Projeto de Compiladores 2021/22

Compilador para a linguagem deiGo

28 de novembro de 2021

Este projeto consiste no desenvolvimento de um compilador para a linguagem deiGo, que é um subconjunto da linguagem Go (`https://golang.org/ref/spec`) de acordo com a especificação de maio de 2018 disponibilizada no material de apoio.

Na linguagem deiGo é possível usar variáveis e literais dos tipos `string`, `bool`, `int` e `float32` (estes dois últimos com sinal). A linguagem deiGo inclui expressões aritméticas e lógicas, instruções de atribuição, operadores relacionais, e instruções de controlo (`if-else` e `for`). Inclui também funções com os tipos de dados já referidos e ainda o tipo especial `[]string`, sendo a passagem de parâmetros sempre feita por valor.

A função `main()` invocada no início de cada programa é declarada na `package main` e não recebe argumentos nem retorna qualquer valor. O programa `package main; func main() {};` é dos mais pequenos possíveis na linguagem deiGo. Os programas podem ler e escrever carateres na consola com as funções pré-definidas `strconv.Atoi(os.Args[...])` e `fmt.Println(...)`, respetivamente.

O significado de um programa na linguagem deiGo será o mesmo que na linguagem Go, assumindo a pré-definição das funções `strconv.Atoi(...)` e `fmt.Println(...)`, bem como da variável `os.Args[...]`. Por fim, são aceites comentários nas formas `/* ... */` e `// ...` que deverão ser ignorados. Assim, por exemplo, o programa que se segue calcula o fatorial de um número passado como argumento:

```
package main;

func factorial(n int) int {
    if n == 0 {
        return 1;
    };
    return n * factorial(n-1);
};

func main() {
    var argument int;
    argument,_ = strconv.Atoi(os.Args[1]);
    fmt.Println(factorial(argument));
};
```

Este programa declara uma variável `argument` do tipo `int` e atribui-lhe o valor inteiro do argumento passado ao programa, usando a função `Atoi` para realizar a conversão (esta função retorna um par de valores e o segundo valor é descartado). De seguida, calcula o fatorial desse valor e invoca a função `Println` para escrever o resultado na consola.

1 Metas e avaliação

O projeto está estruturado em quatro metas encadeadas, nas quais o resultado de cada meta é o ponto de partida para a meta seguinte. As datas e as ponderações são as seguintes:

1. Análise lexical (19%) – 20 de outubro de 2021
2. Análise sintática (25%) – 19 de novembro de 2021 (meta de avaliação)
3. Análise semântica (25%) – 3 de dezembro de 2021
4. Geração de código (25%) – 17 de dezembro de 2021 (meta de avaliação)

A entrega final será acompanhada de um relatório que tem um peso de 6% na avaliação. Para além disso, a entrega final do trabalho deverá ser feita através do Inforestudante, até ao dia seguinte ao da Meta 4, e incluir todo o código-fonte produzido no âmbito do projeto (exatamente os mesmos arquivos `.zip` que tiverem sido colocados no MOOSHAK em cada meta).

O trabalho será verificado no MOOSHAK, em cada uma das metas, usando um concurso criado para o efeito. A classificação final da Meta 1 é obtida em conjunto com a Meta 2 e a classificação final da Meta 3 é obtida em conjunto com a Meta 4. O nome do grupo a registar no MOOSHAK deverá ser obrigatoriamente da forma "uc2019123456_uc2019654321" usando os números de estudante como identificação do grupo na página `https://mooshak.dei.uc.pt/~comp2021` na qual o MOOSHAK está acessível.

1.1 Defesa e grupos

O trabalho será realizado por grupos de dois alunos inscritos em turmas práticas do mesmo docente. Em casos excecionais, a confirmar com o docente, admite-se trabalhos individuais. A defesa oral do trabalho será realizada em grupo entre os dias 3 e 7 de janeiro de 2021. A nota final do projeto diz respeito à prestação individual na defesa e está limitada pela soma ponderada das pontuações obtidas no MOOSHAK em cada uma das metas. Assim, a classificação final nunca poderá exceder a pontuação obtida no MOOSHAK acrescida da classificação do relatório final. Os programas de teste colocados por cada estudante no repositório `https://git.dei.uc.pt/rbarbosa/Comp2021/tree/master` serão contabilizados na avaliação. Aplica-se mínimos de 40% à nota final após a defesa.

