

– INF01147 –  
Compiladores

Apresentação da Disciplina  
Projeto de Compilador  
Introdução Geral

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



# Apresentação da disciplina – Professor

- ▶ Prof. Lucas M. Schnorr
- ▶ Doutorado em co-tutela UFRGS/INPG (2005 – 2009)
- ▶ Pesquisador CNRS na França (2009 – 2013)
  - ▶ Processamento Paralelo
  - ▶ Sistemas Distribuídos
  - ▶ Análise de Desempenho
  - ▶ Visualização de Rastros
- ▶ PajeNG, Viva, SimGrid, Akypuera, Poti, Tupi
- ▶ Contato
  - ▶ Prédio 73, Sala 202
  - ▶ <http://www.inf.ufrgs.br/~schnorr/>
  - ▶ [schnorr@inf.ufrgs.br](mailto:schnorr@inf.ufrgs.br)

# Apresentação da disciplina – Plano

- ▶ Súmula, conteúdo programático e cronograma
- ▶ Procedimentos didáticos, laboratórios
- ▶ Trabalhos, provas e avaliação
- ▶ Moodle da UFRGS: <http://moodleinstitucional.ufrgs.br>
  - ▶ Login: código do aluno + senha do portal do aluno
  - ▶ Comunicação professor/alunos

# Apresentação da disciplina – Bibliografia

- ▶ Compilers: Principles, Techniques and Tools.  
Aho, A.; Sethi, R.; Ullman, J. D.  
(Dragão roxo ou vermelho)
- ▶ Engineering a Compiler.  
Cooper & Torczon  
2nd edition
- ▶ Impl. de Linguagens de Programação: Compiladores.  
Ana Price & Simão Toscani.
- ▶ Lex & Yacc.  
Tony Mason and Doug Brown.
- ▶ Projeto moderno de compiladores.  
D. Grune, H. Bal e K. Langendoen.

# Apresentação da disciplina – Motivação

- ▶ É uma disciplina complexa
- ▶ Forte background teórico/prático
- ▶ Dependências com outras áreas
  - ▶ Programação (Linguagem C, Scripts)
  - ▶ Linguagens Formais (gramáticas, autômatos)
  - ▶ Estruturas de Dados (pilhas, grafos, árvores, tabelas)
  - ▶ Sistemas Operacionais (memória, formato binário)
  - ▶ Organização de Computadores (processador, registros)

# Apresentação da disciplina – Motivação

“Ninguém trabalha com compiladores” → falso

- ▶ Principais compiladores
  - ▶ gcc – <http://gcc.gnu.org/>
  - ▶ LLVM/clang – <http://www.llvm.org/>
  - ▶ Intel (icc)
- ▶ Novas arquiteturas: Celulares, Raspberry Pi, ...
- ▶ Disciplina interessante sob vários aspectos
  - ▶ Compreender a implementação de uma linguagem
    - ▶ Todo programador deve saber isso
    - ▶ Permite programar com eficiência
  - ▶ Usa técnicas e ferramentas avançadas de desenvolvimento
  - ▶ Técnicas de compiladores são aplicáveis em outras áreas
    - ▶ Reconhecimento de padrões
    - ▶ Tratamento automatizado do eventos

# Apresentação da disciplina – Objetivos

- ▶ Entender o funcionamento de um compilador
  - ▶ Teórico: algoritmos e estruturas de dados
  - ▶ Prático: projeto e implementação
- ▶ Projetar e implementar um compilador para uma linguagem de programação imperativa
- ▶ Adquirir experiência de programação na linguagem C

# Apresentação da disciplina – Funcionamento

- ▶ A presença das aulas será aferida em todas as aulas
  - ▶ Feita nos primeiros 10 minutos da aula
  - ▶ Se chegou após a chamada, tem meia presença (se conversar com o professor no final da aula)
  - ▶ 75% de frequência para evitar conceito FF
- ▶ Moodle/bibliografia
  - ▶ Será oferecido material de apoio
  - ▶ Presença em aula é fundamental
- ▶ Durante a aula
  - ▶ Prestar atenção
  - ▶ Realizar anotações
  - ▶ Perguntar em caso de dúvida
- ▶ **Cuidado:** a disciplina é difícil
  - ▶ Tudo se passa muito bem caso se trabalhe regularmente

