Faculdade de ciências e Tecnologia da Universidade de Coimbra

Departamento de Engenharia Informática

Compilador de Linguagem KPascal

João Alexandre Rodrigues, nº 2006125096, jarod@student.dei.uc.pt

Diogo Machado, nº 2006124876, dmachado@student.dei.uc.pt

Índice

Introdução

Neste trabalho pretendeu-se desenvolver um compilador para a linguagem Kpascal, que consiste num pequeno conjunto de funcionalidades da linguagem Pascal. O objectivo deste, era aplicar os conhecimentos obtidos nas aulas teóricas na prática. Para isso, ao longo do semestre, foram sendo disponibilizadas algumas fichas de orientação para o projecto, correspondendo cada conjunto de destas a uma meta do compilador.

No entanto, apesar de o nosso grupo ter-se servido das fichas como orientação, optámos por seguir uma abordagem ligeiramente diferente da aconselhada. Como estamos poucos treinados a programar em C, decidimos "aventurar-nos" e implementar um compilador de Kpascal numa linguagem em que estamos muito mais à vontade: Python.

Claro está, que antes de tomarmos esta decisão, tivemos que nos certificar que existiam bibliotecas em Python equivalentes ao Lex e Yacc usados em C. Assim, depois de descobrirmos o *ply* (http://pypi.python.org/pypi/ply/3.1) tomámos a decisão de seguir em frente com a abordagem, pois para as seguintes metas (Análise Semântica e Geração de código) apenas nos bastaria percorrer a árvore AST gerada.

Procurarmos seguir todas as metas propostas, levando assim o projecto mais ao menos em dia. Trabalhamos sempre em grupo até termos feito a meta 1. A partir daí, devido à elevada subrecarga que somos sujeitos ao longo do semestre, decidimos repartir tarefas. Assim um ficou responsável pela análise semântica (Diogo) e o outro pela geração de código (João), estando os dois no entanto sempre em sintonia.

Em relação às *features*, cumprimos tudo o que o enunciado nos pediu exceptuando os extras, ou seja:

- tipos de dados (integer, real, boolean, char);
- declarações de variáveis;
- declaração de procedimentos e funções (procedure, function);
- expressões envolvendo operações aritméticas (+, -, /, *, div, mod), relacionais (=, <, >, <=, >=, <>), lógicas (and, or, not);
- instrução de atribuição (:=) e de controlo (de selecção e repetição: if, for, while, repeat) e de output (write)

O nosso código final está como o enunciado pediu em C reduzido, não violando nenhuma das regras impostas, e com a visualização do resultado final através de gcc (ficheiro src/gen/c_code/output.c).

Análise Lexical

Para começar com este projecto de desenvolvimento de um compilador, é preciso um analisador lexical. O módulo *lex* da biblioteca *ply* permite criar um extremamente poderoso.
Para a utilização do *lex*, é preciso ter num ficheiro à parte (parser/lexFile.py) as expressões regulares correspondentes aos padrões especificados, tal como se pode ver na *Tabela 1* e *Tabela 2*.

EQUALS	r'='	*NOT_EQUAL*	r'<>'
ADD_OP	r'\+'	*DECLARATOR*	r':='
SUB_OP	r'-'	*SEMICOLON*	r';'
MUL_OP	r'*'	*COMMA*	r','
EXP	r'**'	*COLON*	r':'
DIV_OP	r'/'	*COMMENT*	r' (\{ [^\}]* \}) | (\(* [^(*\))]* *\))'
LEFT_PAREN	r'\(	*STRING*	r" (\' [^\']* \') | (\" [^\"]* \") "
RIGHT_PAREN	r'\)'	*DOT*	r'\.'
GREATER	r'>'	*IDENTIFIER*	r'[a-zA-Z_][\w_]*'
LESS	r'<'	*CHAR*	r'(\'[\w]\') | (\"[\w]\")'
GREATER_OR_EQUAL	r'>='	*REAL*	r'[0-9]+\.[0-9]+'
LESS_OR_EQUAL	r'<='	*INTEGER*	r'[0-9]+'

Tabela 1 – Expressões regulares de Tokens

AND	TRUE	FUNCTION
OR	FALSE	PROCEDURE
NOT	PROGRAM	IN
IF	BEGIN	NILVAR
FOR	END	DO
WHILE	THEN	TO
REPEAT	TYPE	DOWNTO
MOD	CONST	UNTIL
DIV	ELSE	

