Manual do Implementador

Relatório de Implementação de um Compilador do Pascal Simplificado

Curso de Ciências da Computação - Integral
Unesp – Rio Claro

Disciplina: Compiladores
Professor Doutor Eraldo P. Marinho
Alunos: Jonni Borges Surano
Leonardo Mellaci de Matos

Sumário:

1- Introdução:

Esse documento visa explanar a construção do compilador para o Pascal simplificado, que foi o último trabalho definido para a conclusão do curso de compiladores. Tal trabalho foi realizado através da linguagem C, implementada no compilador Turbo C++, versão 1.0. O sistema operacional escolhido foi o Windows, devido à familiaridade que com ele possuímos. Utilizamos como referência o livro "Compiladores: princípios, técnicas e ferramentas", de Alfred V. Aho, Ravi Sethi e Jeffrey D. Ullman. Para a construção do parser que seria utilizado pelo compilador, utilizamos a descrição do Pascal simplificado do Wirth.

2- Considerações iniciais:

Algumas simplificações sobre a linguagem pascal foram adotadas para a realização do trabalho. São elas:

- Foi excluído o operador aritmético "div";
- Foi excluído o apontador "^" (declaração de variáveis dinâmicas);
- Foram excluídos os símbolos de classes de estruturas ("array file");
- Constantes só podem assumir os valores "false", "true". Não é permitida a inclusão de novos tipos de constantes;
- Tipos só podem ser "boolean", "integer", "char", "real" e "text". Não é permitida a inclusão de novos tipos de variáveis.
- Cada condição do "case" deve estar separada, mesmo que resultem no mesmo caso.

Ex:

Caso Proibido:

case <expressão> of

‘A’, ‘B’: <comando>;

Caso Aceito:

case <expressão> of

‘A’: <comando>;

‘B’: <comando>;

3- Metodologia:

A técnica utilizada para o desenvolvimento do analisador sintático foi à técnica “top down” preditiva recursiva. Por “top down”, entende-se que a análise começa com o símbolo inicial da gramática, prevendo o próximo símbolo que será lido e escolhendo regras de produção aceitáveis. Por “preditiva”, entende-se que haverá uma previsão de qual será o próximo “token” a ser lido na linguagem para então escolher a regra de produção correta para a geração da sentença.

4- Sobre o Compilador:

Ao executar o programa, o usuário digita o nome do arquivo que deseja executar. Esse nome pode estar sem extensão ou com a extensão “.pas”. Se o arquivo não tiver extensão alguma, o programa lê o arquivo como se ele tivesse a extensão “.pas”. Por

exemplo, se o usuário digita compilador no nome do arquivo, o programa lê como arquivo de entrada “compilador.pas”. Após a leitura do nome do arquivo, o programa faz a compilação do arquivo através das rotinas que serão explicadas adiante. Feita a compilação, o código de máquina fica armazenado em um arquivo com o mesmo nome do original, porém com extensão “.obj”. No exemplo anterior, o arquivo seria “compilador.obj”.

Para gerarmos o código de máquina utilizamos o padrão Intel, em detrimento do padrão AT&T.

A tabela de símbolos foi criada como um vetor de registro com quatro campos: “symbol”, “tipo”, “objtipo” e “offset”. Os três primeiros são “strings” e o “offset” é um valor inteiro. O campo “symbol” armazena o nome da variável, “tipo” armazena o tipo da variável, que pode ser “integer”, “boolean”, “char”, “text” ou “real”, “objtipo” armazena o valor referente ao tipo de objeto que a variável é, que pode ser “variável”, “procedimento” ou “função”, e “offset” armazena o deslocamento da variável, se for necessário. Cada variável equivale a uma posição no vetor.