# Apresentação da disciplina – Avaliação

- ▶ **Prova** Teórica (P)
  - ▶ Antes do fim do semestre (ver cronograma)
  - ▶ Testará os aspectos teóricos da disciplina (até a data da prova)
  - ▶ Recuperação possível no final do semestre
- ▶ **Projeto** de Compilador (T)
  - ▶ Desenvolvimento contínuo (por etapas) de um compilador
    - ▶ Testes, documentação e comentários no código
    - ▶ Respeito das especificações
    - ▶ Linguagem C
  - ▶ Grupos de **três** alunos
  - ▶ Apresentações regulares
- ▶ Nota final:  $\frac{(P+T)}{2}$   
→ gerando o conceito correspondente

# Introdução Geral

## Compiladores

# Linguagens de Programação

- ▶ Modelos de Linguagem de Programação (MLP)
- ▶ Linguagem de programação
  - ▶ meio de comunicação entre o usuário e o computador



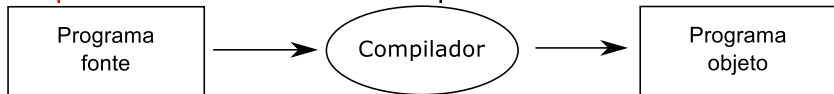
- ▶ Linguagens em diferentes níveis de abstração
  - ▶ Baixo nível (de máquina, usada pelo computador)
  - ▶ Alto nível (próxima ao pensamento humano)

# Níveis de Abstração em Linguagens

- ▶ Linguagens de máquina
  - ▶ Sequência de zeros e uns (0011010000011101)
  - ▶ Notação binária
- ▶ Linguagens Simbólicas
  - ▶ Exemplo: Assembly
  - ▶ Add 4(0), #1
- ▶ Linguagens orientadas ao usuário
  - ▶ Maioria das linguagens de programação
    - ▶ C, Fortran, Pascal, Algol, Ada, Objective-C, C++
    - ▶ `*x += 1;`
- ▶ Linguagens orientadas à aplicação (scripts)
  - ▶ Excel, SQL, Matlab, R, ...

# Tradução de Linguagens de Programação

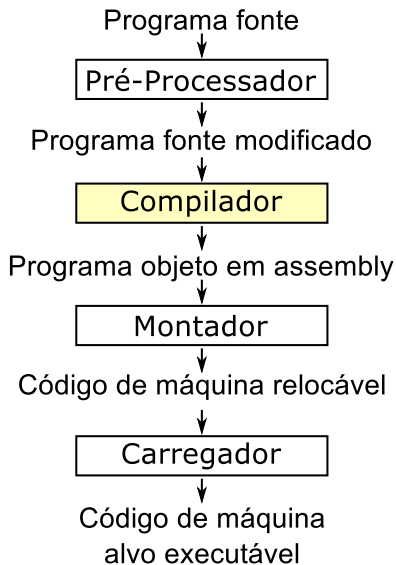
- ▶ Duas definições para **compilador**
- ▶ Forte: é um programa de computador que transforma uma **linguagem de alto nível** qualquer em uma **linguagem de máquina** de uma determinada arquitetura



- ▶ Fraca: é um programa de computador que transforma uma linguagem de alto nível em outra linguagem qualquer
- ▶ Compilador ou interpretador?
  - ▶ Há diferença
- ▶ Combinando as duas abordagens
  - ▶ Exemplo: Java, Python
  - ▶ Por questões de desempenho

# Passos no processo de tradução

- ▶ Várias etapas
- ▶ Programas auxiliares
  - ▶ Normalmente embutidos
- ▶ Exemplo utilizando **gcc**
  - ▶ `gcc -E hello.c > hello-pre.c`
  - ▶ `gcc -c hello-pre.c -o hello.o`
  - ▶ `nm hello.o`
  - ▶ `gcc hello.o -o hello`
  - ▶ `nm hello`
  - ▶ `ldd hello`
  - ▶ `hexdump hello.o | wc -l`
  - ▶ `hexdump hello | wc -l`



# Como fazer a tradução?

## ► Código Fonte C

```
int i;
double x;
i = 0;
x = 1.0;
while (i <= 10) {
    x = x * 2.0;
    i = i + 1;
}
```

## ► Código Assembly

```
movl    $0, -12(%ebp)    ;; i = 0
fldl
fstpl    -24(%ebp)        ;; x = 1.0;
jmp     .L2

.L3:
fldl    -24(%ebp)
fadd     %st(0), %st      ;; x = 2 * x;
fstpl    -24(%ebp)
addl     $1, -12(%ebp)    ;; i+1

.L2:
cmpl     $10, -12(%ebp)   ;; i <= ? 10
jle      .L3              ;; SIM, goto L3
movl     $0, %eax
;; NAO, encerra...
```

