Universidade de Coimbra

Departamento de Engenharia Informática

Compiladores

Compilador para a linguagem mPa (mili-Pascal)

by **PascalMasters**

Autor:
António Carlos Lima
2011166926

Autor:
Inês Lopes Petronilho
2012137900

2 de Junho de 2015

Conteúdo

1 Introdução

O projeto consiste no desenvolvimento de um compilador para mili-Pascal, um subconjunto da linguagem de programação Pascal, que mediante um ficheiro de entrada produza o output resultante ou reporte os erros lexicais, sintáticos e semânticos que este possa conter.

Um programa escrito em mili-Pascal terá o mesmo comportamento e significado que um escrito em Pascal segundo o ISO 7185:1990.

Os tipos de variáveis aceites em mili-Pascal são:

- booleanos
- inteiros
- reais

Para efeitos de impressão é possível também usar literais do tipo string. As operações possíveis em mili-Pascal englobam:

- expressões aritméticas e lógicas
- operações relacionais simples
- instruções de atribuição
- instruções de controlo (if-then-else, while-do e repeat-until)
- instruções de saída (writeln)

2 Analisador Lexical

Tendo em conta que mili-Pascal é uma linguagem case-insensitive, os tokens que predefinimos foram

- ID: sequências alfanuméricas começadas por uma letra
- INTLIT: sequências de dígitos decimais
- REALLIT: sequências de dígitos decimais interrompidas por um ponto e opcionalmente seguidas de um expoente (representado por "e") sendo opcionalmente seguido de um sinal de + ou -, ou sequências de dígitos decimais seguidas apenas de um expoente.
- word [a-zA-Z]+
- number [0-9]+
- NEWLINE "\n"
- WHITESPACE " "| "\t"| "\r"
- ASSIGN ":="
- BEGIN_token [bB][eE][gG][iI][nN]
- COLON ":"
- COMMA ","
- DO [dD][oO]
- DOT "."
- ELSE [eE][lL][sS][eE]
- END [eE][nN][dD]
- FORWARD [fF][oO][rR][wW][aA][rR][dD]
- FUNCTION [fF][uU][nN][cC][tT][iI][oO][nN]
- IF [iI][fF]
- LBRAC "("
- NOT [nN][oO][tT]
- OUTPUT [oO][uU][tT][pP][uU][tT]
- PARAMSTR [pP][aA][rR][aA][mM][sS][tT][rR]
- PROGRAM [pP][rR][oO][gG][rR][aA][mM]
- RBRAC ")"
- REPEAT [rR][eE][pP][eE][aA][tT]
- SEMIC ";"
- THEN [tT][hH][eE][nN]
- UNTIL [uU][nN][tT][iI][lL]
- VAL [vV][aA][lL]

- VAR [vV][aA][rR]
- WHILE [wW][hH][iI][lL][eE]
- WRITELN [wW][rR][iI][tT][eE][lL][nN]
- MOD [mM][oO][dD]
- DIV [dD][iI][vV]
- AND [aA][nN][dD]
- OR [oO][rR]
- OP1 {AND} | {OR}
- OP2 "<>"| "<="| ">="| "<"| ">"| "="
- OP3 [+-]
- OP4 "*"| "/"| {MOD} | {DIV}
- RESERVED = palavras reservadas e identificadores requeridos em Pascal standard não usados. Dado o seu elevado número podem ser encontradas no final deste documento como anexo.

2.1 Tratamento de Comentários

Sempre que o analisador Lexical detecta os tokens "(*"ou "{"está na presença de um início de comentário e activa o estado COMMENT, que é declarado no início do Lex como %x COMMENT e é activado através da instrução BEGIN COMMENT.

Após entrar no estado, o Lex limita-se a actualizar o valor das colunas e linhas sempre que encontrar quaisquer tokens diferentes de "*)", "}"ou EOF. Se detectar "*)"ou "}"o comentário termina, logo aplicamos a instrução BEGIN 0 para sair do estado. Se detectar EOF, atingimos o fim do ficheiro sem terminar o comentário logo é impressa a mensagem de erro

- "Line <num linha>, col <num coluna>: unterminated comment".

2.2 Tratamento de Erros Lexicais

Consideramos caracteres ilegais todos aqueles que não foram identificados como um dos tokens de mili-Pascal que predefinimos à partida. Nesse caso, é impressa a mensagem de erro

- "Line <num linha>, col <num coluna>: illegal character (' <c>')"

O analisador recupera da ocorrência de erros Lexicais a partir do fim desse token.

2.3 Invocação e Output

Se tivermos em conta o programa

```
1 program echo(output);
2 var x: integer;
3 begin
4     val(paramstr(1), x);
5     writeln(x)
6 end.
```

Listing 1: Primeiro exemplo de um programa de entrada em mili-Pascal

O analisador mpascanner lê o ficheiro a processar através do stdin e emite o resultado da análise para o stdout através da invocação no terminal do comando **./mpascanner < input.mpa**

O ouput deste programa seria então

```
PROGRAM
ID(echo)
LBRAC
OUTPUT
RBRAC
SEMIC
VAR
ID(x)
COLON
ID(integer)
SEMIC
BEGIN
VAL
LBRAC
PARAMSTR
LBRAC
INTLIT(1)
RBRAC
COMMA
ID(x)
RBRAC
SEMIC
WRITELN
LBRAC
ID(x)
RBRAC
END
DOT
```

3 Analisador Sintático

3.1 Gramática de mili-Pascal

Após identificar os tokens, o Lex retorna-os para o **YACC**, que os processa de acordo com o especificado pela gramática.