Tabela 2 – Palavras Reservadas

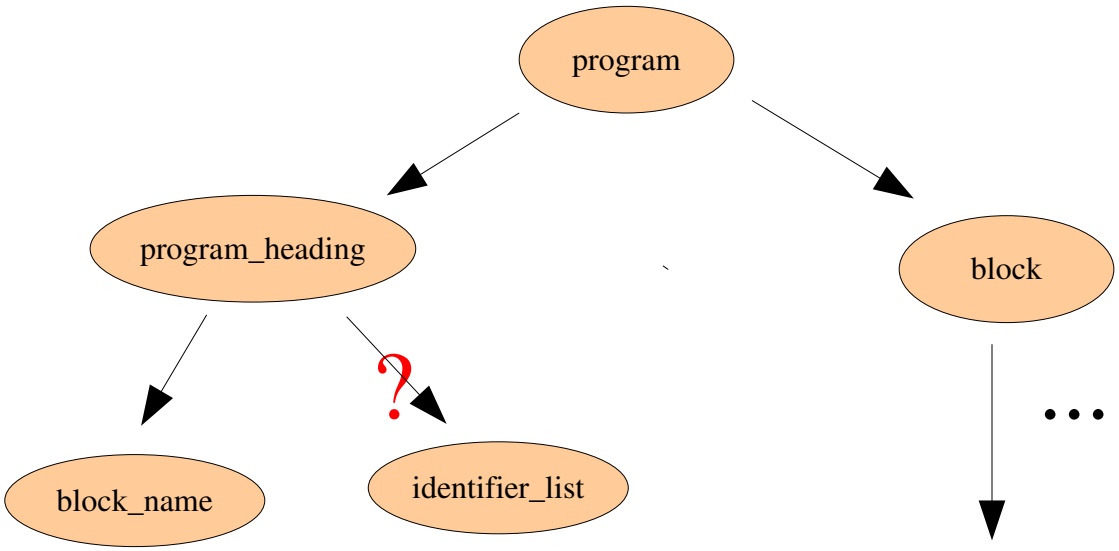
Daqui o *lex* lê o input ditando um analisador que mapeia as expressões regulares em blocos, gerando então código C implementando o analisador, que será utilizado em etapas seguintes. Ele será por exemplo, peça essencial e complementar ao uso do *yacc*, na análise sintáctica.

Análise Sintáctica e geração da AST

O ficheiro parser/yaccFile.py é o responsável por fazer a analise sintáctica e construir a árvore de sintaxe abstracta respectiva. Para isso é usado o modulo *yacc* da biblioteca *ply*, referida na introdução deste projecto. Este módulo é semelhante ao *yacc* usado em C, diferindo apenas um pouco na sintaxe e facilitando em muito a construção da árvore.

Aqui, cada função representa um nó da AST. Comecemos pelo topo da árvore, ou seja, a função *p_program*. Esta função diz que qualquer programa em pascal vai ter de ser constituído pelo seu cabeçalho, seguido de um ponto e virgula, do bloco do programa e de um ponto final. Se algum programa violar esta regra, este vai ser abortado devolvendo erro de sintaxe na linha respectiva. Caso contrário então adicionamos à nossa árvore um nó do tipo "program" e uma lista dos seus filhos cabeçalho do programa e bloco. De seguida aplica-se o mesmo esquema a cada um dos seus filhos e assim sucessivamente até termos o programa todo representado em árvore.

Existem casos no entanto, como o *program_heading*, em que podem haver várias hipóteses de sintaxe (cabeçalho do programa pode ter argumentos ou não). Nestes casos, usamos o comprimento da hipótese escolhida para a conseguir diferenciar da(s) outra(s) e assim saber que filhos adicionar ao novo nó.



A estrutura da árvore encontra-se no ficheiro lib/ast.py e, como se pode observar, é bastante simples: cada nó apenas vai ter um tipo e uma lista de filhos, que pode ou não estar vazia. Na mesma pasta temos o ficheiro get_program.py, que é responsável por transformar o programa de input numa só string, inserindo *\n's* entre as diferentes linhas. Assim, por cada vez que é encontrado um *\n*, uma variável é incrementada, de maneira a que quando o yacc detecte um erro a linha correspondente seja imprimida (função p_error).

Por fim, referir apenas o facto do *yacc* apenas aceitar *tokens* ou palavras reservadas definidas no *lex*. Deste modo, a primeira coisa que faz é o *import* desses mesmos *tokens*, devolvendo erro sempre que encontra um que não está definido.

Análise Semântica

A terceira fase para o desenvolvimento de um compilador é a Análise Semântica. Nesta fase pretende-se efectuar a validação quanto à lógica e significado do programa. Questões como a compatibilidade de tipos em expressões, chamada correcta de uma função e outras, são aqui tidas em consideração e analisadas.

O estudo semântico é feito recorrendo à árvore AST criada na fase anterior. Percorrendo-a, é feita a sua análise de acordo com os nossos parâmetros de validação para esta etapa.

É de explicar o conceito de stack e frame usadas na implementação. Cada programa apresenta uma estrutura, podendo esta ser dividida por blocos ou frames. Este têm o seu próprio ambiente e as suas próprias variáveis locais. Por esta razão, a divisão por frames é importante na detecção de erros semânticos quanto ao alcance das variáveis (locais, globais).

