nao usam arquivos (alem do INPUT e OUTPUT) foram testados.

Exemplos de programas de testes

1. FUNCAO Fibonacci
2. Funcao Ackerman
3. Permutacao
4. Torre de Hanoi
5. Problema de oito rainhas
6. Ordenacao de vetor

7. Programas de estrutura de dados(LISTA, ARVORE BINARIA etc)

Normalmente depuracao do programa no sistema INTEL 8088 e' feita atraves dos comandos de depuracao do monitor. Porem todas as depuracoes de erros foram feitas atraves de programas proprios do compilador. Antes de testar os programas no sistema INTEL 8088, os programas foram executados no sistema DEC 10. Se as saidas nao coincidem, modificamos o programa de teste para soltar os valores intermediarios de execucao para detectar a localizacao de erro. Alem disso o comando HALT que foi adicionado no Compilador ajudou bastante a depuracao. O unico caso em que precisamos de comandos de depuracao do sistema INTEL 8088 foi o caso em que o montador montava codigo errado. Apenas quatro ou cinco erros do compilador foram detectados pelos testes maiores. As opcoes de compilacao, SELECT, TRACE, DUMPT, PRINT(Apendice C) ajudaram bastante a depuracao do compilador.

O maior progarama de teste compilado e executado foi um interpretador LISP, que implementa apenas funcoes basicas da linguagem LISP. A primeira versao do programa foi escrita para o compilador de Hamburgo e testada no sistema DEC 10 e a segunda versao foi modificada para o compilador NBS. Este programa chama muitas funcoes com parametros que sao chamadas de funcoes (O maior numero de chamada de funcoes dentro de chamada global de uma funcao e' 6). Houve apenas um erro detectado no teste deste programa.

6.2 TESTE DE TAMANHO DO CODIGO GERADO

O tamanho de codigo gerado pelo compilador varia muito dependendo do tipo de programa de teste. O tamanho de codigo gerado para um programa que possui bastante indexacao, por exemplo, ficaria bastante grande. Dependendo do tipo de atribuicao, o tamanno do codigo gerado varia muito tambem. Tres programas razoavelmente grandes foram testados para verificar o tamanho do codigo gerado.

1. Compilador NBS, Passo 1 original
   4000 linhas => 43 K bytes

2. Compilador PL/M
   1450 linhas => 15 K bytes

3. Interpretador LISP
   700 linhas => 8 K bytes

Os numeros incluem os codigos de subrotinas de suporte de execucao tambem. E' interessante notar que o tamanho de codigo gerado para o Passo 1 original ficou de tamanho

equivalente ao gerado para o PDP 11. Considerando o fato de que o microprocessador INTEL 8088 tem menos recursos de hardware do que PDP 11, este resultado e' bem satisfatorio para obter o nosso objetivo principal:
gerar codigo compacto.
O numero de bytes de codigo gerado para cada linha ficou na media de oito a treze bytes.

6.3 TESTE DE TEMPO DE EXECUCAO

Apenas dois programas foram testados para esta finalidade.

1. Programa que acha numero perfeito(ate 8128)
   DEC 10 - 6 min. INTEL 8088 - 10 min

2. Programa que acha solucoes de movimento de cavalo de xadrez saturando o tabuleiro de 5 x 5.
   DEC 10 - 40 min. INTEL 8088 - 70 min

CAPITULO 7

PROSPECTIVAS DO PROJETO

Faltam quatro funcoes para o compilador ficar completo:

1. Implementacao de E/S de arquivo em disco
   Para implementar E/S no compilador precisamos de hardware e das rotinas basicas de controle de disco. Infelizmente estas rotinas ainda nao estao prontas.

2. BOOTSTRAP
   A E/S de discos e' indispensavel para o BOOTSTRAP. O Passo 1 original deve ser modificado para adjustar para o Passo 2 nosso que gera codigo para INTEL 8088. O Passo 2 que esta' escrito para o sistema DEC 10(Compilador de Hamburgo) deve ser reescrito para o sistema INTEL 8088(Pascal NBS). O ideal seria gerar o codigo binario diretamente, porem quando o compilador gerar codigo errado, seria bastante dificil detecta-lo.

