INSTITUTO MAUÁ DE TECNOLOGIA

# *Curso de Engenharia de Computação*
# *ECM253 – Linguagens Formais, Autômatos e Compiladores*

## Visão geral sobre compilação

Slides da disciplina ECM253 – Linguagens Formais, Autômatos e Compiladores
Curso de Engenharia de Computação
Instituto Mauá de Tecnologia – Escola de Engenharia Mauá
Prof. Marco Antonio Furlan de Souza
<marco.furlan@maua.br>

# *Compiladores e Interpretadores*

- **Definições**
  - **Compiladores** são programas que **traduzem** uma **linguagem-fonte** em uma **linguagem-alvo**:


  - **Interpretadores** são programas que **leem** um **programa executável** e então **produzem os resultados da execução** daquele programa:


  - **Exemplos**
    - O compilador **Java** (`javac`) **traduz** código-fonte **Java** em **código de máquina** para uma **máquina virtual** (JVM – Java Virtual Machine), isto é, um interpretador que faz o papel em software de um microprocessador;
    - A **linguagem C** é tipicamente **compilada** em código de máquina, mas pode ser interpretada ;
    - Um **programa Python** é primeiro compilado (pyc), e depois interpretado;
    - Um **programa Scheme** é normalmente interpretado.

# *Por que estudar compiladores?*

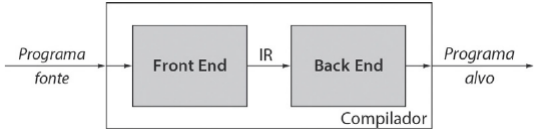
- **Importância**
  - São responsáveis pela eficiência dos programas do usuário e do sistema.
- **Compiladores são interessantes**
  - Muitas das teorias (interessantes!) possuem aplicação prática;
  - Escrever um compilador é desafiador em termos algorítmicos e de engenharia.
- **Aplicações em todo lugar**
  - Linha de comando de terminais, macros, tags de formatação, scripts, máscaras de entrada de dados, linguagens embarcadas (VBA, Python), IDEs de programação, mecanismos de pesquisa etc.

# *Conceitos gerais sobre compiladores*

- **Princípios fundamentais da compilação**
  - **O compilador deve preservar o significado do programa a ser compilado.**:o compilador deve preservar a exatidão, implementando fielmente o "significado" de seu programa de entrada.
  - **O compilador deve melhorar o programa de entrada de alguma forma perceptível**: um compilador tradicional melhora o programa de entrada ao torná-lo executável diretamente em alguma máquina-alvo. Outros melhoram suas entradas de diferentes maneiras (por exemplo, conversor de texto Markdown para HTML).
- **Tarefas gerais de um compilador**
  - Gerar código correto;
  - Gerenciar variáveis e código;
  - Gerar código que o carregador do sistema operacional (*loader*) consiga armazenar na memória e executar;

# *Estrutura de um compilador*

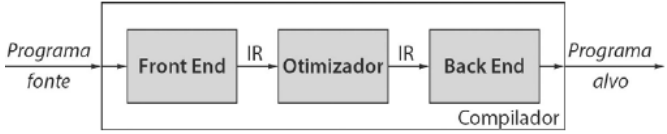
- **Front e Back End**
  - As **operações** que **dependem** somente da **linguagem-fonte** são relacionadas ao **front end** (ou **frente**); aquelas que **dependem** somente da **linguagem-destino** são relacionadas ao **back end** (ou **fundo**):




  - O **front end mapeia** o **código-fonte** em uma **representação intermediária** (**IR** – *intermediate representation*) enquanto que o **back end mapeia IR** em **código destino**;
  - **Vantagens desta separação**: pode-se definir inúmeros back ends para uma mesma linguagem-fonte – por exemplo, o compilador `gcc` (GNU C) **compila** código C (e outros) em uma **mesma representação IR**, que é então traduzida para diversas **linguagens-destino** por diferentes back ends (Intel, ARM etc). Isso se denomina **retargeting**.