2 Analisador lexical – Meta 1

Nesta primeira meta deve ser programado um analisador lexical para a linguagem deiGo. A programação deve ser feita em linguagem C utilizando a ferramenta *lex*. Os "tokens" a ser considerados pelo compilador são apresentados de seguida e deverão estar de acordo com a especificação da linguagem Go, disponível em `https://golang.org/ref/spec#Lexical_elements` e no material de apoio da disciplina.

2.1 Tokens da linguagem deiGo

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

STRLIT: uma sequência de carateres (excepto "carriage return", "newline", ou aspas duplas) e/ou "sequências de escape" entre aspas duplas. Apenas as sequências de escape \f, \n, \r, \t, \\ e \" são especificadas pela linguagem. Sequências de escape não especificadas devem dar origem a erros lexicais, como se detalha mais adiante.

INTLIT: uma sequência de dígitos que representa uma constante inteira. Existe a opção de adicionar um prefixo para especificar outra base que não a decimal: `0` para octal, `0x` ou `0X` para hexadecimal. Nesta última, as letras (`a-f`) e (`A-F`) correspondem aos valores entre 10 e 15.

REALLIT: uma parte inteira seguida de um ponto, opcionalmente seguido de uma parte fracionária e/ou de um expoente; ou um ponto seguido de uma parte fracionária, opcionalmente seguida de um expoente; ou uma parte inteira seguida de um expoente. O expoente consiste numa das letras "e" ou "E" seguida de um número opcionalmente precedido de um dos sinais "+" ou "-". Tanto a parte inteira como a parte fracionária e o número do expoente consistem em sequências de dígitos decimais.

SEMICOLON = ";"

COMMA = ","

BLANKID = "_"

ASSIGN = "="

STAR = "*"

DIV = "/"

MINUS = "-"

PLUS = "+"

EQ = "=="

GE = ">="

GT = “>”

LBRACE = “{”

LE = “<=”

LPAR = “(”

LSQ = “[”

LT = “<”

MOD = “%”

NE = “!=”

NOT = “!”

AND = “&&”

OR = “||”

RBRACE = “}”

RPAR = “)”

RSQ = “]”

PACKAGE = “package”

RETURN = “return”

ELSE = “else”

FOR = “for”

IF = “if”

VAR = “var”

INT = “int”

FLOAT32 = “float32”

BOOL = “bool”

STRING = “string”

PRINT = “fmt.Println”

PARSEINT = “strconv.Atoi”

FUNC = “func”

CMDARGS = “os.Args”

RESERVED: todas as palavras reservadas da linguagem Go não utilizadas em deiGo bem como o operador de incremento (“++”) e o operador de decremento (“−−”).

O analisador deve aceitar (e ignorar) como separador de tokens o espaço em branco (espaços, tabs e mudanças de linha), bem como comentários dos tipos `// ...` e `/* ... */`. Deve ainda detetar a existência de quaisquer erros lexicais no ficheiro de entrada, tal como se especifica mais adiante.

2.2 Programação do analisador

O analisador deverá chamar-se `gocompiler`, ler o ficheiro a processar através do *stdin* e, quando invocado com a opção `-l`, deve emitir os tokens e as mensagens de erro para o *stdout* e terminar. Na ausência de qualquer opção, deve escrever no *stdout* apenas as mensagens de erro. Por exemplo, caso o ficheiro `factorial.dgo` contenha o programa de exemplo dado anteriormente, que calcula o fatorial de números, a invocação:

```
./gocompiler -l < factorial.dgo
```

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

```
PACKAGE
ID(main)
SEMICOLON
FUNC
ID(factorial)
LPAR
ID(n)
INT
RPAR
INT
LBRACE
...
```

Figura 1: Exemplo de output do analisador lexical. O output completo está disponível em: `https://git.dei.uc.pt/rbarbosa/Comp2021/blob/master/meta1/factorial.out`

Sempre que uma categoria lexical contenha mais do que um único símbolo, o token encontrado deve ser impresso entre parêntesis logo a seguir à categoria do token, tal como exemplificado na Figura 1 para ID. Por outras palavras, as categorias ID, STRLIT, INTLIT, REALLIT e RESERVED requerem que o valor encontrado seja impresso a seguir ao nome da categoria lexical.