3. Recuperacao de erros
   A maioria dos metodos adotados para recuperacao de erros para analise do tipo TOPDOWN usam conjuntos de simbolos cuja cardinalidade e' cerca de 60 a 70(Numero de palavras reservadas) dependendo da implementacao. Para implementar recuperacao de erros descente, precisamos no primeiro lugar implementar conjunto de cardinalidade de 64 pelo menos. O compilador Pascal Sueco que e' tambem implementado para o minicomputador PDP 11 implmenta a recuperacao de erros do [wirt]. A recuperacao de erros no compilador NBS e' feita atraves de um procedimento chamado SKIP(stopsy : symboltype) que busca os simbolos ate que encontrar um simbol indicado pelo parametro. Normalmente quando um erro e' detectado, o compilador procura os simbolos ';' ou 'END'. Este tipo de recuperacao de erro mostrou-se impraticavel. Quando o compilador compilou o montador de INTEL 8088 escrito em Pascal de Hamburgo, o compilador detectou os erros muito

ineficientemente. A tabela em baixo mostra o numero de erros detectados em cada compilacao.

	Numero de erros detectados
Primeira ...	45
Segunda ...	32
Terceira ...	17
Quarta ...	5
Quinta ...	145

Note que depois de cada compilacao, todos erros avisados pelo compilador foram corrigidos antes da proxima compilacao.

Para melnorar a qualidade de recuperacao de erros do compilador, algumas pequenas modificacoes foram feitas. Em primeiro lugar, precisamos explicar como a recuperacao de erros e' implementada no compilador.

Existe apenas um procedimento simples denomiado SKIP para esta finalidade:

```
PROCEDURE skip(tosymbol : symbol);
BEGIN
  WHILE sym <> tosymbol do insymbol
END;
```

Este procedimento e' chamado em apenas tres lugares:

1. Chamado pelo programa principal
   Os parametros declarado no programa principal sao reconhecidos mas ignorados. Ao encontrar a declaracao de parametros do programa fonte, o compilador chama SKIP(LPAREN).

2. Chamado pelo procedimento BLOCK
   Na parte de declaracao de um bloco, se o simbolo nao estiver LABEL, CONST, TYPE, VAR, PROCEDURE, FUNCTION e BEGIN, o procedimento BLOCK chama SKIP(SEMICOLON).

3. Chamado pelo procedimento STATELIST.
   Este procedimento analisa a sequencia de comandos. O delimitador da sequencia dos comandos depende do comando que esta' sendo analisado.

   REPEAT => UNTIL

```
LOOP    =>      EXIT ou END
BEGIN   =>      END
```

O esquema principal do procedimento e' descrito a seguir:

```
PROCEDURE statelist(stopper:symbol);
BEGIN
  statement;
  WHILE (sym<>stopper) AND (sym<>endsy) DO
  IF sym = semicolon
  THEN BEGIN
   insymbol; (*pega o proximo simbolo*)
   statement (*analisa um comando*)
       END
  ELSE BEGIN
   error(14); (*simbolo ilegal*)
   skip(stopper) <==statelist(stopper)
       END
END;
```

A ideia do algoritmo e' bastante simples. Ao detectar um erro na sequencia de comandos, o compilador procura o delimitador da sequencia. Este algoritmo possui duas disvantagens imediatas.

1. O compilador nao analisa a parte a partir do primeiro erro detectado ate' o simbolo delimitador.

   Exemplo:

```
REPEAT
  IFF  i=0 THEN ...  <= erro
  ...
  ...  <= Parte nao alanlizada
  ...
UNTIL i < 10;
```

2. O compilador nao consegue distinguir o nivel de comandos. O seguinte exemplo mostra um caso inconveniente.

   Exemplo:

```
REPEAT
  IFF i=0 THEN ...
```

```
...
  ...
REPEAT
  ...
  ...
UNTIL i=j;
...
...
UNTIL i <> j;
```

Note que o segundo UNTIL e' reconhecido como o UNTIL do primeiro REPEAT correspondente neste caso.

