• U  C •

Compiladores

Mili-Pascal

Grupo 1vs1

Christophe Oliveira nº2011154912

Noé Godinho nº 2011159459

2 de Junho de 2015

Conteúdo

Introdução

No âmbito da cadeira de Compiladores, foi nos proposto o desenvolvimento de um projeto, cujo o intuito foi fazer um compilador para a linguagem Mili-Pascal, que é um subconjunto da linguagem Pascal Standard ISO 7185:1990.

Trata-se de uma linguagem procedimental, onde é possível utilizar variáveis e literais dos tipos lógico, inteiro e real. Literais do tipo *string* apenas são para efeitos de impressão no ecrã. Para além disto, esta linguagem também implementa expressões aritméticas e lógicas, operações relacionais simples, instruções de atribuição, instruções de controlo e instruções de saída. Para passar parâmetros são usados literais inteiros e são passados através da linha de comandos. Na impressão é usada a instrução e é possível definir funções, mas não procedimentos.

Este compilador foi desenvolvido essencialmente em linguagem C, com auxílio das ferramentas *LEX*, *YACC* e *LLVM*.

Este projeto está dividido em diversas fases, onde estas foram essencialmente:

- Análise Lexical
- Análise Sintática
- Análise Semântica
- Geração de Código

Na primeira fase tratámos essencialmente da verificação dos *tokens*, vendo se estes são ou não válidos, na segunda fase verificámos se os *tokens* analisados na fase anterior seguiam a gramática definida neste projeto e procedemos à construção da AST[1], na terceira fase foi feita a tabela de símbolos e foi feito o tratamento de erros semânticos e, na última fase, foi gerado o código intermédio (*LLVM*) que implementa a mesma funcionalidade que o programa de entrada.

O login utilizado pelo nosso grupo foi **1vs1**.

[1] AST: Abstract Syntax Tree ou Árvore de Sintaxe Abstrata

Análise Lexical

Na primeira fase foi necessário criar o analisador lexical em C e com recurso à ferramenta *LEX*. Para isso foram considerados os seguintes *tokens*:

- ID: Sequências alfanuméricas começadas por uma letra.
- INTLIT: Sequências de dígitos decimais.
- REALLIT: Sequências de dígitos decimais interrompidas por um único ponto e, opcionalmente, seguidas de um expoente ou sequências de dígitos decimais seguidas de um expoente.
  O expoente consiste na letra **e** e, opcionalmente, seguida de um sinal + ou **-** e de uma sequência de dígitos decimais.
- STRING: Sequências de caracteres (excluindo mudanças de linha) iniciadas por uma aspa simples **'** e terminadas pela primeira ocorrência de uma aspa simples que não seja seguida imediatamente por outra aspa simples.
- ASSIGN = ":="
- BEGIN = "begin"
- COLON = ":"
- COMMA = ","
- DO = "do"
- DOT = "."
- ELSE = "else"
- END = "end"
- FORWARD = "forward"
- FUNCTION = "function"
- IF = "if"
- LBRAC = "("
- NOT = "not"
- OUTPUT = "output"
- PARAMSTR = "paramstr"
- PROGRAM = "program"
- RBRAC = ")"
- REPEAT = "repeat"

- SEMIC = ";"
- THEN = "then"
- UNTIL = "until"
- VAL = "val"
- VAR = "var"
- WHILE = "while"
- WRITELN = "writeln"
- OP1 = and |or
- OP2 = <|>|= |<>|<= |>=
- OP3 = + |-
- OP4 = * |/ |mod |div
- RESERVED: palavras reservadas e identificadores requeridos do Pascal *standard* não usados.

A linguagem mili-Pascal é *case-insensitive*, logo não existem distinção entre letras maiúsculas e minúsculas, exceto quando estas ocorrem no interior de cadeias de caracteres. Os tokens OP1, OP2, OP3 e OP4, estão assim agrupados para facilitar nas precedências de operadores, onde cada grupo é aplicado ao mesmo tipo de dados.

As palavras reservadas para este projeto foram: **abs, arctan, array, case, char, chr, const, cos, dispose, downto, eof, eoln, exp, file, for, get, goto, in, input, label, ln, maxint, new, nil, odd, of, ord, pack, packed, page, pred, procedure, put, read, readln, record, reset, rewrite, round, set, sin, sqr, sqrt, succ, text, to, trunc, type, unpack, with** e **write**.