# Partes do processo de compilação

- ▶ Análise (front-end)

**Reconhecer** o que o usuário quer escrever

- ▶ “Palavras” → análise lexical
- ▶ “Frases” → análise sintática/semântica
- ▶ Detecção de erros

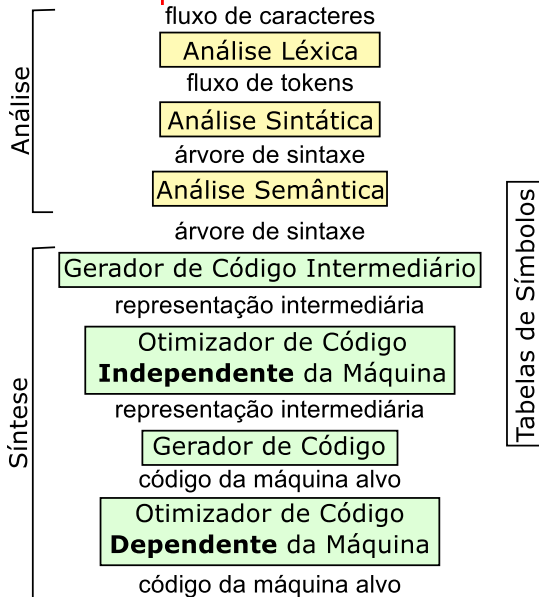
- ▶ Síntese (back-end)

**Produzir** alguma coisa, em função da análise

- ▶ Meta-representação do programa (código intermediário)
- ▶ Tabela de símbolos
- ▶ Código otimizado

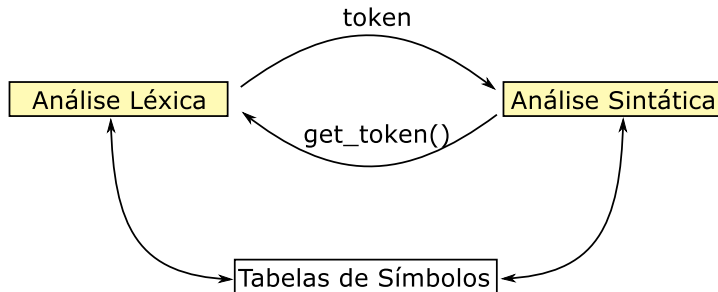
→ Fases de compilação

# Estrutura de um **Compilador** em fases



# Análise Lexical (scanner)

- ▶ Identificar sequências significativas na entrada: **lexemas**
- ▶ Quando um lexema é identificado, gera um **token**
- ▶ Funções adicionais
  - ▶ Começa a construção da tabela de símbolos
  - ▶ Detecção de erros léxicos, gerando mensagens de erro



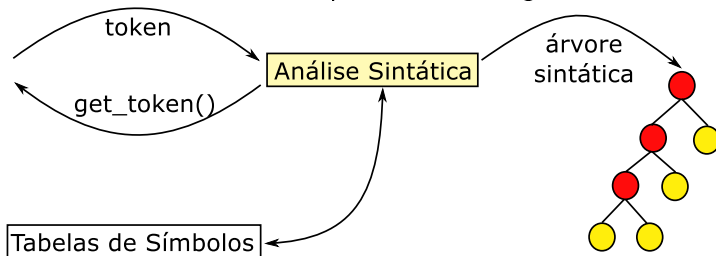
- ▶ Um token é composto de: **<nome-token, valor-atributo>**
- ▶ Exemplo

# Como reconhecer os tokens?

- ▶ Através do uso de **Expressões Regulares**
- ▶ Algumas regras para formação de palavras válidas
  - ▶ Concatenação:  $xy$  (x seguido de y)
  - ▶ Alternância:  $x|y$  (x ou y)
  - ▶ Repetição:  $x^*$  (x repetido 0 ou mais vezes)
  - ▶ Repetição:  $x^+$  (x repetido 1 ou mais vezes)
- ▶ As mesmas expressões regulares usadas correntemente
  - ▶ vim – usando o comando
  - ▶ emacs – Ctrl + Alt + % “Query replace regexp ->”
  - ▶ grep, sed, ...
- ▶ Existe uma multitude de recursos de apoio
  - ▶ Procurar por “Regular Expressions” em qualquer livraria
  - ▶ Manual do SED: <http://www.gnu.org/software/sed/>
  - ▶ man grep (seção “Regular Expressions”)