# *Estrutura de um compilador*

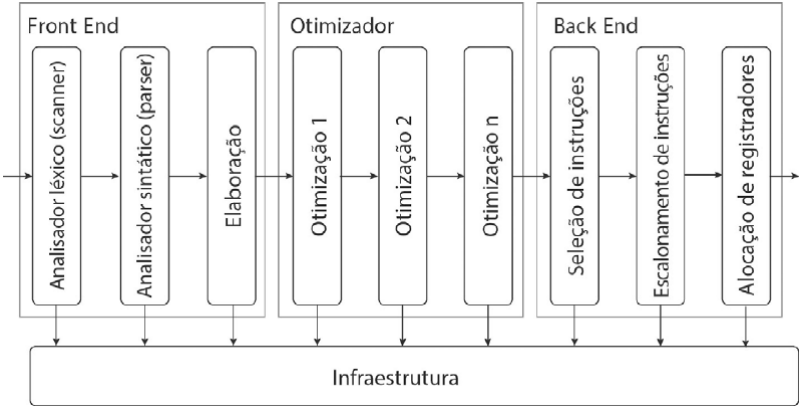
- **Fases de compilação**
  - O modelo do compilador apresentado no slide 5 é denominado compilador de **duas fases** pois o código é transformado duas vezes (fonte para IR e IR para destino);
  - Pode-se melhorar a qualidade do código gerado adicionado-se mais fases. Por exemplo, pode-se adicionar um **otimizador** que gere IR melhorado a partir do IR original:




  - Cada fase, por sua vez, pode ser organizada em **passos**, que representam tarefas específicas a serem realizadas em cada fase.

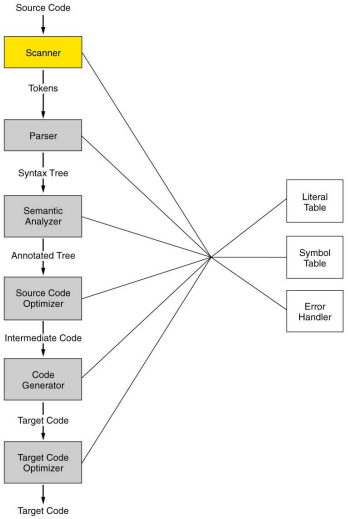
# *Estrutura de um compilador*

- **Passos de compilação**



# *Estrutura de um compilador*

- **Analisador Léxico**

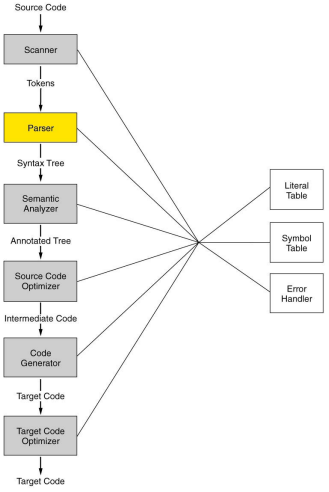



- Também conhecido como "**scanner**" – executa a **análise léxica**.
- **Identifica marcas** ("tokens"), que é uma representação interna conveniente e significativa para o compilador;
- Por exemplo, a **expressão C**, `a[index] = 4 + 2`, poderia ser "tokenizada" assim:

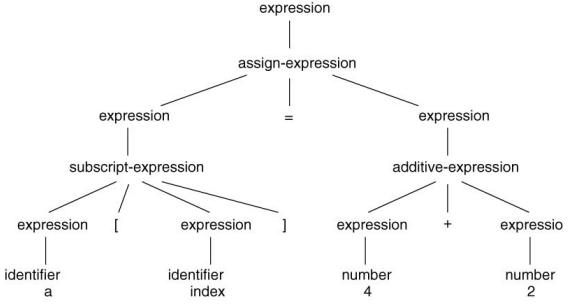
| Marca | Significado |
|---|---|
| **a** | identificador |
| **[** | colchete à esquerda |
| **index** | identificador |
| **]** | colchete à direita |
| **=** | atribuição |
| **4** | número |
| **+** | sinal de adição |
| **2** | número |

# *Estrutura de um compilador*

- **Analisador sintático**

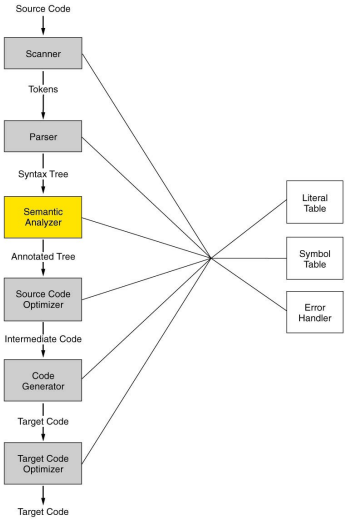



- Também conhecido por "**parser**" – a partir das marcas realiza a **análise sintática** – elementos estruturais do programa;
- Sintetiza a **árvore de análise sintática** ou simplesmente **árvore sintática**.