Para o analisador separar os *tokens* são usados os espaços em branco, *tabs*, mudanças de linhas e comentários iniciados por "(*"ou "{", com a condição de que estes terminem pela primeira ocorrência de "*)"e "}".

Para localizar os erros e caracteres inválidos, usamos duas variáveis globais cuja a função é contar linhas e colunas, assim se uma expressão não passar na análise lexical é emitida uma mensagem de erro para o *stdout* com o tipo de erro e a localização do mesmo no ficheiro de entrada, através das linhas e colunas previamente guardadas.

1.1 Ficheiros

Esta meta conta com um único ficheiro, **mpascanner.l**, que contém todo o código desta meta, sendo feito em *LEX*.
No início do ficheiro, temos a declaração das variáveis de linhas e colunas, a declaração do estado necessário para o tratamento de comentários, a declaração do *token reserved*, com todos os *tokens* reservados e um comando para indicar ao *LEX* que os *tokens* lidos serão *case-insensitive*. Uma vez feito isso, temos a criação do estado dos comentários e a leitura dos *tokens* e comparação com os existentes, para poder imprimir o valor.
No caso de haver erros de símbolos não válidos, irá ser impressa uma mensagem com o tipo de erro (seja de *string* inválida ou *tokens* inválidos).

1.2 Pontuação

Nesta meta foi possível conseguir a pontuação máxima na alínea A.

Análise Sintática

Uma vez feita a análise lexical, é necessária a análise sintática.

Esta análise está dividida em duas partes:

- Gramática
- ÁST

A gramática trata da verificação das regras dadas no enunciado, para verificar se, sintaticamente, o programa está correto.
Caso esteja correto, irá ser impressa a AST.

2.1 Gramática

Nesta parte é analisada a ordem dos *tokens* recebidos para ver se a construção do programa faz sentido.
A partir de um conjunto de regras definidos no enunciado, a gramática definida com notação EBNF[2]. Essa gramática teve de ser convertida para BNF[3], de maneira a ser possível implementá-la no programa.

A gramática, em notação EBNF, é representada da seguinte maneira:

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

[2]EBNF: Extended Backus-Naur Form ou Formalismo de Backus-Naur Estendido.
[3]BNF: Backus-Naur Form ou Formalismo de Backus-Naur.

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 seguida, também pertencente à gramática, temos a tabela que representa as prioridades dos diferentes *tokens*:

Operador	Associatividade
+, -, or	Esquerda
*, /, and, mod, div	Esquerda
not	Esquerda
<, >, =, <>, <=, >=	Sem associação
then	Sem associação
else	Sem associação

Desta maneira, asseguramos que o **not** tem mais prioridade, seguido de *****, /, **and**, **mod**, **div** e +, **-**, **or**. No caso de <, >, =, <>, **<=, >=**, **then** e **else**, não tem associatividade, de maneira a não ter erro de sintaxe se for associado, já que, por exemplo, no caso do **if-then** e **if-then-else** o compilador não sabe para onde ir caso não se aplique um **nonassoc** no **then** e **else**.

2.2 AST

Caso o código, ao percorrer a gramática, não origine nenhum erro de sintaxe, ou seja, siga as regras definidas na gramática previamente, a AST será impressa. A mesma é construída durante a execução do programa, mas quando há um erro, o programa termina.
Esta árvore tem como objetivo armazenar a informação existente no ficheiro num conjunto de estruturas interligadas correspondente a cada regra. Uma vez construida, a árvore é impressa em forma DFS[4].

A árvore tem a seguinte estrutura:

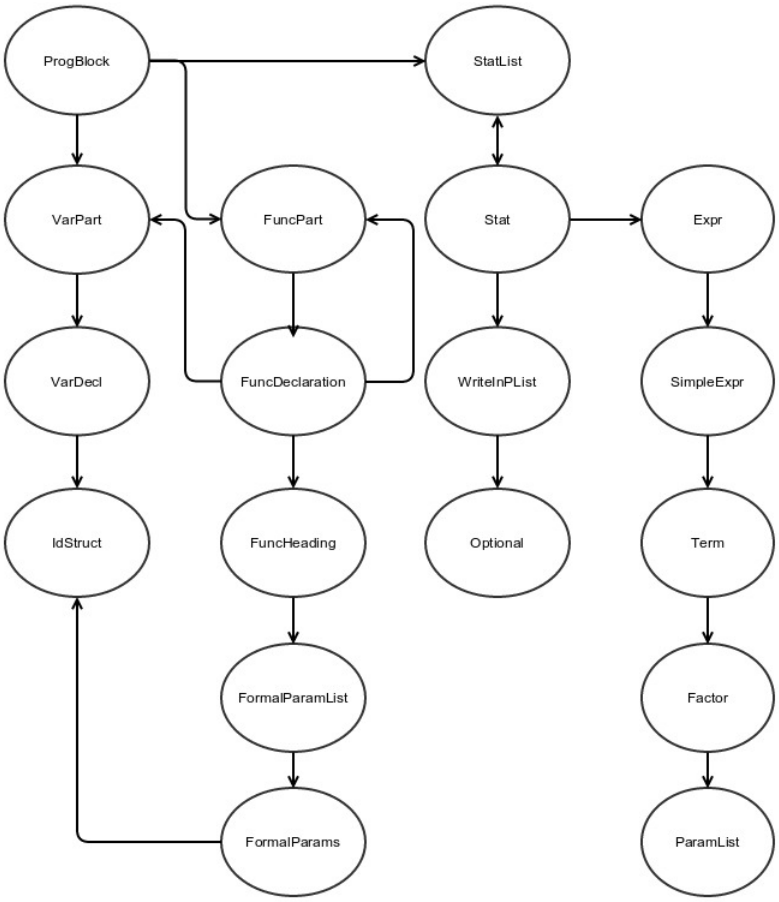

Figura 1: Diagrama das estruturas da AST

[4]DFS: Depth-First Search ou Pesquisa em Profundidade Primeiro

2.3 Ficheiros

Esta meta é constituída por 7 ficheiros diferentes.

- **mpaparser.l:** este ficheiro *LEX* contém a leitura dos *tokens*, com a diferença que, nesta meta, em vez de imprimir o valor do *token*, retorna-o ao ficheiro **mpaparser.y**.
- **mpaparser.y:** este ficheiro *YACC* contém a gramática acima explicada, os *tokens* recebidos e ainda chamadas às funções para criar e imprimir a árvore. Caso exista uma *flag* **-t** como parâmetro na chamada do *parser*, a árvore será impressa.
  Caso haja um erro, a função **yyerror**, que pertence ao *YACC* e fizemos *override*, irá imprimir uma mensagem de erro e terminar o programa.
- **structures.h:** este *header* contém as estruturas necessárias para guardar os *tokens* recebidos na árvore.
- **print.c, print.h:** este ficheiro C e o seu respectivo *header* contém as funções e declarações necessárias para imprimir a AST quando não tem erros e a *flag* **-t** foi usada.
- **functions.c, functions.h:** este ficheiro C e o seu respectivo *header* contém as funções e declarações necessárias para criar a AST.

2.4 Pontuação

Apesar da meta ter começado bem e termos estruturado bem o código e as estruturas necessárias, vários problemas na criação de **StatList** e algumas expressões, não nos permitiram conseguir os pontos todos, ficando com 293 no final, com as alíneas B, C, D, E e F com a pontuação máxima e as alíneas G e I com 30 pontos. Nas restantes não obtivemos nenhum ponto.

2.5 Pós-Meta

Após revisão do código, decidimos refazer a AST desde o início, com uma única estrutura.
Assim, foram eliminados os ficheiros **structures.h**, **print.c**, **print.h** e **functions.c** e foi encapsulado o código da AST num único ficheiro.

- **functions.h:** este ficheiro contém a estrutura usada para armazenar a árvore, a função de criação da árvore, umas funções necessárias para verificar o tipo de *token* recebido e a função de impressão da árvore.

A estrutura é a seguinte:

```
typedef struct Program Program;
struct Program{
    char* type;
    char* value;
    Program *son;
    Program *brother;
};
```

No caso do ficheiro **mpaparser.y**, o código correspondente à criação da AST foi alterado, de maneira a coincidir com as funções no ficheiro acima referido.

Uma vez estruturado o código e criado a árvore com uma única estrutura, foi-nos possível atingir o máximo de pontos existentes para a meta.

Análise Semântica

Usando as ferramentas anteriormente enunciadas, nesta meta o objetivo foi detetar as existências de qualquer tipo de erro, seja a nível lexical, nível sintático ou nível semântico. Isto implica que as análises sintática e semântica têm de estar sem erros, para que seja possível proceder á impressão das tabelas de símbolos.