Para ilustrar as funcionalidades do nosso compilador vamos utilizar o seguinte exemplo de um programa de entrada (Greatest Common Divisor) em mili-Pascal.

```
1  program gcd(output);
2  var a, b: integer;
3  begin
4      val(paramstr(1), a);
5      val(paramstr(2), b);
6      if a = 0 then
7          writeln(b)
8      else
9          begin
10             while b > 0 do
11                 if a > b then
12                     a := a - b
13                 else
14                     b := b - a;
15             writeln(a)
16         end
17 end.
```

Listing 2: Segundo exemplo de um programa em mili-Pascal

3.1.1 Gramática inicial em notação EBNF

A gramática inicialmente fornecida para a linguagem mili-Pascal é a seguinte:

Prog → ProgHeading SEMIC ProgBlock DOT
ProgHeading → PROGRAM ID LBRAC OUTPUT RBRAC
ProgBlock → VarPart FuncPart StatPart
VarPart → [VAR VarDeclaration SEMIC { VarDeclaration SEMIC }]
VarDeclaration→ IDList COLON ID
IDList → ID { COMMA ID }
FuncPart → { FuncDeclaration SEMIC }
FuncDeclaration→ FuncHeading SEMIC FORWARD
FuncDeclaration → FuncIdent SEMIC FuncBlock
FuncDeclaration → FuncHeading SEMIC FuncBlock
FuncHeading → FUNCTION ID [FormalParamList] COLON ID
FuncIdent → FUNCTION ID
FormalParamList → LBRAC FormalParams { SEMIC FormalParams } RBRAC
FormalParams → [VAR] IDList COLON ID
FuncBlock → VarPart StatPart
StatPart → CompStat
CompStat → BEGIN StatList END
StatList → Stat { SEMIC Stat }
Stat → CompStat
Stat → IF Expr THEN Stat [ELSE Stat]
Stat → WHILE Expr DO Stat
Stat → REPEAT StatList UNTIL Expr
Stat → VAL LBRAC PARAMSTR LBRAC Expr RBRAC COMMA ID RBRAC
Stat → [ID ASSIGN Expr]
Stat → WRITELN [WritelnPList]

WritelnPList → LBRAC (Expr | STRING) { COMMA (Expr | STRING) } RBRAC
Expr → Expr (OP1 | OP2 | OP3 | OP4) Expr
Expr → (OP3 | NOT) Expr
Expr → LBRAC Expr RBRAC
Expr → INTLIT | REALLIT
Expr → ID [ParamList]
ParamList → LBRAC Expr {COMMA Expr} RBRAC

em que [...] representa opcional e {...} representa "zero ou mais repetições". No entanto, a gramática é ambígua e como tal é necessário proceder a algumas alterações para eliminar conflitos de shift/reduce através da definição de precedências e regras de associação dos operadores, assegurando ao mesmo tempo a compatibilidade entre as linguagens mili-Pascal e Pascal.

3.1.2 Gramática do YACC adaptada ao Lex

```
%union {
    tokenInfo *info;
    node* node\textunderscore pointer;
}

typedef struct {
    char* string;
    int line;
    int col;
}tokenInfo;
```

Listing 3: Estruturas utilizadas para comunicar informação entre o YAAC e o Lex

Definimos uma union que define os vários tipos de valores que yylval pode tomar, o que posteriormente nos permite aceder à linha e coluna de cada tokens processado pelo Lex no ficheiro de input.

```
%token <info> ID
%token <info> STRING
%token <info> REALLIT
%token <info> INTLIT
%token <info> OP2
%token <info> OR
%token <info> OP3
%token <info> AND
%token <info> OP4
%token <info> NOT
```

Esta informação sobre estes tokens é necessária aquando da verificação de erros semânticos a fim de indicar correctamente qual a operação, identificador ou literal que viola as regras semânticas de mili-Pascal.

Em YACC, **%left** e **%right** servem para denotar, respectivamente, associatividade à esquerda e à direita.

A associatividade à esquerda ou à direita resolve outra situação de ambiguidade na gramática. O seu propósito é indicar, dentro de operadores com a mesma precedência, por que ordem se dará o processamento. Por exemplo, se considerarmos a expressão

$$a + b - c$$

como definimos que a precedência dos operadores "+"e -"é a mesma, ao acrescentar %left OP2 estamos a especificar que o processamento deve ser feito da esquerda para a direita, ou seja, na forma **(a+b) - c** e não da forma **a + (b-c)**.

```
%right THEN
%right ELSE
%left OP2
%left OR OP3
%left AND OP4
```

```
%right NOT
%left '(' ')'
%right ASSIGN
```

A precedência de tokens segue-se de baixo para cima, ou seja, ASSIGN é o token com maior precedência relativamente aos restantes, ao passo que THEN tem a precedência mais baixa.

O leitor atento notará que ambos os tokens THEN e ELSE têm uma associatividade à direita que a princípio parece confusa. No entanto, estas existem para resolver um problema que, como em tantas outras linguagens de programação, pode surgir: ***Se uma statement do tipo If for seguida de uma statement do tipo If e de um Else com qual If deve o Else emparelhar, o mais próximo ou o mais distante?***

Resolvemos este problema, um conflito entre a possibilidade de efectuar shift ou reduce das expressões do tipo If-Else, atribuindo associatividade à direita a ELSE e a THEN, tendo o ELSE mais precedência que o THEN. Desta forma, sempre que temos uma expressão do tipo

$$IF\ expr\ THEN\ stm\ IF\ expr\ THEN\ stm\ .ELSE\ Stm$$

ao atribuir maior precedência ao ELSE, fazemos shift de ELSE e este será agrupado ao IF mais próximo, e não ao mais distante.