Imagem 1 – Stack e Frames

O algoritmo usado faz uso de uma pilha (stack) de frames. Encontrado novo ambiente ao percorrer a AST é adicionada um novo elemento à pilha. Este é retirado quando terminada o seu estudo. É de reparar que só existirá nesta stack uma função ou procedimento (ambientes) de cada vez, o que simplifica certas verificações (p/ex: variáveis locais a uma função não podem ser usadas fora do ambiente em que estão inseridas). No entanto nunca se perde registo de informação das funções ou procedimentos existentes, para o bem de uma completa e correcta análise semântica.

Erros Semânticos:

Quanto aos tipos de erros semânticos considerados, a bem de uma melhor leitura, podemos dividi-los por três grandes grupos: Erros de Variáveis, Erros de Tipos, Erros de Função e Procedimentos.

- Erro na Declaração de Variáveis

A declaração de variáveis, é susceptível a erros não detectados no teste sintáctico. É disso exemplo, a transcrição abaixo

VAR a,b,c : (Integer, Real);

Por esta razão, a análise Semântica tem aqui um especial cuidado.

A partir da árvore AST, podemos distinguir sub-árvores separando nós chave. Neste caso em particular, as sub-árvores destinadas à declaração de variáveis (*var_subtree*), é identificada quando alcançado um nó de tipo '*variable_declaration_part*'. Daqui são recolhidos os dados relativos ao nome das variáveis (*IDENTIFIERS*) e respectivo tipo associado ('*type_denoter'*).

São aceites as seguintes declarações, sendo as variáveis guardadas na frame de topo, com o nome, tipo e valor nulo:

```
VAR   A     :     intEger;
      b,c   :     (reaL, bOOlean)
      D,e   :     ChaR
```

Se violada estas regras, é então levantada a dita excepção:

"VARIABLE_DECLARATION_ERROR: given %d types to %d variables [%s]" %(t,v,f)

- Variável Já Definida

Como já falado, o alcance de uma variável é uma matéria sensível nesta etapa. Mesmo antes de validar a declaração da variável, é verificado se uma outra variável local de mesmo nome já não fora anteriormente declarado.

Se afirmativo, é registado o erro de semântica:

"VARIABLE_ALREADY_DEFINED: %s [%s]" %(var,f)

Caso contrário, poderá se proceder a outras averiguações. De notar, que a procura é local ao ambiente em que a nova variável se encontra inserida. Isto permite que frames possam ter variáveis com o mesmo nome que variáveis globais, mas dando preferência de uso às primeiras.

-Variável Não-Definida

Uma vez mais, é preciso exercer o poder de procura e de verificação do poder de alcance das variáveis. Aquando do uso de uma variável, quer seja numa expressão ou na passagem por parâmetro, é preciso reconhecer primeiro a sua existência.

Para esta análise, a procura não é focada na frame actual. A procura é feita na stack, a começar na frame de topo até à primeira.

Se não encontrada, o regime de excepção é verificado:

"VARIABLE_NOT_DEFINED: %s [%s]" %(var,f)

Se pelo contrário, é reconhecida a sua declaração prévia, então segue-se a validação seguinte.

- Variável Não-Inicializada

Sabendo que a variável em causa existe, cabe agora saber se já fora inicializada, isto é, com valor definido.
Se assim não for, não se poderá fazer-lhe uso, sendo guardada o erro:

"VARIABLE_NOT_ASSIGNED: %s (%s) has no value [%s]" %(v,t,f)

- Tipo Desconhecido Não-Suportado

Quer na declaração de variáveis (com o *type_denoter*) quer na declaração de uma função (com o *returning_type*), o tipo declarado tem que ser reconhecido.

Sujeito às limitações do compilador, são aceites os seguintes tipos, de acordo com o parâmetro *unsigned_constant* do YACC:

| INTEGER
| REAL
| CHAR
| boolean (TRUE | FALSE)

Se o tipo não se encontra entre os mencionados, então obtém-se o erro:

"TYPE_UNKNOW: %s [%s]" %(t,f)

- Incompatibilidade de Tipos

A detecção de erros quanto à incompatibilidade de tipos, é importante para mais tarde se obter uma geração de código correcta.
Este erro ramifica-se em dois problemas: incompatibilidade em expressões e incompatibilidade para com os parâmetro definidos de uma função ou procedimento.