Caso não hajam erros na AST, existem duas fases que servem para a deteção de erros semânticos propriamente ditos e que são, respetivamente:

- Deteção de símbolos duplicados
- Deteção de utilização de símbolos não existentes ou incompatibilidades de tipos

Correspondentes respetivamente às fases de tabela de símbolos e tratamento de erros semânticos a seguir detalhadas.

3.1 Tabela de Símbolos

À medida que vamos fazendo a análise semântica, vamos construindo a tabela de símbolos para cada programa ou função do programa de entrada. Para a criação propriamente dita, existem tabelas por *default* que contém os identificadores requeridos *boolean*, *integer*, *real*, *false* e *true* e o identificador da função pré-definida *paramcount*, que implementará o acesso ao número de parâmetros passados na linha de comandos, ficando com o seguinte aspeto:

```
===== Outer Symbol Table =====
boolean _type_ constant _ boolean_
integer _type_ constant _integer_
real _type_ constant _real_
false _boolean_ constant _false_
true _boolean_ constant _true_
paramcount _function_
program _program_

===== Function Symbol Table =====
paramcount _integer_ return
```

A tabela correspondente ao programa irá conter os identificadores das variáveis e funções nele declaradas e definidas, respetivamente e as tabelas correspondentes às funções irão conter o próprio identificador da função (enquanto valor de retorno) e os identificadores dos respetivos parâmetros formais e variáveis locais. Para isso percorremos a AST e as respetivas tabelas e, para cada símbolo, verifica-se se nenhum dos símbolos anteriormente definidos no mesmo espaço de variáveis apresenta o mesmo nome.

Caso haja, é emitido um erro e a pesquisa para. Para controlar isso dispomos de uma variável global (**hasErrorsSemantic**), existente no ficheiro **mpasemantic.y**, que controla a impressão da tabela de símbolos.

Para a construção da tabela foram usadas duas estruturas denominadas **SymbolTableHeader** e **SymbolTableLine**, onde a primeira trata de fazer a ligação entre as sub-tabelas e trata de guardar a primeira linha e a segunda trata de ligar as várias linhas pertencentes a uma sub-tabela, como se pode observar no seguinte código:

```
typedef struct _SymbolTableLine SymbolTableLine;
typedef struct _SymbolTableHeader SymbolTableHeader;
struct _SymbolTableHeader{
    SymbolTableHeader *next;
    int defined;
    char *header_name;
    SymbolTableLine *symbolTableLine;
};

struct _SymbolTableLine{
    char *name;
    char *type;
    char *flag;
    char *value;
    SymbolTableLine *next;
};
```

Desta maneira, a arquitetura da tabela de símbolos fica da seguinte forma:

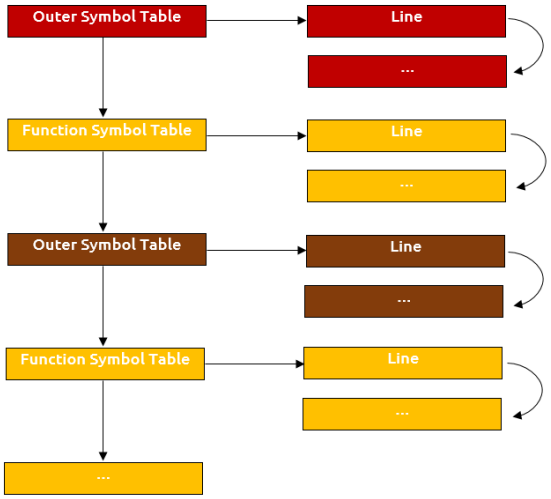

Figura 2: Arquitetura da Tabela de Símbolos

3.2 Tratamento de Erros Semânticos

Esta segunda fase demonstrou ser mais complexa e, inicialmente, para a deteção dos erros, foi necessário localizar o erro no ficheiro de entrada, pois todas as mensagens reportadas para o *stdout* tinham de conter o prefixo: ***"Line <linha >, col <coluna >: "***.

Para isso foram definidas no ficheiro *LEX* algumas variáveis, cuja tarefa é guardar a linha e a coluna em que cada *token* ocorre no ficheiro de entrada. Após isto, para que o *YACC* soubesse as linhas e colunas do respetivo *token* apenas foi necessário acrescentar a linha ***%locations*** ao *YACC* e acrescentar mais duas variáveis em cada nó da árvore para guardar a linha e a coluna. Essa informação é obtida da forma ***@3.first_line***, onde o número corresponde a ordem desse *token* na regra da gramática.

