UNIVERSIDADE DE COIMBRA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
Departamento de Engenharia Informática

Projeto de Compiladores

2014/15– 2º semestre

Licenciatura em Engenharia Informática

Data de Entrega: 1 de Junho 2015

v4.0.1

Nota Importante: Qualquer tentativa de fraude leva à reprovação à disciplina tanto do facilitador como do prevaricador.

Compilador para a linguagem mili-Pascal (mPa)

Este projeto consiste no desenvolvimento de um compilador para a linguagem "mili-Pascal," que é um pequeno subconjunto da linguagem Pascal Standard ISO 7185:1990 com extensões relativas à passagem de parâmetros através da linha de comandos.

Nesta linguagem procedimental, os programas podem incluir dados e operações sobre esses dados. É possível utilizar variáveis e literais dos tipos lógico, inteiro e real. Também é possível usar literais do tipo cadeia de caracteres (string), mas apenas para efeitos de impressão no ecrã. A linguagem implementa expressões aritméticas e lógicas e operações relacionais simples, instruções de atribuição, de controlo (`if-then-else`, `while-do` e `repeat-until`) e de saída (`writeln`).

É possível passar parâmetros, que deverão ser literais inteiros, a um programa em mili-Pascal através da linha de comandos. Os seus valores podem ser recuperados através da construção `val(paramstr(`*i*`),`*v*`)`, que atribui o valor do *i*-ésimo parâmetro à variável *v*. O número de parâmetros pode ser obtido através da função pré-definida `paramcount`. A construção `writeln(...)` permite imprimir valores inteiros, reais (por exemplo, `1.000000000E+00`), lógicos (`TRUE` e `FALSE`), e cadeias de caracteres (por exemplo, `'Bom dia!'`). Finalmente, é possível definir funções, mas não procedimentos. São aceites (e ignorados) comentários dos tipos `(*...*)` e `{...}` (e ainda `{...*)` e `(*...}` !).

O significado de um programa em mili-Pascal será o mesmo que o seu significado em Pascal ISO 7185:1990 com a pré-definição das funções `paramcount`, `paramstr` e do procedimento `val`. Por exemplo, o seguinte programa deverá imprimir o valor do primeiro argumento passado na linha de comandos:

```
program echo(output);
var x: integer;
begin
    val(paramstr(1), x);
    writeln(x)
end.
```

Fases

O projeto será estruturado como uma sequência de quatro metas com ponderação e datas de entrega próprias, a saber:

1. Análise lexical (10%) – até 23 de março de 2015
2. Análise sintática (30%) – até 17 de abril de 2015
3. Análise semântica (25%) – até 4 de maio de 2015
4. Geração de código (20%) + relatório (15%) – até 1 de junho de 2015

Em cada uma das metas, o trabalho será obrigatoriamente validado no mooshak usando um concurso criado especificamente para o efeito. Para além disso, a entrega final do projeto deverá ser feita no inforestudante até às **23h59** do dia **1 de junho**, e incluir:

- Todo o código fonte produzido no âmbito do projeto (*exatamente* os mesmos zip que tiverem sido submetidos atempadamente ao mooshak em cada meta, bem como os que, eventualmente, tiverem sido submetidos às pós-metas) e o login utilizado no mooshak.
- O relatório, que deverá descrever a estratégia de implementação adotada, detalhando aspetos como a especificação das categorias lexicais e da gramática concreta utilizada, a construção da árvore de sintaxe abstrata, o formato interno da tabela de símbolos, o tratamento de erros lexicais, sintáticos e semânticos, o acesso aos parâmetros de entrada, a geração de código, etc.

Defesa e grupos

O trabalho será normalmente realizado por grupos de dois alunos, admitindo-se também que o seja a título individual. A **defesa oral** do trabalho será **individual** e terá lugar entre os dias **8 e 12 de junho de 2015**. A nota da defesa (entre 0 e 100%) multiplica pela média ponderada das pontuações obtidas no mooshak e no relatório à data de entrega de cada uma das metas. *Excecionalmente*, e por motivos justificados (como, por exemplo, falha técnica), poderão ser atribuídas notas superiores a 100% na defesa, mas a classificação final nunca poderá exceder a pontuação obtida no mooshak para as diversas fases à data da última entrega.

