UNESP – Universidade Estadual Paulista

Relatório de um Compilador Pascal Simplificado

Alunos: Caio Nunes
Willian Makarenko
Disciplina: Compiladores
Docente: Dr. Eraldo Pereira Marinho

Sumário

1 . Introdução

Este documento tem como objetivo explicar o funcionamento de um Compilador, desenvolvido em linguagem C, que reconhece instruções válidas na linguagem de entrada definida, Pascal simplificado, e produz o código em linguagem de máquina Assembly correspondente, em um arquivo objeto.

Vale ressaltar as seguintes observações quanto ao Compilador implementado e as simplificações na linguagem Pascal adotadas:

1) São tratadas apenas variáveis globais, declaradas no início do programa;
2) Não há procedimentos dentro de procedimentos, ou seja, existem apenas 2 níveis de escopo;
3) As variáveis podem ser de apenas dois tipo: Integer e/ou Boolean;
4) Não é permitida a declaração de tipos estruturados e variáveis dinâmicas, arrays, records e ponteiros;
5) Foi omitido o comando *mod*.

2 . Análise Léxica

A análise léxica é a primeira das três fases que compõem a análise do programa-fonte. O analisador léxico atua como uma interface entre o código-fonte e o analisador sintático, convertendo a seqüência de caracteres que constituem o programa em uma seqüência de átomos, símbolos terminais da gramática, que serão utilizados pelo analisador sintático. Através da varredura do programa da esquerda para direita, o analisador léxico agrupa os símbolos de cada item léxico determinando sua classe, elimina delimitadores e comentários e identifica palavras reservadas. Há a interação com a tabela de símbolos, uma vez que as palavras ou lexemas são guardados na tabela e classificados de acordo com a linguagem, em palavras reservadas, comandos, variáveis e tipos básicos.

Basicamente, a análise léxica é feita através dos procedimentos "*gettoken*" e "*match*", desenvolvidos em linguagem C e contidos no módulo "*lexerob.c*".

O procedimento "*gettoken*" lê um lexema do arquivo de entrada (programa fonte) determinado pelo usuário, sendo que se este lexema for do tipo alpha, ele verifica na tabela de símbolos a sua existência (através do procedimento "*lookup*"). Caso o símbolo se localize no escopo global e esteja sendo feita declaração de variáveis, e o lexema não for encontrado por "*lookup*", é feita a inserção na tabela com o token de identificador (ID). Se não estiver sendo feita a declaração de variáveis e o lexema for localizado, o token do lexema é enviado para o analisador sintático, sendo ainda, que se o símbolo não for localizado na tabela de símbolos, é emitida uma mensagem de erro indicando que o símbolo não foi previamente declarado. No caso do símbolo lido por "*gettoken*" ser um espaço em branco, tabulação ou quebra de linha, o mesmo é ignorado e a varredura

continua normalmente. Caso o caractere lido seja um dígito, é enviado ao analisador sintático o token CT, e caso seja um operador ou pontuação, é enviado o código ASCII do mesmo, ou o token associado a ele.

O procedimento "*match*" verifica se o token passado como parâmetro equivale ao símbolo lido, que está contido na variável global *lookahead*. Caso sejam iguais, um novo lexema é lido através do procedimento "*gettoken*", e caso contrário, é emitida uma mensagem de erro, e uma vez que o token não é o esperado, a compilação é interrompida.

3 . Análise Sintática

Cada linguagem possui suas regras que definem a estrutura sintática de programas *bem-formados*. O analisador sintático processa a seqüência de átomos extraídos do programa-fonte pelo analisador léxico, efetuando uma verificação quanto à ordem em que eles se apresentam, de acordo com a gramática na qual se baseia o reconhecedor. Basicamente, o analisador sintático cuida da forma das sentenças da linguagem, procurando levantar suas estruturas com base na gramática.

