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

Projeto de Compiladores

2015/16 – 2º semestre

Licenciatura em Engenharia Informática

Data de Entrega: 6 de Junho 2016

v3.0.1

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

Compilador para a linguagem mC

Este projeto consiste no desenvolvimento de um compilador para a linguagem mC (mini-C), que é um pequeno subconjunto da linguagem ANSI C (C89/C90).

Nesta linguagem procedimental, os programas podem incluir dados e operações sobre esses dados. É possível utilizar variáveis e literais dos tipos inteiro e caracter, ambos com sinal, ponteiros para dados destes tipos, para outros ponteiros, e ponteiros genéricos (`void*`), variáveis do tipo array unidimensional de inteiros, caracteres ou ponteiros, e literais do tipo string).

A linguagem implementa expressões aritméticas e lógicas, operações relacionais simples, operações sobre ponteiros, e instruções de atribuição, de controlo (`if-else` e `for`) e de saída (`puts`). Implementa ainda funções envolvendo os tipos de dados referidos acima, onde a ausência de parâmetros de entrada ou de valor de retorno é representada pelo tipo `void`. Todos os parâmetros são passados por valor.

É possível passar parâmetros a um programa em mC através da linha de comandos. Supondo que os nomes dados aos argumentos da função `main()` são `argc` e `argv`, os seus valores podem ser recuperados (como strings) indexando o array `argv`. O número de parâmetros pode ser obtido através do parâmetro `argc`. A função `puts()` permite imprimir valores do tipo string. É possível converter de valores inteiros para strings e de strings para inteiros através das funções `itoa()` e `atoi()`, respetivamente.

Finalmente, são aceites (e ignorados) comentários do tipo `/*...*/`.

O significado de um programa em mC será o mesmo que o seu significado em ANSI C, supondo a pré-definição das funções `puts()`, `atoi()` e `itoa()` (cujo código fonte será dado posteriormente). Por exemplo, o seguinte programa:

```
char buf[10];
int main(int argc, char **argv) {
     puts("A resposta e:");
     puts(itoa(1234, buf));
     return 0;
}
```

imprime na consola:

```
A resposta e:
1234
```

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 7 de março de 2016
2. Análise sintática (30%) – até 4 de abril de 2016
3. Análise semântica (30%) – até 2 de maio de 2016
4. Geração de código (30%) – até 6 de junho de 2016

Em cada uma das metas, o trabalho será obrigatoriamente validado no mooshak usando um concurso criado especificamente para o efeito. O código submetido ao mooshak deve estar devidamente comentado de modo a permitir compreender a estratégia de implementação adotada. Para além disso, a entrega final do projeto deverá ser feita no inforestudante até às **23h59** do dia **6 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 um ficheiro `grupo.txt` contendo o login utilizado no mooshak na primeira linha e, nas duas linhas seguintes, o número e nome de cada elemento do grupo, separados por vírgula. Os ficheiros zip correspondentes a cada submissão devem chamar-se `1.zip`, …, `4.zip` para as submissões regulares, e `1pos.zip`, `2pos.zip`, …, para as submissões às pós-metas.

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 realizada **em grupo** e terá lugar entre os dias **13 e 17 de junho de 2016**. A nota da defesa (entre 0 e 100%) diz respeito à prestação **do grupo** e multiplica pela soma das pontuações obtidas no mooshak 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.

Tokens da linguagem mC

ID: Sequências alfanuméricas começadas por uma letra, onde o símbolo "_" conta como uma letra. Letras maiúsculas e minúsculas são consideradas letras diferentes.

INTLIT : sequências de dígitos decimais.

CHRLIT: um único caracter (excepto *newline* ou aspa simples) ou uma "sequência de escape" entre aspas simples. Apenas as sequências de escape \n, \t, \\, \', \" e *ooo* são definidas pela linguagem, onde *ooo* representa uma sequência de 1 a 3 dígitos entre 0 e 7. A ocorrência de uma sequência de escape inválida ou de mais do que um caracter ou sequência de escape entre aspas simples deve dar origem a um erro lexical.

STRLIT: uma sequência de caracteres (excepto *newline* ou aspas duplas) e/ou das sequências de escape definidas acima entre aspas duplas. Sequências de escape não definidas devem dar origem a um erro lexical.