A declaração de funções e procedimentos é feita da seguinte forma: antes da declaração de cada subrotina, há um salto para a próxima posição após a declaração da subrotina. Isso é feito para que o programa, quando executado, siga normalmente sem passar pela subrotina. Na declaração, a primeira coisa a se fazer é colocar o rótulo referente à subrotina. Esse rótulo é o próprio nome da subrotina. Se o nome da subrotina for “adicionar”, então o rótulo será “adicionar”. Após isso, é feita a declaração da subrotina e no final é escrito no arquivo de saída a função “ret”. A chamada de uma subrotina é simples: basta empilhar os parâmetros e depois chamar a subrotina através de “call <nome_subrotina>”.

A conversão de tipo é feita da seguinte maneira: “tipoa” armazena o tipo da última expressão. Se ele for “–1”, a expressão atual é a primeira expressão, então “tipoa” é o tipo atual. Essa variável é necessária porque, quando é feita uma atribuição, o tipo da variável é comparado com “tipoa”, e se eles forem equivalentes, a atribuição pode ser feita. Senão, o programa é encerrado. Se for feita uma operação, “tipoa” é comparada com o tipo da última variável lida, e se eles forem equivalentes, a atribuição pode ser feita. Senão, o programa é encerrado.

5- Variáveis Globais:

Tabela – tipo que define uma tabela de símbolos. É definido por 4 campos: “symbol”, “tipo”, “objtipo” e “offset”;

tabsimb[100] – tabela de símbolos com um máximo de 100 posições;

buffer – define o arquivo a ser compilado. Esse arquivo será do tipo “*.pas”;

buffer2 – define o arquivo que receberá o código objeto. Esse arquivo será do tipo “*.obj”;

lookahead – indica qual é o próximo token;

indsimb – representa a última posição usada na tabela de símbolo. Inicialmente é “–1”, pois não há elementos na tabela;

L – representa o próximo rótulo a ser definido. Inicialmente é “0”, ou seja, o próximo rótulo é “0”;

Argumentos – armazena o número de argumentos da definição de uma função ou procedimento;

Compare – variável auxiliar usada no procedimento expressão. Se ela for “0”, não deve haver comparação, ou seja, não deve-se escrever no buffer de saída um desvio condicional, ao fim do procedimento expressão; caso contrário deve haver comparação

Princ – variável auxiliar que indica se a análise que está sendo feita pertence ao programa principal ou não. Se for “1”, quer dizer que o compilador está no programa principal; caso contrário, não está;

Indtroca - representa a última posição usada na tabela de trocas. Inicialmente é “–1”, pois não há elementos na tabela;

Nível – nível léxico atual do programa. Inicialmente é zero;

trocas[20][2] – tabela que armazena as variáveis que devem ter seu nome trocado pelo seu offset em relação à pilha,que é do tipo “8(%ebp)”;

lexeme – guarda o valor do último lexeme lido pelo analisador léxico;

simbolo[54] – armazena as palavras reservadas em pascal.

6- Módulos do Compilador:

6.1- Analisador léxico:

O analisador léxico de nosso compilador contém três funções: “gettoken”, “match” e “converte”.

A função “gettoken” encontra o próximo token que o compilador reconhece, ou seja, o próximo buffer presente no buffer de entrada. A função funciona da seguinte forma: a função lê o buffer de entrada caracter por caracter até que o caracter encontrado não seja um espaço em branco. Ao encontrar esse caracter não branco, é necessário saber de que

tipo ele é. Portanto, são feitas sucessivas comparações, inicialmente para saber se o caracter é uma letra, e, se não for, é feita uma comparação para saber se ele é um número. Se não for um número, o caracter será um sinal ou um símbolo de pontuação.