Aplicam-se mínimos de 47,5% à nota final após a defesa.

Fase I – Analisador lexical

O analisador lexical deve ser implementado em C utilizando a ferramenta `lex`. Os tokens da linguagem são apresentados de seguida.

NOTA: Em Pascal, não é feita qualquer distinção entre letras maiúsculas e minúsculas, salvo quando estas ocorrem no interior de cadeias de caracteres. Onde a seguir se usam letras minúsculas, também se devem considerar as correspondentes maiúsculas.

Tokens da linguagem mili-Pascal

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", opcionalmente seguida de um sinal de + ou de - , seguida 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. Por exemplo, "`'abc'`" e "`'texto entre ''aspas'''`".

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. NOTA: os identificadores requeridos `boolean`, `false`, `integer`, `real` e `true` serão usados em fases posteriores do projeto, e não deverão ser RESERVED.

Implementação

O analisador deverá chamar-se `mpascanner`, ler o ficheiro a processar através do `stdin`, e emitir o resultado da análise para o `stdout`. Caso o ficheiro `echo.mpa` contenha o programa de exemplo dado anteriormente, a invocação:

```
./mpascanner < echo.mpa
```

deverá imprimir a correspondente sequência de tokens no ecrã. Neste caso:

```
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
```

O analisador deve aceitar (e ignorar) como separador de tokens espaço em branco (espaços, tabs e mudanças de linha), bem como comentários iniciados por `(*` ou `{` e terminados pela primeira ocorrência de `*)` ou `}`. Deve ainda detetar a existência de quaisquer erros lexicais no ficheiro de entrada. Sempre que um token possa admitir mais do que um valor semântico, o valor encontrado deve ser impresso entre parêntesis logo a seguir ao nome do token, como se exemplificou acima para `ID` e `INTLIT`.

Tratamento de erros

Caso o ficheiro de entrada contenha erros lexicais, o programa deverá imprimir uma das seguintes mensagens no `stdout`, conforme o caso:

- "Line <num linha>, col <num coluna>: illegal character ('<c>')\n"
- "Line <num linha>, col <num coluna>: unterminated string\n"
- "Line <num linha>, col <num coluna>: unterminated comment\n"

onde <c>, <num linha> e <num coluna> devem ser substituídos pelos valores correspondentes ao *início* do token que originou o erro. O analisador deve recuperar da ocorrência de erros lexicais a partir do *fim* desse token.

Submissão

O trabalho deverá ser validado no mooshak, usando o concurso criado especificamente para o efeito em `http://mooshak2.dei.uc.pt/~comp2015`. Será tida em conta apenas a última submissão ao problema A desse concurso. Os restantes problemas destinam-se a ajudar na validação do analisador. No entanto, o mooshak não deve ser utilizado como ferramenta de debug!

O ficheiro lex a submeter deve chamar-se `mpascanner.l` e ser colocado num ficheiro zip com o nome `mpascanner.zip`. O ficheiro zip não deve conter quaisquer diretórios.

Fase II – Analisador sintático

O analisador sintático deve ser implementado em C utilizando as ferramentas `lex` e `yacc`. A gramática seguinte define a sintaxe da linguagem miliPascal.

Gramática inicial em notação EBNF

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

Uma vez que a gramática dada é ambígua e é apresentada em notação EBNF, onde […] representa "opcional" e {…} representa "zero ou mais repetições," esta deverá ser modificada para permitir a análise sintática ascendente com o `yacc`. Será necessário ter em conta a precedência e as regras de associação dos operadores, entre outros aspetos, de modo a garantir a compatibilidade entre as linguagens miliPascal e Pascal.

O analisador deverá chamar-se `mpaparser`, ler o ficheiro a processar através do `stdin`, e detetar a existência de quaisquer erros lexicais ou de sintaxe no ficheiro de entrada.