A gramática do YACC adaptada ao Lex fica então com o seguinte aspecto

```
Prog:
    ProgHeading ';' ProgBlock '.'

ProgHeading:
    PROGRAM ID '(' OUTPUT ')'

ProgBlock:
    VarPart FuncPart StatPart

VarPart:
    VAR VarDeclaration ';' VarDeclarationSemicRepeat
  |
  ;

VarDeclarationSemic_Repeat:
    VarDeclaration ';' VarDeclarationSemic_ Repeat
  |
  ;

VarDeclaration:
    IDList ':' ID

IDList:
    ID CommaID_Repeat
CommaID_Repeat:
    ',' ID CommaID_Repeat
  |
  ;

FuncPart:
    FuncDeclaration_Repeat

FuncDeclaration_Repeat:
    FuncDeclaration ';' FuncDeclaration_Repeat
  |
  ;

FuncDeclaration:
    FuncHeading ';' FORWARD
  | FuncIdent ';' FuncBlock
  | FuncHeading ';' FuncBlock
  ;
```

```
43
44      FuncHeading:
45          FUNCTION ID FormalParamList ':' ID
46        | FUNCTION ID ':' ID
47        ;
48
49      FuncIdent:
50          FUNCTION ID
51
52      FormalParamList:
53          '(' FormalParams SemicFormalParams_Repeat ')'
54
55      SemicFormalParams_Repeat:
56          ';' FormalParams SemicFormalParams_Repeat
57        |
58        ;
59
60      FormalParams:
61          VAR IDList ':' ID
62        | IDList ':' ID
63        ;
64
65      FuncBlock:
66          VarPart StatPart
67
68      StatPart:
69          CompStat
70
71      CompStat:
72          BEGIN_token StatList END
73
74      StatList:
75          Stat SemicStat_Repeat
76
77      SemicStat_Repeat:
78          ';' Stat SemicStat_Repeat
79
80      Stat:
81          CompStat
82        | IF Expr THEN Stat ELSE Stat
83        | IF Expr THEN Stat
84        | WHILE Expr DO Stat
85        | REPEAT StatList UNTIL Expr
86        | VAL '(' PARAMSTR '(' Expr ')' ',' ID ')'
87        | IDAssignExpr_Optional
88        | WRITELN WritelnPList_Optional
89        ;
90
91      IDAssignExpr_Optional:
92          ID ASSIGN Expr
93        |
94        ;
95
96      WritelnPList_Optional:
97          WritelnPList
98        |
99        ;
100
101     WritelnPList:
102         '(' Expr CommaExprString_Repeat ')'
103       | '(' STRING CommaExprString_Repeat ')'
104       ;
105
106     CommaExprString_Repeat:
107         ',' Expr CommaExprString_Repeat
108       | ',' STRING CommaExprString_Repeat
109       |
```

```
110         ;
111
112     Expr:
113         SimpleExpr
114       | SimpleExpr OP2 SimpleExpr
115       ;
116
117     SimpleExpr:
118         SimpleExpr OP3 Term
119       | SimpleExpr OR Term
120       | OP3 Term
121       | Term
122       ;
123
124     Term:
125         Term OP4 Term
126       | Term AND Term
127       | Factor
128       ;
129
130     Factor:
131         NOT Factor
132       | '(' Expr ')'
133       | INTLIT
134       | REALLIT
135       | ID ParamList
136       | ID
137       ;
138
139     ParamList:
140         '(' Expr CommaExpr_Repeat ')'
141
142     CommaExpr_Repeat:
143         ',' Expr CommaExpr_Repeat
144       |
145       ;
```

Livre de conflitos do tipo shift/reduce na gramática, podemos começar a construir a Árvore de Sintaxe Abstracta.

3.2 Árvore de Sintaxe Abstracta

A implementação dos nós está de acordo com as seguintes definições impostas pelo enunciado deste projecto:

Programa
$Program(4)$ (Id $VarPart$ $FuncPart$ $< statement >$)
$VarPart(\geq 0)$ ($VarDecl$)
$FuncPart(\geq 0)$ ($FuncDecl$ | $FuncDef$ | $FuncDef2$)

Declaração de variáveis
$VarDecl(\geq 2)$ (Id Id Id)

Declaração de funções
$FuncDecl(3)$ (Id $FuncParams$ Id)
$FuncDef(5)$ (Id $FuncParams$ Id $VarPart$ $< statement >$)
$FuncDef2(3)$ (Id $VarPart$ $< statement >$)
$FuncParams(\geq 0)$($Params$ $VarParams$)
$Params(\geq 2)$ (Id Id Id)
$ValParams(\geq 2)$ (Id Id Id)

Statements
$Assign(2)$ $IfElse(3)$ $Repeat(2)$ $StatList(\geq 0)$
$Valparam(2)$ $While(2)$ $WriteLn(\geq 0)$

Operações
$Add(2)$ $And(2)$ $Call(\geq 2)$ $Div(2)$ $Eq(2)$ $Geq(2)$ $Gt(2)$ $Leq(2)$ $Lt(2)$
$Minus(1)$ $Mod(2)$ $Mul(2)$ $Neq(2)$ $Not(1)$ $Or(2)$ $Plus(1)$ $RealDiv(2)$ $Sub(2)$

Terminais
$IdIntLitRealLitString$

3.2.1 Nós implementados

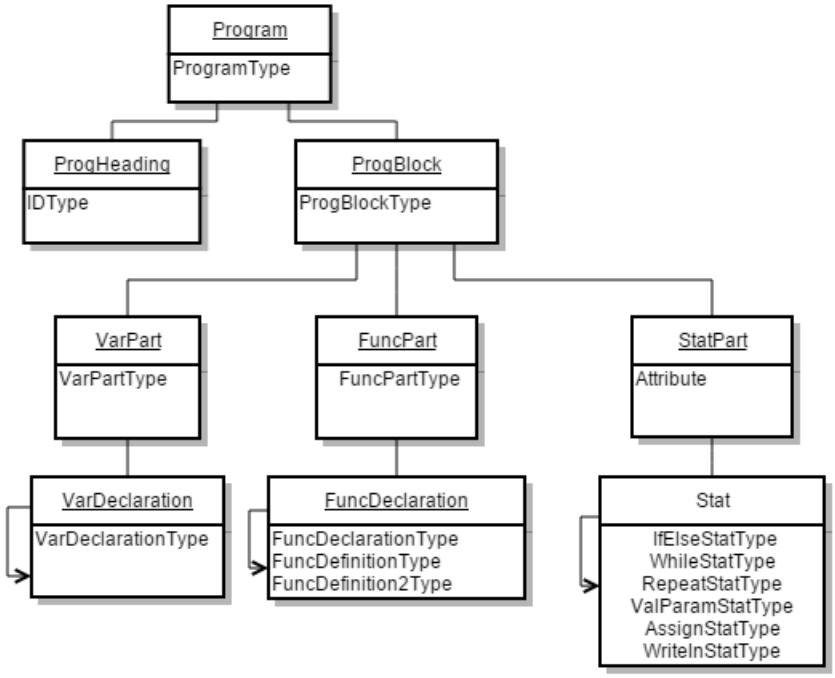
Em vez de criar várias estruturas para cada tipo, optámos por fazer apenas uma estrutura do nó genérico, que é dada por

```
1 typedef struct {
2     nodeType type_of_node;
3     void* field1;
4     void* field2;
5     void* field3;
6     int line;
7     int col;
8 } node;
```

Listing 4: Estrutura utilizada para os nós da AST

em que type_of_node representa o tipo de nó, field1, field2 e field3 são os seus filhos, e line e col representam a linha e coluna onde está situado o nó.

