Licenciatura em Engenharia Informática
Ano letivo 2021/2022
Professor: Raul Barbosa

Adriana Bernardo 2019218086
Pedro Henriques 2019217793

UNIVERSIDADE DE COIMBRA

Relatório final - Compilador de Linguagem Dei GO

Gramática re-escrita - Meta 2

A nível da gramática re-escrita seguimos o esquema gramatical providenciado no enunciado da meta 2.

Começámos por transformar as produções gramaticais dadas, em produções lineares à esquerda, isto é, os símbolos terminais do lado esquerdo da produção. Tivemos ainda de adaptar as produções que continham componentes opcionais [...], ou componentes com 0 ou mais repetições {...}. Para o primeiro caso, optamos, na maioria das vezes, por adicionar novas produções ao próprio símbolo não terminal, já para o segundo caso, desdobramos a transformação em duas, com o uso de símbolos auxiliares. Como no caso do símbolo "*varSpec*", que deu origem aos símbolos "*varSpec*" e "*commaID*".

Neste ponto, conseguimos reduzir muitos dos conflitos "shift/reduce" existentes até então, no entanto faltava verificar as precedências e associatividade das mesmas, para isto e tendo ido consultar a documentação enumeramos as regras de precedência de GO. E, posteriormente, identificamos um problema com os operadores "!", "-" e "+": Tanto o operador "-" como o operador "+" são apenas prioritários ao "!", quando aplicados a duas expressões. Para resolver este problema utilizámos o operador "*%prec*" para igualar as precedências nos restantes casos.

Concluindo este passo, passámos à construção da árvore AST não anotada.

Licenciatura em Engenharia Informática
Ano letivo 2021/2022
Professor: Raul Barbosa

Adriana Bernardo 2019218086
Pedro Henriques 2019217793

UNIVERSIDADE DE COIMBRA

Algoritmos e estruturas de dados da AST e da tabela de símbolos - Meta 2 e 3

Começámos por criar uma estrutura de dados para cada nó da árvore AST, assim, cada estrutura que contém:

- ➔ A classe do nó, sob a forma de string (“VarDecl”, “RealLit”,...);
- ➔ O valor do nó, se aplicável (1, 3, “a”, …);
- ➔ O tipo de nó, se aplicável (int, float32, string, …);
- ➔ A linha e a coluna dos identificadores;
- ➔ Ponteiros para os nós filho e irmão.

Para adicionar nós na árvore, utilizamos o YACC, que, ao ser um parser LALR(1), permite ter uma abordagem bottom->up e definir os filhos/irmãos de cada nó antes de lhes ter sido feita a análise sintática (parse), usámos esta possibilidade nas funções “*new_node*” e “*new_brother*”.

Passámos, então, à criação das tabelas de símbolos, útil para a verificação de erros e anotação de tipos na árvore AST.

De forma a proceder à criação das tabelas criámos uma estrutura de tabela, que contém:

- O tipo (tabela global ou de função);
- O identificador (no caso de ser função);
- Uma lista ligada de parâmetros, que contêm:
  - O identificador;
  - O tipo.
- Uma lista de elementos, que contém, para cada um:
  - O identificador;
  - O tipo;
  - A linha e coluna do identificador;
  - Um booleano que identifica se o elemento é um parâmetro da função;
  - Um booleano que identifica se o elemento é uma função;
  - Um booleano que identifica se o elemento é usado (para verificação de erros semânticos);
  - Um booleano que identifica se o elemento é declarado (para verificação de erros semânticos);

Depois da criação das tabelas, percorremos a árvore em pré-ordem para efetuar, em simultâneo, a verificação de erros semânticos e anotação de tipos na AST.

Criámos, também, funções para imprimir o conteúdo da árvore, anotada ou não, percorrendo a mesma em pré-ordem, partindo do nó raiz.

Adriana Bernardo 2019218086
Pedro Henriques 2019217793

Geração de código - Meta 4

Para efetuar a geração de código LLVM IR, percorremos, da mesma forma da verificação semântica a AST.

Começámos por criar duas estruturas:

- Uma para guardar os registos de cada função, ordenados de 0 a n, que contém:
  - O número do registo;
  - Um identificador;
  - Um tipo;
  - Um booleano que identifica se um determinado registo é constante*;

- Uma lista ligada com informação de strings constantes usadas (StrLits), bem como as strings da função "*printf*" usadas para impressão de inteiros, floats contidos no enunciado.
  Para a impressão de booleanos como "*true*" ou "*false*" utilizamos duas strings constantes, definidas caso necessário e a instrução "*select*" do LLVM, para escolher entre uma ou outra consoante o valor do booleano.

Para proceder à implementação de funções, utilizámos a função para emitir código LLVM a partir de código da linguagem C do Clang, para assim, conhecer as funções a utilizar. Começámos por códigos simples como declarações de variáveis, prints, etc…; tendo posteriormente testado funções mais complexas.

Sobre controlo de fluxo (*if e for*), que não conseguimos implementar, pretendíamos usar a funcionalidade "label" do LLVM, para, por exemplo, efetuar as execuções dinâmicas e as incrementações às variáveis.

Acerca da função Atoi, não implementámos por impossibilidade de testar a mesma, isto porque, na gramática definida, não conseguimos atribuir a uma variável uma string constante (em assign), nem colocar uma string diretamente na função Atoi.

*****Como forma de optimização, reutilizámos os registros constantes (IntLit, RealLit), de forma a "poupar", caso seja necessário utilizar a mesma constante.