Tratamento de erros

Caso o ficheiro de entrada contenha erros lexicais, o programa deverá imprimir no `stdout` as mensagens definidas na Fase I, e continuar. Caso seja encontrado um erro de sintaxe, o analisador deve imprimir uma mensagem de erro e parar. A mensagem a imprimir será

- "Line <num linha>, col <num coluna>: syntax error: <token>\n"

onde <num linha>, <num coluna> e <token> devem ser substituídos pelos números de linha e de coluna, e pelo valor semântico do token que dá origem ao erro. Isto pode ser conseguido definindo a função:

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

Árvore de sintaxe abstrata

Caso o ficheiro `gcd.mpa` contenha o programa:

```
program gcd(output);
var a, b: integer;
begin
    val(paramstr(1), a);
    val(paramstr(2), b);
    if a = 0 then
        writeln(b)
    else
        begin
            while b > 0 do
                if a > b then
                    a := a - b
                else
                    b := b - a;
            writeln(a)
        end
end.
```

a invocação

```
./mpaparser -t < gcd.mpa
```

deverá gerar a árvore de sintaxe abstrata correspondente, imprimi-la no `stdout` conforme a seguir se explica, e terminar. A árvore de sintaxe abstrata só deverá ser impressa se não houver erros de sintaxe. Caso haja erros lexicais que não causem também erros de sintaxe, a árvore deverá ser impressa imediatamente a seguir às correspondentes mensagens de erro.

As árvores de sintaxe abstrata geradas durante a análise sintática devem incluir apenas nós dos tipos indicados abaixo. Entre parêntesis à frente de cada nó indica-se o número de filhos desse nó e, onde necessário, também o tipo de filhos.

Programa

```
Program(4)          ( Id VarPart FuncPart <statement> )
VarPart(>=0)        ( { VarDecl } )
FuncPart(>=0)       ( { FuncDecl | FuncDef | FuncDef2 } )
```

Declaração de variáveis

```
VarDecl(>=2)        ( Id {Id} Id )    /* último Id é o tipo */
```

Declaração de funções

```
FuncDecl(3)         ( Id FuncParams Id )
FuncDef(5)          ( Id FuncParams Id VarPart <statement> )
FuncDef2(3)         ( Id VarPart <statement> )
FuncParams(>=0)     ( { Params | VarParams } )
Params(>=2)         ( Id {Id} Id )
VarParams(>=2)      ( Id {Id} Id )
```

Statements

```
Assign(2) IfElse(3) Repeat(2) StatList(>=0) ValParam(2)
While(2) WriteLn(>=0)
```

Operações

```
Add(2) And(2) Call(>=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

```
Id IntLit RealLit String
```

Nota: não deverão ser gerados nós supérfluos, nomeadamente `StatList` com menos de dois *statements* no seu interior, salvo quando um *statement* requerido por algum nó for o *statement* vazio.

No caso do programa dado, o resultado deve ser:

```
Program
..Id(gcd)
..VarPart
....VarDecl
......Id(a)
......Id(b)
......Id(integer)
..FuncPart
..StatList
....ValParam
......IntLit(1)
......Id(a)
....ValParam
......IntLit(2)
......Id(b)
....IfElse
......Eq
........Id(a)
........IntLit(0)
......WriteLn
........Id(b)
```

