

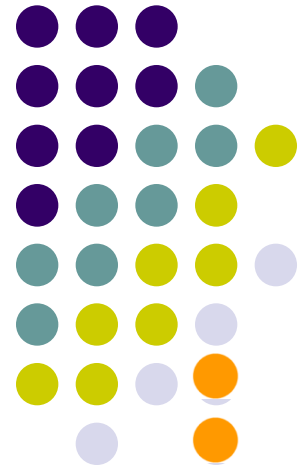
Arquitetura de Software

Interpretadores

Compiladores e Interpretadores

José Motta Lopes

josemotta@bamplicom

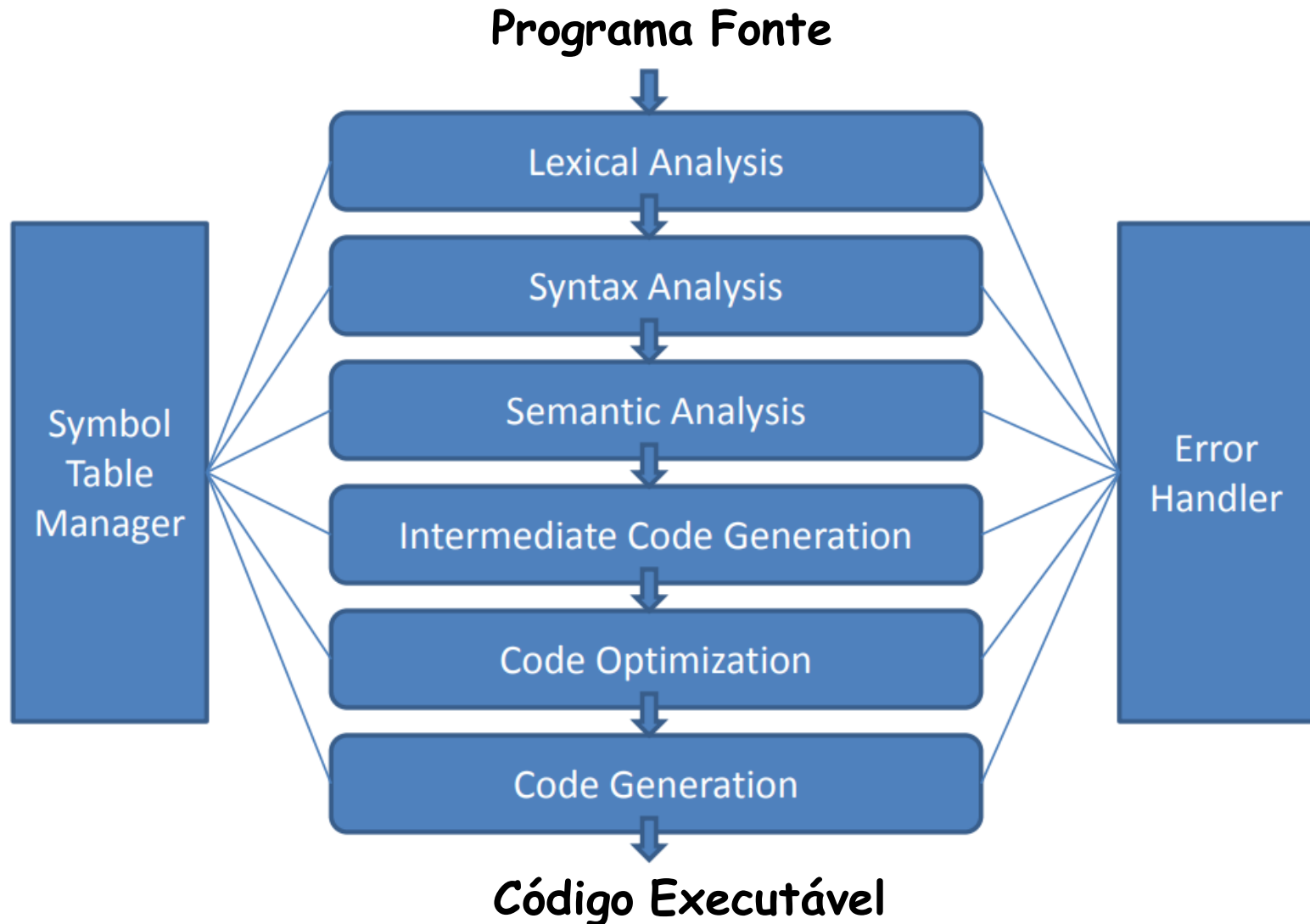


Agenda

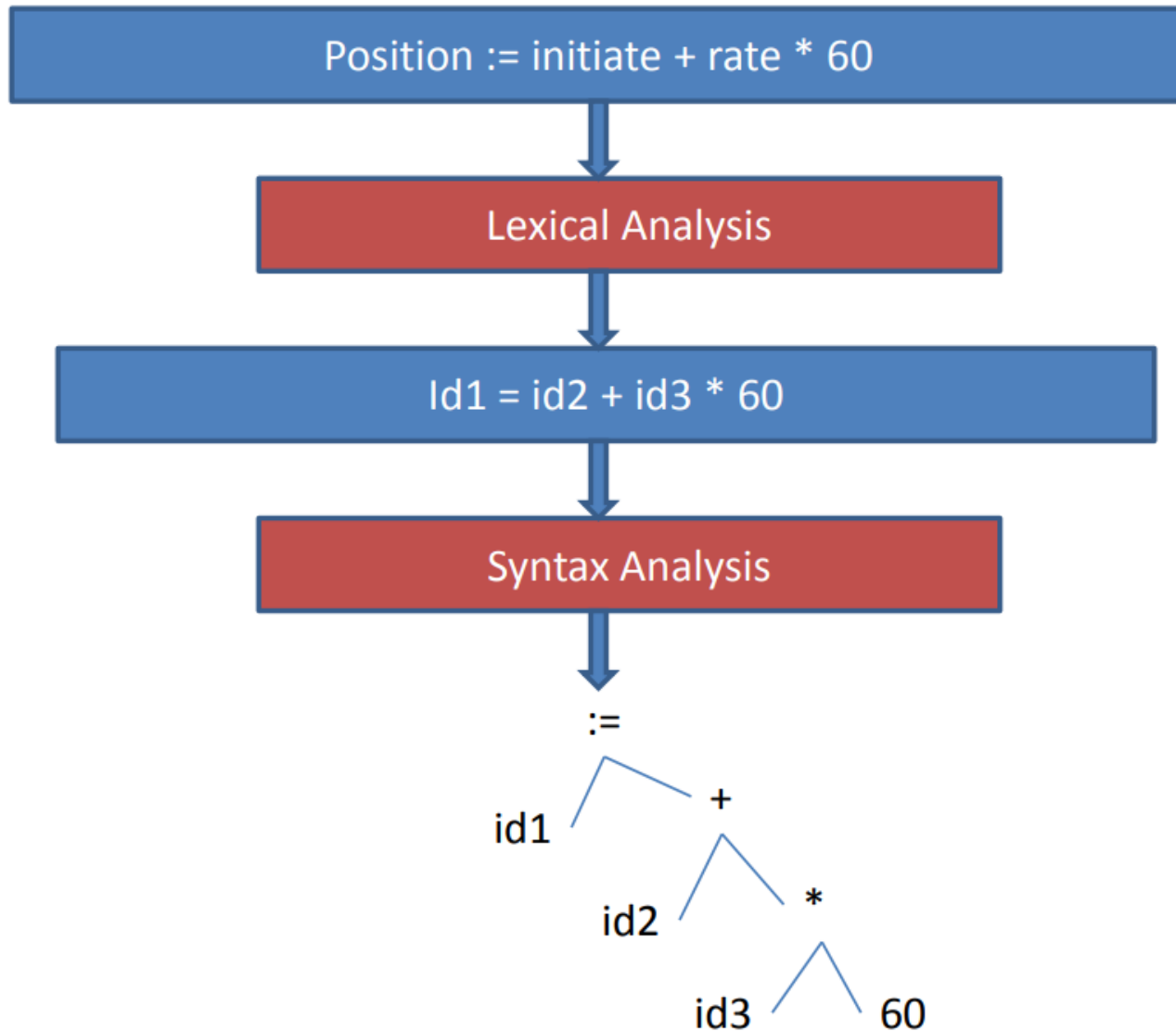


- **Compilador**
 - Tabela de Símbolos
 - Tratamento de Erros
 - Análise
 - Geração Código Intermediário
 - Otimização de Código
 - Geração de Código
- **Aspectos da Compilação**
- **Linguagem de Programação**
 - Tipo de Dado
 - Estrutura de Dados
 - Regras de Escopo
 - Estruturas de Controle
 - Alocação Estática e Dinâmica de Memória
- **Interpretador**
 - Compilador x Interpretador
 - Porque usar Interpretador?
 - Componentes
 - Interpretadores Puros
 - Interpretadores Impuros