Foi utilizada a técnica *top-down preditiva recursiva* para construção deste analisador sintático. O termo *top-down* vem do fato de que a construção da árvore sintática é feita da raiz para as folhas, e a análise é *preditiva* porque é feita a previsão do próximo token a ser lido, a fim de que assim seja escolhida a regra de produção utilizada para a geração da sentença. Resumindo, parte-se do símbolo inicial da gramática prevendo qual será o próximo símbolo a ser lido, escolhendo-se então as regras de produção corretas.

A análise é sempre feita da esquerda para direita e devemos utilizar uma gramática que não possua recursão à esquerda, ou pelo menos usar técnicas que eliminem este tipo de recursão sem alterar a regra de formação. Este cuidado deve ser tomado pois além do fato de produções com recursão esquerda serem indesejáveis em análises *top-down*, estamos lidando com um analisador preditivo, onde a fatoração a esquerda é uma transformação gramatical utilizada por ser interessante deixar as decisões para mais tarde (quando um número suficiente de símbolos de entrada forem processados), a fim de que a produção seja escolhida corretamente.

Por uma regra geral de eliminação de recursão esquerda, produções do tipo:

$A \rightarrow A\alpha|\beta$

Ficam assim:

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A'|\varepsilon$

A gramática utilizada para descrever este analisador sintático está definida a seguir:

programa::= **PROGRAM ID ;** *declara_variável* **;** *subprogramas***;** *bloco* **.**

*bloc*o::= **BEGIN** *comando* **;** {*comando* **;**}* **END**

*declara_variave*l::= **VAR ID** {**,ID**}* **:** (**INTEGER** | **BOOLEAN**) **;**

subprogramas::= **PROCEDURE ID** () *; bloco ;* | **FUNCTION ID** ()**:** (**INTEGER** | **BOOLEAN**)**;** *bloco* **;** | *subprogramas*

comando: bloco | **ID ( )** | **ID ATRIB** *expressão_simples*
| *ifthen ( )*
| *whiledo ( )*
| *repeatuntil ( )*
| *declara_for ( )*
| *declara_read ( )*
| *declara_write ( )*
| *declara_case ( )*

ifthen::= ∋ | **IF** *expressão* **THEN** *comando {***ELSE** *comando}*

whiledo::= ∋ | **WHILE** *expressão* **DO** *comando*

repeatuntil::= ∋ | **REPEAT** *(comando* **;**)*** UNTIL** *expressão*

declara_for::= ∋ | **FOR ID ATRIB CT TO CT** *comando*

declara_read::= ∋ | **READ (ID)**

declara_write::= ∋ | **WRITE ("ID")**

declara_case::= ∋ | **CASE ID OF CT** : *comando* {**CT** : *comando* **;**} **END**

lista-expressão::= *expressão {, expressão}**

*expressã*o::= *expressão_simples* { **= | > | < | >= | <= | <>** } *expressão_simples*

expressão_simples::= **+** *termo* | **-** *termo* | *termo* { **+ | - | OR | AND** } *termo*

termo::= fator | ***** { **ID | CT** *lista_expressão* } | **/** { **ID | CT** *lista_expressão* } | **DIV { ID | CT** *lista_expressão* **}**

fator::= **ID (***lista_expressão***) | CT | (***expressão***)**

Símbolos não terminais foram transformados em procedimentos contidos no módulo "*parserob.c*", e explicados a seguir:

Procedimento *programa*: após reconhecer o token PROGRAM e o identificador do programa, realiza a chamada do procedimento *declara_variável*, seguido dos procedimentos *subprogramas* e *bloco*, finalizando pelo token ".".

Procedimento *declara_variável*: reconhece o token VAR e em seguida todos os identificadores, determinando ainda seu tipo, que pode ser inteiro ou booleano.