Em relação ao primeiro, são precisas procuras e verificações, quanto à existência e inicialização, de todas as variáveis em causa.
Se encontrado um tipo de variável diferente ao tipo da variável em *'assignment*', é registada a excepção:

"DIFFERENT_TYPES_IN_ASSIGNMENT: %s and %s [%s]" %(a,b,f)

Em relação ao segundo, na chamada de uma função ('*procedure_statement*' ou '*function_designator*'), são registados os parâmetro a passar (em *params_subtree*) e comparados com a informação dos parâmetros esperados, anteriormente guardada.
Se falhada a comparação:

"ARGUMENT_TYPE_INCOMPATIBILITY: Type of argument #%d must be %s [%s]" %(arg,type,f)

Neste mesmo método é também usada para a excepção seguinte

- Número Errado de Argumentos na Função ou Procedimento

Fazendo uso do mesmo processo imediatamente acima descrito, é tão simplesmente comparado o número de argumentos com o número de parâmetros.

Se negativo, o erro é registado:

"WRONG_NUMBER_OF_ARGUMENTS: %s takes exactly %d arguments (%d given)" %(f,a,b)

- Função ou Procedimento não definido

Aproveitando a utilidade do nó '*function_designator*' ou do '*procedure_statement'*, comprovada na detecção dos dois erros anteriores, também aqui a usaremos.

Se chamada uma função ou procedimento, é em primeiro lugar preciso verificar a sua declaração. Este processo é idêntico ao mencionado no 'Variável não definida'. No entanto a sua procura não é feita na stack.
Como dito ao início desta secção do relatório, as funções são retiradas da stack mal acabe a sua análise. Mas, como também já mencionado, a sua informação não é desperdiçada. É guardado num registo à parte, todas as funções ou procedimentos e respectivas informações (quanto aos parâmetros e tipo de retorno), sendo então aqui, efectuada a procura

Se a procura não for bem sucedida, ter-se-á o seguinte:

"FUNCTION_OR_PROCEDURE_NOT_DEFINED: %s [%s]" %(id,f)

- Função ou Procedimento já definido

Na declaração de funções ou procedimentos, tal como na de variáveis, é preciso também verificar se tal nome não fora já reservado.

Efectuando a procura como mencionado na descrição do erro anterior, poder-se-a obter a seguinte excepção:

"FUNCTION_OR_PROCEDURE_ALREADY_DEFINED: %s [%s]" %(id,f)

- Repetição de Parâmetro na Função ou Procedimento

O tratamento de parâmetros é feita como se de simples variáveis locais se tratassem. Assim sendo, é aqui também verificada a unicidade do nome de entre todo os outros parâmetro.

"NAME_REPEATED_IN_PARAMS: %s [%s]" %(n,f)

Geração de Código C

É no ficheiro gen/generate.py que se encontra a função *generate*, a função principal da geração de código C. Se a análise semântica for efectuada com sucesso, então esta função vai ser chamada, sendo passado por parâmetro o primeiro nó da AST.

Antes de passarmos à especificação do código, convém explicar de uma forma geral quais as estruturas de dados utilizadas, que incidem basicamente em seis dicionários e uma classe:

dic_typ (dic) – faz a correspondência entre os tipos de variáveis de Pascal com os de C;
dic_type_c (dic) – para cada tipo de variável em C, tem a sua representação em percentagem usado por exemplo na função *printf;*
dic_trans (dic) – faz a correspondência entre as operações aritméticas, relacionais e lógicas de Pascal com as de C;
global_vars (dic) – faz o mapeamento entre as variáveis globais definidas no programa em Pascal, com as que vão ser utilizadas no programa em C (string "g" + offset);
var_type (dic) – possui o tipo em Pascal de uma determinada variável global utilizada no programa em C. Por exemplo, se determinada variável global *y* em Pascal é um inteiro e é traduzida para C como *g0*, este dicionário tem a correspondência {g0 : integer};
Frame (classe) – esta classe contém estruturas semelhantes ao global_vars e ao var_type, mas para variáveis locais a uma determinada frame (proc/function). Além disso, tem um contador próprio de variáveis (offset local, que é incrementado sempre que é definida uma nova variável) e em alguns casos o tipo de valor que retorna. O nome das variáveis também vai ser diferente, já que no caso das frames é utilizada a lista *locals* definida na estrutura "frame.h" sugerida nas aulas para as guardar.
frames (dic) – este dicionário apenas faz a correspondência entre de terminada frame e a sua classes com os respectivos dados.

Conhecendo as estruturas definidas, mais facilmente entenderemos o resto da estrutura do programa. Basicamente, a árvore vai ser percorrida em profundidade primeiro, sendo cada nó analisado de cada vez, e o respectivo código C gerado ao mesmo tempo.

Caso o nó seja uma folha, só temos interesse em retornar o seu valor. Apenas se for *false* ou *true* queremos que ele retorne a *string* em maiúsculas, devido aos *define's* presentes no código C que vão representar o tipo *boolean*, no fundo não passando de representações em inteiros (0 ou 1).

Quando apanhamos o nó pai (*program*), escrevemos para o ficheiro o nosso *header* e *footer*, processando o resto dos nós no meio deles e fechando o ficheiro no fim. Para sabermos qual é o bloco de código (frame) em que nos encontramos, cada vez que detectamos o nó *block_name* actualizamos o nome do bloco corrente, permitindo assim aceder ao seu dicionário *frames*.