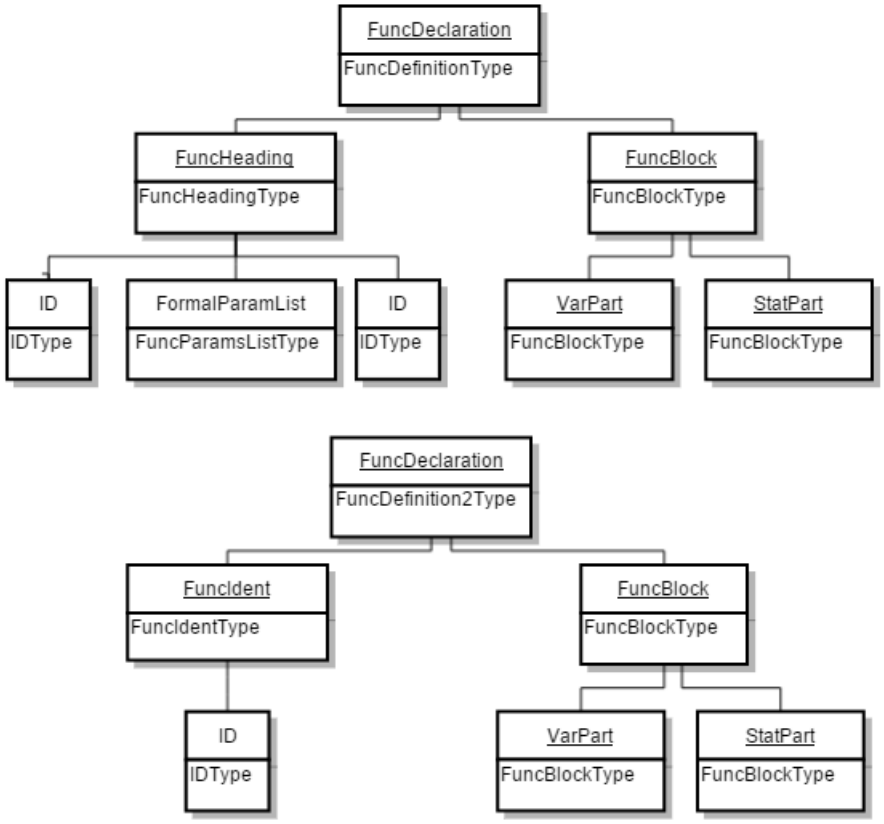
Segue-se uma explicação da nossa Árvore de Sintaxe Abstracta, tomando como ficheiro de entrada o programa **gcd** inserido anteriormente.



A AST pode conter vários nós do tipo **Program**. Cada nó deste tipo tem dois filhos, ProgHeading e ProgBlock. **ProgHeading** é uma folha de tipo ID que contém o identificador do programa. No exemplo de input, o ID seria 'gcd'. **ProgBlock** é um nó que contém o resto do programa em si e que se subdivide em VarPart, FuncPart e StatPart.

VarPart é um nó que engloba o conjunto de declarações de nós do tipo **VarDeclaration**. Estes possuem dois filhos, IDList e ID, que representam respectivamente a lista de IDs de variáveis que são declaradas no início ("a, b") e o seu tipo (neste caso, integer).

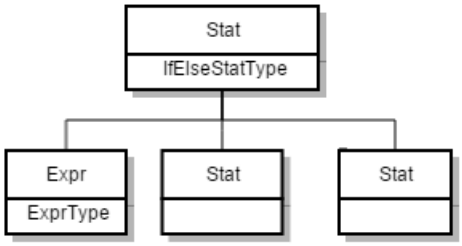
FuncPart tem um filho que é uma repetição de FuncDeclarations. No programa exemplo, não há declaração de funções e é apenas criado um nó do tipo FuncPart com os filhos a NULL.



Há três tipos de nós referentes às declarações de função: FuncDeclarationType, FuncDefinitionType e FuncDefinition2Type.

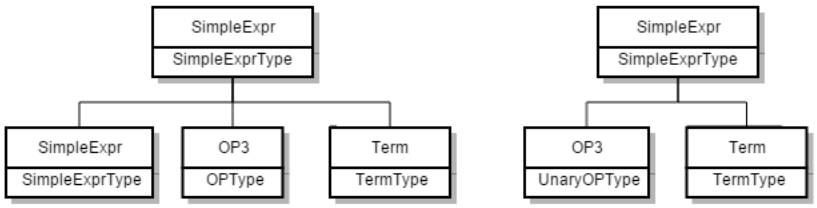
- ***FuncDefinitionType*** define a função e é constituída, por exemplo, por
  function gcd(a, b: integer);
  [bloco da função]
- ***FuncDeclarationType***, é apenas uma declaração da função, pois espera-se que esta seja definida posteriormente no programa. Refere-se a afirmações do tipo
  function gcd(a, b: integer); forward;
- ***FuncDefinion2Type*** indica que a função já foi declarada anteriormente e agora irá ser definida. Um exemplo é
  function gcd;
  [bloco da função]

Cada bloco de função é constituído pela declaração de variáveis e por uma repetição de Statements, englobada pelo nó de tipo **StatPart**. Os seus filhos variam de acordo com o tipo de statement em causa (IfElse, While, Repeat, ValParam, Assign, Writeln). Um nó do tipo IfElseStat pode descrever-se em



É de realçar que estes nós têm sempre três campos. Se a Statement for apenas da forma If sem ser seguido de Else, o terceiro campo será NULL.