Algumas pequenas modificacoes foram feitas para melhorar a recuperacao de erros. Uma delas foi feita no procedimento STATELIST. A modificacao feita para o procedimento STATELIST e' bastante simples. Ao inves de chamar SKIP(STOPPER), o proprio procedimento STATELIST(STOPPER) e' chamado recursivamente. Repare que SKIP e STATELIST sao iguais no sentido de condicao de parada. Sendo a modificacao bastante simples, a recuperacao de erros ficou substancialmente melhor em relacao 'a original.

A tabela a seguir mostra os numeros de mensagens de erros dadas pelo Passo 1 original e o Passo 1 modificado sobre um programa simples de 41 linhas.

Passo 1 original	Passo 1 modificado
15	42

4. Debug

A implementacao de DEBUG e' muito interessante. Como o sistema de desenvolvimento do INTEL 8088 nao possui recursos de depuracao de erros, isto seria uma ferramanta muito importante. Para implementar o DEBUG, precisariamos modificar a INTERFACE de execucao primeiro. Alem de mais, precisamos implementar a checagem do limites de dados do tipo SCALAR(SCALAR OUT OF RANGE) que esta' faltando. Se nao o DEBUG mesmo nao

teria muito sentido. Atualmente o compilador pode verificar opcionalmente indices de ARRAY, overflow, underflow e referencia ilegal pelo apontador. Uma maneira mais facil de implementar o DEBUG seria adaptar o programa de DEBUG feito para compilador de Hamburgo para o compilador NBS como o compilador Sweco fez. O DEBUG requer os seguintes itens:

1. Tabela de LOCATION COUNTER de cada linha do programa fonte.

2. Tabela de todos os identificadores declarados no programa com as informacoes de tipo, atributo, limites, tamanho etc.

3. O apontador de inicio da tabela de simbolos declarados na rotina corrente na pilha.

4. Um pequeno interpretador para comunicar com o usuario durante o DEBUG. O interpretador deve ter uma rotina que calcula o endereco de cada variavel ou subcampo no registro.

[Poul78]

APENDICE A

TABELA DE CODIGOS-P DO COMPILADOR NBS

Existem seis Codigos-P do tipo PSEUDO. Estes codigos servem para controlar a montagem de arvore de sintaxe.

0 ... NOP : Faz nada
1 ... XCH : Troca os dois elementos que estao no topo da pilha de montagem.
2 ... DEL : Se a pilha nao estiver vazia, entao abaixa o topo da pilha de um.
5 ... IDENT : Pega o no' da rotina correntemente montada.
6 ... PROC : Reconhecimento de nova rotina para montar. O nivel lexico e' somado de um e o tipo da rotina e' verificado.
7 ... ENDOP : Reconhecemento de fim de montagem da rotina correntemente montada. Alem disso, pega as informacoes adicionais:
1. Numeracao da rotina
2. Memoria alocada para parametros
3. Memoria alocada para variaveis
4. Tamanho de valor da funcao

Existem tres tipos de Codigod-P basicamente:

1. <u>Codigos para comandos :</u>
STCL .. STOF : Atribuicao
MOVEM : Atribuicao
INVOK : Chamada de Builtin rotina
DTEMP,RTEMP : Comando WITH
IFOP : Comando IF
CASEOP,ENTRY : Comando CASE
LOOPOP,EXITOP: Comandos WHILE,REPEAT,LOOP

FORDP : Comando FOR
SEQ : Comando composto

2. Codigos para operacoes :

Existem cinco tipos de operacoes:

1. Operacao ou comparacao de um byte.
   UCEQ .. UMIN

2. Operacoes ou comparacao de dois bytes.
   IADD .. IMIN

3. Operacao ou comparacao sobre tipo booleano.
   EQV .. ANDOP

4. Operacao ou comparacao sobre tipo conjunto.
   UNION .. SANY

5. Operacao ou comparacao sobre vetores de caracteres.
   VCEQ .. VCLT

3. Codigos para dados ou enderecos :
Existem dois tipos basicos:

1. Codigo para calcular enderecos.
   FIELD .. INDEX

2. Codigos de enderco ou dados basicos.
   VARBL .. VARBL15 , PARAM .. PARAM15 ,LITER , RDATA
   (Os numeros significam niveis lexicos.)