Quando chegamos à parte de declarar funções ou procedimentos, sabemos de antemão que no final desta estamos no bloco *main*, daí actualizarmos o nome. Depois, consoante seja um ou outro vamos sempre primeiro gerar o seu nome e o seu valor de retorno (se tiver), criando seguidamente o objecto respectivo e guardando no dicionário usando como chave o seu nome. Seguidamente, escrevemos o *header* e o *footer*, gerando no meio o parâmetros e o corpo da função/procedimento.

Para as instruções de controlo e ciclos (*if,while,for e repeat*), apenas geramos a instrução de condição e o corpo, escrevendo o respectivo código C usando apenas a instrução *goto<label>*.

Por cada vez que encontramos uma operação aritmética, relacional ou lógica vamos ao dicionário *dic_trans* verificar se existe correspondência em C e, se houver, substituímos. O mesmo se aplica às variáveis quando encontramos um nó do tipo *primary*. No entanto, neste segundo caso é um pouco mais complicado. Temos de verificar primeiro se a variável tem um nome de uma função, pois se tiver apenas queremos retornar o seu nome. Em caso negativo, vamos ver em que bloco nos encontramos. Se nos encontrarmos no *MAIN*, então a variável é global e usamos o dicionário *global_vars* para devolvermos a sua correspondência no nosso código. Se não estivermos no bloco *MAIN*, então é porque estamos numa função ou procedimento, por isso vamos usar o dicionário *frames* para aceder ao dicionário *global_vars* correspondente à frame corrente, e assim devolver o nome da variável no nosso código C. Ainda se pode dar o caso de ele não estar no bloco *MAIN* e a variável não existir na base de dados da frame. Neste caso é porque a função/procedimento está a usar uma variável global ao programa e por isso comportamos-nos como se estivéssemos no bloco principal. Este raciocínio aplica-se sempre que vamos traduzir uma variável para o nosso código C.

Existem casos, tais como os nós *identifier_list* ou *actual_parameter_list* em que nos interessa recolher os vários parâmetros ou variáveis para uma lista, daí concatenarmos os vários valores. Sempre que recebemos uma lista com variáveis e operações (*expression*), uma lista de parâmetros, etc, usamos sempre a função *get_list* que nos vai devolver essa lista em forma de parêntesis de maneira a facilitar a cópia de expressões para o nosso código C.

Quando temos um *assignment_satement*, este pode ser uma atribuição a uma variável de qualquer coisa, incluindo valores de retorno de funções. Assim, sempre que chegamos a este nó, vamos verificar que tipo de atribuição é. Se for a uma função, então vamos invoca-la. Primeiro vamos gerar os parâmetros e para cada um deles vamos alocar memoria e guarda-los na lista *outgoing* da frame corrente. De seguida, redireccionamos o código para a *label* apropriada e no final, se for mesmo uma função (pode ser procedimento), vamos guardar na variável o valor que tivermos na primeira posição da lista *return_val* da nossa frame, que entretanto foi lá colocada pela frame invocada. Caso a atribuição seja de outro tipo, apenas a fazemos normalmente usando o raciocínio descrito à pouco para verificarmos qual a variável correcta a imprimir.

Para o caso dos procedimentos é diferente, pois antes invoca-los (tal como as funções), temos de verificar primeiro se estamos na presença de um *write* ou de um *writeln*. Caso seja o caso, vamos ver se queremos imprimir uma variável ou uma frase. Se for uma frase fazemos o *printf* correspondente em C, caso seja variável temos que usar o mesmo raciocínio de sempre tal como em etapas anteriores para verificar em que frame está e qual a sua correspondência no nosso código.

Finalmente, a declaração de variáveis. É aqui, quando encontramos este nó, que atribuímos um nome às variáveis e as guardamos nos respectivos dicionários, incrementando o *offset*. Caso a variável seja global então fazemos uma declaração normal em C e colocamos-la no dicionário do bloco *MAIN*, caso esteja numa frame vamos alocar memória na lista *locals* da frame corrente para guardamos lá as nossas variáveis e guardamos os dados na classe da frame correspondente à frame.

Conclusão

No geral, os objectivos do trabalho foram cumpridos, pois conseguimos fazer tudo o que o enunciado nos pedia exceptuando os extras, que eram opcionais. Embora não saiba quando tempo demoraríamos a fazer o projecto C, penso que compensou fazer em Python.

Apesar de termos perdido bastante tempo no inicio, a pesquisar bibliotecas equivalentes ao *lex* e ao *yacc*, a descobrir como estas funcionavam e a tentar aplica-las na prática fomos recompensados mais tarde, pois como temos bastante prática nesta linguagem, tivemos com certeza muitas mais facilidades de implementação nas metas seguintes do que teríamos se tivéssemos feito em C.