Um nó do tipo **Expr** pode derivar em SimpleExpr.



Demonstramos estes dois exemplos de SimpleExpr com o intuito de fazer a distinção de OP3 enquanto operador simples ou operador unário. Os operadores unários apenas podem surgir no início das expressões.

Por fim, um nó do tipo **Term** pode desdobrar-se em algumas operações entre Terms, ou apenas **Factor**, que por sua vez terá como filhos nós do tipo Expr ou folhas que correspondem a inteiros, reais ou IDs.

3.3 Tratamento de erros Sintácticos

Se for encontrado um erro de sintaxe, o analisador imprime a mensagem de erro

- "Line <num linha>, col <num coluna>: syntax error: (' <token>')"

onde <token> é o valor semântico do token que gera o erro. Esta funcionalidade está implementada pela função

```
1 void yyerror (char *s) {
2     printf ("Line %d, cold %d: %s: %s\n", <num linha>, <num coluna>, s, yytext);
3 }
```

3.4 Invocação e Output

A função *printNode* é responsável pela impressão do output. Percorre a árvore após a sua construção e é uma função recursiva na qual o nó é impresso e chamando-se a si própria para cada filho do nó.

Sempre que o analisador se depara com um nó do tipo StatList, examina os tipos dos seus três filhos. O nó é apenas impresso pela função se não possuir qualquer filho do tipo Stat (lista vazia) ou dois ou mais filhos desse tipo, pois um Statement isolado não constitui uma lista.

Ao correr o comando ./mpasemantic -t < gcd.mpa, o programa imprime o seguinte output do programa mencionado anteriormente será então

```
1   Program
2   ..Id(gcd)
3   ..VarPart
4   ....VarDecl
5   ......Id(a)
6   ......Id(b)
7   ......Id(integer)
8   ..FuncPart
9   ..StatList
10  ....ValParam
11  ......IntLit(1)
12  ......Id(a)
13  ....ValParam
14  ......IntLit(2)
15  ......Id(b)
16  ....IfElse
17  ......Eq
18  ........Id(a)
19  ........IntLit(0)
20  ......WriteLn
21  ........Id(b)
22  ......StatList
23  ........While
24  ..........Gt
25  ............Id(b)
26  ............IntLit(0)
27  ..........IfElse
28  ............Gt
29  ..............Id(a)
30  ..............Id(b)
31  ............Assign
32  ..............Id(a)
33  ..............Sub
34  ................Id(a)
35  ................Id(b)
36  ............Assign
37  ..............Id(b)
38  ..............Sub
39  ................Id(b)
40  ................Id(a)
41  ........WriteLn
42  ..........Id(a)
```

4 Analisador Semântico

4.1 Tabelas de Símbolos

Durante a análise semântica, deve ser construída uma tabela para cada região (programa ou função) do programa de entrada, assim como uma tabela exterior. Esta tabela é constituída pelos identificadores possíveis *boolean, integer, real, false* e *true*, o identificador da função pré-definida *paramcount*, e uma referência ao próprio programa.
As tabelas correspondentes aos programas devem conter os identificadores das variáveis e funções definidas, enquanto que as tabelas de funções deverão possuir o próprio identificador da função e os identificadores dos respectivos parâmetros formais e variáveis locais.

Considere-se então a estrutura para definir um símbolo que é dada por

```
1     typedef struct {
2         char* name;
3         PredefType type;
4         PredefFlag flag;
5         char* value;
6         char* value;
7         int isDefined;
8         table* declarationScope;
9         void* nextSymbol;
10    }symbol;
```

Listing 5: Estrutura utilizada para os símbolos de ST

e os enumeradores

```
1     typedef enum {
2         _boolean_, _integer_, _real_, _function_, _program_, _type_, _true_, _false_,
              _string_, NULL_
3     }PredefType
4
5     typedef enum {
6         constantFlag, returnFlag, paramFlag, varparamFlag, NULLFlag
7     }PredefFlag
```

Listing 6: Enumeradores utilizados na ST

em que type representa o tipo do identificador e a flag 'constant' identifica se o símbolo é constante ou não, ou seja, se o seu valor pode ser modificado ou é fixo. As flags são do tipo 'return', 'param' e 'valparam' e são usadas para indicar o valor de retorno e parâmetros formais das funções. O campo declarationScope representa a scope em que o símbolo está inserido e isDefined é usado para indicar se o símbolo já foi definido anteriormente no programa, o que será importante para o tratamento de erros.

Para as tabelas, criámos a seguinte estrutura

```
1     typedef struct {
2         PredefTable type;
3         void* symbol_variables;
4         void* childrenTableList;
5         void* nextSiblingTable;
6         void* parentTable;
7     } table;
8
9     typedef enum {
10        outerTable, programTable, functionTable
11    }predefTable;
```

Listing 7: Estrutura utilizada para uma ST

Cada tabela tem um tipo (exterior, de programa ou de função), um pai, um ponteiro para o primeiro filho da lista childrenTableList (lista de tabelas incluídas no seu scope) e um ponteiro para o seu próximo irmão (a próxima tabela do scope actual).

