

# PCS730 - Prova

Lucas Virgili

# 1. Introdução e conceitos básicos

## ► Slides 1.20 - 1.21

Os slides acima descrevem, em “alto nível”, os passos executados pelo compilador durante o processo de compilação, que funcionam como um “pipeline”.

Inicialmente, é necessário que o compilador consiga acessar o arquivo-fonte fisicamente, para poder, então, acessá-lo através de uma forma mais abstrata. De posse dessa representação lógica do arquivo-fonte, o compilador pode extrair caracteres do fonte e alimentar o analisador léxico, que extrai os átomos do código. Esses átomos são, então, passados ao analisador sintático que pode, assim, criar a árvore sintática abstrata que representa o programa. Essa árvore, por sua vez, é usada como entrada pelo otimizador, que gera uma árvore reduzida, a qual é, finalmente, utilizada pelo gerador de código para gerar o código-objeto.

## 2. Linguagens formais e autômatos

### ► Elementos do slide 2.20

O slide 2.20 descreve os tipos de transição possíveis em autômatos de pilha estruturados.

1. Transições internas, ou seja, sem chamadas de submáquina:
  - 1.1 Transições em vazio, sem consumo de átomo, do estado  $j$  para o estado  $k$
  - 1.2 Transições com consumo de átomo, também do estado  $j$  para o estado  $k$ .
2. Transições externas, ou seja, com chamadas de submáquina:
  - 2.1 Chamada de submáquina pela máquina  $i$  no estado  $j$ , indo para a submáquina  $m$  no estado 0
  - 2.2 Retorno da submáquina  $i$  no estado  $j$  para a submáquina que a chamou,  $m$ , no estado seguinte dela, no caso  $n$ .

## 4. Técnicas clássicas de análise (b) bottom-up

### ► Simplificação de autômatos

A sequência de slides 4.18 a 4.28 descrevem a sequência de operações para simplificar os autômatos. Os passos são os seguintes:

1. Eliminar as transições em vazio: Se um estado  $i$  possui transição em vazio para o estado  $j$ , removemos a transição de  $i$  para  $j$  e, para toda transição de  $j$  para outros estados  $k_n$ , criamos uma transição de  $i$  para  $k_j$  com o consumo de átomo correto
2. Eliminação de estados inatingíveis: construindo-se uma árvore de atingibilidade do autômato, a partir do nó principal, repetindo-se o seguinte processo: para cada estado não atingido ainda de cada folha, acrescentamos uma nova folha à árvore. Quando não pudermos mais acrescentar novas folhas, estados que não estão na árvore não são atingíveis e podem ser removidos.
3. (Continua no próximo)

3. Eliminação de não-determinismos: podem sobrar transições para mais de um estado com o mesmo consumo de átomo. Para removermos tais transições, criamos um estado auxiliar para cada transição desse tipo. Tais estados devem ter as transições que “saem” dos estados de origem.

## 6. Automatização da construção de compiladores

- ▶ Flex é um programa gerador de tokenizers a partir de expressões regulares e código em uma linguagem base, como C<sup>1</sup>. Podemos usá-lo para extrair os terminais de um código fonte, ou seja, construir analisadores léxicos com ele.<sup>2</sup>
- ▶ Bison é o equivalente sintático do Flex: ele converte uma gramática livre de contexto em um autômato LR. Podemos usar o Bison para escrever o reconhedor sintático de uma linguagem.
- ▶ Podemos usar o par acima para criar um compilador, bastando escrever as funções semânticas na linguagem base para cada regra sintática ou léxica.<sup>3</sup>

---

<sup>1</sup>Usar o de SML foi horrível.

<sup>2</sup>Ref.: <http://flex.sourceforge.net/>

<sup>3</sup>Ref.: <http://www.gnu.org/software/bison/>

## 7. Recuperação de erros

### ► Um método simples de recuperação de erros<sup>4</sup>

Uma maneira que eu imagino que seja razoavelmente simples para se implementar é criar um estado “lixão”. Todos os estados que não são o “lixão” têm transições para o lixão com todos os símbolos que não representam transições “reais”, ou seja, as transições para estados não lixão.

Já o estado lixão tem apenas duas transições: uma para algum estado inicial com algum dos símbolos que ele sabe consumir e outra transição de “fecho”, consumindo quaisquer outros símbolos. Ou seja, até encontrar o símbolo correto, o estado “lixão” remove símbolos da entrada.

A implementação seria um tanto monótona, contudo, já que seria mais simples adicionar o estado lixo na própria matriz de transição do que na gramática.

---

<sup>4</sup>Note que eu ainda acho que não se recuperar do erro e abandonar o programador com sua incompetência é o melhor a se fazer

## 8. Funcionalidades do compilador

- ▶ Grande parte da semântica de um programa não é feita pelos componentes mencionados. O compilador deve:
  - ▶ Manter a tabela de símbolos, o que pode ser feito pelo analisador léxico;
  - ▶ Atribuir atributos aos símbolos;
  - ▶ Guardar informações sobre o escopo;
  - ▶ Representar tipos;
  - ▶ Checar se o uso dos identificadores está correto;
  - ▶ Gerenciar memória;
  - ▶ Comunicação com o ambiente de execução.



## 9. Formas intermediárias

### ► Vantagens:

1. Geração de código objeto especializado para diferentes arquiteturas;
2. Várias linguagens diferentes gerando essa linguagem e utilizadas ao mesmo tempo;
3. Otimizações baseadas nela podem usadas para todas as linguagens que geram a linguagem intermediária.

### ► Desvantagens:

1. Origem da máquina virtual: é código-aberto?, quem a mantém?, uso comercial?, etc.
- Não, pode-se converter essa linguagem intermediária depois para outras diferentes
  - Diversas formas intermediárias podem ser usadas em conjunto quando cada uma facilita algum passo da compilação. Por exemplo, uma permite otimizações locais de forma simples e outra permite uma redução do tamanho do objeto.

## 10. Análise semântica

- ▶ Semântica estática é o “acochambramento” feito pelo compilador para poder lidar com gramáticas dependentes de contexto. Nesse caso, é de responsabilidade da semântica analisar e garantir a correta associação entre identificadores e valores, tipos e etc.
- ▶ O nome é inadequado pois isso deveria ser um aspecto sintático.
- ▶ Exemplos são o gerenciamento de memória, de escopos, comunicação com o ambiente de execução.



## 12. Otimização de código

- ▶ Otimização global se aplica considerando-se o programa como um todo, ao contrário das otimizações locais, que baseiam-se em blocos básicos. Exemplo de global é a eliminação de regiões inacessíveis e de local é *tail-call recursion*.
- ▶ Otimizações dependentes de máquina são as que só podem ser feitas (ou têm efeito) em determinadas arquitetura. Por exemplo, usar alguma instrução de manipulação vetorial. Já otimizações independentes baseiam-se em propriedades matemáticas do programa, procurando uma representação equivalente porém mais eficiente.
- ▶ Todos estes são objetivos diferentes de otimização: quer-se minimizar o tempo de execução ou o tamanho do executável ou a quantidade de comunicação?
- ▶ Programas sequenciais podem ser otimizados de forma diferente de paralelos, já que funcionam de forma diferente. Por exemplo, podemos querer minimizar o número de acessos à memória compartilhada e sincronização de caches em um programa paralelo, enquanto em um sequencial esses