Tabela de Codigos-P no Compilador N&S

0 NOP	1 XCH	2 DEL
4 NEW NOTA: Este codigo nao consta no original		
5 IDENT	6 PROC	7 ENDOP
8 NULL	9 REFER	10 STOL
11 STOR	12 STOF	16 SUCCOP
17 PREDOP	24 UCEQ	25 UCNE
26 UCGT	27 UCLE	28 UCGE
29 UCLT	30 UMAX	31 UMIN
32 IADD	33 ISUB	34 IMUL
35 IDIV	36 IMOD	40 INEG
41 IABS	42 IODD	44 CEIL
45 FLOOR	56 ICEQ	57 ICNE
58 ICGT	59 ICLE	60 ICGE
61 ICLT	62 IMAX	63 IMIN
64 FADD	65 FSUB	66 FMUL
67 FDIV	72 FNEG	73 FABS
74 FLOATOP	75 TRUNCOP	76 ROUNDOP
88 FCEQ	89 FCNE	90 FCGT
91 FCLE	92 FCGE	93 FCLT
94 FMAX	95 FMIN	96 NOTOP
104 EQV	105 XOR	106 NIMP
107 RIMP	108 IMP	109 NRIMP
110 OROP	111 ANDOP	112 COMPL
113 UNION	114 INTER	115 SDIFF
117 SGENS	118 SADEL	119 EMPTY
120 SCEQ	121 SCNE	122 SCGT
123 SCLE	124 SCGE	125 SCLT
126 SIN	127 SANY	131 FIELD
132 OFSET	133 INDIR	134 INDEX
135 MOVEM	138 INVOK	140 RTEMP
141 DTEMP	144 IFOP	145 CASEOP
146 ENTRY	147 LOOPOP	148 EXITOP
149 FOROP	152 SEQ	162 LITEP
163 RDATA	164 LITD	168 VCEQ
169 VCNE	170 VCGT	171 VCLE
172 VCGE	173 VCLT	176 VARBL
176 VARBL0	177 VARBL1	178 VARBL2
179 VARBL3	180 VARBL4	181 VARBL5
182 VARBL6	183 VARBL7	184 VARBL8
185 VARBL9	186 VARBL10	187 VARBL11
188 VARBL12	189 VARBL13	190 VARBL14
191 VARBL15	192 PARAM	192 PARAM0
193 PARAM1	194 PARAM2	195 PARAM3
196 PARAM4	197 PARAM5	198 PARAM6
199 PARAM7	200 PARAM8	201 PARAM9
202 PARAM10	203 PARAM11	204 PARAM12
205 PARAM13	206 PARAM14	207 PARAM15
208 CALL	208 CALL0	209 CALL1
210 CALL2	211 CALL3	212 CALL4
213 CALL5	214 CALL6	215 CALL7
216 CALL8	217 CALL9	218 CALL10
219 CALL11	220 CALL12	221 CALL13
222 CALL14	223 CALL15	

APENDICE B

Lista de rotinas de suporte durante execucao

1. CHKMEM :
Compara o nivel da pilha no SP e proximo endereco disponivel para area de HEAP que esta' armazenado no endereco HEAPPTR. Se o valor no SP estiver maior do que o valor que esta' no HEAPPTR entao dar a mensagem "NO MORE MEMORY TO ALLOCATE" e volta o controle ao monitor. Esta rotina e' chamada na entrada de rotina e chamada do comando NEW incondicionalmente.

2. NILREF :
Ao referir um endereco atraves de um apontador, esta rotina verifica se o valor do pontador e' NIL. Se o valor estiver NIL dar a mensagem "REFERENCE TO NIL" e' volta o controle ao monitor. Esta rotina e' chamada opcionalmente. Ao executar o Passo 2, o programador pode indicar a option RUNCHK. O valor NIL e' representado como 0 no caso do compilador NBS.

3. ILLREF :
Ao referir um endereco que esta' indicado atraves de um apontador, esta rotina verifica se o endereco indicado esta' na area HEAP ou nao. Esta rotina tambem e' chamada opcionalmente.