Se o caracter for uma letra, o caracter lido é devolvido ao buffer e lê-se a palavra inteira correspondente a ele. Após ler essa palavra, a mesma é comparada com as palavras reservadas, que são pré-definidas no vetor de strings “simbolo”. Se elas forem iguais, a função retorna o token correspondente à palavra reservada. Se a palavra lida não for igual a nenhuma palavra reservada, verifica-se se ela já foi adicionada na tabela de símbolos “tabsimb”. Se já, retorna o tipo definido na tabela de símbolos. Esse tipo, na tabela de símbolos é uma “string”. Portanto, ela deve ser convertida para inteiro, e isso é feito através da função “converte”. Essa função verifica se o tipo do token definido na tabela de símbolos é uma variável, um procedimento ou uma função, e então é convertido através da função para o tipo correspondente, retornando esse tipo. Se a palavra lida não está presente na tabela de símbolos, a função insere a palavra na tabela de símbolos, incrementando o seu índice, e retorna ID.

A função “match” confere se duas variáveis, o “lookahead”, que representa o próximo token, e a variável passada como parâmetro, são iguais. Se elas forem, é lido o próximo token, sendo esse armazenado em “lookahead”. Se elas forem diferentes, o programa é encerrado e é exibida uma mensagem de erro.

6.2- Rotulo:

O subprograma rótulo.c possui dois procedimentos: “emitlabel” e “emitgo”. O procedimento “emillabel” insere um rótulo da forma “.L <num>”, onde “<num>” é um

parâmetro passado na chamada da rotina. O procedimento “emitgo” insere uma condição de desvio de acordo com a variável relação, que, apesar de não ser usada como parâmetro, é usada no procedimento, pois é uma variável global. Essa variável armazena o valor da comparação para qual é feito o desvio, e emite a instrução de desvio de acordo com a relação. Se a relação for “=”, é emitida a condição de desvio “jnz”; se a relação for “<>”, é emitida a condição de desvio “jz”; se a relação for “<”, é emitida a condição de desvio “jnb”; se a relação for “>”, é emitida a condição de desvio “jna”; se a relação for “<”, é emitida a condição de desvio “jb”; se a relação for “>”, é emitida a condição de desvio “ja”. O desvio é feito para o rótulo “label”, passado como parâmetro no procedimento.

6.3- Tipo:

O subprograma “tipo.c” define a conversão e comparação de tipos no compilador Pascal. Este subprograma possui dois procedimentos, “tipoop” e “tipoat”, e uma função, “convtipo”.

O procedimento “tipoop” verifica o tipo de uma operação, como, por exemplo, uma soma ou uma multiplicação. Esse procedimento verifica se “tipoa”, que representa o tipo atual da operação, e “tipo1”, que é um parâmetro, são equivalentes. Se a primeira operação feita é a atual, “tipoa” passa a ser o tipo do token atual. Senão, verifica-se a equivalência entre “tipoa” e “tipo1”. Se eles forem iguais, são equivalentes. Senão, verifica-se se o tipo atual é inteiro. Se for, o “tipo1” pode ser real, pois nesse caso há equivalência de tipos. Se não houver equivalência, exibe-se uma mensagem de erro e o programa é finalizado.

O procedimento “tipoat” verifica se é possível realizar uma atribuição. Isso ocorrerá se o “tipoa”, que representa o tipo atual, for igual ao tipo passado como parâmetro. Se eles

forem diferentes, há a possibilidade de ser feita a atribuição se o tipo atual for inteiro e o tipo da variável que terá a ela um valor atribuído for real. Se não houver equivalência na atribuição, será exibida uma mensagem de erro e o programa será finalizado.

A função “convtipo” converte um tipo e string para inteiro. Isso deve ser feito para a utilização dos procedimentos de conversão de tipo “tipoop” e “tipoat”. A conversão é feita da seguinte maneira: a string “tipo1” é passada como parâmetro e é feita uma comparação dela com os tipos definidos “boolean”, “integer”, “char”, “real” e “text”. Se “tipo1” for “boolean”, o valor retornado será “0”; se “tipo1” for “integer”, o valor retornado será “1”; se “tipo1” for “char”, o valor retornado será “2”; se “tipo1” for “real”, o valor retornado será “3”; e se “tipo1” for “text”, o valor retornado será “4”.