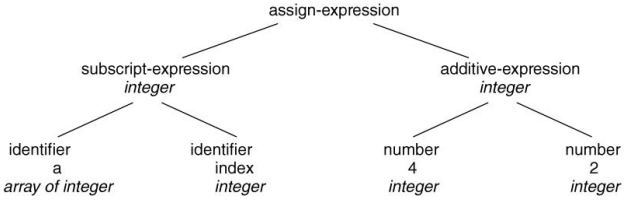


# *Estrutura de um compilador*

- **Analisador semântico** ou **elaboração**

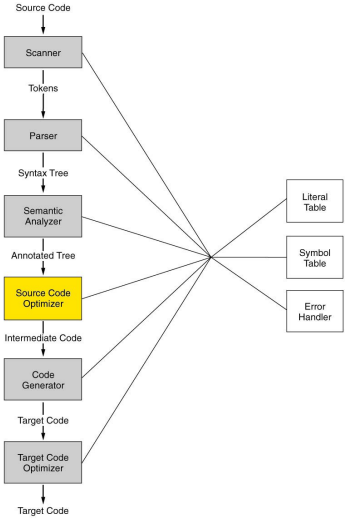



– Extrai o significado (semântica) do programa a partir das árvores sintáticas. Existe a **semântica estática** (tarefa do compilador) e a **semântica dinâmica** (obtida somente com a execução).

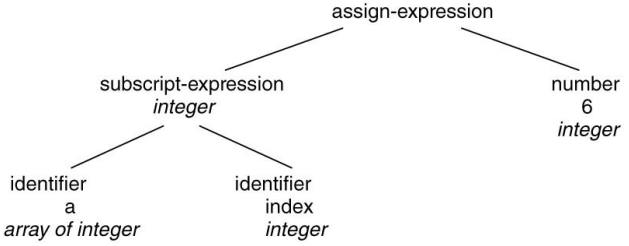


# *Estrutura de um compilador*

- **Otimizador de código-fonte**

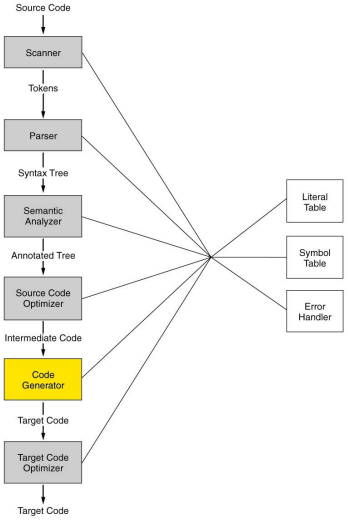



- **Melhora a organização** das **estruturas** obtidas na **análise semântica**. A árvore obtida no exemplo anterior poderia ser "otimizada" substituindo-se o ramo com a expressão `4+2` por um nó com valor `6`;
- É aqui que se produz, no final, **código intermediário – IR**. Podem ser definidas diversas etapas de otimização.



# *Estrutura de um compilador*

- **Gerador de código**

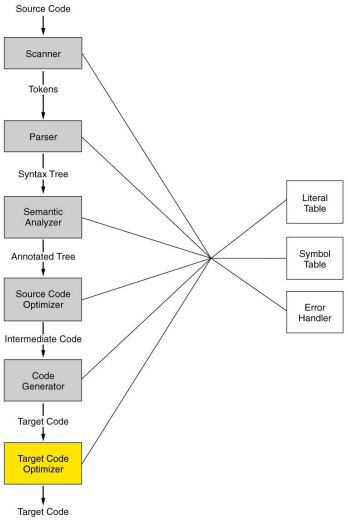



- Na geração do código, a partir do **código intermediário** cria-se o código para a **máquina-alvo**;
- Além do conjunto de instruções da máquina-alvo, é necessário saber, também, os **tamanhos dos objetos de dados** para essa máquina, bem como a ordem dos bits. **Exemplo**:

```
MOV R0, index ;;R0 recebe valor de index
MOV R1, &a    ;;R1 recebe endereço de a
ADD R1, R0    ;;Adiciona R0 a R1
MOV *R1,6     ;;Atribui cte. 6 a R1
```

