Compiladores (CC3001)
Aula 11: Implementação de código intermédio

Mário Florido

DCC/FCUP

2024

[dcc] FC FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Esta aula

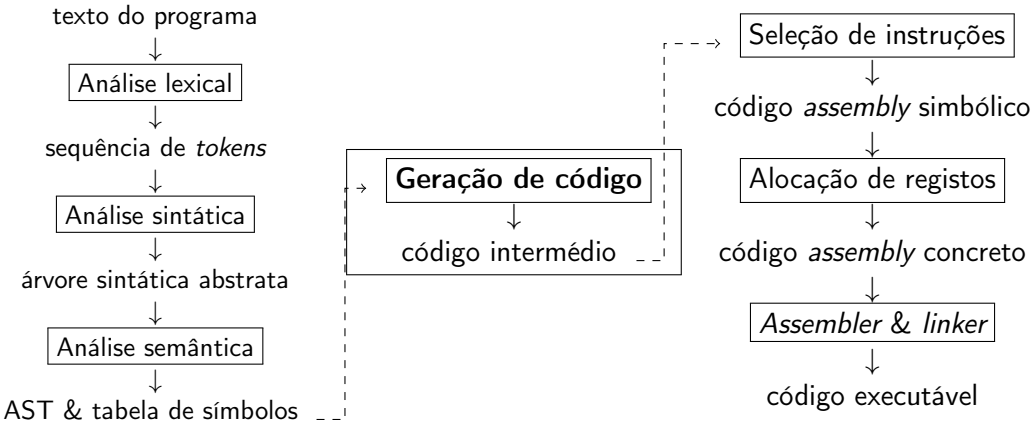
Representação de instruções

Geração de código

Geração de código em C

Geração de código em Haskell

Re-lembrar: fases dum compilador



- Na aula passada: esquemas de tradução de uma linguagem imperativa para código intermédio de 3 endereços
- Nesta aula: vamos ver algumas formas para implementar esses esquemas de tradução em Haskell e C

Esta aula

Representação de instruções

Geração de código

Geração de código em C

Geração de código em Haskell

- Representamos as instruções usando um tipo algébrico `Instr`
- Representamos Os operadores da linguagem por uma enumeração

```
data Instr = MOVE Temp Temp              -- temp1 := temp2
           | MOVEI Temp Int              -- temp1 := num
           | OP BinOp Temp Temp Temp     -- temp1 := temp2 op temp3
           | OPI BinOp Temp Temp Int     -- temp1 := temp2 op num
           | LABEL Label
           | JUMP Label
           | COND Temp BinOp Temp Label Label

type Temp = String
type Label = String

-- operadores binários
data BinOp = Plus | Minus | ...  | Lt | Lteq | Eq | ...
```

- Em C temos de "simular" os tipos algébricos
- Vamos ver uma solução simples mas suficiente

```
#include <stdint.h>
typedef struct { Opcode opcode;
                 Addr arg1, arg2, arg3, ...; // argumentos
               } Instr;
typedef enum { MOVE, MOVEI, ... } Opcode; // opcode da instrução
typedef intptr_t Addr;  // endereço (inteiro ou apontador)
```

- `Instr` é um estrutura com um número fixo de campos
- Preenchemos com 0 ou NULL os campos não usados
- Definimos um *opcode* para cada tipo de instrução (`MOVE`, `MOVEI`, etc.)
- A maioria da instruções usa 2 ou 3 argumentos (exceção: `COND`)

Esta aula

Representação de instruções

Geração de código

Geração de código em C

Geração de código em Haskell

- Em Haskell vamos implementar os esquemas de tradução como funções cujo resultado são listas de instruções
- Em C cada vamos acumular instruções num vector global

Esta aula

Representação de instruções

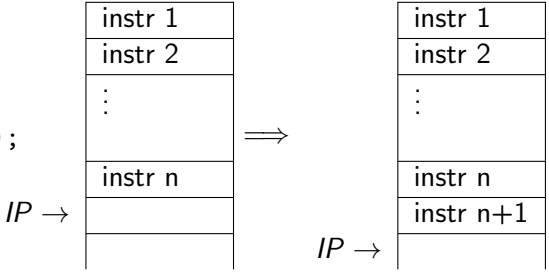
Geração de código

Geração de código em C

Geração de código em Haskell

- Usamos algumas auxiliares para acrescentar uma nova instrução ao final (marcado pelo *instruction pointer*)
- Intercalando *emit* com as chamadas recursivas podemos evitar a necessidade de concatenações de listas

```
void emit2(opc,arg1,arg2);
void emit3(opc,arg1,arg2,arg3);
...
```



```
void transExp(Exp exp, dest) {
 switch(...) {
 case BINOP:
   t1 = newTemp();
   t2 = newTemp();
   transExp(exp->binop.left, t1);   // lado esquerdo
   transExp(exp->binop.right, t2);  // lado direito
   emit3(opcode(exp->binop.op), dest, t1, t2);  // instrução final
   break;
 }
}
```