```
......StatList
........While
..........Gt
............Id(b)
............IntLit(0)
..........IfElse
............Gt
..............Id(a)
..............Id(b)
............Assign
..............Id(a)
..............Sub
................Id(a)
................Id(b)
............Assign
..............Id(b)
..............Sub
................Id(b)
................Id(a)
........WriteLn
..........Id(a)
```

Desenvolvimento do analisador

Sugere-se que o desenvolvimento do analisador seja efetuado em duas fases. A primeira deverá visar a tradução da gramática para o `yacc` de modo a permitir detetar e reportar eventuais erros de sintaxe. A segunda deverá incidir sobre a construção da árvore de sintaxe abstrata e sua impressão na saída.

Submissão

O trabalho deverá ser validado no mooshak, usando o concurso criado especificamente para o efeito em `http://mooshak2.dei.uc.pt/~comp2015`. Como na fase anterior, será tida em conta apenas a última submissão ao problema A desse concurso. Os restantes problemas destinam-se a ajudar na validação do analisador, nomeadamente no que respeita à deteção de erros de sintaxe e à construção da árvore de sintaxe abstrata, de acordo com a estratégia de desenvolvimento proposta.

Os ficheiros `lex` e `yacc` a submeter devem chamar-se `mpaparser.l` e `mpaparser.y` e ser colocados juntamente com quaisquer ficheiros adicionais necessários à compilação do analisador num único ficheiro zip com o nome `mpaparser.zip`. O ficheiro zip não deve conter quaisquer diretórios.

Fase III – Análise semântica

A análise semântica da linguagem miliPascal deve ser implementada em C utilizando as ferramentas `lex` e `yacc` tendo por base o analisador sintático desenvolvido na fase anterior. O analisador deverá chamar-se `mpasemantic`, ler o ficheiro a processar através do `stdin`, e detetar a existência de quaisquer erros (lexicais, de sintaxe, ou de semântica) no ficheiro de entrada. Caso o ficheiro `gcd2.mpa` contenha o programa:

```
program gcd2(output);
var x, y: integer;
function gcd(A, B: integer): integer;
    begin
        if A = 0 then
            gcd := B
        else
            begin
                while B > 0 do
                    if A > B then
                        A := A - B
                    else
                        B := B - A;
                gcd := A
            end
    end;
begin
    if paramcount >= 2 then
        begin
            val(paramstr(1), x);
            val(paramstr(2), y);
            writeln(gcd(x, y))
        end
    else
        writeln('Error: two parameters required.')
end.
```

a invocação

```
./mpasemantic < gcd2.mpa
```

deverá levar o analisador a proceder à análise sintática do programa e, sendo este sintaticamente válido, a proceder também à análise semântica.

Por uma questão de compatibilidade com a fase anterior, a invocação

```
./mpasemantic -t < gcd2.mpa
```

deverá gerar a árvore de sintaxe abstrata correspondente, imprimi-la no `stdout` como anteriormente, e terminar *sem* proceder a análise semântica.

Sendo o programa sintática e semanticamente válido, a invocação

```
./mpasemantic -s < gcd2.mpa
```

deve levar o analisador a imprimir no `stdout` a(s) tabela(s) de símbolos

correspondentes, como a seguir se especifica. Caso sejam passadas ambas a opções (`-t` e `-s`), a árvore de sintaxe abstrata deve ser impressa primeiro, seguida de uma linha em branco e da(s) tabela(s) de símbolos.

Tabelas de símbolos

Durante a análise semântica, deve ser construída uma tabela de símbolos para cada *região* (programa ou função) do programa de entrada, incluindo uma tabela exterior contendo os identificadores requeridos `boolean`, `integer`, `real`, `false` e `true`, o identificador da função pré-definida `paramcount` (que implementará o acesso ao número de parâmetros passados na linha de comandos), e uma referência ao próprio programa. Por sua vez, a tabela correspondente ao programa irá conter os identificadores das variáveis e funções nele declaradas e definidas, respetivamente. Finalmente, 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. De notar que o nome do programa não tem significado para o próprio programa, pelo que não figura em qualquer tabela! Todos os identificadores a inserir nas tabelas devem ser previamente convertidos para minúsculas.

Para o programa dado, as tabelas de símbolos a imprimir são as seguintes:

```
===== 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

===== Program Symbol Table =====
x     _integer_
y     _integer_
gcd   _function_