# Análise Sintática (parsing)

- ▶ Tem como entrada um fluxo de tokens
- ▶ Mapeia sequências de tokens para estruturas sintáticas
- ▶ Cria uma **Árvore de Sintaxe**
  - ▶ Nós intermediários representam operações
  - ▶ Filhos desses nós representam os argumentos



- ▶ Funções
  - ▶ Verificar a estrutura gramatical do programa
  - ▶ Detecção de erros sintáticos, gerando mensagens de erro
  - ▶ Tentar sobreviver a um erro sintático

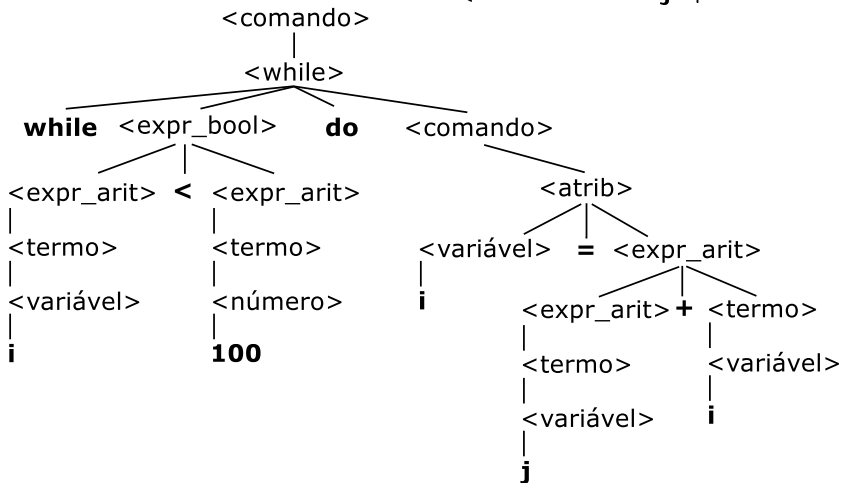
▶ Exemplo

# Como construir a árvore de sintaxe?

- ▶ Através do uso de **Gramáticas Livres de Contexto**
  - ▶ Conjunto de símbolos terminais (T)
  - ▶ Conjunto de símbolos não-terminais (NT)
  - ▶ Conjunto de produções (ou Regras de derivação)  
 $\langle NT \rangle \rightarrow \text{sequência de } \langle T \rangle \text{ ou } \langle NT \rangle$
  - ▶ Um  $\langle NT \rangle$  como o símbolo inicial da gramática
- ▶ Notação para gramáticas: **BNF** (Backus-Naur Form)
  - $\langle \text{comando} \rangle \rightarrow \langle \text{while} \rangle \mid \langle \text{atrib} \rangle \mid \dots$
  - $\langle \text{while} \rangle \rightarrow \text{while } \langle \text{expr\_bool} \rangle \text{ do } \langle \text{comando} \rangle$
  - $\langle \text{atrib} \rangle \rightarrow \langle \text{variável} \rangle = \langle \text{expr\_arit} \rangle$
  - $\langle \text{expr\_bool} \rangle \rightarrow \langle \text{expr\_arit} \rangle < \langle \text{expr\_arit} \rangle$
  - $\langle \text{expr\_arit} \rangle \rightarrow \langle \text{expr\_arit} \rangle + \langle \text{termo} \rangle \mid \langle \text{termo} \rangle$
  - $\langle \text{termo} \rangle \rightarrow \langle \text{número} \rangle \mid \langle \text{variável} \rangle$
  - $\langle \text{variável} \rangle \rightarrow i \mid j$
  - $\langle \text{número} \rangle \rightarrow 100$

# Árvore de derivação

- ▶ Ilustra a **derivação das regras** de uma gramática
- ▶ Considerando a entrada: **while i < 100 do i = j + i**



# Análise Semântica

- ▶ Avaliar a **consistência semântica** do programa
- ▶ Verificação de tipos
  - ▶ Métodos de coerção (caso a definição da linguagem autorisar)
- ▶ Exemplo

# Geração de Código Intermediário

- ▶ Usa a **representação interna** do compilador
  - ▶ Exemplo: LLVM Language Reference Manual  
<http://www.llvm.org/docs/LangRef.html>
- ▶ Gera código objeto ou intermediário
- ▶ Se for um código intermediário
  - ▶ não especifica detalhes arquiteturais
  - ▶ registradores
  - ▶ endereçamento, etc
- ▶ Exemplo com código de três endereços
- ▶ Exemplo considerando a entrada: **while  $i < 100$  do  $i = j + i$**

L0: if  $i < 100$  goto L1

goto L2

L1: temp =  $i + j$

$i = \text{temp}$

goto L0

L2: ...