4.2 Tratamento de erros semânticos

Os diversos tipos de erros semânticos que é possível detectar são

1. Cannot write values of type <type>
2. Function identifier expected
3. Incompatible type for argument <num> in call to function <token> (got <type>, expected <type>)
4. Incompatible type in assignment to <token> (got <type>, expected <type>)
5. Incompatible type in <statement> statement (got <type>, expected <type>)
6. Operator <token> cannot be applied to type <type>
7. Operator <token> cannot be applied to type <type>, <type>
8. Symbol <token> already defined
9. Symbol <token> not defined
10. Type identifier expected
11. Variable identifier expected
12. Wrong number of arguments in call to function <token> (got <number>, expected <number>)

Passamos a explicar os erros mencionados:

1. Tentamos imprimir um ID que não representa um dos tipos pré-definidos (inteiro, real ou booleano). Quando durante a travessia da árvore o analisador se depara com um nó do tipo Writeln, verificamos na scope se o tipo do campo que será imprimido pelo nó é válido (deve ser da forma integer, boolean, real, string).

```
1    writeln(integer)
```

2. Chamadas de funções com identificadores que não constam da scope actual nem das scopes superiores. Sempre que nos deparamos com um nó do tipo CallType, procuramos o ID da função na tabela actual e nos pais até ser encontrado. Se chegar à raiz sem ser identificado, é impresso o erro.

```
1    a := undefinedFunction();
```

3. O argumento de uma Call não está de acordo com o tipo esperado para esse argumento.
   Quando detectamos um nó do tipo CallType, verificamos na lista de tabelas qual o ID que corresponde à função que tentamos evocar. Dentro dessa tabela, possuímos a lista de parâmetros da função (identificados pela flag 'param') e, um a um, verificamos se os tipos de cada parâmetro encontrado coincidem com os tipos passados à função.

```
1    var exemplo: string, a: integer
2    function my_func(var b: integer): integer;
3    a := myFunc(exemplo);
```

4. Está a ser atribuído a um ID um tipo diferente do da sua declaração.
   É aplicável a nós do tipo AssignStat, que são do tipo ID := Expr. Verificamos o tipo dos dois símbolos e, se não coincidir, imprimimos a mensagem de erro. A excepção a esta regra será a atribuição de inteiros a reais, que é válido.

```
1    var exemplo: integer;
2    exemplo := true;
```

5. Um statement espera um type e recebe outro.
   Os nós IfElseStat, WhileStat, RepeatStat e Valparams todos esperam uma expressão com um tipo definido. Nos três primeiros casos a expressão deve ser do tipo 'boolean', enquanto que em nós do tipo ValParams é necessário que este receba um inteiro forçosamente como primeiro argumento. Se algum dos tipos não coincidir, é gerada a mensagem de erro.

```
1    if a + b then
```

6. Operações unárias inválidas.
   Estas operações só possuem um operando à direita, logo estamos na presença de uma operação unária ("+", -", "not"). Em função do operador, verificamos se os tipos são válidos. "+"e -"aplicam-se apenas a inteiros e reais, e "not"aplica-se a booleans.

```
1    a := -true;
```

7. Operações inválidas (por exemplo, somas não podem ser aplicadas a booleanos, operadores lógicos não podem ser aplicados a strings, integers e reals.
   Comparamos o tipo da expressão à esquerda do operador com a expressão que se encontra do lado direito e verificamos se os tipos são válidos de acordo com o operador.

```
1    a := 2 + true;
```

8. Existe uma variável definida e tentamos definir uma com o mesmo nome.
   Aplicado a nós do tipo VarDeclaration, VarParams e Params. Procuramos o ID do símbolo e se estiver declarado previamente numa das tabelas é impresso o erro.

```
1    var x, X: integer
```

9. Um símbolo não consta da tabela de símbolos actual ou das que lhe são exteriores.
   Sempre que encontramos um ID, verificamos na scope actual ou nas scopes anteriores se este está declarado e, caso não seja encontrado, é impressa a mensagem de erro.

```
1    var a, c: integer;
2    b := a + c;
```

10. Atribuição de um tipo que não esteja pre-definido numa declaração de variáveis.
    Aplicado aos tipos VarDeclaration, VarParams e ParamsType, que são todos constituídos por uma lista de variáveis e um ID que identifica o seu tipo. Ocorre sempre que, após comparação, este ID não for compatível com os tipos predefinidos.

```
1    var a, b: c;
```

11. Atribuir um valor a uma variável constante.
    Aplicado ao tipo AssignStat. Pesquisamos o ID que está a ser atribuído na scope actual e nos pais até ser encontrado e verificamos a sua flag. Se for do tipo constantFlag significa que o valor do símbolo não pode ser alterado e o erro é impresso.

```
1    integer := 1;
```

12. Quando recebemos o número errado de parâmetros.
    Quando detectamos um nó do tipo CallType, verificamos na lista de tabelas qual o ID que corresponde à função que tentamos evocar. Dentro dessa tabela, temos a lista de parâmetros necessários e contamo-los. Se for diferente do numero de parâmetros que passamos à função, imprimimos esta mensagem de erro.

```
1    function myFunc(var a, b: integer): integer;
2    c := myFunc(a);
```

4.3 Invocação e Output

Consideremos o seguinte programa de entrada

```
1   program gcd2(output);
2   var x, y: integer;
3   function gcd(A, B:integer): integer;
4       begin
5           if A = 0 then
6               writeln(B)
7           else
8               begin
9                   while B > 0 do
10                      if A > b then
11                          A := A - B
12                      else
13                          B := B - A;
14                  writeln(A)
15              end
16      end;
17  begin
18      if paramcount >= 2 then
19          begin
20              val(paramstr(1), x);
21              val(paramstr(2), y);
22              writeln(gcd(x, y))
23          end
24      else
25          writeln('Error: two parameters required.')
26  end.
```

Listing 8: Terceiro exemplo de um programa em mili-Pascal

Ao correr o comando ./mpasemantic -s < gcd2.mpa, o programa imprime o seguinte output

```
1   ===== Outer Symbol Table =====
2   boolean _type_ constant _boolean_
3   integer _type_ constant _integer_
4   real _type_ constant _real_
5   false _boolean_ constant _false_
6   true _boolean_ constant _true_
7   paramcount _function_
8   program _program_
9
10  ===== Function Symbol Table =====
11  paramcount _integer_ return
12
13  ===== Program Symbol Table =====
14  x _integer_
15  y _integer_
16  gcd _function_
17
18  ===== Function Symbol Table =====
19  gcd _integer_ return
20  a _integer_ param
21  b _integer_ param
```

5 Geração de Código

Infelizmente, sendo apenas possível avaliar de forma automática a etapa de geração de código através do output gerado pelo mesmo e não por análise do ficheiro com código **LLVM**, não conseguimos demonstrar os frutos do nosso esforço a fim colmatar as restantes etapas deste projecto. No entanto, não quer isto dizer que não tenhamos tentado, e como tal, segue-se a explicação do que foi implementado para tentar traduzir o código fonte em mPa para a **Representação Intermédia (IR)**.