Depois de testes exaustivos ao nosso programa, descobrimos apenas uma pequena falha, que diz respeito à geração de código. No nosso programa, é impossível colocar expressões dentro de chamadas de funções, como por exemplo função1(4*5,função2()). No entanto, esta pequena falha não é muito grave, já que se pode associar cada expressão a um variável. Neste caso, poderíamos resolver o problema do seguinte modo:

```
a := 4*5;
b := função2();
c := função1(a,b);
```

Tirando esta pequena falha, o nosso programa passou em todos os restantes testes, devolvendo sempre os resultados esperados e gerando sempre o código correcto sem erros.

Caso seja de interesse a consulta do código online, estatísticas, lista de *commits*, etc, esta poderá ser feita em http://github.com/joaor/pcompiler/tree/master.

Bibliografia

Fichas Práticas

http://www.moorecad.com/standardpascal/yacclex.html

http://doc.gnu-darwin.org/ply/ply.html

Anexos

Especificação da gramática da sintaxe abstracta

```
#Estrutura do programa e o seu bloco de codigo---------------------------------
def p_program(t):
        #program a;  .
        'program : program_heading SEMICOLON block DOT'
        t[0] = AST("program", [t[1],t[3]] )

def p_block(t):
        'block : variable_declaration_part procedure_and_function_declaration_part compound_statement'
        t[0] = AST("block", [t[1],t[2],t[3]] )

#auxiliar para conseguir distinguir nomes de programa, funcao e procedimentos
def p_block_name(t):
        'block_name : IDENTIFIER'

        t[0] = AST("block_name", [t[1]])

#Declaracao do heading do programa----------------------------------------------
def p_program_heading(t):
        '''program_heading : PROGRAM block_name
                                        | PROGRAM block_name LEFT_PAREN identifier_list RIGHT_PAREN'''
        if len(t)==3:
                t[0] = AST("program_heading", [t[2]] )
        else:
                t[0] = AST("program_heading", [t[2],t[4]] )


def p_identifier_list(t):
        '''identifier_list : identifier_list COMMA IDENTIFIER
                                        | IDENTIFIER'''
        if len(t)==2:
                t[0] = AST("identifier_list", [t[1]] )
        else:
                t[0] = AST("identifier_list", [t[1],t[3]] )

#Declaracao de variaveis--------------------------------------------------------
def p_variable_declaration_part(t):
        '''variable_declaration_part        : VAR variable_declaration_list SEMICOLON
                                                                                |'''
        if len(t)==4:
                t[0] = AST("variable_declaration_part", [t[2]] )

def p_variable_declaration_list(t):
        '''variable_declaration_list        : variable_declaration_list SEMICOLON variable_declaration
                                                                        | variable_declaration'''
        if len(t)==2:
                t[0] = AST("variable_declation_list", [t[1]] )
        else:
                t[0] = AST("variable_declation_list", [t[1],t[3]] )

def p_variable_declaration(t):
        'variable_declaration : identifier_list COLON type_denoter'
        t[0] = AST("variable_declaration", [t[1],t[3]] )

def p_type_denoter(t):
        '''type_denoter         : IDENTIFIER
                                        | LEFT_PAREN identifier_list RIGHT_PAREN'''
        if len(t)==2:
                t[0] = AST("type_denoter", [t[1]] )
        else:
                t[0] = AST("type_denoter", [t[2]] )

#Declaracao de procedimentos e funcoes------------------------------------
def p_procedure_and_function_declaration_part(t):
```

```
        '''procedure_and_function_declaration_part : proc_or_func_declaration_list SEMICOLON
                                                                                                |'''
        if len(t)==3:
                t[0] = AST("procedure_and_function_declaration_part", [t[1]] )

def p_proc_or_func_declaration_list(t):
        '''proc_or_func_declaration_list   : proc_or_func_declaration_list SEMICOLON proc_or_func_declaration
                                                                                        | proc_or_func_declaration'''
        if len(t)==2:
                t[0] = AST("proc_or_func_declaration_list", [t[1]] )
        else:
                t[0] = AST("proc_or_func_declaration_list", [t[1],t[3]] )

def p_proc_or_func_declaration(t):
        '''proc_or_func_declaration         : procedure_declaration
                                                                        | function_declaration'''
        t[0] = AST("proc_or_func_declaration", [t[1]] )

#Declaracao de procedimentos----------------------------------------
def p_procedure_declaration(t):
        #procedure a; ;
        '''procedure_declaration : PROCEDURE block_name SEMICOLON block
                                                                | procedure_heading SEMICOLON block'''
        if len(t)==5:
                t[0] = AST("procedure_declaration", [t[2],t[4]] )
        else:
                t[0] = AST("procedure_declaration", [t[1],t[3]] )

def p_procedure_heading(t):
        #procedure a(c,v : real); ;
        '''procedure_heading   : PROCEDURE block_name formal_parameter_list'''
        t[0] = AST("procedure_heading", [t[2],t[3]] )

#Declaracao de funcoes----------------------------------------------------------