# Otimização de Código

- ▶ Realizar **otimizações** sobre o código intermediário
  - ▶ Desempenho durante a execução
  - ▶ Eficiência na ocupação dos recursos
    - diminuir quantidade de memória, de registradores
- ▶ Exemplo a partir do código de três endereços
- ▶ Exemplo considerando a entrada: **while i < 100 do i = j + i**

## ▶ Código Inicial

```
L0: if i < 100 goto L1
    goto L2
L1: temp = i + j
    i = temp
    goto L0
L2: ...
```

## ▶ Código Otimizado

```
L0: if i >= 100 goto L2
    i = i + j
    goto L0
L2:
```

# Geração de Código Objeto

- ▶ Gerar código objeto considerando
  - ▶ Qual é a arquitetura alvo
  - ▶ Alocação de memória
  - ▶ Seleção de registradores
- ▶ Exemplo considerando a entrada: **while  $i < 100$  do  $i = j + i$**

## ▶ Código Otimizado

```
L0: if i >= 100 goto L2
    i = i + j
    goto L0
L2:
```

## ▶ Código Objeto para PC8086

```
L0: MOV AX, i
    CMP AX, 100
    JGE L2      //jump condicional
    MOV AX, j
    MOV BX, i
    ADD BX
    MOV i, AX
    JMP L0      //jump não condicional
L2: ...
```

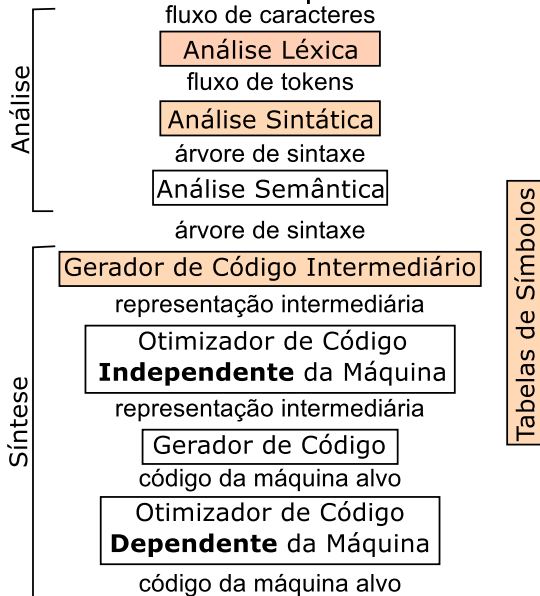
# Gerência da Tabela de Símbolos

- ▶ Acompanha todas as fases do compilador
- ▶ Guarda atributos das **variáveis** e **funções** do programa
- ▶ Atributos de variáveis
  - ▶ Espaço de memória
  - ▶ Tipo
  - ▶ Escopo
- ▶ e de funções
  - ▶ Quantidade e tipos de argumentos
  - ▶ Método de passagem de parâmetro (valor, referência, ...)
  - ▶ Tipo de retorno
- ▶ **Acesso eficiente**
  - ▶ Inserção
  - ▶ Extração

# Tratamento e Recuperação de Erros

- ▶ O que fazer quando um erro é detectado?  
(considerando apenas erros léxicos e sintáticos?)
- ▶ **Sobreviver**, se recuperando da seguinte forma
  - ▶ Fazer uma suposição a respeito do erro
  - ▶ Continuar a análise confiando na suposição feita
- ▶ Como sobreviver a um erro léxico?
- ▶ Como sobreviver a um erro sintático?
- ▶ E sobre erros de geração/otimização de código?

# Estrutura Geral de um Compilador



# Geradores de Compiladores

- ▶ Análise Léxica – **lex** e **flex**
- ▶ Análise Sintática – **yacc** e **bison**
- ▶ Gerador de Código

# Projeto de Compilador

## Lançamento da Etapa 0

# Conclusão da Aula de Hoje

- ▶ Temos trabalho pela frente
- ▶ Leituras recomendadas
  - ▶ Capítulo 1 de Aho et. al. (Dragão Roxo ou Vermelho)
    - ▶ Cuidado na versão em português
  - ▶ Capítulo 1 de Price & Toscani (2008)
  - ▶ <http://dinosaur.compilertools.net>  
Toda a turma: Lex | Yacc | Flex | Bison
- ▶ Próxima aula  
Análise Léxica e Expressões Regulares