6.4- Símbolo:

O subprograma “simbolo.c” serve para a manipulação da tabela de símbolos. Ela tem um procedimento, “insert”, e uma função, “encontra”. O procedimento “insert” serve para inserir elementos na tabela de símbolo. Isso é feito incrementando o índice da tabela “(indsimb)” e copiando o atual lexeme para a posição atual da tabela de símbolos, no campo “symbol”. Não são necessários parâmetros nesse procedimento porque a inserção é feita usando o lexeme lido antes da leitura do próximo token..

A função encontra retorna a posição relativa a um identificador na tabela de símbolos. É bom ressaltar que essa posição refere-se ao último identificador com nome igual ao procurado encontrado na tabela. A busca é feita através de um laço Para, que percorre toda a tabela de símbolos, e retorna quando encontrado, o índice relativo à posição atual. Se houver dois símbolos com o mesmo nome na tabela, a posição retornada será a do

último símbolo, porque o laço continua até o fim da tabela. Como essa função só é acessada se o símbolo existe, não há a necessidade de verificar qualquer tipo de condição para o caso de o símbolo não existir na tabela.

6.5- Main:

O programa “main.c”, que é o programa principal, define o arquivo que deve ser compilado e inicia a compilação. Inicialmente, o usuário escolhe o arquivo que deseja compilar. Esse nome de arquivo pode ser escrito de duas formas: sem qualquer extensão, ou seja, se o usuário deseja compilar o arquivo “Compilador.pas”, ele digitaria “Compilador”. O programa transforma o nome em “Compilador.pas” e esse é o arquivo compilado. A outra é digitar “Compilador.pas”, e o programa compilará esse arquivo. O arquivo com o código em assembler gerado será o arquivo com o mesmo nome do anterior, porém com extensão “.obj”. Outra ação de “main.c” é o inicio da compilação, com a chamada de programa.

6.6- Analisador sintático:

O analisador sintático, ou “parser”, é a parte mais extensa do código do compilador. Ele possui várias funções que serão melhor explicadas na seqüência.

Uma delas é a função “tamanho”, que retorna o tamanho de um tipo de variável. Se a variável for “boolean”, o valor retornado será “5”; se for “integer”, o valor retornado será “4”; se for “char”, o valor retornado será “1”; se for “real”, o valor retornado será “8”; e se for “text”, o valor retornado será “255”. Essa função é importante no cálculo do tamanho

reservado de cada variável e também no cálculo dos valores de deslocamento em relação à pilha, do modo “8(%ebp)” e “–4(%ebp)”.

A função “troca” procura na tabela “trocas”, que guarda as posições em relação à pilha das variáveis usadas como parâmetro e também como locais nas funções e procedimentos, o nome “nome1” de uma variável. Se esse nome estiver presente na tabela, então “nome1” passa a ser a posição da variável na pilha, ou seja, passa a ser uma expressão do tipo “8(%ebp)” ou “–8(%ebp)”. Isso deve ser feito, pois nos procedimentos e funções as variáveis são chamadas dessa forma, e não pelo nome.

O procedimento “num_sem_sinal” encontra um número sem sinal, sendo esse inteiro ou real. No nosso compilador Pascal simplificado, quando um número for real ele é convertido para inteiro ao final do procedimento. No compilador, o resultado, ou seja, o número sem sinal é armazenado em “%eax”.

O procedimento “cons_sem_sinal” encontra um número sem sinal, uma constante entre aspas da forma “ ‘constante’ ” ou um identificador de constante que pode ser “false” ou “true”. No compilador, o resultado dessa operação é armazenado em “%eax”.

O procedimento constante encontra um número com sinal, um identificador de constante ou uma expressão qualquer entre aspas. No compilador, o resultado dessa operação é armazenado em “%eax”.

O procedimento “fator” encontra uma constante sem sinal, um identificador de variável, uma função, um fator negado ou uma expressão entre parênteses. No compilador, o resultado da operação é armazenado em “%eax”.

