FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D COIMBRA

Relatório Trabalho Prático

Compilador para a linguagem deiGo

Compiladores

2021/2022

Maria da Conceição Silva Dias, 2018274188, mddias@student.dei.uc.pt
Simão Carvalho Monteiro, 2019215412, uc2019215412@student.uc.pt

1. Reescrita da gramática

Na primeira fase da construção do analisador começamos por identificar os tokens da linguagem a serem considerados. De seguida, iniciámos a reescrita da gramática, à qual foram aplicadas algumas alterações de modo a resolver as ambiguidades presentes.

Por exemplo, na produção:

$$Declarations \rightarrow \{VarDeclaration\ SEMICOLON \mid FuncDeclaration\ SEMICOLON\}$$

Foi aplicada recursividade à direita, uma vez que a produção pode repetir-se zero ou mais vezes. Por outro lado, foram criadas duas produções distintas uma vez que temos tokens opcionais. Para as restantes produções, foi aplicada uma lógica semelhante.

Por outro lado, foi também criada uma produção auxiliar denominada $StatementList1$, que em nada difere da produção $StatementList$ mas que nos permite identificar de modo mais simples quais os locais onde temos um $Block$.

Por fim, foram aplicadas as precedências definidas para a linguagem Go, presentes nas especificações da linguagem. Um caso "exceção" a estas precedências está presente no uso dos operadores unários, que nesse caso possuem prioridade superior à divisão, multiplicação e divisão inteira.

2. Algoritmos e estruturas de dados da AST e tabela de símbolos

Nesta fase de construção da AST e das tabelas de símbolos foram usadas algumas estruturas auxiliares, entre elas, as apresentadas de seguida.

$$typedef\ struct\ \_28\ \{char\ *\ id;\ int\ line;\ int\ column;\}\ id\_token;$$

$$typedef\ struct\ \_t4\{\ bool\ seen;\ char\ *\ id;\ char\ *\ type;\ struct\ \_t4\ *\ next;\}\ t\_params;$$

$$typedef\ struct\ \_t3\{\ bool\ seen;\ char\ *\ id;\ char\ *\ type;\ t\_params\ *\ params;\}\ t\_funcdec;$$

$$typedef\ struct\ \_t2\{\ int\ line;\ int\ column;\ bool\ seen;\ bool\ used;\ char\ *\ id;\ char\ *\ type;\ bool\ param;\}\ t\_id;$$

$typedef\ struct\ \_t1\{\ t\_id\ *\ id;\ t\_funcdec\ *\ funcdec;\ struct\ \_t1\ *\ next;\}\ table\_element;$

A estrutura id_token foi necessária para guardar as linhas, colunas e o valor do token enviados pelo lex que nos permitem ao fazer a impressão dos erros dar print à coluna e linha onde o erro ocorre.

A estrutura t_params foi usada para guardar os parâmetros passados na declaração de uma função e possui uma variável denominada seen que nos permite identificar se uma determinada variável foi vista ou não.

A estrutura $t\_funcdec$ permite guardar o nome, tipo e parâmetros de uma função. Possui à semelhança da estrutura anterior uma variável que permite identificar se a função foi vista ou não.

À semelhança da estrutura anterior, a estrutura t_id permite guardar o nome de uma variável, o tipo e verificar se a variável foi vista. Permite também verificar se a variável foi acedida, se é um parâmetro de uma função e saber qual a linha e coluna em que a variável foi declarada.

Por fim, a estrutura $table\_element$ permite guardar os elementos da tabela de símbolos.

3. Geração de código

Nesta última fase da construção do analisador, para garantir que o código intermédio seja gerado corretamente foram definidos os seguintes passos:

1. Declarar inicialmente a função printf e atoi para o seu uso posterior;
2. Declarar strings auxiliares, explicadas no enunciado, como *"%d\n"* para o tipo *int* , *".08f\n"* para o tipo *float*, *"%s\n"* para o tipo *string* e por fim *“false\n”* e *“true\n”* para o tipo *bool*;
3. Definir a função global fmt.Println em LLVM para imprimir variáveis do tipo bool;
4. Percorrer a tabela de símbolos global para declarar funções e variáveis globais;

5. Percorrer AST, inicializar contador de variáveis a 0 e gerar código de cada função.

Gerar Código de Funções

Depois de definidas estas funções, irá ser gerado o código para o corpo da função da seguinte forma:

- **VarDecl** - Usa-se a instrução *"alloca"* e guarda-se o número do registo na tabela de símbolos da variável;
- **Return** - É dado return usando a instrução *"ret"* sempre do último registo usado, ou seja, o último registo irá ter sempre o valor da variável a retornar;
- **FuncInvocation** - (Não implementado totalmente) Teria de ser usada a instrução *"call"* para chamar uma função com os seus argumentos e é guardado num registo o return dessa mesma função ;
- **ParseArgs** - (Não está implementado) Teria de se verificar qual argumento estaria a ser requerido e fazer *"load"* do mesmo e chamar a função atoi sobre o registro para converter para um inteiro.
- **Print** - (Não está implementado); Poderíamos fazer uma instrução "*call*" à função printf usando uma das strings de formatação definidas no início da geração do código dependente do tipo que seja para imprimir e caso seja do tipo bool chama-se a função *fmt.Prinln* definida no início da geração;
- **If** - (Não implementado totalmente) É usada a seguinte estrutura:

  br i1 <Valor Expr>: <True>%d, <False>%d
  True:%d:
  <Código If>
  False:%d:
  <Código Else>
  End:%d:
  <Código Continuação>

Caso o valor da expressão seja verdadeiro passa para o bloco *True* e de seguida para o *End*, caso seja falso passa para o *False* e de seguida para o *End*;

- **For** - (Não está implementado); Teria de ser usada a seguinte estrutura:

```
br <For>%d
For:%d:
br i1 <Valor Expr>: <Begin>%d, End%d
Begin:%d:
<Código For>
br For%
End:%d:
<Código Continuação>
```

Verifica-se a expressão na *label For* e enquanto esta for verdadeira passa-se para a *label Begin* executando o código neste bloco e voltando para a *label For*, quando a expressão for falsa passa para a *label End* e continua a execução do código normalmente;

- **Operators** - A instrução utilizada para cada um difere entre as seguintes: *"add"*, *"sub"*, *"mul"*, *"sdiv"*, *"srem"* para os diferentes cálculos e *"eq"*, *"ne"*, *"sge"*, *"sgt"*, *"sle"*, *"slt"* para as comparações. O resultado das mesmas é guardado num registo;
- **Literals** - É usada a instrução "*store*" para guardar o valor do literal no registro previamente alocado para a variável em questão;
- **Id** - É usada a instrução "*load*" para guardar o valor de uma variável previamente colocada num registo noutro registo;