===== Function Symbol Table =====
gcd   _integer_   return
a     _integer_   param
b     _integer_   param
```

Os símbolos (e as tabelas) devem ser apresentados por ordem de declaração no programa fonte. O formato das linhas é "`Name\tType[\tFlag[\tValue]]`". Consideram-se pré-definidos os tipos `_boolean_`, `_integer_` e `_real_`, os pseudo-tipos `_function_`, `_program_` e `_type_`, e ainda os valores booleanos `_false_` e `_true_`. A palavra `constant` é usada para indicar que o identificador é uma constante com um valor pré-definido (que não pode ser modificado, por exemplo, por atribuição), e as palavras `return`, `param` e `varparam` são usadas para indicar o valor de retorno e os parâmetros formais das funções (a passar por valor e por referência), respetivamente. Deve ser deixada uma linha em branco entre tabelas consecutivas.

Tratamento de erros semânticos

Eventuais erros de semântica deverão ser detetados e reportados no `stdout` de acordo com o catálogo de erros abaixo,[1] onde cada mensagem deve ser antecedida pelo prefixo "`Line <linha>, col <coluna>: `" e terminada com um caracter de fim de linha.

```
Cannot write values of type <type>
Function identifier expected
Incompatible type for argument <num> in call to function <token> (got
<type>, expected <type>)
Incompatible type in assigment to <token> (got <type>, expected
<type>)
Incompatible type in <statement> statement (got <type>, expected
<type>)
Operator <token> cannot be applied to type <type>
Operator <token> cannot be applied to types <type>, <type>
Symbol <token> already defined
Symbol <token> not defined
Type identifier expected
Variable identifier expected
Wrong number of arguments in call to function <token> (got <number>,
expected <number>)
```

Caso seja detetado algum erro durante a análise semântica do programa, o analisador deverá terminar com a mensagem de erro apropriada, sem imprimir a tabela de símbolos. Os tipos de dados (`<type>`) a reportar nas mensagens de erro deverão ser os mesmos usados na impressão das tabelas de símbolos, e todos os tokens (`<token>`) deverão ser apresentados tal como aparecem no código fonte (sem conversão para minúsculas). Relativamente aos statements (`<statement>`), deverão ser usadas as designações `if`, `while`, `repeat-until` e `val-paramstr`, em minúsculas.

Os números de linha e coluna a reportar dizem respeito ao primeiro caracter dos seguintes tokens:

- O identificador que dá origem ao erro
- O operador cujos argumentos são de tipos incompatíveis
- O operador ou o identificador da função invocada correspondente à raiz da AST da expressão que é incompatível com a forma como é usada
- O identificador da função invocada quando o número de parâmetros estiver errado

Desenvolvimento do analisador

Sugere-se que o desenvolvimento do analisador seja efetuado em duas fases. A primeira deverá consistir na construção das tabelas de símbolos e sua impressão, e a segunda no tratamento de erros semânticos.

Submissão

O trabalho deverá ser validado no mooshak, usando o concurso criado especificamente

1 Por lapso, falta um "n" na palavra "assignment" na quarta mensagem de erro. Esta gralha propagou-se aos casos de teste no mooshak, pelo que não deverá ser corrigida na implementação do analisador.

para o efeito em `http://mooshak2.dei.uc.pt/~comp2015`. Como nas fases anteriores, será tida em conta apenas a última submissão ao problema A desse concurso. Os restantes problemas destinam-se a ajudar na validação do analisador, nomeadamente no que respeita à construção das tabelas de símbolos e à deteção de erros de semântica, de acordo com a estratégia de desenvolvimento proposta.

Os ficheiros `lex` e `yacc` a submeter devem chamar-se `mpasemantic.l` e `mpasemantic.y` e ser colocados juntamente com quaisquer ficheiros adicionais necessários à compilação do analisador num único ficheiro zip com o nome `mpasemantic.zip`. O ficheiro zip não deve conter quaisquer diretórios.

Fase IV – Geração de código intermédio

O gerador de código intermédio deve ser implementado em C utilizando as ferramentas `lex` e `yacc` a partir do código desenvolvido nas metas anteriores. Deverá chamar-se `mpacompiler`, ler o programa a compilar do `stdin`, e emitir para o `stdout` um programa na representação intermédia do LLVM que implemente a mesma funcionalidade que o programa de entrada.

Por exemplo, a invocação:

```
./mpacompiler < gcd2.mpa > gcd2.ll ↵
```

deverá processar o programa `gcd2.mpa` e escrever o código IR LLVM correspondente em `gcd2.ll`. Este poderá ser executado diretamente com parâmetros de entrada na linha de comandos:

```
lli gcd2.ll 12 8 ↵
```

ou compilado e ligado com:

```
llc gcd2.ll
cc -o gcd2 gcd2.s
```

podendo o executável resultante ser então invocado a partir da linha de comandos:

```
./gcd2 12 8 ↵
```

Em qualquer dos casos, deverá ser impresso no ecrã o resultado:

```
4
```

Para efeitos de verificação, o compilador deve fornecer ainda as seguintes opções definidas nas metas 2 e 3:

`-t` : imprime a árvore de sintaxe abstrata construída durante a análise sintática do programa (se não houver erros sintáticos).

`-s` : imprime o conteúdo da(s) tabela(s) de símbolos após a análise semântica do programa (se não houver erros sintáticos nem semânticos).

Caso sejam passadas ambas a opções, a árvore de sintaxe abstrata deve ser impressa primeiro, seguida das tabelas de símbolos (com uma linha em branco a separá-las). Só deverá ser gerado código caso não haja erros de qualquer tipo nem sejam passadas quaisquer opções na linha de comandos.

Questões de implementação

Os tipos de dados `boolean`, `integer` e `real` da linguagem mili-Pascal deverão ser implementados como os tipos `i1`, `i32` e `double` da representação intermédia LLVM. Valores booleanos deverão ser impressos pelo comando `writeln` como `FALSE` e `TRUE`, enquanto valores inteiros e reais deverão ser impressos no formato "`%d`" e "`%.12E`" da função `printf` da linguagem C, respetivamente.

O corpo do programa em mili-Pascal deve ser traduzido para o corpo da função `main` no código LLVM. A função pré-definida `paramcount` deve ser implementada e incluída diretamente no código gerado, tendo em conta que poderá ter que coexistir com outra função ou variável com o mesmo nome no *scope* do programa.

Submissão do trabalho

O trabalho deverá ser validado no mooshak, usando o concurso criado especificamente para o efeito em `http://mooshak2.dei.uc.pt/~comp2015`. Como nas fases anteriores, será tida em conta apenas a última submissão ao problema A desse concurso. Os restantes problemas destinam-se a ajudar na validação do gerador de código.

Os ficheiros `lex` e `yacc` a submeter devem chamar-se `mpacompiler.l` e `mpacompiler.y` e ser colocados juntamente com quaisquer ficheiros adicionais necessários à compilação do analisador num único ficheiro zip com o nome `mpacompiler.zip`. O ficheiro zip não deve conter quaisquer diretórios.

Relatório e entrega final

Relembra-se que a entrega final do projeto deverá ser feita no inforestudante até às **23h59** do dia **1 de junho**, e incluir:

- Todo o código fonte produzido no âmbito do projeto (*exatamente* os mesmos zip que tiverem sido submetidos atempadamente ao mooshak em cada meta, bem como os que, eventualmente, tiverem sido submetidos às pós-metas), o login utilizado no mooshak, e o nome dos membros do grupo previamente constituído.
- O relatório, que deverá descrever a estratégia de implementação adotada, detalhando aspetos como a especificação das categorias lexicais e da gramática concreta utilizada, a construção da árvore de sintaxe abstrata, o formato interno da tabela de símbolos, o tratamento de erros lexicais, sintáticos e semânticos, o acesso aos parâmetros de entrada, a geração de código, etc.

Caso nem toda a funcionalidade pedida tenha sido implementada, o relatório deverá identificar claramente os aspetos não implementados.

ATENÇÃO: Apenas serão considerados para avaliação, discussão e defesa os projetos previamente avaliados no mooshak e submetidos no inforestudante!