def p_function_declaration(t):
        #funcao a;  ;-> nao devolve nada -> foi apagada
        '''function_declaration             : function_heading SEMICOLON block'''
        if len(t)==5:
                t[0] = AST("function_declaration", [t[2],t[4]] )
        else:
                t[0] = AST("function_declaration", [t[1],t[3]] )

def p_function_heading(t):
        #funcao a: real;  ; -> devolve real
        #funcao b(a,b : real ; VAR g:integer ) : real;  ;  -> devolve real, mas tem argumentos
        '''function_heading     : FUNCTION block_name COLON function_returning
                                                                | FUNCTION block_name formal_parameter_list COLON function_returning'''
        if len(t)==5:
                t[0] = AST("function_heading", [t[2],t[4]] )
        else:
                t[0] = AST("function_heading", [t[2],t[3],t[5]] )

def p_function_returning(t):
        'function_returning : IDENTIFIER'
        t[0] = AST("function_returning", [t[1]])

#Corpo das funcoes e procedimentos----------------------------------------------------
def p_formal_parameter_list(t):
        'formal_parameter_list : LEFT_PAREN formal_parameter_section_list RIGHT_PAREN'
        t[0] = AST("formal_parameter_list", [t[2]] )

def p_formal_parameter_section_list(t):
        '''formal_parameter_section_list  : formal_parameter_section_list SEMICOLON formal_parameter_section
                                                                                        | formal_parameter_section'''
        if len(t)==2:
                t[0] = AST("formal_parameter_section_list", [t[1]] )
        else:
                t[0] = AST("formal_parameter_section_list", [t[1],t[3]] )

def p_formal_parameter_section(t):
```

```
        '''formal_parameter_section         : identifier_list COLON type_denoter
                                                                        | VAR identifier_list COLON type_denoter
                                                                        | procedure_heading
                                                                        | function_heading'''
        if len(t)==2:
                t[0] = AST("formal_parameter_section", [t[1]] )
        elif len(t)==4:
                t[0] = AST("formal_parameter_section", [t[1],t[3]] )
        else:
                t[0] = AST("formal_parameter_section", [t[1],t[2],t[4]] )

#Statements----------------------------------------------------------------------------
def p_compound_statement(t):
        'compound_statement : BEGIN statement_sequence END'
        t[0] = AST("compound_statement", [t[2]] )

def p_statement_sequence(t):
        '''statement_sequence : statement_sequence SEMICOLON statement
                                                                | statement'''
        if len(t)==2:
                t[0] = AST("statement_sequence", [t[1]] )
        else:
                t[0] = AST("statement_sequence", [t[1],t[3]] )

def p_statement(t):
        '''statement                : open_statement
                                        | closed_statement'''
        t[0] = AST("statement", [t[1]] )

def p_closed_statement(t):
        '''closed_statement : assignment_statement
                                                | procedure_statement
                                                | compound_statement
                                                | repeat_statement
                                                | closed_if_statement
                                                | closed_while_statement
                                                | closed_for_statement
                                                | '''
        if len(t)==2:
                t[0] = AST("closed_statement", [t[1]] )

def p_open_statement(t):
        '''open_statement       : open_if_statement
                                                | open_while_statement
                                                | open_for_statement'''
        t[0] = AST("open_statement", [t[1]] )

#While dentro do for -> program a; begin for count:=1 downto 100 do begin while a<6 do count:=count+1; end; end.

#REPEAT e WHILE------------------------------------------------------------------------------
#program a(d,c); var a:integer; begin a:=1; repeat a:=a+1; until a=6; end.
def p_repeat_statement(t):
        'repeat_statement : REPEAT statement_sequence UNTIL expression'
        t[0] = AST("repeat_statement", [t[2],t[4]] )

#program a; begin a:=5; while a<6 do a:=a+1; end.
#program a; begin a:=5; while a<6 do begin a:=a+1; end; end.
def p_open_while_statement(t):
        'open_while_statement : WHILE expression DO open_statement'
        t[0] = AST("open_while_statement", [t[2],t[4]] )

def p_closed_while_statement(t):
        'closed_while_statement : WHILE expression DO closed_statement'
        t[0] = AST("closed_while_statement", [t[2],t[4]] )

#FOR-----------------------------------------------------------------------------------------
#program a; var sum:integer; begin sum:=0; for count:=1 to 100 do sum:=sum + count; end.
def p_open_for_statement(t):
        'open_for_statement : FOR assignment_statement direction expression DO open_statement'
        t[0] = AST("open_for_statement", [t[2],t[3],t[4],t[6]] )
```

```
def p_closed_for_statement(t):
        'closed_for_statement : FOR assignment_statement direction expression DO closed_statement'