# *Estrutura de um compilador*

- **Otimizador de código-alvo**




– Tenta **melhorar** o **código-alvo gerado**: escolher outros modos de endereçamento, substituição de instruções lentas, eliminação de operações redundantes, agendamento de instruções, alocação de registradores.

```
MOV R0, index ;;RO recebe valor de index
MOV &a[R0], 6 ;;atribui cte. 6 a end. a + R0
```

# *Estrutura de um compilador*

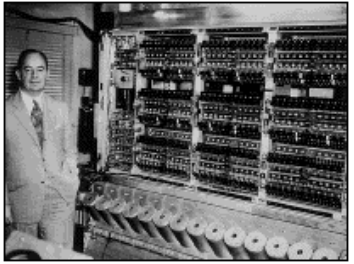
- **Estruturas de dados em um compilador**
  - **Marcas**
    - Normalmente **cada marca** obtida pelo sistema de varredura pode ser **armazenada** em **uma única variável**. Assim, somente é necessário olhar para a frente um único símbolo (chamado de "**lookahead symbol**"). Existem linguagens que utilizam "lookahead symbol" com mais de um caractere;
  - **Árvore sintática**
    - **Composta** dinamicamente (ponteiros) por **nós** (interior e folha), onde cada nó é um **registro** com **informações** provindas do **analisador sintático** e depois o **analisador semântico**. Algumas informações, para salvar espaço, também podem ser alocadas em uma **tabela de símbolos**;
  - **Tabela de literais**
    - Armazena **constantes** e **cadeias de caracteres**.

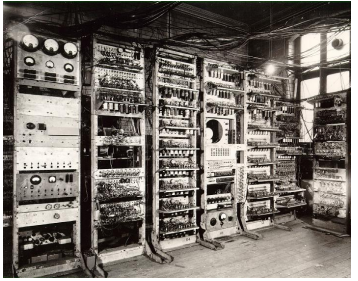
# *Estrutura de um compilador*

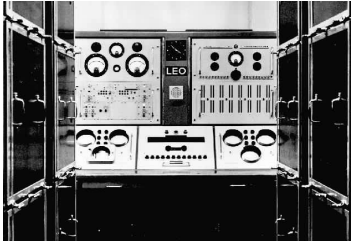
- **Estruturas de dados em um compilador**
  - **Tabela de símbolos**
    - Armazena **informações dos identificadores** (funções, variáveis, constantes, tipos de dados), sendo **acessada** em **quase todas as fases** de compilação. Esquemas como “hashing” são comumente empregados em sua criação;
  - **Código intermediário**
    - Pode ser armazenado como uma **matriz de caracteres** ou um **arquivo-texto temporário** ou ainda como uma **lista ligada de estruturas**. Com otimização, a decisão do tipo de estrutura é essencial;
  - **Arquivos temporários**
    - São utilizados quando não é possível manter o processo de compilação inteiramente na memória, principalmente quando é necessário **revisar** o **código gerado** para determinar **endereços corretos**, como, por exemplo, na tradução de uma estrutura **if-then-else**.

# *Histórico*

- Linha do tempo
  - No início (~1940–1950): **programação por chaves e fios**.








Onde estão o mouse, o teclado e o monitor?