Procedimento *subprogramas*: este procedimento é responsável por determinar tanto os procedimentos quanto as funções, podendo inclusive chamar a si mesmo. Caso seja um procedimento, reconhece-se o token PREOCEDURE e seu identificador, seguido do procedimento *bloco*. Caso seja uma função, reconhece-se o token FUNCTION, seu identificador e o tipo da função, que pode ser inteiro ou booleano, seguido do procedimento *bloco*.

Procedimento *bloco*: reconhece o token BEGIN seguido dos procedimentos *comando* relativos a todos os comandos do bloco em questão, finalizando pelo reconhecimento do token END.

Procedimento *comando*: este procedimento pode realizar a chamada do procedimento *bloco*, ou a chamada de um identificador de procedimento/função, ou ainda a chamada de um identificador seguido de atribuição e o procedimento *expressão_simples*. O procedimento comando pode também realizar a chamada dos procedimentos de estrutura de comandos, *ifthen, whiledo, repeatuntil, declara_for, declara_read, declara_write, declara_case.*

Procedimento *ifthen*: caso lookahead seja igual a IF, reconhece o token IF, realiza a chamada do procedimento *expressão*, reconhece o token THEN e realiza a chamada do procedimento *comando*. Em seguida pode, caso necessário, reconhecer o token ELSE e realizar a chamada do procedimento *comando*.

Procedimento *whiledo*: caso lookahead seja igual a WHILE, reconhece o token WHILE e realiza a chamada do procedimento *expressão*, reconhece o token DO e por fim realiza a chamada do procedimento *comando*.

Procedimento *repeatuntil*: caso lookahead seja igual a REPEAT, reconhece o token REPEAT, faz a chamada do procedimento *comando* até que todos os comandos sejam executados, reconhece então o token UNTIL e realiza a chamada do procedimento *expressão*.

Procedimento *declara_for*: caso lookahead seja igual a FOR, reconhece os tokens FOR, ID, ATRIB, CT, TO, CT e realiza a chamada do procedimento *comando*, que será executado tantas vezes quanto foi definido pela diferença entra as constantes.

Procedimento *declara_read*: caso lookahead seja igual a READ, ele reconhece o token READ e em seguida reconhece o identificador.

Procdimento *declara_write*: caso lookahead seja igual a WRITE, ele reconhece o token WRITE e em seguida o identificador, colocado entre aspas.

Procedimento *declara_case*: caso lookahead seja igual a CASE, são reconhecidos os tokens CASE, ID, OF, sendo que em seguida são reconhecidos os tokens CT e chamados os

procedimentos *comando* quantas vezes quanto forem necessárias, finalizando com o reconhecimento do token END.

Procedimento *lista-expressão*: realiza a chamada do procedimento *expressão* uma ou mais vezes.

Procedimento *expressão: após fazer a chamada do procedimento expressão_simples*, pode ou não reconhecer os tokens relativos aos sinais dos operadores relacionais seguidos da chamada do procedimento *expressão_simples*.

Procedimento *expressão_simples*: este procedimento pode reconhecer o token "+" ou "-" seguido da chamada do procedimento *termo*, ou simplesmente realizar apenas a chamada do procedimento *termo.* Pode ainda, reconhecer algum do tokens "+", "-", OR ou AND seguido da chamada ao procedimento *termo*.

Procedimento *termo*: pode realizar a chamada do procedimento *fator*, ou então reconhece um dos tokens "*", "/" ou DIV, seguido do reconhecimento do token ID ou do reconhecimento do token CT seguido da chamada do procedimento *lista_expressão*.

Procedimento *fator*: pode reconhecer o token ID seguido da chamada do procedimento *lista_expressão*, reconhecer o token CT ou ainda, realizar a chamada do procedimento *expressão*.

