Projeto de Compiladores 2024/25

Compilador para a linguagem deiGo

17 de outubro de 2024

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%) – 17 de outubro de 2024
2. Análise sintática (25%) – 10 de novembro de 2024 (meta de avaliação)
3. Análise semântica (25%) – 21 de novembro de 2024
4. Geração de código (25%) – 14 de dezembro de 2024 (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 "uc2021123456_uc2021654321" usando os números de estudante como identificação do grupo na página `https://mooshak.dei.uc.pt/~comp2024` na qual o MOOSHAK está acessível. Será tidaem conta apenas a última versão apresentada ao problema A de cada concurso do MOOSHAK para efeitos de avaliação.

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 8 de janeiro de 2025. 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/Comp/tree/master/go` 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

IDENTIFIER: 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.

NATURAL: 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.

DECIMAL: 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
IDENTIFIER(main)
SEMICOLON
FUNC
IDENTIFIER(factorial)
LPAR
IDENTIFIER(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/Comp/blob/master/go/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 IDENTIFIER. Por outras palavras, as categorias IDENTIFIER, STRLIT, NATURAL, DECIMAL 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 `NATURAL`, `DECIMAL` ou `STRLIT`
- um `IDENTIFIER`
- 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/Comp/tree/master/go` 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 IDENTIFIER SEMICOLON Declarations

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

VarDeclaration $\longrightarrow$ VAR VarSpec

VarDeclaration $\longrightarrow$ VAR LPAR VarSpec SEMICOLON RPAR

VarSpec $\longrightarrow$ IDENTIFIER {COMMA IDENTIFIER} Type

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

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

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

FuncBody $\longrightarrow$ LBRACE VarsAndStatements RBRACE

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

Statement $\longrightarrow$ IDENTIFIER 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 $\longrightarrow$ IDENTIFIER COMMA BLANKID ASSIGN PARSEINT LPAR CMDARGS LSQ Expr RSQ RPAR

FuncInvocation $\longrightarrow$ IDENTIFIER 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 $\longrightarrow$ NATURAL | DECIMAL | IDENTIFIER | 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 ⟶ error
ParseArgs ⟶ IDENTIFIER COMMA BLANKID ASSIGN PARSEINT LPAR error RPAR
FuncInvocation ⟶ IDENTIFIER LPAR error RPAR
Expression ⟶ 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`, `Natural`, `Decimal`, `Identifier`, `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
......Identifier(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/Comp/blob/master/go/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/Comp/tree/master` contendo casos de teste.