- Representamos os contadores de variáveis e etiquetas temporárias como inteiros
- Usamos contadores globais para gerar novos temporários e etiquetas:

```
int temp_count = 0, label_count = 0;

int newTemp() {
    return temp_count ++;
}

int newLabel () {
    return label_count ++;
}
```

Esta aula

Representação de instruções

Geração de código

Geração de código em C

Geração de código em Haskell

- Em Haskell não podemos variáveis globais
- Vamos ver duas soluções:
  - Passar explicitamente contadores de temporários;
  - Passar implicitamente contadores usando *State*

```
type Supply = (Int, Int)  -- contadores de temporários e etiquetas

newTemp :: Supply -> (Temp, Supply)
newTemp (temps, labels) =
      ("t"++show temps, (temps+1,labels))

newLabel :: Supply -> (Label, Supply)
newLabel (temps, labels) =
      ("L"++show labels, (temps,labels+1))
```

Com passagem explicita dos contadores.

```
transExpr :: Expr -> Table -> Temp -> Supply -> ([Instr], Supply)
transExpr (Num n) table dest supply
  = ([MOVEI dest n], supply)

transExpr (Op op e1 e2) table dest supply0
  = let (t1, supply1) = newTemp supply0
        (t2, supply2) = newTemp supply1
        (code1, supply3) = transExpr e1 table t1 supply2
        (code2, supply4) = transExpr e2 table t2 supply3
        code = code1 ++ code2 ++ [OP op dest t1 t2]
    in (code, supply4)
```

Passagem implícita dos contadores

- A passagem explícita é trabalhosa e dada a erros
  - usar $supply_0$ para obter $supply_1$, $supply_1$ para obter $supply_2$, ...
  - em caso de engano vamos re-utilizar temporários e gerar código errado!
- Vamos ver uma alternativa para que a passagem seja implicita

Passagem implícita dos contadores (cont.)

- Importamos `Control.Monad.State`
- Alteramos o tipo da função

  ```
  transExpr :: Expr -> Table -> Temp -> Supply -> ([Instr], Supply)
  ```

  para

  ```
  transExpr :: Expr -> Table -> Temp -> State Supply [Instr]
  ```

- O tipo `State Supply` significa que a função tem:
  - um argumento implícito (`Supply` inicial)
  - um resultado implícito (`Supply` final)

Casos terminais: removemos os contadores e introduzimos `return`.

```
-- versão explícita
transExpr (Num n) table dest supply = ([MOVEI dest n], supply)


-- versão implícita
transExpr (Num n) table dest        = return [MOVEI dest n]
```

Passagem implícita dos contadores (cont.)

Casos recursivos: removemos os contadores e usamos notação-do em vez de `let`.

```
-- versão explícita
transExpr (Op op e1 e2) table dest supply0
  = let (t1, supply1) = newTemp supply0
        (t2, supply2) = newTemp supply1
        (code1, supply3) = transExpr e1 table t1 supply2
        (code2, supply4) = transExpr e2 table t2 supply3
        in (code1 ++ code2 ++ [OP op dest t1 t2], supply4)

-- versão implícita
transExpr (Op op e1 e2) table dest
  = do t1 <- newTemp
       t2 <- newTemp
       code1 <- transExpr e1 table t1
       code2 <- transExpr e2 table t2
       return (code1 ++ code2 ++ [OP op dest t1 t2])
```

Passagem implícita dos contadores (cont.)

Para executar a tradução usamos runState ou evalState com o valor iniciais dos contadores:

```
> let expr = Op Mult (Num 3) (Op Plus (Num 4) (Num 5))
> runState (transExpr expr Map.empty "r") (0,0)
([MOVEI "t0" 3,  MOVEI "t2" 4,  MOVEI "t3" 5,
  OP Plus "t1" "t2" "t3", OP Mult "r" "t0" "t1"], (4,0))
> evalState (transExpr expr Map.empty "r") (0,0)
[MOVEI "t0" 3, "MOVEI "t2" 4,  MOVEI "t3" 5,
  OP Plus "t1" "t2" "t3", OP Mult "r" "t0" "t1"]
```

Passagem implícita dos contadores (cont.)

Temos ainda de modificar as funções auxiliares.

```
import Control.Monad.State

type Count = (Int,Int)

newTemp :: State Count Temp
newTemp =  do (temps,labels) <- get
              put (temps+1,labels)
              return ("t"++show temps)

newLabel :: State Count Label
newLabel = do (temps,labels) <- get
              put (temps,labels+1)
              return ("L"++show labels)
```