

– INF01147 –
Compiladores

Geração de Código para Controle de fluxo
Suporte ao ambiente de Execução

Prof. Lucas M. Schnorr
– Universidade Federal do Rio Grande do Sul –



Plano da Aula de Hoje

- ▶ Geração de código para controle de fluxo
- ▶ Suporte ao ambiente de execução
 - ▶ Introdução
 - ▶ Alocação de memória
 - ▶ Registros de Ativação
 - ▶ Sequência de Chamada
 - ▶ Passagem de parâmetros
 - ▶ Acesso a variáveis (Checagem de Escopo)

Geração de código para
controle de fluxo

Controle de Fluxo

- ▶ Controlar o fluxo de execução
 - ▶ Gerar código de controle
 - ▶ Utiliza rótulos e desvios
- ▶ Estudaremos três situações (if, if else, while)
- ▶ Gramática (B é uma expressão booleana)

$$S \rightarrow \text{if (B) } S_1$$
$$S \rightarrow \text{if (B) } S_1 \text{ else } S_2$$
$$S \rightarrow \text{while (B) } S_1$$

Controle de Fluxo – if

```
S  →  if { B.t=rot(); B.f=S.next; }  
      (B) { S1.next=S.next; }  
      S1 { S.code=B.code || gera(B.t:) || S1.code }
```

Fluxo de Execução – if else

```
S  →  if  { B.t=rot(); B.f=rot(); }  
      (B) { S1.next=S.next; }  
      S1 else { S2.next=S.next; }  
      S2 { S.code=B.code || gera(B.t:) || S1.code || gera(goto S.next) ||  
          gera(B.f:); || S2.code }
```

Controle de Fluxo – while

```
S  →  while  { B.f=S.next; B.t=rot(); }  
      (B)  { S.begin=rot(); S1.next=S.begin; }  
      S1  { S.code=gera(S.begin:) || B.code ||  
           gera(B.t:) || S1.code || gera(goto S.begin) }
```

Gramática para Exercício

```
S → attr { S.code=gera(attr.lexval) || gera(goto S.next) }
S → if { B.t=rot(); B.f=rot(); }
    (B) { S1.next=S.next; }
    S1 else { S2.next=S.next; }
    S2 { S.code=B.code || gera(B.t:) || S1.code ||
        gera(B.f:); || S2.code }
S → while { B.f=S.next; B.t=rot(); }
    (B) { S.begin=rot(); S1.next=S.begin; }
    S1 { S.code=gera(S.begin:) || B.code ||
        gera(B.t:) || S1.code || gera(goto S.begin) }
B → { B1.t=rot(); B1.f=B.f; } B1 and { B2.t=B.t; B2.f=B.f; } B2
    { B.code=B1.code || label(B1.t) || B2.code }
B → E1 relop E2 { B.code=E1.code || E2.code ||
    gera(if E1.local relop.lexval E2.local goto B.t) ||
    gera(goto B.f); }
```

Controle de Fluxo – Exercício

- Gere o TAC para o código seguinte

```
while (a < b && e != f) {  
    if (c < d){  
        x = y + z;  
    }else{  
        x = x - z;  
    }  
}
```

Suporte ao Ambiente de Execução

- ▶ Geração de Código Intermediário resolve parte do problema
 - ▶ Cálculo de endereçamento para arranjos
 - ▶ Endereçamento para variáveis locais
 - ▶ Expressões aritméticas e booleanas
 - ▶ Construções de fluxo de controle
 - ▶ Atribuições para variáveis
- ▶ O que falta?
 - ▶ Implementação da chamadas de funções
 - ▶ Gerenciar parâmetros formais e reais
 - ▶ Método de passagem de parâmetro
 - ▶ Alocação dinâmica no monte
 - ▶ Acesso a variáveis e dados
 - ▶ Qual a primeira instrução a ser executada?

Suporte ao Ambiente de Execução

- ▶ Conjunto de rotinas chamado **Pacote de Suporte à Execução**
 - ▶ Código adicional embutido no binário
 - ▶ Gerado pelo compilador
- ▶ Envolve uma série de conceitos e técnicas
 - ▶ Composição do binário do programa (e carga em memória)
 - ▶ Alocação dinâmica de endereços de memória
 - ▶ Acesso a variáveis e dados
 - ▶ Chamadas de funções

Organização de Memória

Organização da Memória

- ▶ Composição do binário gerado pelo compilador
 - ▶ Variáveis estáticas
 - ▶ Código executável
 - Incluindo o Pacote de Suporte à Execução
- ▶ Na execução, o sistema operacional cria um processo
 - ▶ **Segmento de Código**: código executável
 - ▶ **Segmento de Dados**: variáveis estáticas
 - ▶ **Monte**
 - ▶ Alocação dinâmica de memória
 - ▶ **Pilha**
 - ▶ Registros de Ativação (RA) para chamadas de funções
 - ▶ Alocação dinâmica de memória para variáveis locais

Alocação de Memória – Estática

- ▶ Alocação estática do **Segmento de Código**
 - ▶ Reserva de memória é realizada durante a compilação
 - ▶ Tamanho do código executável é conhecido antes da execução
- ▶ Alocação estática do **Segmento de Dados**
 - ▶ Tipo (e tamanho) dos dados é conhecido em compilação
 - ▶ Tamanho de tipos é estável (não muda na execução)

Alocação de Memória – Dinâmica

- ▶ Alocação dinâmica no **Monte**
 - ▶ Áreas alocadas explicitamente pelo programa
 - Com uso de funções `malloc` e `free`
 - ▶ A área cresce no sentido contrário ao da pilha
 - ▶ Alocação em geral é caótica
- ▶ Gerenciamento do monte pode ser
 - ▶ Em nível de usuário (no caso da `libc`)
 - ▶ Em nível de sistema operacional (chamada de sistema)
 - ▶ Existe uma multitude de técnicas diferentes