O procedimento “termo” encontra um fator e pode multiplicar, dividir encontrar o resto ou fazer operação lógica e desse fator encontrado com outro fator. No compilador, o resultado da operação é armazenado em “%eax”.

O procedimento “expressão_simples” encontra um termo e pode adicionar, subtrair e ou fazer operação lógica ou desse termo com outro termo. No compilador, o resultado da operação é armazenado em “%eax”.

O procedimento “expressão” encontra uma expressão simples e pode fazer uma comparação, sendo esta igual, diferente, maior que, menor que, maior ou igual que ou menor ou igual que, dessa expressão simples com outra expressão simples. No compilador, o resultado da operação é armazenado em “%eax”.

O procedimento “lista” encontra uma lista de parâmetros, usada na declaração de procedimentos e funções. São lidos os parâmetros um a um, e o número total de parâmetros é dado pelo contador “nvar”, que é incrementado a cada vez que é feita a verificação de que existe um novo parâmetro. Nesse procedimento é definido também o tipo da variável, que sempre será “VAR”, já que não é possível declarar uma função ou um procedimento como parâmetro. A variável “offs” representa o offset máximo atual, e só é alterada quando é encontrado o tipo do da variável e conseqüentemente seu tamanho. “Argumento” é uma variável global que conta o número de argumentos e é usada para contar quantos parâmetros foram usados na lista de parâmetros. Após a leitura de todos os parâmetros de um certo tipo, o campo “offset” referente a cada símbolo armazenado é alterado com o offset atual, que é incrementado do tamanho da variável. Além disso, cada variável declarada deve ser armazenada na tabela de trocas, já que, quando for feita uma referência a ela, o compilador deve entender que está sendo feita uma referencia à pilha na posição do offset da variável. Por isso, deve-se incrementar o índice de troca e copiar na posição atual da tabela de trocas o nome da variável e seu offset concatenado com o ponteiro da pilha “%ebp”. Após isso, verifica-se se o próximo item é um parâmetro. Se não for, o procedimento é encerrado. Se for, é feita a verificação do(s) próximo(s) parâmetro(s).

O procedimento comando encontra e executa um comando da linguagem Pascal. Esse comando pode ser um conjunto de comando, que é feito através da instrução “Begin-End”, um laço, que pode ser “If-Then_else”, “While-Do”, “Repeat-Until”, “For-to” ou “Case”, uma atribuição, um rótulo ou uma chamada de rotina ou função Há dez condições possíveis de execução desse procedimento:

- Se a instrução for “Begin-end”, o procedimento comando é realizado continuamente até que o próximo token não seja “;”.
- Se for um laço “If-Then-Else”, a variável compare, que indica quando deve ser colocada uma comparação ao final do procedimento expressão, é tornada verdadeira(==1). Após a chamada de expressão, o tipo atual é tornado inválido e é emitido um rótulo de desvio, que será criado no início do “else”, se esse houver, ou no final do laço. Após isso o procedimento comando é executado novamente. Se depois dessa execução houver um comando “Else”, é feito um salto para o final do laço, usado para instruções que passaram pelo laço “if-then”, e realizado um novo comando , referente às instruções que devem passar pelo laço “else”.
- Se for um laço “Case”, a variável “compare”, que indica quando deve ser colocada uma comparação ao final do procedimento expressão, é tornada verdadeira “(==1)”. Após a chamada da expressão, o tipo atual se torna inválido e são feitas comparações da expressão com as constantes de cada condição do laço, sendo emitidas condições de comparação. Se a comparação for verdadeira, é executado o comando referente à condição e, após ele ser encerrado, o programa é saltado para o fim do laço. Se a

comparação for falsa, o programa salta para a próxima condição, fazendo a comparação dela com a expressão, sendo essa comparação feita até o que se encontre alguma comparação verdadeira ou o laço chegue ao seu fim.