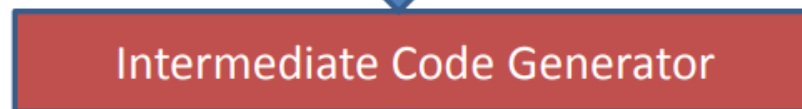
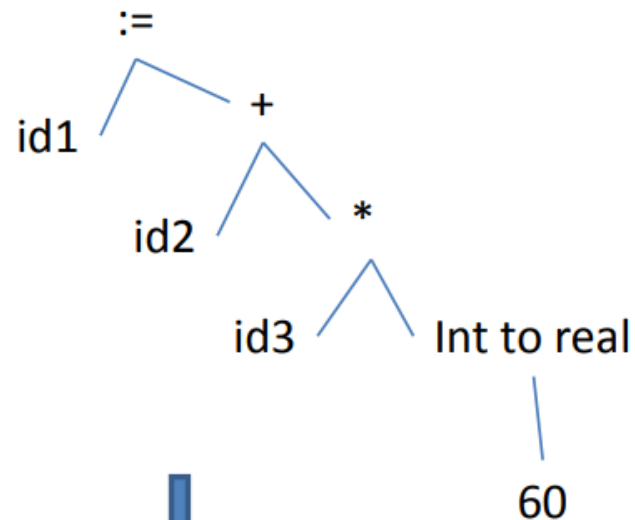
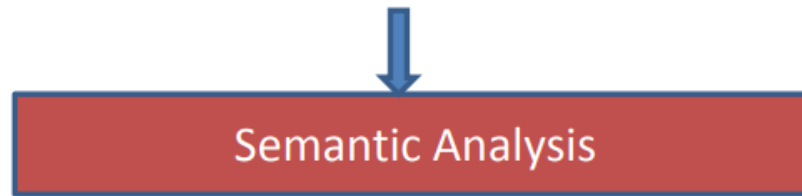
Compilador



Exemplo



Exemplo



```
Temp1 := int to real (60)
Temp2 := id3 * Temp1
Temp3 := id2 + Temp2
id1 := Temp3
```

