

COMPILADORES

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

UNIVERSIDADE DE COIMBRA

Compilador para a linguagem deiGo



Autores:

João Costa

Hipólito Lopes

Professores:

Prof. Raúl Barbosa

Prof. João Fernandes

Contents

1	Introdução	2
2	Meta 1 - Análise lexical	3
3	Meta 2 - Análise Sintática	4
4	Meta 3 - Análise Semântica	6

1 Introdução

Este projeto tem como objetivo o desenvolvimento de um compilador para a linguagem deiGO. Para tal foi necessária a instalação e aprendizagem de ferramentas como o Lex e o Yacc.

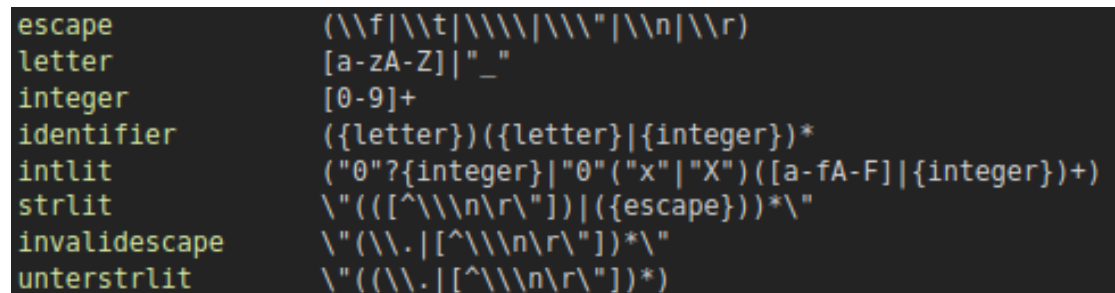
O Lex é uma ferramenta de análise lexical que tem como objectivo receber código como input e transformá-lo em tokens, de modo a serem utilizados mais tarde para a análise sintática. Caso estes tokens não estejam definidos pela linguagem o programa transmite errors lexicais.

O Yacc, por sua vez, complementa o Lex na medida em que é capaz de definir uma gramática usando os tokens devolvidos pelo Lex. O mesmo, pode emitir erros sintáticos, caso a gramática não seja cumprida segundo as regras definidas.

As ferramentas anteriormente apresentadas foram usadas ao longo do projeto para cumprimento das metas 1 (análise lexical), meta 2(análise sintática) e meta 3 (análise semântica). .

2 Meta 1 - Análise lexical

No ficheiro `gocompiler.l`, utilizando a ferramenta `lex`, foram criados os diversos tokens especificados na linguagem `deiGo`. Nas regras, descreve-se um conjunto de padrões e são definidas as acções associadas a cada padrão. Estes padrões são descritos em expressões regulares. Na imagem a baixo estão apresentadas algumas das regras definidas.



```
escape      (\\f|\\t|\\\\|\\\\\\\"|\\\\n|\\\\r)
letter      [a-zA-Z]|\"_\"
integer     [0-9]+
identifier   ({letter})({letter}|{integer})*
intlitt     (\"0\"?{integer}|\"0\"(\"x\"|\"X\")([a-fA-F]|{integer}))+
strlitt     \\\"([\\^\\\\\\\\n\\\\r\\\\\"]|({escape}))*\\\\\"
invalidescape  \\\"(\\\\\\\\.|[\\^\\\\\\\\n\\\\r\\\\\"])*\\\\\"
unterstrlitt \\\"(\\\\\\\\.|[\\^\\\\\\\\n\\\\r\\\\\"])*\"
```

Figure 1: Gramática

Para este projeto existem 4 tipos de erros lexicais diferentes. Os erros definem-se como:

- **invalid escape sequence**, quando existem escape sequences numa string que sejam diferentes das que estão especificadas na Figura 1, com o nome "escape". Para dar print do erro, percorre-se a string à procura de invalid escape sequences. Assim que se encontra uma destas, imprime-se o erro onde está identificada a escape sequence e sua linha e coluna;
- **unterminated strlit**, significa que existe uma string onde não é fechada a última aspa. Esta está especificada na gramática da Figura 1 com o nome "unterstrlit";
- **unterminated comment**, este ocorre quando um comentário inicializado com o símbolo `"/*`, não termina com o símbolo `*/`. Para resolver este erro foi usada a opção `%x COMMENTS` e o seguinte código:

```


/*
{BEGIN COMMENTS; tempColumn = column; tempLine = yylineno; column+=yyleng;}
<COMMENTS>\n
{column = 1;}
<COMMENTS>.
{column+=yyleng;}
<COMMENTS>"/"
{column+=yyleng; BEGIN 0;}
<COMMENTS><<EOF>>
{printf("Line %d, column %d: unterminated comment\n", tempLine, tempColumn); column+=yyleng; return 0;}