5.1 Estruturas de Dados

O principal problema ao traduzir código de mili-Pascal para a IR em **LLVM** tem por base o facto de nem todas as operações válidas em mPa poderem ser desdobradas em apenas uma instrução de 3 endereços. Por exemplo, a seguinte atribuição

```
1    a := 7 * ( 2 + ( b / 2 ) )
```

tem de ser desdobrada da seguinte forma

```
1    %1 = sdiv i32 %b, 2
2    %2 = add i32 2, %1
3    %a = mul i32 7, %2
```

Deste modo, é necessário guardar contar o número de variáveis que são utilizadas bem como guardar o índice e o tipo de dados das variáveis temporárias onde ficam armazenados os resultados intermédios das operações.

```
1    #define COUNTER_TYPE unsigned int
2
3    COUNTER_TYPE localVarCounter;
4    COUNTER_TYPE labelCounter;
5    COUNTER_TYPE tabCounter;
6    table* curFunctionScope;
7
8    typedef enum {
9      llvm_i1, llvm_i8, llvm_i32, llvm_double, llvm_null
10   } LLVMType;
11
12   typedef struct {
13     COUNTER_TYPE returnVarNum;
14     LLVMType returnVarType;
15   } LLVMReturnReff;
```

Listing 9: Estruturas utilizadas para gerar código LLVM

Com a estrutura de dados **LLVMReturnReff** podemos obter o efeito desejado e ir armazenando uma referência que nos permite recursivamente indicar onde foram armazenados os resultados dos operandos envolvidos em cada operação.

Como se pode notar, o mesmo problema de referência também surge aquando da criação de labels para fabricar o flow control das statements de mili-Pascal.

5.2 Operações

Nesta secção descrevemos como obtivemos o efeito pretendido por cada operação disponível em mPA utilizando as funções disponibilizadas para LLVM.

5.2.1 Unárias

A adição unária pode ser considerada uma simples adição de 0 com o valor passado

```
1    %res = add <tipo de %a> 0, %a
```

ao passo que, analogamente, a subtração unária pode ser obtida por subtração do valor passado a 0

```
1    %res = sub <tipo de %a> 0, %a
```

A negação já requer um pouco mais de engenho dada inexistência de uma operação em LLVM para o efeito. No entanto, o mesmo efeito pode ser obtido comparando o valor passado ao valor de lógico *false* com a operação de diferença e consequentemente efectuando um XOR com o valor lógico *true*.

```
1    %temp1 = icmp ne i1 %a, 0
2    %res = xor i1 %temp1, true
```

Podemos resumir este comportamento da seguinte forma:

	icmp ne a, 0	xor temp, 1
a = 1	temp = 1	res = 0
a = 0	temp = 0	res = 1

A	B	XOR A, B
1	1	0
1	0	1
0	1	1
0	0	0

5.2.2 Binárias

As operações mais fáceis de traduzir devido à existência de equivalentes são:

- Diferença

```
1    %res = icmp ne <i1, i32> %a, %b
2    %res = fcmp ne double %a, %b
```

- Menor ou igual ($a \leq b$)

```
1    %res = icmp sle <i1, i32> %a, %b
2    %res = fcmp sle double %a, %b
```

- Maior ou igual ($a \geq b$)

```
1    %res = icmp sge <i1, i32> %a, %b
2    %res = fcmp sge double %a, %b
```

- Menor ($a < b$)

```
1    %res = icmp slt <i1, i32> %a, %b
2    %res = fcmp slt double %a, %b
```

- Maior ($a > b$)

```
1    %res = icmp sgt <i1, i32> %a, %b
2    %res = fcmp sgt double %a, %b
```

- Igualdade ($a = b$)

```
1    %res = icmp eq <i1, i32> %a, %b
2    %res = fcmp eq double %a, %b
```

- Soma $(a + b)$

```
%res = add <i32, double> %a, %b
```

- Subtração $(a - b)$

```
%res = sub <i32, double> %a, %b
```

- Divisão de inteiros $(a\ div\ b)$

```
%res = sdiv i32 %a, %b
```

- Multiplicação $(a * b)$

```
%res = sdiv i32 %a, %b
%res = fdiv double %a, %b
```

- Divisão (a/b)

```
%res = mul i32 %a, %b
%res = fmul double %a, %b
```

Um pouco mais complexa, temos a tradução da operação **mod**

- Módulo $(a\ mod\ b)$
  Em mili-Pascal o resultado do módulo é sempre um inteiro positivo pelo que, caso **a** seja negativo temos de adicionar **b** ao resultado da operação do módulo dado que em llvm o módulo pode retornar valores negativos.

```
   %res = srem i32 %a, %b
 %temp1 = icmp slt i32 %a, 0
 br i1 %temp1, label %if.then, label %if.else
 if.then:
    %final_res = add i32 %res, %b
     br label %if.cont
 if.else:
     %final_res = add i32 %res, 0
   br label %if.cont
 if.cont:
```

5.3 Statements

Seguem-se as statements **If-Else**, **While** e **Assignment** cuja implementação está assente também sobre o contador de labels, a fim de garantir que os emparelhamentos entre as labels são consistentes.