Em deiGo, o ";" é utilizado como terminador em muitas situações. No entanto, a linguagem permite que grande parte destes ";" sejam omitidos. Para isso, quando o programa está a ser analisado lexicalmente é emitido, de forma automática, um token `SEMICOLON` sempre que o último token de uma linha seja:

- um literal `INTLIT`, `REALLIT` ou `STRLIT`
- um `ID`
- o símbolo `RETURN`
- ou um dos operadores de pontuação `RPAR`, `RSQ` ou `RBRACE`

2.3 Tratamento de erros

Caso o ficheiro contenha erros lexicais, o programa deverá imprimir exatamente uma das seguintes mensagens no *stdout*, consoante o caso:

```
Line <num linha>, column <num coluna>: illegal character (<c>)\n
Line <num linha>, column <num coluna>: invalid octal constant (<c>)\n
Line <num linha>, column <num coluna>: unterminated comment\n
Line <num linha>, column <num coluna>: unterminated string literal\n
Line <num linha>, column <num coluna>: invalid escape sequence (<c>)\n
```

onde `<num linha>` e `<num coluna>` devem ser substituídos pelos valores correspondentes ao *início* do token que originou o erro, e `<c>` deve ser substituído por esse token. O analisador deve recuperar da ocorrência de erros lexicais a partir do *fim* desse token. Tanto as linhas como as colunas são numeradas a partir de 1.

2.4 Entrega da Meta 1

O ficheiro *lex* a entregar deverá obrigatoriamente listar os autores num comentário colocado no topo desse ficheiro, contendo o nome e o número de estudante de cada elemento do grupo. Esse ficheiro deverá chamar-se `gocompiler.l` e ser enviado num arquivo de nome `gocompiler.zip`, que não deverá ter quaisquer diretorias.

O trabalho deverá ser verificado no MOOSHAK, usando o concurso criado especificamente para o efeito e cuja página está acima indicada na Secção 1. 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 depuração. Os estudantes deverão usar e contribuir para o repositório que está disponível em `https://git.dei.uc.pt/rbarbosa/Comp2021/tree/master` contendo casos de teste.

3 Analisador sintático – Meta 2

O analisador sintático deve ser programado em C utilizando as ferramentas `lex` e `yacc`. A gramática que se segue especifica a sintaxe da linguagem deiGo.

3.1 Gramática inicial em notação EBNF

Program $\longrightarrow$ PACKAGE ID SEMICOLON Declarations

Declarations $\longrightarrow$ {VarDeclaration SEMICOLON | FuncDeclaration SEMICOLON}

VarDeclaration $\longrightarrow$ VAR VarSpec

VarDeclaration $\longrightarrow$ VAR LPAR VarSpec SEMICOLON RPAR

VarSpec $\longrightarrow$ ID {COMMA ID} Type

Type $\longrightarrow$ INT | FLOAT32 | BOOL | STRING

FuncDeclaration $\longrightarrow$ FUNC ID LPAR [Parameters] RPAR [Type] FuncBody

Parameters $\longrightarrow$ ID Type {COMMA ID Type}

FuncBody $\longrightarrow$ LBRACE VarsAndStatements RBRACE

VarsAndStatements $\longrightarrow$ VarsAndStatements [VarDeclaration | Statement] SEMICOLON | ϵ

Statement $\longrightarrow$ ID ASSIGN Expr

Statement $\longrightarrow$ LBRACE {Statement SEMICOLON} RBRACE

Statement $\longrightarrow$ IF Expr LBRACE {Statement SEMICOLON} RBRACE [ELSE LBRACE {Statement SEMICOLON} RBRACE]