Os erros tratados nesta fase estão a seguir descritos, onde apenas foi possível detetar alguns:

- Cannot write values
- Function identifier expected
- Incompatible type for argument
- Incompatible type in assigment

- Incompatible type in statement
- Operator cannot be applied
- Symbol already defined
- Symbol not defined
- Type identifier expected
- Variable identifier expected
- Wrong number of arguments in call to function

Assim que o primeiro erro é detetado, a mensagem de erro associada é emitida e a variável global (**hasErrorsSemantic**) é atualizada para o valor 1. Esta variável, como já foi referida anteriormente, é usada para impedir que outros erros sejam emitidos, ou que o código seja gerado havendo erros, pois o compilador verifica sempre se a variável está com o valor 0 ou 1, sendo o valor 0 indicativo de ausência de erros e o valor 1 o oposto.

3.3 Ficheiros

Esta meta é constituída por 4 ficheiros diferentes.

- **mpasemantic.l:** este ficheiro *LEX* contém a leitura dos *tokens*, com a diferença que, nesta meta, foram acrescentadas algumas variáveis para a contagem de linhas e colunas, cujo o seu valor é retornado para o ficheiro **mpasemantic.y**.
- **mpasemantic.y:** este ficheiro *YACC* contém a gramática acima explicada, os *tokens* recebidos e ainda chamadas às funções para criar e imprimir a árvore. Caso exista uma *flag* **-t** como parâmetro na chamada do *parser*, a árvore será impressa, caso exista também a *flag* **-s** e não haja erros, será impressa a tabela de símbolos.
  Caso haja um erro, a função **yyerror**, que pertence ao *YACC* e fizemos *override*, irá imprimir uma mensagem de erro e terminar o programa.
  Caso haja um erro semântico, o tipo será impresso e o programa irá ser terminado.
- **semantic.h:** este *header* contém as estruturas necessárias para guardar a estrutura da tabela de símbolos e contém o código para a construção da mesma, assim como para a análise semântica.
- **functions.h:** este *header* contém as funções e declarações necessárias para criar a AST.

Nesta meta, os ficheiros **mpasemantic.l**, **mpasemantic.y** e **functions.h** foram alterados de maneira a ser possível encontrar a linha de cada *token* no *YACC*.

3.4 Pontuação

Apesar da meta ter começado bem e termos estruturado bem o código e as estruturas necessárias, vários problemas foram surgindo a quando a impressão dos erros semânticos o que fez com que não pudéssemos ter os pontos totais nesta meta e ficando assim com um total de 283 no final, com as alíneas B, C e D com a pontuação máxima e a alínea G com 33 pontos. Nas restantes não obtivemos nenhuma pontuação.
Apesar de não ter obtido pontuação nas alíneas E e F, ainda tentámos fazer a deteção de alguns erros, porém, tenham sido detetados por ordem errada ou outros tipos de bugs dos quais não descobrimos, não tivemos sucesso nas mesmas.
Também não ajudou o facto de termos perdido algum tempo na Pós-Meta 2, o que só nos deixou 4 dias para fazer a Meta 3.

3.5 Pós-Meta

Neste caso, não nos foi possível obter pontos extra por falta de tempo com projetos de outras cadeiras.

Geração de Código

Esta meta teve como objetivo a geração de código intermédio para a posterior compilação no *LLVM*.

Para ser possível, necessitamos da AST e da tabela de símbolos, para obter os tipos das variáveis e funções e, também, as funções e variáveis existentes de uma maneira mais simples.
A AST também é necessária para obter todas as operações realizadas no *StatList* de cada função.