- Se for um laço “while-do”, é emitido um label que designa o início do laço. Após isso, a variável “compare”, que indica quando deve ser colocada uma comparação ao final do procedimento expressão, se torna verdadeira “(==1)” e o tipo atual se torna inválido. Se a comparação feita for verdadeira, o programa salta para o fim do laço. Senão ele executa um comando e retorna para o inicio do laço “while”, através do rótulo definido anteriormente.
- Se for um laço “repeat”, é emitido um rótulo que designa o ínicio do laço “repeat”. Após isso, é executado um comando e, depois, a variável “compare”, que indica quando deve ser colocada uma comparação ao final do procedimento expressão, se torna verdadeira “(==1)”. Se a comparação feita for verdadeira, é feito um salto para o início do laço “repeat”. Senão, o laço é finalizado.
- Se for um laço “for”, a variável “compare”, que indica quando deve ser colocada uma comparação ao final do procedimento expressão, se torna falsa “(==0)”. Essa expressão indica o limite mínimo do laço. Depois, é feita a verificação de tipo para saber se a expressão é aceita pela variável que deve recebê-la. Após isso, é encontrado limite máximo do laço e então é feita a subtração entre os dois limites, que representa o número de passos que deve ser feito pelo laço “for”, sendo este valor armazenado no contador

"%ecx". Então, o rótulo inicial é definido, e é executado um comando. Se o contador for maior que zero, o laço é executado novamente através de um salto para o rótulo de início do laço através da instrução "loop", que decrementa o contador automaticamente. Se o contador "%ecx" for zero, o laço se encerra.

- Se for um laço "goto", é feito um salto incondicional para o rótulo definido por lexeme.
- Se for um identificador de função ou um identificador de variável, a variável é encontrada na tabela de símbolos, e, se necessário, é feita a troca do nome que será exibido em linguagem de máquina através da tabela trocas. Então, a variável "compare" se torna falsa "(==0)". Após isso, é feita uma verificação de tipo, e se o tipo for aceito, é feita a atribuição do identificador com o valor pela expressão, através da instrução "mov".
- Se for um identificador de rotina, a variável "compare" se torna falsa "(==0)". Então, os parâmetros encontrados por expressão são empilhados e a função é chamada através da função "call".
- Se for uma constante, simplesmente é definido um rótulo para essa constante.

O procedimento bloco define a declaração de novas variáveis, de novos procedimentos de novas funções e do programa em si.

Se a ação realizada for uma declaração de variáveis, verifica-se se a declaração faz parte do programa principal ou não. Se fizer, deve-se escrever ".section .bss" no arquivo gerado pelo compilador e são escritas as variáveis com os seus respectivos nomes e

tamanhos. Se não fizer parte do programa principal, deve-se calcular o offset e com ele criar a tabela “trocas”, usando para isso o nome da variável no primeiro item da matriz e no segundo o offset concatenado com a variável referente à pilha “%ebp”. Após a fim da declaração de variáveis, verifica-se se o próximo token define um procedimento ou uma função.

Se definir um procedimento ou função, a variável “nível”, que define o nível léxico atual, é incrementada, e é criado um rótulo para que seja feito um salto incondicional “ret<nome_var>”, ou seja, se for feito um procedimento com o nome “func”, o rótulo criado será “retfunc”. Esse rótulo será criado no final da declaração do procedimento ou função. Isso deve ser feito, pois quando o programa é executado, ele não deve passar por essa parte do código, que só deve ser acessada se houver uma chamada à função ou procedimento através da instrução “call”. Após essa declaração, é criado o procedimento, que se inicia com o rótulo “<nome_var>”, que é feito de modo análogo ao anterior. O prólogo de uma subrotina é definido pelas instruções “push %ebp”, “mov %ebp, %esp”, “add %esp, $-4”. Depois é chamado o procedimento “lista” e o procedimento “bloco”, que representa um bloco de instruções que serão analisadas pelo compilador. Após isso, é introduzido o epílogo do programa, definido pelas instruções “mov%esp, %ebp”, “pop %ebp”, “ret”, esta última saindo da execução da função ou procedimento. O índice da tabela de trocas recebe o valor anterior ao inicio da definição do subprograma, já que após o procedimento ou função ser encerrado, não há a necessidade de se acessar as variáveis definidas no subprograma. Se o próximo token indicar que há outra função ou procedimento a ser executado, ele é executado do mesmo modo que o anterior.