Statement $\longrightarrow$ FOR [Expr] LBRACE {Statement SEMICOLON} RBRACE

Statement $\longrightarrow$ RETURN [Expr]

Statement $\longrightarrow$ FuncInvocation | ParseArgs

Statement $\longrightarrow$ PRINT LPAR (Expr | STRLIT) RPAR

ParseArgs $\longrightarrow$ ID COMMA BLANKID ASSIGN PARSEINT LPAR CMDARGS LSQ Expr RSQ RPAR

FuncInvocation $\longrightarrow$ ID LPAR [Expr {COMMA Expr}] RPAR

Expr $\longrightarrow$ Expr (OR | AND) Expr

Expr $\longrightarrow$ Expr (LT | GT | EQ | NE | LE | GE) Expr

Expr $\longrightarrow$ Expr (PLUS | MINUS | STAR | DIV | MOD) Expr

Expr $\longrightarrow$ (NOT | MINUS | PLUS) Expr

Expr $\longrightarrow$ INTLIT | REALLIT | ID | FuncInvocation | LPAR Expr RPAR

A gramática apresentada é ambígua e está escrita em notação EBNF, onde [...] significa "opcional" e {...} significa "zero ou mais repetições". Portanto, a gramática deverá ser modificada para permitir a análise sintática ascendente com o `yacc`. Será necessário ter em conta as regras

de associação dos operadores e as precedências, entre outros aspetos, de modo a garantir a compatibilidade entre as linguagens deiGo e Go.

3.2 Programação do analisador

O analisador deverá chamar-se `gocompiler`, ler o ficheiro a processar através do *stdin* e emitir todos os resultados para o `stdout`. Quando invocado com a opção `-t` deve imprimir a árvore de sintaxe tal como se especifica nas secções seguintes. Para manter a compatibilidade com a fase anterior, se o analisador for invocado com a opção `-l` deverá realizar apenas a análise lexical, emitir os tokens e as mensagens de erro para o *stdout* e terminar.

Se não for passada qualquer opção, o analisador deve apenas escrever no *stdout* as mensagens de erro correspondentes aos erros lexicais e de sintaxe.

3.3 Tratamento e recuperação de erros

Caso o ficheiro de entrada contenha erros lexicais, o programa deverá imprimir no `stdout` as mensagens já especificadas na meta 1 e continuar. Caso sejam encontrados erros de sintaxe, o analisador deve imprimir mensagens de erro com o seguinte formato:

```
Line <num linha>, column <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 escrevendo a função:

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

O analisador deve ainda incluir 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):

Statement $\longrightarrow$ error
ParseArgs $\longrightarrow$ ID COMMA BLANKID ASSIGN PARSEINT LPAR error RPAR
FuncInvocation $\longrightarrow$ ID LPAR error RPAR
Expression $\longrightarrow$ LPAR error RPAR

3.4 Árvore de sintaxe abstrata (AST)

Caso seja feita a seguinte invocação:

```
./gocompiler -t < factorial.dgo
```

deverá gerar a árvore de sintaxe abstrata correspondente e imprimi-la no `stdout` de acordo com a descrição que se segue. 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 (>=0) (<variable and/or function declarations>)`

Declaração de variáveis

`VarDecl (<typespec> Id)`

Declaração/definição de Funções

```
FuncDecl(2) (FuncHeader FuncBody)
FuncHeader(>=2) (Id [<typespec>] FuncParams)
FuncParams(>=0) (ParamDecl)
FuncBody(>=0) (<declarations> | <statements>)
ParamDecl(2) (<typespec> Id)
```

Statements

`Block(>=0)` `If(3)` `For([Expr] Block)` `Return(>=0)` `Call(>=1)` `Print(1)` `ParseArgs(2)`

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)` `Assign(2)` `Call(>=1)`

Terminais

`Int`, `Float32`, `Bool`, `String`, `IntLit`, `RealLit`, `Id`, `StrLit`

Nota: Não deverão ser gerados nós supérfluos, nomeadamente Block com menos de dois statements no seu interior, exceto para representar blocos obrigatórios em nós If e For. Os nós Program, FuncParams e FuncBody não deverão ser considerados redundantes mesmo que tenham menos de dois nós filhos.

No caso do programa `package main; func main() {};` o resultado deve ser:

```
Program
..FuncDecl
....FuncHeader
......Id(main)
......FuncParams
....FuncBody
```

Figura 2: Exemplo de output do analisador sintático.

Para o caso do programa que calcula o fatorial de um número, na primeira página, encontra-se em `https://git.dei.uc.pt/rbarbosa/Comp2021/blob/master/meta2/factorial.out` o output completo da análise sintática.

3.5 Desenvolvimento do analisador

Sugere-se que desenvolva o analisador de forma faseada. Deverá começar por re-escrever para o `yacc` a gramática acima apresentada de modo a detetar erros de sintaxe (isto é, inicialmente sem a árvore de sintaxe). Após terminada esta fase, e já garantindo que a gramática está correta, deverá focar-se no desenvolvimento do código necessário para a construção da árvore de sintaxe abstrata e a sua impressão para o `stdout`. O relatório final deverá descrever as opções tomadas na escrita da gramática, pelo que se recomenda agora a documentação dessa parte.

Para promover uma boa divisão de tarefas entre membros do grupo, sugere-se que comecem por analisar produções diferentes. Observando o não-terminal Declarations, um membro começaria por Declarations $\longrightarrow$ {VarDeclaration SEMICOLON} enquanto o outro começaria por Declarations $\longrightarrow$ {FuncDeclaration SEMICOLON}. Outra possibilidade seria um membro começar pelo topo da gramática, em Program, e o outro membro começar pela base, em Expr. Teriam de coordenar o trabalho a partir do momento em que chegassem a não-terminais comuns na gramática.

Deverá ter em atenção que toda a memória alocada durante a execução do analisador deve ser libertada antes deste terminar, devendo ter em conta as situações em que a construção da AST é interrompida por erros de sintaxe.

3.6 Entrega da Meta 2

O ficheiro *lex* entregue deverá obrigatoriamente listar os autores num comentário colocado no topo desse ficheiro, contendo o nome e o número de estudante de cada membro do grupo. Os ficheiros `lex` e `yacc` a entregar deverão chamar-se `gocompiler.l` e `gocompiler.y` e ser colocados num único arquivo com o nome `gocompiler.zip` juntamente com quaisquer outros ficheiros necessários para compilar o analisador.

O trabalho deverá ser avaliado no MOOSHAK, usando o concurso criado especificamente para o efeito e cuja página está acima indicada na Secção 1. Para efeitos de avaliação, 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. No entanto, o MOOSHAK não deve ser utilizado como ferramenta de depuração. Os estudantes deverão usar e contribuir para o repositório disponível em `https://git.dei.uc.pt/rbarbosa/Comp2021/tree/master` contendo casos de teste.

4 Analisador semântico – Meta 3

O analisador semântico deve ser programado em C tendo por base o analisador sintático desenvolvido na meta anterior com as ferramentas `lex` e `yacc`. O analisador deverá chamar-se `gocompiler`, ler o ficheiro a processar através do `stdin`, e detetar a ocorrência de quaisquer erros (lexicais, sintáticos ou semânticos) no ficheiro de entrada. Considere a invocação

```
./gocompiler < factorial.dgo
```

deverá levar o analisador a proceder à análise lexical e sintática do programa, e apenas caso este seja válido, proceder à análise semântica. No caso em que não é passada qualquer opção, o analisador deve apenas escrever no *stdout* as mensagens de erro.

Por uma questão de compatibilidade com a fase anterior, se o analisador for invocado com a opção `-t`, deverá realizar *apenas* a análise sintática, e 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 à análise semântica.

Sendo o programa sintaticamente válido, a invocação