4 . Análise Semântica

O principal objetivo do analisador semântico é captar o significado das ações a serem tomadas no texto-fonte. Desta maneira, a principal função da análise semântica é criar uma interpretação do código-fonte. Através da técnica de *tradução dirigida por sintaxe*, utilizada por este compilador, ações semânticas foram adicionadas à gramática da linguagem A tradução dirigida por sintaxe é uma técnica que permite a associação de atributos/regras semânticas às produções gramaticais, as quais serão avaliadas quando tais produções forem utilizadas no reconhecimento de uma sentença. Vale ressaltar que é também durante a fase de análise semântica que se verificam os erros semânticos no programa-fonte e são capturadas as informações de tipo para geração de código. O analisador semântico utiliza a estrutura hierárquica determinada pela fase de análise sintática, a fim de identificar os operadores e operandos das expressões e enunciados.

É realizada a tradução de um arquivo de entrada definido pelo usuário, da linguagem Pascal simplificado para linguagem de máquina Assembly, sendo que as traduções são encontradas no arquivo de saída de extensão “.s”.

5 . Tabela de Símbolos

Uma tabela de símbolos é uma estrutura de dados contendo um registro para cada identificador, incluindo os campos sobre seus atributos. Uma das funções essenciais do compilador é registrar os identificadores usados no programa fonte.

Foi utilizada uma fita (array) onde são armazenados todos os lexemas. A tabela de símbolos contém um campo ponteiro que aponta para o endereço do lexema no array, um campo inteiro onde é armazenado o token do lexema, outro campo inteiro que determina o tipo do token (que pode ser variável ou função) como integer ou boolean e um último campo inteiro que verifica se uma variável foi ou não declarada em *var*. A estrutura da tabela de símbolos assim como as funções para manipulação da mesma estão contidas no módulo "*simbolo.c*". A seguir serão explicadas as funções de manipulação da tabela:

- Insert: insere um lexema e seu token correspondente, ambos passados como parâmetro, na tabela de símbolos. É feita uma busca pela primeira posição vazia da tabela a fim de se realizar a operação, sendo que caso a tabela ou o array de lexemas estejam cheios, a inserção não pode ser realizada. É retornado o token inserido.
- Lookup: realiza a busca de um lexema, passado como parâmetro, na tabela de símbolos. O valor retornado é o valor relativo à posição do lexema encontrado, e caso ele não tenha sido encontrado, é retornado o valor 0.
- Verifica: verifica se uma variável foi declarada ou não em *var*. Caso ela não tenha sido declarada, é emitida uma mensagem de aviso.

- Declara: este procedimento atribui o valor 1 ao campo *dec* da tabela de símbolos, do último identificador reconhecido(a posição deste está na variável *lastentry*) sinalizando que o identificador foi declarado.

6 . Principais Variáveis

Lastentry – variável inteira que contém a posição relativa a última entrada na tabela de símbolos.

Nextentry – variável inteira que contém a posição da próxima entrada na tabela se símbolos.

Lexentry – variável inteira que contém a posição onde o lexema lido será inserido na fita.

Symboltable – variável do tipo struct que contém a estrutura da tabela de símbolos.

Lookahead – variável que armazena o token do último lexema.

Lexema – variável do tipo char[33] onde são armazenados os lexemas.

Rótulo – variável inteira responsável pelo controle dos rótulos.

7 . Funcionamento Básico

Esse compilador foi dividido em 5 módulos: *mainob.c*, *lexerob.c*, *parserob.c*, *símbolo.c* e *global.h*, explicados a seguir:

Mainob.c: módulo que inicia a execução do processo de compilação. É feita a inicialização da tabela de símbolos, onde são colocadas as palavras reservadas do Pascal, tokens estes, definidos em *global.h*. É então reconhecido o arquivo de entrada de extensão ".pas" para que seja gerado o arquivo-objeto de saída, de extensão ".s". Este módulo também é responsável pela inicialização do analisador léxico (*lexerob.c*) através da instrução `lookahead=gettoken()`, e dá início ao analisador sintático (*parserob.c*) através da chamada do procedimento *programa()*. Terminada a compilação, realiza o fechamento dos arquivos.