Caso a ação executada for a declaração do programa em si, verifica-se se o nível léxico é zero; se for escreve-se no arquivo de saída “global _start”,”_start:”. Após isso, são executados sucessivos comando, até que seja encontrado o fim da execução do arquivo.

O procedimento programa executa o procedimento bloco, ou seja, executa um bloco de instruções. Ao fim desse bloco, ele escreve no arquivo de saída “int $128”, que delimita o fim da execução do programa em linguagem de máquina.

O procedimento rotinas define as rotinas read e write do Pascal no compilador. Essas funções foram criadas como exemplo de como criar rotinas predefinidas no compilador.

7- Análise Sintática:

A gramática utilizada para a construção da análise sintática do compilador, já com as devidas simplificações, foi a seguinte:

identificador := **letra** (**letra** | **digito**)*

inteiro sem sinal := (**dígito**)$^{+}$

número sem sinal := *inteiro sem sinal* (. (**digito**)$^{+}$)? (**E** (**+** | **-**)? *inteiro sem sinal*)?

constante sem sinal := *identif. de constan.* | *número sem sinal* | ‘ (**caracter**)$^{+}$ ‘

constante := (**+** | **-**)? *número sem sinal* | (**+** | **-**)? *identif. de constan.* | ‘ (**caracter**)$^{+}$ ‘

fator := *constante sem sinal* | *identif. de variável* | *identif. de função* (**(** *expressão* (**,** *expressão*)* **)**)? | **(** *expressão* **)** | ¬ *fator*

termo := *fator* ((***** | / | **mod** | ^) *fator*)*

expressão simples := (+ | -)? *termo* ((+ | - | ∨) *termo*)*

expressão := *expressão simples* ((= | ≠ | < | ≤ | > | ≥) *expressão simples*)?

lista de parâmetros := (**(** (**var**)? *identif.* (**,** *identif*)* **:** *identif. de tipo* (**;** (**var**)? *identif.* (**,** *identif*)* **:** *identif. de tipo*)* **)** | ε)

comando := (*inteiro sem sinal* **:**)? ((*identif. de variável* | *identif. de função*) := *expressão* | *identif. de rotina* (**(** *expressão* (**,** *expressão*)* **)**)? | **begin** *comando* (**;** *comando*)* **end** | **if** *expressão* **then** *comando* (**else** *comando*)? | **case** *expressão* **of** *constante* (**,** *constante*)* **:** *comando* (**;** *constante* (**,** *constante*)* **:** *comando*)* **end** | **while** *expressão* **do** *comando* | **repeat** *comando* (**;** *comando*)* **until** *expressão* | **for** *identif. de variável* **:=** *expressão* **to** *expressão* **do** *comando* | **goto** *inteiro sem sinal* | ε)

bloco := (*inteiro sem sinal* (**,** *inteiro sem sinal*)* **;**)? (**var** *identif.* (**,** *identif.*)* : *tipo* ; (*identif.* (**,** *identif.*)* **:** *tipo* **;**)*)? (**procedure** *identif. lista de parêmet.* **;** *bloco* **;**)* (**function** *identif. lista de parêmet* **:** *identif.* **;** *bloco* **;**)* **begin** *comando* (**;** *comando*)* **end**

programa := *bloco* **.**

8- Referências Bibliográficas:

AHO, A. V. **Compiladores – Princípios, Técnicas e Ferramentas.** Editora Guanabara Koogan, 1995.

Adam's Assembler Tutorial 1.0 disponível em **http://www.faroc.com.au/~blackcat.**