```

Figure 2: Erros Comentários

Ao encontrar um `"/` o código é percorrido até encontrar o símbolo terminal `"/`. Se este não for encontrado, é dado o print do erro, juntamente com a linha e coluna do início do comentário, como identificado na Figura 2;

- **illegal character**, este ocorre quando há um pedaço de código onde não existem tokens correspondentes. Este erro é colocado no final da gramática com o identificador `""`, significando que vai entrar tudo o que não está especificado acima deste.

3 Meta 2 - Análise Sintática

Nesta secção é descrita detalhadamente a estrutura da gramática utilizada usando a ferramenta lex e yacc. O primeiro passo foi devolver os tokens no ficheiro `go-compiler.l`, para estes serem utilizados na análise sintática. Assim, quando o lex encontra uma variável, poderá enviar ao yacc o respectivo identificador. Foi usada a variável `yyval` em alguns tokens (`Id`, `Intlit`, `Strlit`, `RealLit`), para enviar o valor associado aos tokens, necessários para a construção da árvore de sintaxe. Na linguagem C, existe uma estrutura apropriada a este efeito, a `union`. A `union` permite partilhar, no mesmo espaço de memória, vários tipos diferentes. Na nossa `union` foram colocadas as seguintes variáveis:

```


%union{
    char* strToken;
    struct node *node;
}


```

Figure 3: Union

O `"char * strToken"` diz respeito aos valores enviados pelo `yyval` dos tokens já descritos em cima. E a `"struct node *node"` está associado aos nós que vão ser criados para a construção da árvore. Esta estrutura resume-se ao seguinte:

```
typedef struct node{
    char *id;
    char *type;
    struct node *brother;
    struct node *son;
}node;
```

Figure 4: Estrutura node

A cada nó está associado o tipo dele, ou seja, o que ele representa segundo a gramática; o id, que é o valor do token se este for um Id, Intlit, Strlit ou RealLit. Cada nó tem também na sua estrutura dois ponteiros para possíveis nós son e brother. Retiremos um excerto da gramática para explicar o seu funcionamento:

Program: PACKAGE ID SEMICOLON Declarations	{head = create("Program",NULL,0,0); createSon(head,\$4);}
;	
Declarations:%empty	{\$\$ = create("NULL",NULL,0,0);}
VarDeclaration SEMICOLON Declarations	{\$\$ = \$1; createBrother(\$1,\$3);}
FuncDeclaration SEMICOLON Declarations	{\$\$ = \$1; createBrother(\$1,\$3);}
;	
VarDeclaration:VAR VarSpec	{\$\$ = \$2;}
VAR LPAR VarSpec SEMICOLON RPAR	{\$\$ = \$3;}
;	
VarSpec:ID auxVarSpec Type	{\$\$ = create("VarDecl",NULL,0, 0);
	aux = create("Id",\$1->id, \$1->line, \$1->column);
	createSon(\$\$, \$3);
	createBrother(\$\$,aux);
	createBrother(\$\$, \$2);
	if (brother == 1)
	createTypes(\$\$);
	brother=0;
	}
;	

Figure 5: Gramática

A árvore de sintaxe é gerada de baixo para cima. Quando existe mais do que um elemento do mesmo tipo a gramática é recursiva à direita. Esta foi feita de maneira a não existirem ambiguidades, o que significa que a cada momento só existe um caminho possível. A função create() vai criar um nó com os parâmetros especificados na estrutura. A createBrother() recebe dois nós como parâmetros, adicionando ao primeiro nó um ponteiro *brother para o segundo. A função createSon() faz o mesmo que a anterior mas adiciona um ponteiro *son em vez do *brother.

Se nas regras existem acções que referenciem o topo da pilha (\$\$), implica que é obrigatório definir o tipo com a declaração type, que no nosso caso é "%type<node>" (mesmo tipo dos nós criados).