- If Expression Then Statement Else Statement

```
    br i1 %expression_bool_value, label %if.then<num>, label %if.else<num>

    if.then<num>:
        /*
         * generateLLVMStatement(ifStatement);
         */
        br label %if.end<num>

    if.else<num>:
        /*
         * generateLLVMStatement(elseStatement);
         */
        br label %if.end<num>

    if.end<num>:
```

- While Expression Do Statement

```
    br label %while.start<num>

    while.start<num>:
        /*
         * generateLLVMExpression(expression);
         */
        br i1 %expression_bool_value, label %while.do<num>, label %while.end<num>

    while.do<num>:
        /*
         * generateLLVMStatement(statement);
         */
        br label %while.start<num>

    while.end<num>:
```

- Atribuição ($a := b$)

```
    store <type>* %a, <type>* %b
```

5.4 Funções

Aquando da definição de funções é necessário tomar várias precauções, nomeadamente a alocação das variáveis que representam os parâmetros da função na stack do programa e o armazenamento dos valores que lhes são passados, a fim de poderem ser utilizados pela função.

Assim, a seguinte função

```
function gcd(A, B: integer): integer;
    begin
        gcd := 0
    end;
```

é traduzida em

```
define i32 @gcd(i32 %a.param, i32 %b.param){

  %a = alloca i32
  store i32 %a.param, i32* %a
  %b = alloca i32
  store i32 %b.param, i32* %b

  ret i32 0
}
```

6 Problemas e funcionalidades não implementadas

Nesta secção abordamos os problemas que ficaram por solucionar ao longo da implementação do compilador.

6.1 Árvore de Sintaxe Abstracta

Ao criar a AST não conseguimos eliminar todos os nós supérfluos do tipo StatementList, nomeadamente os que resultam de sequências encadeadas de Begin End sem conteúdo.

6.2 Tabela de Símbolos

Na verificação de erros semânticos, após a criação da **Tabela de Símbolos**, temos várias falhas e todas elas relacionadas com a utilização e definição de funções em mili-Pascal.

Não efectuamos correctamente a verificação de redefinições de funções que sejam definidas inicialmente por **forward declaration**. A verificação do **tipo de retorno de uma função** no seu cabeçalho também não é feita correctamente para símbolos que não tenham sido previamente declarados. Chamadas a **funções sem parâmetros** sem utilizar parêntesis não são tratadas como tal mas sim, erroneamente, como uma referências a identificadores de variáveis.

Ainda no que toca a chamadas a funções, estamos a permitir a **passagem de resultados de expressões a parâmetros do tipo *Var*** quando tal deveria ser um erro do tipo *"Variable identifier expected"*. Por último, não permitimos a **passagem de valores do tipo *Integer* a parâmetros do tipo *Real*** embora em mili-Pascal isto não seja considerado um problema.

Appendices

A Palavras Reservadas em mili-Pascal

- ABS [aA][bB][sS]
- ABSOLUTE [aA][bB][sS][oO][lL][uU][tT][eE]
- ARCTAN [aA][rR][cC][tT][aA][nN]
- ARRAY [aA][rR][rR][aA][yY]
- ASM [aA][sS][mM]
- CASE [cC][aA][sS][eE]
- CHAR [cC][hH][aA][rR]
- CHR [cC][hH][rR]
- CONST [cC][oO][nN][sS][tT]
- CONSTRUCTOR [cC][oO][nN][sS][tT][rR][uU][cC][tT][oO][rR]
- COS [cC][oO][sS]
- DESTRUCTOR [dD][eE][sS][tT][rR][uU][cC][tT][oO][rR]
- DISPOSE [dD][iI][sS][pP][oO][sS][eE]
- DOWNTO [dD][oO][wW][nN][tT][oO]
- EOF [eE][oO][fF]
- EOLN [eE][oO][lL][nN]
- EXIT [eE][xX][iI][tT]
- EXP [eE][xX][pP]
- FILE [fF][iI][lL][eE]
- FOR [fF][oO][rR]
- GET [gG][eE][tT]
- GOTO [gG][oO][tT][oO]
- IMPLEMENTATION [iI][mM][pP][lL][eE][mM][eE][nN][tT][aA][tT][iI][oO][nN]
- IN [iI][nN]
- INHERITED [iI][nN][hH][eE][rR][iI][tT][eE][dD]
- INLINE [iI][nN][lL][iI][nN][eE]
- INPUT [iI][nN][pP][uU][tT]
- LABEL [lL][aA][bB][eE][lL]
- LN [lL][nN]
- MAXINT [mM][aA][xX][iI][nN][tT]

- NEW [nN][eE][wW]
- NIL [nN][iI][lL]
- OBJECT [oO][bB][jJ][eE][cC][tT]
- ODD [oO][dD][dD]
- OF [oO][fF]
- OPERATOR [oO][pP][eE][rR][aA][tT][oO][rR]
- ORD [oO][rR][dD]
- PACK [pP][aA][cC][kK]
- PACKED [pP][aA][cC][kK][eE][dD]
- PAGE [pP][aA][gG][eE]
- PRED [pP][rR][eE][dD]
- PROCEDURE [pP][rR][oO][cC][eE][dD][uU][rR][eE]
- PUT [pP][uU][tT]
- READ [rR][eE][aA][dD]
- READLN [rR][eE][aA][dD][lL][nN]
- RECORD [rR][eE][cC][oO][rR][dD]
- REINTRODUCE [rR][eE][iI][nN][tT][rR][oO][dD][uU][cC][eE]
- RESET [rR][eE][sS][eE][tT]
- REWRITE [rR][eE][wW][rR][iI][tT][eE]
- ROUND [rR][oO][uU][nN][dD]
- SELF [sS][eE][lL][fF]
- SET [sS][eE][tT]
- SHL [sS][hH][lL]
- SHR [sS][hH][rR]
- SIN [sS][iI][nN]
- SQR [sS][qQ][rR]
- SQRT [sS][qQ][rR][tT]
- SUCC [sS][uU][cC][cC]
- TEXT [tT][eE][xX][tT]
- TO [tT][oO]
- TRUNC [tT][rR][uU][nN][cC]
- TYPE [tT][yY][pP][eE]
- UNIT [uU][nN][iI][tT]

- UNPACK [uU][nN][pP][aA][cC][kK]
- WITH [wW][iI][tT][hH]
- WRITE [wW][rR][iI][tT][eE]
- XOR [xX][oO][rR]