4. IDXCHK :
Ao referir um campo do tipo ARRAY, esta rotina verifica se o valor de indice calculado durante execucao esta' entre nos limites inferior e superior. Se nao da' a mensagem "ARRAY INDEX OUT OF RANGE AT ..." e devolve o controle ao monitor. A diferenca entre o Compilador de Hamburgo do DEC-10 e o nosso compilador neste caso e' que o nosso informa o numero de linha do programa fonte onde este tipo de erro ocorreu durante execucao. Esta rotina e' chamada opcionalmente tambem.

5. CSCHK :
Ao calcular o valor da expressano do comando CASE, esta rotina verifica se o valor calculado e' menor do que de 1 byte. (O valor da expressao do comando CASE no caso do Compilador NeS e' representado por um byte apenas). Esta rotina e' chamada opcionalmente.

6. WRINI :
Rotina que imprime o valor que esta' no registrador AX para TTY.

7. RDINI :
Rotina que le um numero inteiro de TTY e devolve o valor lido no registrador AX. Se haver um error na leitura, a rotina da' a mensagem "ILLEGAL INPUT DATA" e devolve o controle para o monitor.

8. WRIBL :
Rotina que imprime o valor booleana que esta' no registrador AX.

9. RDEOLN :
Rotina que le ate a fim da linha no TTY.

10. WRISTR :
Rotina que imprime uma cadeia de caracteres no TTY. O endereco inicial da cadeia esta' no registrador SI e o numero de caracteres para imprimir esta' no registrador AX. Estes valores sao inicializdos antes da chamada da rotina pelo Passo 2.

11. RDCHAR :
Rotina que le um caracter do TTY e armazenar o codigo do caracter lido no registrador AL.

12. WICHAR :
Rotina que imprime o caracter correspondente do valor que esta' no registrador AL para o TTY.

13. WIREAL :
Rotina que imprime o valor do tipo real que esta' no registrador AX no format FOCUS para TTY.

14. EMUL :
Rotina que multiplica os valores do tipo real que estao no AX e CX e armazena o resultado no AX.

15. EDIV :
Rotina que multiplica os valores do tipo real que estao no AX e CX e armazenar o resultado no AX.

16. ESQR :
Rotina que calcula o quadrado que esta' no AX e devolve o resultado no AX.

17. EMAX :
Rotina que calcula o valor maximo dos numeros reais que estao no AX e CX e devolve o resultado no AX.

18. EMIN :
Rotina que calcula o valor minimo dos numeros reais que estao no AX e CX e devolve o resultado no AX.

19. VCEQ VCNE VCGI VILE VCGE VCLI :
Rotina que compara duas cadeias de caracteres e devolve o resultado booleano no registrador AL. Os enderecos iniciais de cadeias estao nos registradores SI e DI e o comprimento de cadeias esta' no registrador CX.

APENDICE C

Lista de opcoes de compilacao do compilador

1. SELECT :
   O Passo 2 pergunta se cada rotina do programa fonte deve ser compilado interativamente pelo terminal.

2. CHECK :
   Verifica se a arvore de sintaxe montada no Passo 2 esta' correta ou nao.

3. KEEP :
   Normalmente o Passo 2 apaga o arquivo que contm codigo intermediario gerado pelo Passo 1 ao terminar a execucao. Com esta' opcao, o Passo 2 deixa o arquivo. Esiste dois programas de suporte; TREE.PAS e PCODE.PAS que analiza o codigo intermediario.

4. TRACE :
   O Passo 2 marca o inicio e o fim de cada comando compilado para o codigo gerado em linguagem de montagem.

5. RUNCHK :
   O Passo 2 gera codigo para verificacao em tempo de excucao do tipo:

   1. ARRAY INDEX OUT OF BOUND
   2. REFERENCE TO NIL
   3. ILLEGAL REFERENCE OF MEMORY
   4. CASE SELECTER BIGGER THAN ONE BYTE
   5. OVERFLOW E UNDERFLOW

6. DUMPI :
O Passo 2 mostra todos os passos de modificacao de arvore de expressao feita para a otimizacao de codigo. Desta maneira podemos verificar se a modificacao de arvore de expressao e' feita corretamente.