Em relação aos erros, estes são dados sempre que algum pedaço de código esteja em falta ou a mais de acordo com a gramática definida. No entanto esta permite recuperação do programa caso existam apenas certos erros.

```

FuncInvocation:ID LPAR RPAR                                {$$=create("Call",NULL,0, 0);aux = create("I
                |ID LPAR Expr auxFuncInvocation RPAR      {$$=create("Call",NULL,0, 0);aux = create("I
                |ID LPAR error RPAR                        {$$=create("NULL", NULL,0,0);}
                ;

```

Figure 6: Recuperação de erros

Por exemplo, no caso da Figura 8 quando ocorre algo não especificado na gramática dentro de "error" é criado um nó do tipo "NULL", e assim ao percorrer a árvore todos os nós do tipo "NULL" são saltados e o programa continua a sua execução.

4 Meta 3 - Análise Semântica

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, para isto foram criadas novas estruturas as representar, entre as quais:

```

typedef struct global{
    struct varDecl *vardecl;
    struct funcDecl *funcdecl;
}global;

```

Figure 7: Estrutura global

Na estrutura acima estão representados ponteiros para declarações globais e funções, que vai conter a totalidade das tabelas de um dado programa.

```

typedef struct varDecl{
    char *name;
    char *type;
    int line;
    int column;
    int order;
    int check;
    struct varDecl *next;
}varDecl;

```

Figure 8: Estrutura varDecl

Acima está apresentada a estrutura varDecl, que vai conter as variáveis globais para a tabela global e as variáveis de cada função para a tabela da própria. Para esta vão ser precisos guardar o tipo e nome da variáveis declaradas, assim como a linha e coluna para mais tarde serem usados para a impressão de erros.

```
typedef struct funcDecl{
    char * name;
    char * type;
    struct parameters *params;
    struct funcDecl *next;
    struct varDecl *vardecl;
    int order;
}funcDecl;
```

Figure 9: Estrutura funcDecl

Para as tabelas das funções foi criada uma estrutura funcDecl, onde são guardados o nome, o tipo, a lista parâmetros e variáveis locais desta, necessários para a impressão da tabela.

```
typedef struct parameters{
    char *type;
    char *name;
    struct parameters *next;
}parameters;
```

Figure 10: Estrutura parameters

Esta estrutura diz respeito aos parâmetros de cada função, onde são guardados os seus tipos e nomes.

Devido maioritariamente á necessidade da impressão de erros durante o desenvolvimento desta meta, foram realizadas algumas mudanças na estrutura node e nos ficheiros lex e yacc, para poder ter acesso á linha e coluna de alguns tokens.

```
typedef struct node{
    char *id;
    char *type;
    char *anote;
    int line;
    int column;
    int here;
    int skip;
    struct node *brother;
    struct node *son;
}node;
```

Figure 11: Nova estrutura node

Foi adicionado também, á estrutura do nó, uma string anote, que é utilizado para a geração da árvore de sintaxe anotada, uma flag skip para mais tarde irá envitar conflitos entre funções com o mesmo id.

Para fazer as tabelas de simbolos, foram percorridos os nós da árvore e adicionados os elementos para a construção das mesmas. No fim de obter as tabelas, ficamos com os dados necessários para a criação da árvore anotada.

Para anotar a árvore, esta é percorrida até encontrar nós que correspondam a expressões. Para isto é também usada a tabela como auxilio para a anotação. Depois de encontrar as expressões são anotadas os filhos de forma recursiva de baixo para cima.

O tratamento de erros foi realizado correspondente ao que estava no enunciado. Apenas um dos erros é encontrado durante a construção das tabelas, que é

- **Symbol <token> already defined**, quando é encontrada uma variável ou função que já foi adicionada á tabela. Ao imprimir este erro, o token em referência não é adicionado á tabela.

Todos os restantes foram tratados durante a realização da árvore anota, sendo eles:

- **Cannot find symbol <token>**, usado quando existem ids em expressões ou chamadas de funções na árvore que não foram definidas na tabela de simbolos;
- **Operator <token> cannot be applied to type <type>**, usado apenas nos operadores que têm apenas um filho, quando o tipo deste não é compatível com o operador;
- **Operator <token> cannot be applied to types <type> , <type>**, é utilizado em operadores com dois filhos quando estes não são iguais ou quando não são aceites pelo operador;
- **Incompatible type <type> in <token> statement**, quando nos statements for ou if, o filho destes não é do tipo bool;
- **Invalid octal constant : <token>**, este erro é causado por um intLit com prefixo 0 se este contiver um 8 ou um 9;
- **Symbol <token> declared but never used**, quando uma variável local é declarada e não é utilizada dentro da sua função;