Alocação de Memória – Dinâmica

- ▶ Alocação dinâmica na **Pilha**
 - ▶ Variáveis locais e suporte a chamada de funções
 - ▶ **Pilha de Registros de Ativação (RA)**
 - ▶ Tamanho da Pilha
 - ▶ Tamanho do registro * Número máximo de ativações
- ▶ Gerenciamento realizado pelo compilador
 - ▶ Prepara-se um RA para cada procedimento/função
 - ▶ Gera-se código para se criar instâncias (que serão empilhadas)
 - ▶ **Sequência de Chamada e de Retorno**

Registro de Ativação

- ▶ Conteúdo de um Registro de Ativação (do topo para baixo)
 - ▶ Temporários (parte variável do RA)
 - ▶ Variáveis Locais
 - ▶ Estado da máquina salvo
 - ▶ Vínculo Estático (ponteiro para o RA do pai estático)
 - ▶ Vínculo Dinâmico (ponteiro para o RA do pai dinâmico)
 - ▶ Valor retornado
 - ▶ Argumentos
- ▶ Na invocação, criada e colocada no topo da pilha
- ▶ Removida da pilha no momento do retorno

Registro de Ativação – Organização

- ▶ Membros devem ser endereçáveis com deslocamento sobre fp
- ▶ Variáveis locais de tamanho estático
 - ▶ Fazem parte da parte de tamanho fixa
- ▶ Variáveis locais de tamanho dinâmico
 - ▶ Alocados na parte de tamanho variável
 - ▶ Ponteiro e Descritor na parte de tamanho fixo

Sequência de Chamada

- ▶ Responsável pelo gerenciamento de registro de ativação
- ▶ Podemos quebrá-la em duas partes
 - ▶ Sequência de chamada (executada pelo chamador e chamado)
 - ▶ Sequência de retorno (executado pelo chamado, no final)

Sequência de Chamada (sugestão)

- ▶ Sequência de chamada (chamador e chamado)
 1. Cria um novo registro de ativação
 2. Calcula o vínculo estático
 3. Passa os parâmetros (organizando-os na pilha)
 4. Passa o endereço de retorno para o chamado
 5. Transfere o controle para o chamado
 6. Salva o estado de execução atual (registradores)
 7. Salva o antigo fp na pilha (como vínculo dinâmico)
 8. Aloca variáveis locais
- ▶ Sequência de retorno (chamado, no final)
 1. Prepara os parâmetros de retorno
 2. Disponibiliza o valor de retorno para o chamador
 3. Atualiza o fp e o sp
 4. Atualiza o estado de execução do chamador
 5. Transfere o controle

Passagem de Parâmetros

Passagem de Parâmetros

► Por valor

- Método mais simples trivial (maioria das linguagens o tem)
- Cria-se uma cópia do parâmetro real para o parâmetro formal
 - Usa-se o parâmetro formal como se fosse uma variável local

► Por referência

- Função chamadora passa o endereço de cada parâmetro real
 - Independe do tamanho, menos ocupação de espaço
- Parâmetro real for
 - Um identificador (endereço é fornecido)
 - Endereço pode ser pilha, do monte, do segmento de dados
 - Uma expressão
 - Avalia-se a expressão
 - Coloca-se seu resultado em um temporário
 - Endereço do temporário é fornecido
- Uso dos parâmetros formais é feito através de indireção

Acesso a Dados

Checagem de Escopo

Acesso a dados

- ▶ Compilador deve definir **endereçamento de variáveis**
 - ▶ Consta na tabela de símbolos
- ▶ **Escopo Estático**
 - ▶ **Sem** procedimentos aninhados
 - ▶ Variáveis estáticas alocadas no segmento de dados
→ Endereçamento absoluto
 - ▶ Variáveis locais alocadas na pilha de ativação
→ Endereçamento referente ao fp (*frame pointer*)
 - ▶ **Com** procedimentos aninhados
 - ▶ Árvore de tabela de símbolos
 - ▶ Na declaração da variável
→ registra-se seu nível de profundidade
 - ▶ No acesso
 1. Procura-se na árvore de tabela de símbolos a declaração
 2. Calcula-se a diferença do escopo atual para o da declaração
 3. Gera código de acesso considerando uma tupla
→ Endereçamento é (diferença_escopo, desloc_fp)
 - ▶ O suporte utiliza a tupla para encontrar a variável na pilha
- ▶ **Escopo Dinâmico** (bonus)

Acesso a dados

- ▶ Considerando escopo estático, **durante a execução**
 - ▶ Deve-se dereferenciar `diferença_escopo` vezes
 - “Perda de tempo” se aninhamento profundo
 - Principalmente para acessar variáveis globais
- ▶ Possível solução (em tempo de execução)
 - ▶ Manter um arranjo com RA associado a cada escopo
 - ▶ Cada variável tem associada o seu escopo
 - calculado em tempo de compilação
 - ▶ No acesso ao conteúdo de uma variável
 - ▶ Basta consultar a posição no arranjo para achar a variável
 - Endereçamento (`escopo`, `desloc_fp`)
 - (onde `escopo` é o índice do arranjo)

Conclusão

- ▶ Leituras Recomendadas para a aula de hoje
 - ▶ Livro do Dragão
 - ▶ Seções 6.6 e Capítulo 7
 - ▶ Série Didática
 - ▶ Seção 5.4
- ▶ Próxima Aula
 - ▶ Suporte ao Ambiente de Execução
 - ▶ Geração de Código Assembly
 - ▶ Otimizações