Lexerob.c: como já definido anteriormente, este módulo, conhecido como analisador léxico, e´ composto do procedimentos *gettoken()* e *match()*, e é responsável pela análise léxica do programa de entrada. Ele utiliza o módulo "símbolo.c", onde estão armazenadas as palavras reservadas do Pascal.

Parserob.c: também já definido anteriormente, este módulo, conhecido como analisador sintático, é inicializado pelo procedimento *programa* e realiza a análise sintática em cima dos átomos enviados pelo analisador léxico. Vale observar que a análise semântica está aninhada as regras de produção, logo, é realizada pelo parser. Contém os procedimentos *programa(), bloco(), variaveis(), subprogramas(), declara_variavel(), comando(), ifthen(), thenelse(), whiledo(), declara_for(), repeatuntil(), expressao(), expressao_simples(), lista_expressao(), termo(), fator(), declara_read() e declara_write().*

Símbolo.c: módulo que contém os procedimentos de manipulação da tabela de símbolos, são eles *insert(), lookup(), declara()* e *verifica()*. Realiza consulta no módulo "*global.h*", onde estão os tokens reservados do Pascal.

Global.h: módulo que contém as palavras reservadas do Pascal e realiza a inicialização dos procedimentos dos demais módulos.

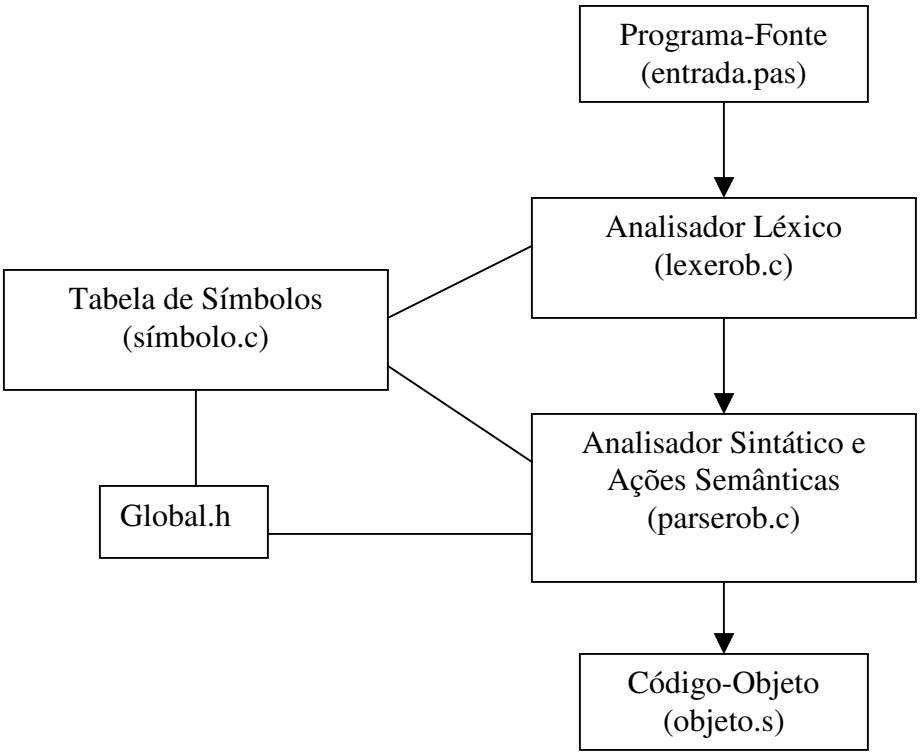

Diagrama de funcionamento do compilador

8 . Referências

Aho, A.V.; Sethi, R.; Ullman, J.D. Compiladores – Princípios, Técnicas e Ferramentas. Ed. JC, 1995

Crenshaw, J. Let´s Build a Compiler! http://compilers.iecc.com/crenshaw/, 1995

Soranz, F. Vamos Construir um Compilador! http://geocities.yahoo.com.br/feliposz/, 2002