        t[0] = AST("closed_for_statement", [t[2],t[3],t[4],t[6]] )

def p_direction(t):
        '''direction : TO
                                        | DOWNTO'''
        t[0] = AST("direction", [t[1]] )

#IF e atribuicao de valores a variaveis-----------------------------------------------------------------------
# program a; begin a:=0; b:=1; if a=b then begin a:=1 end; end.
def p_open_if_statement(t):
        '''open_if_statement    : IF expression THEN statement
                                                                | IF expression THEN closed_statement ELSE open_statement'''
        if len(t)==5:
                t[0] = AST("open_if_statement", [t[2],t[4]] )
        else:
                t[0] = AST("open_if_statement", [t[2],t[4],t[6]] )

def p_closed_if_statement(t):
        'closed_if_statement : IF expression THEN closed_statement ELSE closed_statement'
        t[0] = AST("closed_if_statement", [t[2],t[4],t[6]] )

def p_assignment_statement(t):
        'assignment_statement : IDENTIFIER DECLARATOR expression'
        t[0] = AST("assignment_statement", [t[1],t[3]] )

#chamada de procedimentos-------------------------------------------------------------------------------
#program a; begin a(d); end.
def p_procedure_statement(t):
        '''procedure_statement          : IDENTIFIER params
                                                                | IDENTIFIER'''
        if len(t)==2:
                t[0] = AST("procedure_statement", [t[1]] )
        else:
                t[0] = AST("procedure_statement", [t[1],t[2]] )

def p_params(t):
        'params : LEFT_PAREN actual_parameter_list RIGHT_PAREN'
        t[0] = AST("params", [t[2]] )

def p_actual_parameter_list(t):
        '''actual_parameter_list : actual_parameter_list COMMA actual_parameter
                                                                | actual_parameter'''
        if len(t)==2:
                t[0] = AST("actual_parameter_list", [t[1]] )
        else:
                t[0] = AST("actual_parameter_list", [t[1],t[3]] )

def p_actual_parameter(t):
        '''actual_parameter : expression
                                                | expression COLON expression
                                                | expression COLON expression COLON expression'''
        if len(t)==2:
                t[0] = AST("actual_parameter", [t[1]] )
        elif len(t)==4:
                t[0] = AST("actual_parameter", [t[1],t[3]] )
        else:
                t[0] = AST("actual_parameter", [t[1],t[3],t[5]] )

#expressoes legais nos statements-----------------------------------------------------------------------
def p_expression(t):
        '''expression           : simple_expression
                                        | simple_expression relop simple_expression'''
        if len(t)==2:
                t[0] = AST("expression", [t[1]] )
        else:
                t[0] = AST("expression", [t[1],t[2],t[3]] )
```

```
def p_simple_expression(t):
        '''simple_expression    : term
                                                                | simple_expression addop term'''
        if len(t)==2:
                t[0] = AST("simple_expression", [t[1]] )
        else:
                t[0] = AST("simple_expression", [t[1],t[2],t[3]] )

def p_term(t):
        '''term       : primary
                        | term mulop primary'''
        if len(t)==2:
                t[0] = AST("term", [t[1]] )
        else:
                t[0] = AST("term", [t[1],t[2],t[3]] )

def p_primary(t):
        '''primary   : IDENTIFIER
                                | unsigned_constant
                                | function_designator
                                | LEFT_PAREN expression RIGHT_PAREN
                                | NOT primary'''
        if len(t)==2:
                t[0] = AST("primary", [t[1]] )
        elif len(t)==3:
                t[0] = AST("primary", [t[1],t[2]] )
        else:
                t[0] = AST("primary", [t[2]] )

def p_function_designator(t):   #functions with no params will be handled by plain identifier
        'function_designator : IDENTIFIER params'
        t[0] = AST("function_designator", [t[1],t[2]] )

#Alguns conjuntos de operacoes---------------------------------------
def p_unsigned_constant(t):
        '''unsigned_constant : INTEGER
                                        | REAL
                                        | CHAR
                                        | boolean
                                        | STRING'''
        t[0] = AST("unsigned_constant", [t[1]] )

def p_boolean(t):
        '''boolean :   FALSE
                                | TRUE'''
        t[0] = AST("boolean", [t[1]] )

def p_relop(t):
        '''relop : EQUALS
                        | NOT_EQUAL
                        | LESS
                        | GREATER
                        | LESS_OR_EQUAL
                        | GREATER_OR_EQUAL'''
        t[0] = AST("relop", [t[1]] )

def p_addop(t):
        '''addop  : ADD_OP
                        | SUB_OP
                        | OR'''
        t[0] = AST("addop", [t[1]] )

def p_mulop(t):
        '''mulop  : MUL_OP
                        | DIV_OP
                        | DIV
                        | MOD
                        | AND'''
        t[0] = AST("mulop", [t[1]] )
```