# *Histórico*

- **Linha do tempo**
  - Depois (~1950s), sugiram as **linguagens de montagem** (“assembly”):

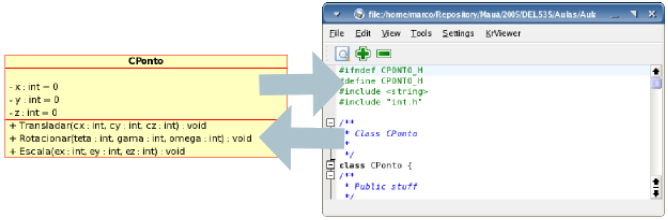

  - **Classificação das gramáticas** (~1950) segundo sua complexidade, por **Noam Chomsky** ⇒ identificar algoritmos para reconhecê-las;
  - Surgimento da **primeira linguagem** de programação (1954): **Fortran**;
  - Estudos para reconhecimento de **linguagens livres de contexto** (aquelas que usamos normalmente) ⇒ **técnicas de análise sintática**. (~1960s–1970s)

# *Histórico*

- **Linha do tempo**
  - De ~1970s até hoje, a **teoria da compilação** tende para:
    - Técnicas de **otimização** de código (melhoria seria mais apropriado);
    - Geração e otimização de código para arquiteturas não convencionais;
    - Programas **geradores de sistema de varredura**. Flex (C/C++) e JFlex (Java) são apenas dois exemplos;
    - Programas **geradores de analisadores sintáticos**. O Bison (C/C++) e Antlr (Java) são dois exemplos;
    - **Máquinas virtuais** tem incentivado cada vez mais o aprimoramento da geração de código.

# Histórico

- **Linha do tempo**
  - De ~1970s até hoje, a **teoria da compilação** tende para:
    - **Ambientes integrados de desenvolvimento (IDE)**: integração do compilador com um conjunto de **ferramentas de desenvolvimento** como **depuradores**, **geradores de perfil** etc. Técnicas de **interpretação** são utilizadas maciçamente por ambientes IDEs atuais, quando se executa o que se conhece por "**round-trip software engineering**" ⇒ do código para o projeto e vice-versa.



# *Programas relacionados a compiladores*

- **Interpretadores**
  - Existem **dois tipos básicos** de interpretadores:
    - **Aqueles** que **traduzem e executam prontamente** as instruções em código-fonte (ex BASIC antigo);
    - **Aqueles** que **geram** um "pseudocódigo" (**p-code**), com alguma otimização, e **depois interpretam** esse código. Esse é o caso dos primeiros compiladores Pascal e, mais recentemente, do Visual Basic.
- **Montadores** ("assemblers")
  - São programas que **traduzem** uma **representação simbólica** do **código de máquina** (e mais um conjunto de diretivas para organizar o programa) **em código-objeto** da máquina

# *Programas relacionados a compiladores*

- **Carregadores** (“loaders”)
  - É o programa que todo sistema operacional possui ⇒ é o responsável por **carregar o programa** a ser executado na memória, **realocando** convenientemente seu **código** e **dados**.
- **Organizadores** (“linkers”)
  - **Sistemas** de médio e grande porte são construídos a partir de um conjunto de **unidades compiladas separadamente**;
  - O organizador tem como finalidade **agrupar os objetos** gerados pelo processo de compilação, bem como **agregar bibliotecas** que essas unidades necessitem, criando o executável final.

# *Programas relacionados a compiladores*

- **Depuradores** (“debuggers”):
  - Auxiliam na tarefa de **encontrar erros**, permitindo a definição de **pontos de interrupção** (“break-points”) e a **verificação** do valor de **variáveis** e **expressões**.
- **Pré-processadores**
  - São ativados pelo compilador em uma **fase anterior à da compilação**;
  - Nesta fase, normalmente há a resolução de **macros** pela substituição literal de seus conteúdos, bem como a execução de comandos denominados **diretivas**, que instruem o compilador em alguma ação. Instruções que alteram o funcionamento do compilador, denominadas de **pragmas**, também figuram no pré-processamento.

# *Programas relacionados a compiladores*

- **Editores**
  - São integrados com compiladores em **ambientes IDE**.
- **Geradores de perfil** (“profilers”)
  - São softwares que nos auxiliam na **avaliação do desempenho** de um programa, permitindo determinar pontos críticos na sua eficiência.
- **Gerenciadores de projeto**
  - São softwares que acrescentam outras funcionalidades importantes ao ambiente de desenvolvimento, como **sistemas de controle de versão**.