7. NOLST :
O Passo 1 nao gera o arquivo de listagem de compilacao.

8. NOTAB :
O Passo 1 nao mostra a tabela de simbolos e seu enderecos na listagem.

9. NOMAC :
O Passo 2 nao copia as rotinas intrinsecas no inicio do codigo gerado.

10. PRINI :
O Passo 2 imprime a arvore de sintaxe de cada rotina do programa fonte antes de gerar codigo em linguagem de montagem do INTEL 8088.

11. STA :
O Passo 2 mostra a estatisticas de tipos de atribuicoes e a otimizacao PEEPHOLE feita no registrador AX e BX sobre o programa fonte.

APENDICE D

BIBLIOGRAFIA

1. Aho, Alfred
   Principles of Compiler Design
   Addison-Wiley 1977

2. Amman, V. Nori. U
   The Pascal <P> Compiler: Implementation Notes:
   Berichte des Instituts fur Informatik

3. Amman, U.R.S
   On Code Generation in a Pascal Compiler
   Sowtware-Practice and experiences, vol. 7, 1977

4. Atkinson, L.V.
   Optimizing Two-state Case Statements in Pascal
   Sowtware-Practice and experiences, vol. 12, 1982

5. Barr, John R.
   Pascal for RSX-11D and RSX-11M
   Proceddings od Digital Equipment Computer Users Society
   1977

6. Barron, David William
   Pascal: The language and its implementation
   Wiley, 1981

7. Bell, James R.
   Threaded Code
   Comm. ACM jun 1973, vol 16

8. Berry, R.E.
   Experiences with the Pascal P-Compiler
   Sowtware-Practice and experiences, vol. 8, 1978

9. Bron, C.
   A Pascal Compiler for PDP 11 Minicomputers
   Sowtware-Practice and experiences, vol. 6, 1976

10. Cael, William
RT/RSX : RT - 11 as a SRX-11/M TASK
Jorn. Proceeedings of Digital Equipment Computer Users Society
1977

11. Colin, A.J.T.
Note on coding Reverse Polish expression for Single-address computers with one accumulator
Computer Journal, vol 6, 1963

12. Cornelius, B.J.
Modification of the Pascal-P compiler for a Single-accumulator One-address Minicomputer
Software-Practice and experience, vol 10,1980

13. Floyed, Robert W.
An Algorithm for Coding Eficient Arithmatic Operations
Comm. ACM

14. Forsyth, Charles
Compiler and Pascal in the new Microprocessores
BYTE august 1979

15. Gries, David
Compiler Construction for Digital Computers
Wiley, 1971

16. Hartman, A.C.
Architectural issues in the design of the INTEL 8086
Electronic engineering, Dcember 1980

17. Hopgood, F.R.A.
Compiler Technics
Mac Donald, 1969

18. Kowaltowski, Tomasz
Runtime systems for programing language

19. Kowaltowski, Tomasz
Implementacao de linguagem de programacao
Escola de Computacao, 1979

20. Lecarme, oliver
Self-compiling Compilers: An Appraisal of their Implementation and Portability
Software-Practice and Experiences, vol 8, 1978

21. Mckeeman, W.M.
Peephole Otimization
Comm. ACM. vol 8. November 1965

22. Poulsen, C. Steven
A high level symbolic DEBUGGER for Pascal
Proceedings of Digital Equipment Computer Users Society

1978

23. Ravenel
Toward a Pascal Standard
Computer, april 1979

24. Richmond
Proposals for Pascal
Pascal Newsletter, No.8 May 1977

25. Sale, Arthur
The Implementation of CASE Statements in Pascal
Sowtware-Practice and experiences, vol. 11, 1981

26. Schneider, G. Michael
Pascal: An overview
Computer, April 1979

27. Stevenson, Gordon
Code Genaration with a Recursive Optimizer
Sowtware-Practice and experiences, vol. 10, 1980

28. Ullman,
The generation of optimal code for arithmatic expressions
J. ACM . vol 17, 1970

29. Wilson, I.R.
Pascal for School and Hobby Use
Software-Practice and Experiences, vol 10 1980

30. Wirth, Niklaus
Pascal: User a Manual and Report
Springer, 1976

31. Wirth
Algorithms + Data structures = Programs
Printice-Hall, 1976

32. Wirth
Systematic Programming
Printice-Hall, 1973