FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D COIMBRA

Compiladores

Relatório do Projeto de Compiladores

2020/2021

Licenciatura em Engenharia Informática

Duarte Dias	2018293526	duartedias@student.dei.uc.pt
Miguel Rabuge	2018293728	rabuge@student.dei.uc.pt

I. Gramática Re-escrita

- Primeiramente, a estratégia foi transformar as produções do enunciado cujo lado direito continha “{...}” em produções auxiliares que representam o efeito cíclico das chavetas.
- De modo semelhante, todos os “... | ... ” foram transformados em produções distintas. Ex: S ⇒ a(A|B)b ⇐⇒ S ⇒ aAb e S ⇒ aBb
- Por fim, relativamente aos “[...]” foi, regra geral, duplicada a produção de modo que uma continha o que estava dentro destes parênteses e outra não.

Para o conflito derivado das produções

- Statement -> error SEMI
- Declaration -> error SEMI

Pensamos em expandir estas produções, por forma a eliminar o mesmo, havendo uma única produção que resultasse em "error SEMI” mais próxima do início da gramática.

Sem sucesso, devido ao facto da “Declaration” ser passível de ser chamada fora do “function body”, constatamos que a produção - “error SEMI” - tinha de estar na Declaration. Deste modo, o lado direito da regra - “error SEMI” - não podia estar, em simultâneo, diretamente no Statement, senão causaria erro.

Neste momento, tendo apenas um “error SEMI” na “Declaration”, pela forma como fomos construindo a gramática, é possível reduzir apenas o primeiro statement do function body por “error SEMI”, porém, os seguintes statements não teriam a capacidade de se reduzir também a isso, o que é suposto.

Isto leva-nos a introduzir e seguidamente expandir a produção Statement-> error SEMI para todos os sítios onde aparece Statement, através de um símbolo não terminal auxiliar “ErrorOrStat” que permite produzir um “Statement” ou “error SEMI”, exceto na DeclarationsOrStatements (e para níveis superiores) pois já é possível derivar “error SEMI“ por “DeclarationsOrStatements->Declaration->error SEMI”. Deste modo, o error SEMI do primeiro Statement fica coberto pelo “error SEMI” na “Declaration” e os subsequentes por este mecanismo.

II. Algoritmos e Estruturas de Dados da AST e da Tabela de Símbolos

- AST

- A estrutura de dados utilizada na AST foi uma árvore de estruturas, estruturas estas que são listas ligadas em alguns casos, onde cada estrutura procura representar os terminais e não terminais da gramática utilizada
  .
- A inserção da AST é feita através do yacc, de forma semelhante às fichas práticas (ou seja, existem várias funções de inserção, consoante o tipo de nó).

- O print da AST é feito percorrendo a árvore de estruturas a partir da sua raíz até às suas folhas, segundo a respetiva gramática.

- **Tabela de Símbolos**

  - Para a Tabela de Símbolos foi utilizada uma lista ligada de scopes que contém, dentro de cada nó desta lista, uma lista ligada de “table elements”.

  - O primeiro Scope na lista de scopes é o Scope Global, os restantes scopes são os scope locais a cada função, com o mesmo nome do que a própria função.

  - A inserção da Tabela de Símbolos é feita enquanto é feita a análise semântica, onde se adiciona um elemento a um determinado scope no caso de não haver nenhum erro lexical, sintático ou semântico (isto feito também percorrendo a AST da raíz às folhas)

  - O print da Tabela de Símbolos é feita percorrendo a lista de scopes, que por sua vez percorre a lista de table elements.

III. Geração de Código

A geração de código é feita, caso nenhum erro seja detetado, percorrendo a AST da raíz até às folhas, produzindo o respetivo código LLVM IR.

São mantidas 2 variáveis globais, o “varcounter” e o “labelcounter” que são contadores do número dos registos e do número do label (os labels têm o formato “labelX”, onde X é o valor do labelcounter) respetivamente. São tidos em conta os diferentes tipos de LLVM e o código correspondente para que tudo funcione como é suposto.

A geração de código é feita de forma semelhante ao print da AST:

Percorre-se recursivamente a AST e em cada nó dá-se print dum pedaço de código.

Dado que a optimização do código não era objetivo do projeto, optou-se por adicionar bastante código supérfluo que garanta que o programa corre sem problemas, nomeadamente:

- Todas as funções tem um return value por defeito, mesmo que estas especifiquem um return value;

- Todas as variáveis são inicializadas a 0;

- No scope local, em todas as expressões são geradas variáveis extra, de forma a simplificar o cálculo, e também porque inicialmente não tínhamos feito operações in-place, por exemplo:

  int a = 2,ficaria:

  ```
  %a = alloca i32
   %1 = add i32 2, 0
   store i32 %1, i32* %a
  ```

  A mesma lógica aplica-se às restantes operações;

- No caso das variáveis Globais não é possível usar este esquema e faz-se o cálculo in-place do valor atribuído à variável.

- O esquema simplificado mencionado acima pode levantar problemas no que toca à conversão de tipos e dessa forma todas as operações verificam o seus operandos convertendo tipos quando necessário.

  double a = 2, ficaria:

  ```
  %a = alloca double
  %1 = add i32 2, 0
  %2 = sitofp i32 %1 to double
  store double %2, double* %a
  ```

- De forma a simplificar a localização e uso das variáveis em llvm, em vez de numerais, estas usam o próprio nome da variável uC como valor da variável llvm. Da mesma forma, parâmetros passados a uma função são declarados sob a forma de %arg.{varname} e no início da mesma função convertidos para o sistema de nome de variável descrito:

  int foo(int a){...}, ficaria:

  ```
  define i32 @foo(i32 %arg.a){
  %a = alloca i32
  store i32 %arg.a, i32* %a
  ...
  ```