AMP = "&"
AND = "&&"
ASSIGN = "="
AST = "*"
CHAR = char
COMMA = ","
DIV = "/"
ELSE = else
EQ = "=="
FOR = for
GE = ">="
GT = ">"
IF = if
INT = int
LBRACE = "{"
LE = "<="
LPAR = "("
LSQ = "["
LT = "<"
MINUS = "-"
MOD = "%"
NE = "!="
NOT = "!"
OR = "||"
PLUS = "+"
RBRACE = "}"
RETURN = return
RPAR = ")"
RSQ = "]"
SEMI = ";"

VOID = void
RESERVED = keywords do C não utilizadas em mC, bem como os operadores decremento ("--") e incremento ("++").

Implementação

O analisador deverá chamar-se `mccompiler`, ler o ficheiro a processar através do `stdin` e, se invocado com a opção `-l`, emitir o resultado da análise para o `stdout` e terminar.[1] Caso o ficheiro `simple.mc` contenha o programa de exemplo dado anteriormente, a invocação:

```
./mccompiler -l < simple.mc
```

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

```
CHAR
ID(buf)
LSQ
INTLIT(10)
RSQ
SEMI
INT
ID(main)
LPAR
INT
ID(argc)
COMMA
CHAR
AST
AST
ID(argv)
RPAR
LBRACE
ID(puts)
LPAR
STRLIT("A resposta e:")
RPAR
SEMI
ID(puts)
LPAR
ID(itoa)
LPAR
INTLIT(1234)
COMMA
ID(buf)
RPAR
RPAR
SEMI
RETURN
INTLIT(0)
SEMI
```

1 Na ausência da opção `-l`, nada deve ser escrito no stdout à excepção de possíveis mensagens de erro.

`RBRACE`

O analisador deve aceitar (e ignorar) como separador de tokens espaço em branco (espaços, tabs e mudanças de linha), bem como comentários iniciados por `/*` e terminados pela primeira ocorrência de `*/`. 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, INTLIT e STRLIT.

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

onde <num linha> e <num coluna> devem ser substituídos pelos valores correspondentes ao *início* do token que originou o erro, e <c> e <s> devem ser substituídos por esse token. 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 `https://mooshak2.dei.uc.pt/~comp2016`. 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 `mccompiler.l` e ser colocado num ficheiro zip com o nome `mccompiler.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 mC.

Gramática inicial em notação EBNF

Start: → (FunctionDefinition | FunctionDeclaration | Declaration) {FunctionDefinition | FunctionDeclaration | Declaration}
FunctionDefinition → TypeSpec FunctionDeclarator FunctionBody
FunctionBody → LBRACE {Declaration} {Statement} RBRACE
FunctionDeclaration → TypeSpec FunctionDeclarator SEMI
FunctionDeclarator → {AST} ID LPAR ParameterList RPAR
ParameterList → ParameterDeclaration {COMMA ParameterDeclaration}
ParameterDeclaration → TypeSpec {AST} [ID]
Declaration → TypeSpec Declarator {COMMA Declarator} SEMI
TypeSpec → CHAR | INT | VOID
Declarator → {AST} ID [LSQ INTLIT RSQ]
Statement → [Expr] SEMI
Statement → LBRACE {Statement} RBRACE
Statement → IF LPAR Expr RPAR Statement [ELSE Statement]
Statement → FOR LPAR [Expr] SEMI [Expr] SEMI [Expr] RPAR Statement
Statement → RETURN [Expr] SEMI
Expr → Expr (ASSIGN | COMMA) Expr
Expr → Expr (AND | OR) Expr
Expr → Expr (EQ | NE | LT | GT | LE | GE) Expr
Expr → Expr (PLUS | MINUS | AST | DIV | MOD) Expr
Expr → (AMP | AST | PLUS | MINUS | NOT) Expr
Expr → Expr LSQ Expr RSQ
Expr → ID LPAR [Expr {COMMA Expr}] RPAR
Expr → ID | INTLIT | CHRLIT | STRLIT | LPAR Expr RPAR

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 mC e C.

NOTAS:

1. O operador COMMA é associativo à *esquerda*.
2. Relativamente ao operador ASSIGN, o standard limita as expressões do lado esquerdo do sinal "=" a "unary-expressions," enquanto as expressões do lado direito podem ser mais gerais ("conditional-expressions"). Por exemplo, em ANSI C, "a+1=2;" é sintaticamente inválido, embora "(a+1)=2;" já seja válido. Por uma questão de simplicidade, essa distinção não será feita neste projeto.

Implementação

O analisador deverá chamar-se `mccompiler`, ler o ficheiro a processar através do

`stdin` e emitir quaisquer resultados para o `stdout`. Por uma questão de compatibilidade com a fase anterior, se o analisador for invocado com uma das opções `-l` ou `-1`, deverá realizar *apenas* a análise lexical, emitir o resultado dessa análise para o `stdout` (erros lexicais e, no caso da opção `-l`, os tokens encontrados) e terminar.