```
./gocompiler -s < factorial.dgo
```

deve fazer com que o analisador imprimia 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, como a seguir se especifica.

4.1 Tabelas de símbolos

Durante a análise semântica, deve ser construída uma tabela de símbolos global, 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. Note que caso uma função não tenha tipo de retorno, deverá à mesma ser inserida na tabela com o tipo de dados `none`.

Para o programa de exemplo dado, as tabelas de símbolos a imprimir são as que se seguem. O formato das linhas é "`Name\t[ParamTypes]\tType[\tparam]`", sendo que `[]` denota uma componente opcional, `\t` denota um espaçamento TAB, `Name` denota o nome do símbolo, `ParamTypes` denota os tipos dos parâmetros de uma função, que devem ser impressos entre parêntesis e separados por uma vírgula, `Type` denota o tipo da variável ou o tipo de retorno da função, e `param` é uma *keyword* que deve ser impressa se a variável em causa for um parâmetro da função.

```
===== Global Symbol Table =====
factorial       (int)   int
main    ()      none

===== Function factorial(int) Symbol Table =====
return          int
n               int     param

===== Function main() Symbol Table =====
return          none
argument                int
```

Os símbolos (e as tabelas) devem ser apresentados por ordem de declaração no programa fonte. No essencial, a notação para os tipos segue as convenções de Go. Deve ser deixada uma linha em branco entre tabelas consecutivas, e entre as tabelas e a árvore de sintaxe abstrata anotada.

4.2 Árvore de sintaxe 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 que se apresenta na Figura 3.

```
Program
..FuncDecl
....FuncHeader
......Id(factorial)
......Int
......FuncParams
........ParamDecl
..........Int
..........Id(n)
....FuncBody
......If
........Eq - bool
..........Id(n) - int
..........IntLit(0) - int
........Block
..........Return
............IntLit(1) - int
........Block
......Return
........Mul - int
..........Id(n) - int
..........Call - int
............Id(factorial) - (int)
............Sub - int
..............Id(n) - int
..............IntLit(1) - int
...
```

Figura 3: Exemplo de output do analisador semântico. O resultado completo está disponível em: `https://git.dei.uc.pt/rbarbosa/Comp2021/blob/master/meta3/factorial.out`

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

4.3 Tratamento de erros semânticos

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

`Symbol <token> already defined`

```
Cannot find symbol <token>
Operator <token> cannot be applied to type <type>
Operator <token> cannot be applied to types <type>, <type>
Incompatible type <type> in <token> statement
Symbol <token> declared but never used
```

Caso seja detetado algum erro durante a análise semântica do programa, **o analisador deverá imprimir a mensagem de erro apropriada e continuar, atribuindo 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. 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 caractere 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 (considerar que o tipo esperado pelas condições das construções `if` e `for` é `bool`),
- o identificador da função invocada quando o número de parâmetros estiver errado,
- o operando ou constante que dá origem ao erro.

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.

4.4 Programação 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.

4.5 Entrega da Meta 3

O trabalho deverá ser avaliado no MOOSHAK, usando o concurso criado especificamente para o efeito e cuja página está acima indicada na Secção 1. Para efeitos de avaliação, 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 semântica e à construção da árvore de sintaxe abstrata anotada. No entanto, o MOOSHAK não deve ser utilizado como ferramenta de depuração. Os estudantes deverão usar e contribuir para o repositório disponível em `https://git.dei.uc.pt/rbarbosa/Comp2021/tree/master` contendo casos de teste.

O ficheiro *lex* entregue deverá obrigatoriamente listar os autores num comentário colocado no topo desse ficheiro, contendo o nome e o número de estudante de cada membro do grupo. Os ficheiros `lex` e `yacc` a entregar deverão chamar-se `gocompiler.l` e `gocompiler.y` e ser colocados num único arquivo com o nome `gocompiler.zip` juntamente com quaisquer outros ficheiros necessários para compilar o analisador.

5 Gerador de código intermédio – Meta 4

O gerador de código intermédio deve ser programado em C utilizando as ferramentas `lex` e `yacc` a partir do código desenvolvido nas metas anteriores. Deverá chamar-se `gocompiler`, como anteriormente, ler o programa a compilar do `stdin`, e emitir para o `stdout` um programa na representação intermédia do LLVM (v7) que tenha a mesma funcionalidade que o programa de entrada. Por exemplo, a invocação:

```
./gocompiler < factorial.dgo > factorial.ll
```

deverá processar e analisar o programa `factorial.dgo` e escrever o código IR LLVM correspondente no ficheiro `factorial.ll`. Este poderá ser executado diretamente na linha de comandos:

```
lli factorial.ll 7
```

ou compilado e ligado com:

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

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

```
./factorial 7
```

Neste caso, ao executar o programa `factorial.dgo` com o argumento `7` deverá ser impresso no ecrã o seguinte:

```
5040
```

Para efeitos de verificação, o compilador deve fornecer ainda as seguintes opções especificadas nas metas anteriores:

- `-l` : executa a análise lexical, reportando os tokens encontrados e eventuais erros lexicais, e termina.
- `-t` : executa a análise sintática, reportando eventuais erros lexicais ou sintáticos, imprime a árvore de sintaxe abstrata construída durante a análise sintática do programa (se não houver erros sintáticos) e termina.
- `-s` : executa a análise semântica (se não houver erros sintáticos), reportando eventuais erros semânticos, imprime o conteúdo da(s) tabela(s) de símbolos e a árvore de sintaxe abstrata anotada e termina.

Só deverá ser gerado código LLVM IR caso não haja erros de qualquer tipo nem sejam passadas quaisquer opções na linha de comandos.

5.1 Programação do gerador de código

Os tipos de dados `bool`, `int` e `float32` da linguagem deiGo deverão ser codificados através dos tipos `i1`, `i32` e `double` da representação intermédia LLVM. Valores do tipo `bool` devem ser impressos pela função `fmt.Println(...)` como `true` e `false`, e valores dos tipos `int` e

`float32` devem ser impressos nos formatos `"%d\n"` e `".08f\n"` usando a função `printf(...)` da linguagem C. As cadeias de carateres (STRLITs) deverão ser impressas no formato `"%s\n"` (tipo `i8*` do LLVM). A conversão de strings para inteiros com a função `strconv.Atoi(...)` deverá ser feita usando a função `atoi(...)` da linguagem C.

5.2 Entrega da Meta 4

O trabalho deverá ser avaliado no MOOSHAK, usando o concurso criado especificamente para o efeito e cuja página está acima indicada na Secção 1. Para efeitos de avaliação, 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 depuração. Os estudantes deverão usar e contribuir para o repositório disponível em `https://git.dei.uc.pt/rbarbosa/Comp2021/tree/master` contendo casos de teste.

O ficheiro *lex* entregue deverá obrigatoriamente listar os autores num comentário colocado no topo desse ficheiro, contendo o nome e o número de estudante de cada membro do grupo. Os ficheiros `lex` e `yacc` a entregar deverão chamar-se `gocompiler.l` e `gocompiler.y` e ser colocados num único arquivo com o nome `gocompiler.zip` juntamente com quaisquer outros ficheiros necessários para compilar o analisador.

6 Entrega final e relatório

A entrega final do projeto será feita no Inforestudante até ao dia seguinte ao da Meta 4, e deve incluir todo o código-fonte produzido no âmbito do projeto: precisamente os quatro arquivos `.zip` que tiverem sido apresentados no MOOSHAK em cada meta. Os ficheiros `.zip` correspondentes a cada submissão devem chamar-se `1.zip`, `2.zip`, `3.zip`, `4.zip`, para as submissões às Metas 1, 2, 3 e 4, respetivamente.

Em todas as entregas no MOOSHAK o ficheiro `gocompiler.l` deve identificar os autores num comentário acrescentado ao topo do ficheiro. Sem a identificação dos autores de cada trabalho não será possível atribuir a respetiva classificação.

O relatório final terá três secções limitadas a 1200 palavras (400 palavras por cada secção), sendo que deverá documentar concisamente as opções técnicas relativas

(i) à gramática re-escrita,

(ii) aos algoritmos e estruturas de dados da AST e da tabela de símbolos, e

(iii) à geração de código.