Projeto de Compiladores 2018/19

Compilador para a linguagem deiGo

14 de fevereiro de 2019

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.

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%) – 4 de março de 2019
2. Análise sintática (25%) – 29 de março de 2019 (meta de avaliação)
3. Análise semântica (25%) – 29 de abril de 2019
4. Geração de código (25%) – 24 de maio de 2019 (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 `http://mooshak2.dei.uc.pt/~comp2019` 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 27 de maio e 7 de junho de 2019. 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/Comp2019/tree/master` serão contabilizados na avaliação. Aplica-se mínimos de 40% à nota final após a defesa.

2 Meta 1 – Analisador lexical

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.

INTLIT: é 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.

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 definidas pela linguagem. Sequências de escape não definidas devem dar origem a erros lexicais, como se detalha mais adiante.

SEMICOLON = ";"

BLANKID = "_"

PACKAGE = "package"

RETURN = "return"

AND = "&&"

ASSIGN = "="

STAR = "*"

COMMA = ","

DIV = "/"

EQ = "=="

GE = ">="

GT = ">"

LBRACE = "{"

LE = "<="

LPAR = "("

LSQ = "["

LT = “<”

MINUS = “-”

MOD = “%”

NE = “!=”

NOT = “!”

OR = “||”

PLUS = “+”

RBRACE = “}”

RPAR = “)”

RSQ = “]”

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

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/Comp2019/blob/master/meta1/factorial.out`

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. 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 exemplificado na figura acima para ID.

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 `ID`
- um literal `INTLIT`, `REALLIT` ou `STRLIT`
- 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>: unterminated comment\n
Line <num linha>, column <num coluna>: illegal character (<c>)\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>` devem ser substituídos 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/Comp2019/tree/master` contendo casos de teste.