# *Outros aspectos*

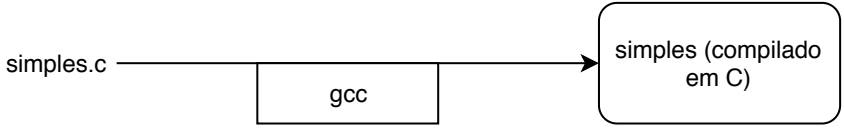
- **Definição da linguagem e compiladores**
  - É necessário definir a linguagem (**manual de referência da linguagem**) antes de implementá-la;
  - Isso envolve, inclusive, a verificação de **ambiguidades**;
  - Linguagens amplamente utilizadas devem possuir um **padrão** (ANSI, ISO etc);
  - A definição pode ser realizada em diversos **graus de formalização**, de **diagramas de sintaxe** e **EBNF**, até **semântica denotacional**.

# *Outros aspectos*

- **Opções e interfaces de um compilador**
  - São mecanismos para **interfacear** com o **sistema operacional** e fornecer opções ao usuário;
  - **Opções** que **instruem** o **compilador** para **modificar** seu **comportamento padrão** são denominadas de **pragmáticas** ou simplesmente **pragmas**.
- **Tratamento de erros**
  - **Erros** podem ser **detectados** durante quase todas as **fases** da compilação – **erros estáticos**;
  - A coleta e o tratamento de erros identificados nas fases é tarefa de um **sistema de tratamento de erros**;
  - Uma boa linguagem também deve fornecer algum tipo de **tratamento de erros dinâmicos**.

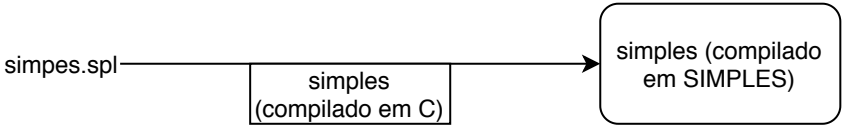
# *Como criar linguagens e compiladores*

- Supor que se deseja **criar** um compilador para a **linguagem** (hipotética) `SIMPLES`. Supondo que se tem apenas o compilador `gcc`, tem-se o processo a seguir (conhecido como **diagrama-T**):



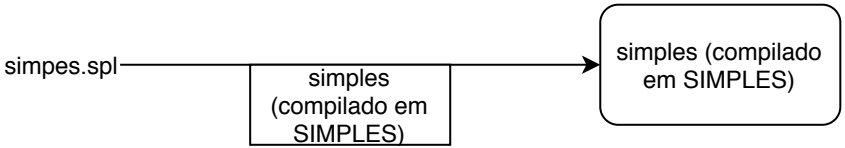
# *Como criar linguagens e compiladores*

- Agora, deseja-se projetar um novo compilador `SIMPLES` escrito nesta mesma linguagem, para adicionar características adicionais que não se encontram em C:



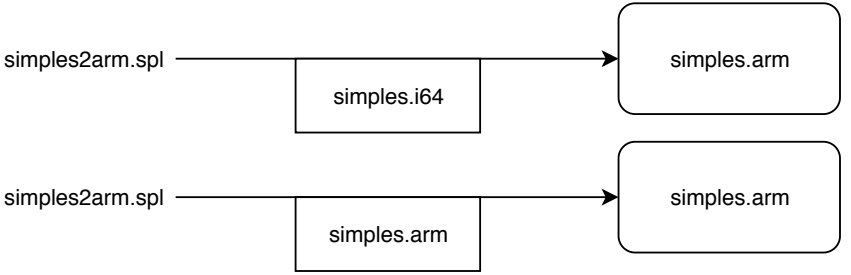
# *Como criar linguagens e compiladores*

- Com o compilador escrito em **SIMPLES**, pode-se criar um novo compilador `SIMPLES`, mais otimizado com este compilador:



# *Como criar linguagens e compiladores*

- Pode-se escrever compiladores para outras arquiteturas além da atual:



# *Referências bibliográficas*

COOPER, K.; TORCZON, L. **Construindo compiladores**. 2. ed. Rio de Janeiro: Elsevier, 2014.

LOUDEN, K. **Compiladores: princípios e práticas**. [S.l.]: Pioneira Thomson Learning, 2004.