Nesta meta foram utilizadas duas estruturas.
A primeira corresponde ao armazenamento das *strings* com um respectivo *ID*. Como é necessário percorrer a AST e imprimir as strings todas com um determinado *ID*, utilizamos uma lista para as poder distinguir:

```
struct SaveStrings{
    struct SaveStrings *next;
    char *value;
    int id;
};
```

A segunda corresponde ao armazenamento das variáveis do tipo *boolean*. Existem dois tipos de *print* para os *booleans*, conforme tenham valor *true* ou *false*:

```
struct BooleanValue{
    struct BooleanValue *next;
    int value;
    char *name;
    char *function;
};
```

Tal como a meta anterior, existem uns *prints* por *default* ao início de cada programa:

```
@.strNewLine = private unnamed_addr constant [2 x i8] c"\0A\00"
@.strInt = private unnamed_addr constant [3 x i8] c"%d\00"
@.strDouble = private unnamed_addr constant [6 x i8] c"%.12E\00"
@.str = private unnamed_addr constant [3 x i8] c"%s\00"
@.strTrue = private unnamed_addr constant [5 x i8] c"TRUE\00"
@.strFalse = private unnamed_addr constant [6 x i8] c"FALSE\00"

declare i32 @printf(i8*, ...)
declare i32 @atoi(i8*)

@paramcount = common global i32 0
@paramstr = common global i8** null
```

Em seguida, irão ser impressas as variáveis globais (caso existam), as funções (caso existam) com as suas variáveis locais e as operações realizadas e, por último, a função *main*, que contém o corpo global do programa.

4.1 Ficheiros

Esta é constituida por 5 ficheiros diferentes.

- **mpacompiler.l:** este ficheiro *LEX* contém a leitura dos *tokens*, com a diferença que, nesta meta, foram acrescentadas algumas variáveis para a contagem de linhas e colunas, cujo o seu valor é retornado para o ficheiro **mpacompiler.y**.
- **mpacompiler.y:** este ficheiro *YACC* contém a gramática acima explicada, os *tokens* recebidos e ainda chamadas às funções para criar e imprimir a árvore. Caso exista uma *flag* **-t** como parâmetro na chamada do *parser*, a árvore será impressa, caso exista também a *flag* **-s** e não haja erros, será impressa a tabela de símbolos.
  Caso haja um erro, a função **yyerror**, que pertence ao *YACC* e fizemos *override*, irá imprimir uma mensagem de erro e terminar o programa.
  Caso haja um erro semântico, o tipo será impresso e o programa irá ser terminado.
  Só se não existir nenhuma *flag* ativa e nenhum erro, irá ser gerado o código intermédio.
- **semantic.h:** este *header* contém as estruturas necessárias para guardar a estrutura da tabela de símbolos e contém o código para a construção da mesma, assim como para a análise semântica.
- **functions.h:** este *header* contém as funções e declarações necessárias para criar a AST.
- **generator.h:** este *header* contém as funções e estruturas necessárias para a criação de código intermédio.

Nesta meta, nenhum ficheiro foi alterado, exceto a criação do ficheiro **generator.h**.

4.2 Pontuação

Nesta meta tivemos menos pontuação que nas outras, 60 em 400, maioritariamente por falta de tempo. Tivemos a pontuação máxima na alínea B, 3 pontos na alínea D e 7 pontos na F.
Apesar não termos tido pontuação nas alíneas C e E, fizemos o código para declaração de funções e das suas variáveis e, ainda, fizemos o código das expressões, mas tivemos um bug quando existiam *integer* e *real* numa mesma operação e não conseguimos resolver a tempo.

Conclusão

Em conclusão, este projeto esteve longe dos nossos ideais, pois o nosso objetivo foi sempre obter a pontuação máxima em todas as metas e tal só foi possível fazer na primeira meta. Talvez esta discrepância esteja relacionada com a conjugação com outras cadeiras feitas paralelamente, que nos fez atrasar um pouco o começo de cada uma das metas, tendo assim menos tempo para fazer uma boa gestão do projeto, que devido ao curto prazo entre as entregas tem de ser bem definida.

Mesmo com todas as dificuldades, o grupo nunca perdeu a vontade e sempre lutou por cada ponto, conseguindo assim ainda acabar a pós-meta2, indispensável para as metas seguintes, mas não conseguindo tocar na pós-meta3 devido ao tempo escasso. No final, acabámos por ter um total de 836 pontos na plataforma mooshak.

Contudo, apesar de o nosso compilador não estar completo, deu para ter uma boa noção do que é um compilador, como funciona e como se implementa. Tendo o compilador sido maioritariamente desenvolvido na linguagem C, este seria mais complicado de desenvolver, se não tivéssemos o auxílio das ferramentas *LEX*, *YACC* e *LLVM*, que também fizeram com que acrescentássemos e percebêssemos mais informações sobre a área de compiladores.