Se não for passada qualquer opção, o analisador deve detetar a existência de quaisquer erros lexicais ou de sintaxe no ficheiro de entrada, e emitir as mensagens de erro correspondentes para o `stdout`.

Tratamento e recuperação 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 sejam encontrados erros de sintaxe, o analisador deve imprimir mensagens de erro com o seguinte formato:

- "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);
}
```

A analisador deve ainda implementar recuperação local de erros de sintaxe através da adição das seguintes regras de erro à gramática (ou de outras com o mesmo efeito dependendo das alterações que a gramática dada vier a sofrer):

FunctionBody → LBRACE error RBRACE
Declaration → error SEMI
Statement → error SEMI
Statement → LBRACE error RBRACE
Expr → ID LPAR error RPAR
Expr → LPAR error RPAR

NOTA: As regras "Declaration → error SEMI" e "Statement → error SEMI" poderão dar origem a um conflito reduce/reduce no FunctionBody, uma vez que a ocorrência de um erro nas declarações tanto pode ser entendida como uma Declaration ou como o primeiro Statement. Neste caso deve ser dada preferência à Declaration.

Árvore de sintaxe abstrata (AST)

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

```
char buffer[20];

int main(int argc, char **argv) {
    int a, b;
    a = atoi(argv[1]);
    b = atoi(argv[2]);
    if (a == 0) {
        puts(itoa(b, buffer));
    } else {
```

```
        for ( ; b > 0 ; )
            if (a > b)
                a = a-b;
            else
                b = b-a;
        puts(itoa(a, buffer));
    }
    return 0;
}
```

a invocação

```
./mccompiler -t < gcd.mc
```

deverá gerar a árvore de sintaxe abstrata correspondente, e imprimi-la no `stdout` conforme a seguir se explica. 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.

Nó raiz

```
Program(>=1)          (<variable and/or function declarations>)
```

Declaração de variáveis

```
Declaration(>=2)      ( <typespec> {Pointer} Id )
ArrayDeclaration(>=3) ( <typespec> {Pointer} Id IntLit )
```

Declaração/definição de funções

```
FuncDeclaration(>=3)  ( <typespec> {Pointer} Id ParamList)
FuncDefinition(>=4)   ( <typespec> {Pointer} Id ParamList FuncBody)
ParamList(>=1)        ( {ParamDeclaration} )
FuncBody(>=0)         ( {<declarations> | <statements>} )
ParamDeclaration(>=1) ( <typespec> {Pointer} [Id] )
```

Statements

```
StatList(>=2) If(3) For(4) Return(1)
```

Operadores

```
Or(2) And(2) Eq(2) Ne(2) Lt(2) Gt(2) Le(2) Ge(2) Add(2) Sub(2) Mul(2)
Div(2) Mod(2) Not(1) Minus(1) Plus(1) Addr(1) Deref(1) Store(2)
Comma(2) Call(>=1)
```

Terminais

```
Char ChrLit Id Int IntLit Pointer StrLit Void
```

Especial

```
Null       (na ausência de um nó filho obrigatório)
```

Nota: não deverão ser gerados nós supérfluos, nomeadamente `StatList` com menos de dois *statements* no seu interior. Os nós `Program`, `ParamList` e `FuncBody`, não deverão ser considerados redundantes mesmo que tenham menos de dois nós filhos.

No caso do programa dado, o resultado deve ser:

```
Program
..ArrayDeclaration
....Char
....Id(buffer)
....IntLit(20)
..FuncDefinition
....Int
....Id(main)
....ParamList
......ParamDeclaration
........Int
........Id(argc)
......ParamDeclaration
........Char
........Pointer
........Pointer
........Id(argv)
....FuncBody
......Declaration
........Int
........Id(a)
......Declaration
........Int
........Id(b)
......Store
........Id(a)
........Call
..........Id(atoi)
..........Deref
............Add
..............Id(argv)
..............IntLit(1)
......Store
........Id(b)
........Call
..........Id(atoi)
..........Deref
............Add
..............Id(argv)
..............IntLit(2)
......If
........Eq
..........Id(a)
..........IntLit(0)
........Call
..........Id(puts)
..........Call
............Id(itoa)
............Id(b)
............Id(buffer)
........StatList
..........For
............Null
............Gt
..............Id(b)
..............IntLit(0)
............Null
............If
..............Gt
```

```
..............Id(a)
..............Id(b)
............Store
..............Id(a)
..............Sub
................Id(a)
................Id(b)
............Store
..............Id(b)
..............Sub
................Id(b)
................Id(a)
.........Call
...........Id(puts)
...........Call
............Id(itoa)
............Id(a)
............Id(buffer)
......Return
.......IntLit(0)
```

Notar que, em C, `argv[1]` é equivalente a `*(argv+1)`.

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.

Toda a memória alocada durante a execução do analisador deve ser libertada antes deste terminar, devendo ser dada especial atenção aos casos em que a construção da AST é interrompida devido à ocorrência de erros de sintaxe. Aconselha-se vivamente a utilização do analisador estático do Clang (scan-build) e do *memory debugger/profiler* valgrind durante o processo de desenvolvimento.

Submissão

O trabalho deverá ser validado no mooshak, usando o concurso criado especificamente para o efeito em `https://mooshak2.dei.uc.pt/~comp2016`. 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 `mccompiler.l` e `mccompiler.y` e ser colocados juntamente com quaisquer ficheiros adicionais necessários à compilação do analisador num único ficheiro zip com o nome `mccompiler.zip`. O ficheiro zip não deve conter quaisquer diretórios.

Fase III – Análise semântica

A análise semântica da linguagem mini-C deve ser implementada em C tendo por base o analisador sintático desenvolvido com as ferramentas `lex` e `yacc` na fase anterior. O analisador deverá chamar-se `mccompiler`, 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.mc` contenha o programa:

```
char buffer[20];

int gcd(int a, int b) {
    if (a == 0) {
        return b;
    } else {
        for ( ; b > 0 ; )
            if (a > b)
                a = a-b;
            else
                b = b-a;
        return a;
    }
}

int main(int argc, char **argv) {
    int a, b;
    if (argc > 2) {
        a = atoi(argv[1]);
        b = atoi(argv[2]);
        puts(itoa(gcd(a,b), buffer));
    } else
        puts("Error: two parameters required.");
    return 0;
}
```

a invocação

```
./mccompiler < gcd2.mc
```

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, se o analisador for invocado com uma das opções `-t` ou `-2`, deverá realizar *apenas* a análise sintática (e a análise lexical subjacente), emitir o resultado para o `stdout` (erros lexicais e/ou sintáticos e, no caso da opção `-t`, a árvore de sintaxe abstrata se não houver erros de sintaxe) e terminar *sem* proceder a análise semântica.

Sendo o programa sintaticamente válido, a invocação

```
./mccompiler -s < gcd2.mc
```

deve levar o analisador a imprimir no `stdout` a(s) tabela(s) de símbolos correspondentes seguida(s) de uma linha em branco e da árvore de sintaxe abstrata anotada com os tipos das variáveis, funções e expressões, etc, como a seguir se especifica.

Tabelas de símbolos

Durante a análise semântica, deve ser construída uma tabela de símbolos global contendo os identificadores das funções pré-definidas `atoi`, `itoa` e `puts`, bem como os identificadores das variáveis e/ou funções declaradas e/ou definidas no programa. Por sua vez, as tabelas correspondentes às funções definidas no programa irão conter a string "`return`" (usada para representar o valor de retorno) e os identificadores dos respetivos parâmetros formais e variáveis locais.

Para o programa de exemplo dado, as tabelas de símbolos a imprimir são as seguintes. O formato das linhas é "`Name\tType[\tparam]`", onde `[]` quer dizer opcional.

```
===== Global Symbol Table =====
atoi  int(char*)
itoa  char*(int,char*)
puts  int(char*)
buffer      char[20]
gcd   int(int,int)
main  int(int,char**)

===== Function gcd Symbol Table =====
return      int
a     int   param
b     int   param

===== Function main Symbol Table =====
return      int
argc  int   param
argv  char**      param
a     int
b     int
```

Os símbolos (e as tabelas) devem ser apresentados por ordem de *declaração* no programa fonte. Em particular, caso uma função `f1` seja declarada antes e definida depois de outra função `f2`, a tabela da função `f1` deverá ser impressa antes da tabela da função `f2`. Caso uma função seja declarada mas não seja definida, o seu nome e tipo devem aparecer na tabela de símbolos global, mas não deve ser impressa qualquer tabela para essa função. É o caso das funções pré-definidas `itoa`, `atoi` e `puts`.

No essencial, a notação para os tipos segue as convenções do C. Para um array de pointeiros declarado como "`int *a[10];`", o tipo a indicar na tabela de símbolos deverá ser "`int*[10]`" (sem quaisquer espaços). Deve ser deixada uma linha em branco entre tabelas consecutivas, e entre as tabelas e a árvore de sintaxe abstrata anotada.

Árvore de sintaxe abstrata anotada

Para o programa dado, a árvore de sintaxe abstrata anotada a imprimir a seguir às tabelas de símbolos quando é dada a opção `-s` seria a seguinte:

```
Program
..ArrayDeclaration
....Char
....Id(buffer)
....IntLit(20)
..FuncDefinition
....Int
```

```
....Id(gcd)
....ParamList
......ParamDeclaration
........Int
........Id(a)
......ParamDeclaration
........Int
........Id(b)
....FuncBody
......If
........Eq - int
..........Id(a) - int
..........IntLit(0) - int
........Return
..........Id(b) - int
........StatList
..........For
............Null
............Gt - int
..............Id(b) - int
..............IntLit(0) - int
............Null
............If
..............Gt - int
................Id(a) - int
................Id(b) - int
..............Store - int
................Id(a) - int
................Sub - int
..................Id(a) - int
..................Id(b) - int
..............Store - int
................Id(b) - int
................Sub - int
..................Id(b) - int
..................Id(a) - int
..........Return
............Id(a) - int
..FuncDefinition
....Int
....Id(main)
....ParamList
......ParamDeclaration
........Int
........Id(argc)
......ParamDeclaration
........Char
........Pointer
........Pointer
........Id(argv)
....FuncBody
......Declaration
........Int
........Id(a)
......Declaration
........Int
........Id(b)
......If
........Gt - int
..........Id(argc) - int
..........IntLit(2) - int
```

```
........StatList
.........Store - int
...........Id(a) - int
...........Call - int
.............Id(atoi) - int(char*)
.............Deref - char*
...............Add - char**
.................Id(argv) - char**
.................IntLit(1) - int
.........Store - int
...........Id(b) - int
...........Call - int
.............Id(atoi) - int(char*)
.............Deref - char*
...............Add - char**
.................Id(argv) - char**
.................IntLit(2) - int
.........Call - int
...........Id(puts) - int(char*)
...........Call - char*
.............Id(itoa) - char*(int,char*)
.............Call - int
...............Id(gcd) - int(int,int)
...............Id(a) - int
...............Id(b) - int
.............Id(buffer) - char[20]
........Call - int
.........Id(puts) - int(char*)
.........StrLit("Error: two parameters required.") - char[32]
......Return
........IntLit(0) - int
```

Deverão ser anotados apenas os nós correspondentes a expressões. Declarações ou statements que não sejam expressões não devem ser anotados.

Tratamento de erros semânticos

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

```
Conflicting types (got <type>, expected <type>)
Invalid use of void type in declaration
Lvalue required
Operator <token> cannot be applied to type <type>
Operator <token> cannot be applied to types <type>, <type>
Symbol <token> already defined
Symbol <token> is not a function
Unknown symbol <token>
Wrong number of arguments to function <token> (got <number>, required
<number>)
```

Caso seja detetado algum erro durante a análise semântica do programa, o analisador deverá imprimir a mensagem de erro apropriada e continuar, dando o pseudo-tipo `undef` a quaisquer símbolos desconhecidos e aos resultados de operações cujo tipo não possa ser determinado devido aos seus operandos (inválidos), o que pode dar origem a novos erros semânticos. De resto, 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.

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 (conversões entre inteiros e ponteiros, ou entre ponteiros de tipos diferentes que gerem "Warnings" em C, devem dar origem a erros de incompatibilidade de tipos)
- O operador ou o identificador da função invocada correspondente à raiz da AST da expressão que é incompatível com a forma como é usada (considerar que o tipo *esperado* pelas condições das construções `if` e `for` é `int`, embora alguns outros tipos também sejam aceitáveis)
- O identificador da função invocada quando o número de parâmetros estiver errado

A impressão das tabelas de símbolos e da AST anotada (se for o caso) deve ser feita depois da impressão de todas as mensagens de erro.

Desenvolvimento do analisador

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

Submissão

O trabalho deverá ser validado no mooshak, usando o concurso criado especificamente para o efeito em `https://mooshak2.dei.uc.pt/~comp2016`. 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 `mccompiler.l` e `mccompiler.y` e ser colocados juntamente com quaisquer ficheiros adicionais necessários à compilação do analisador num único ficheiro zip com o nome `mccompiler.zip`. O ficheiro zip não deve conter quaisquer diretórios.