Exemplo

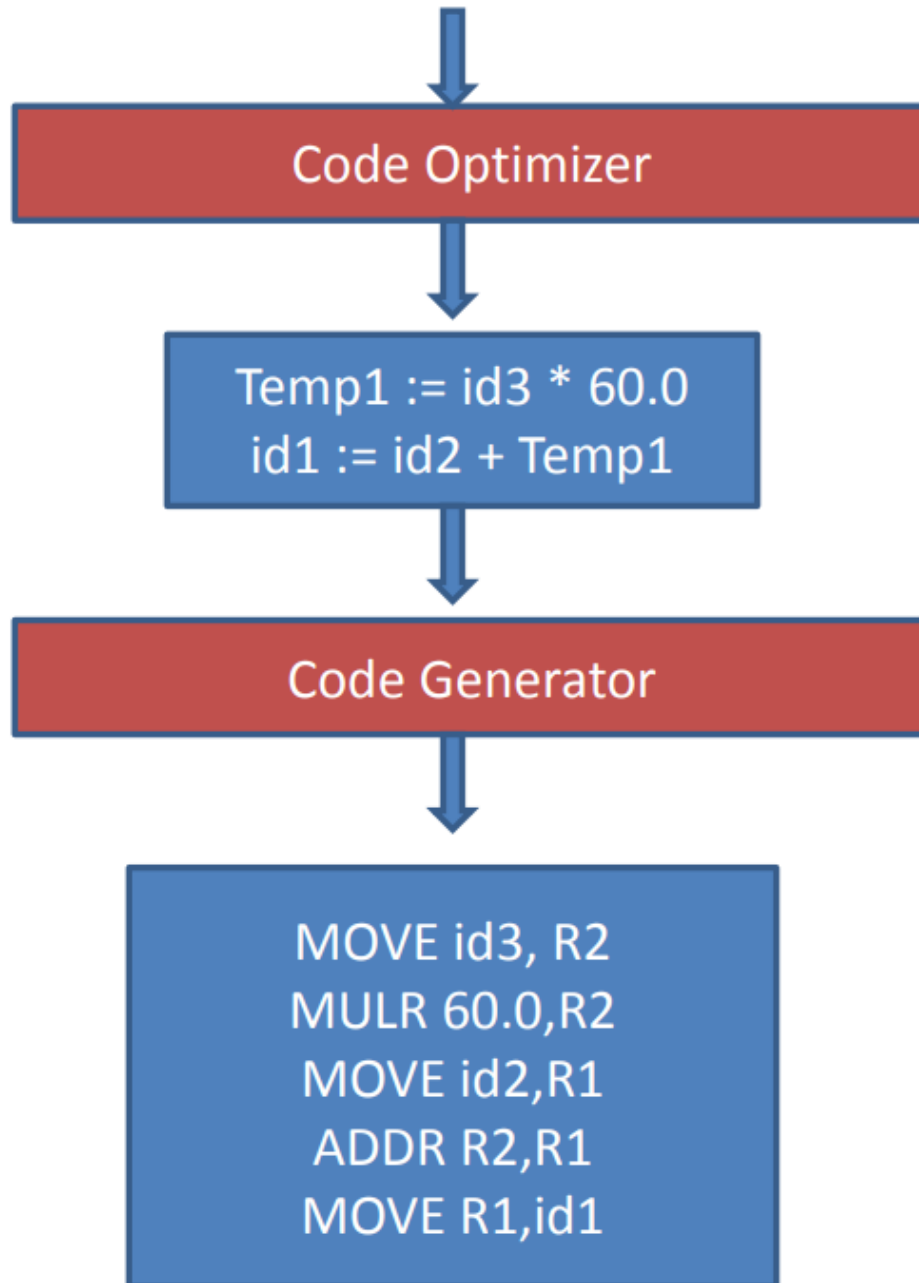
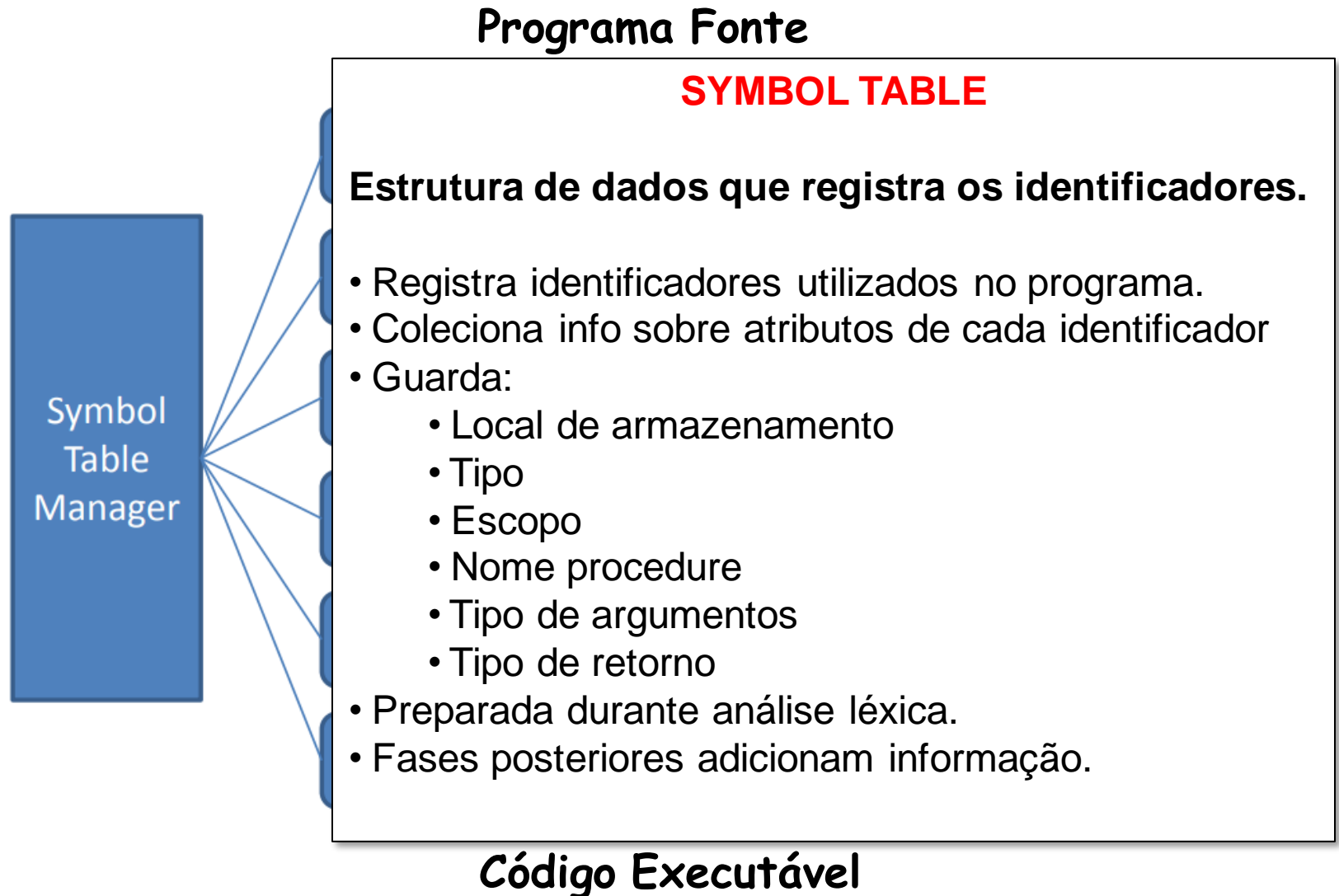




Tabela de Símbolos





Tratamento de Erros

Programa Fonte

ERROR HANDLER

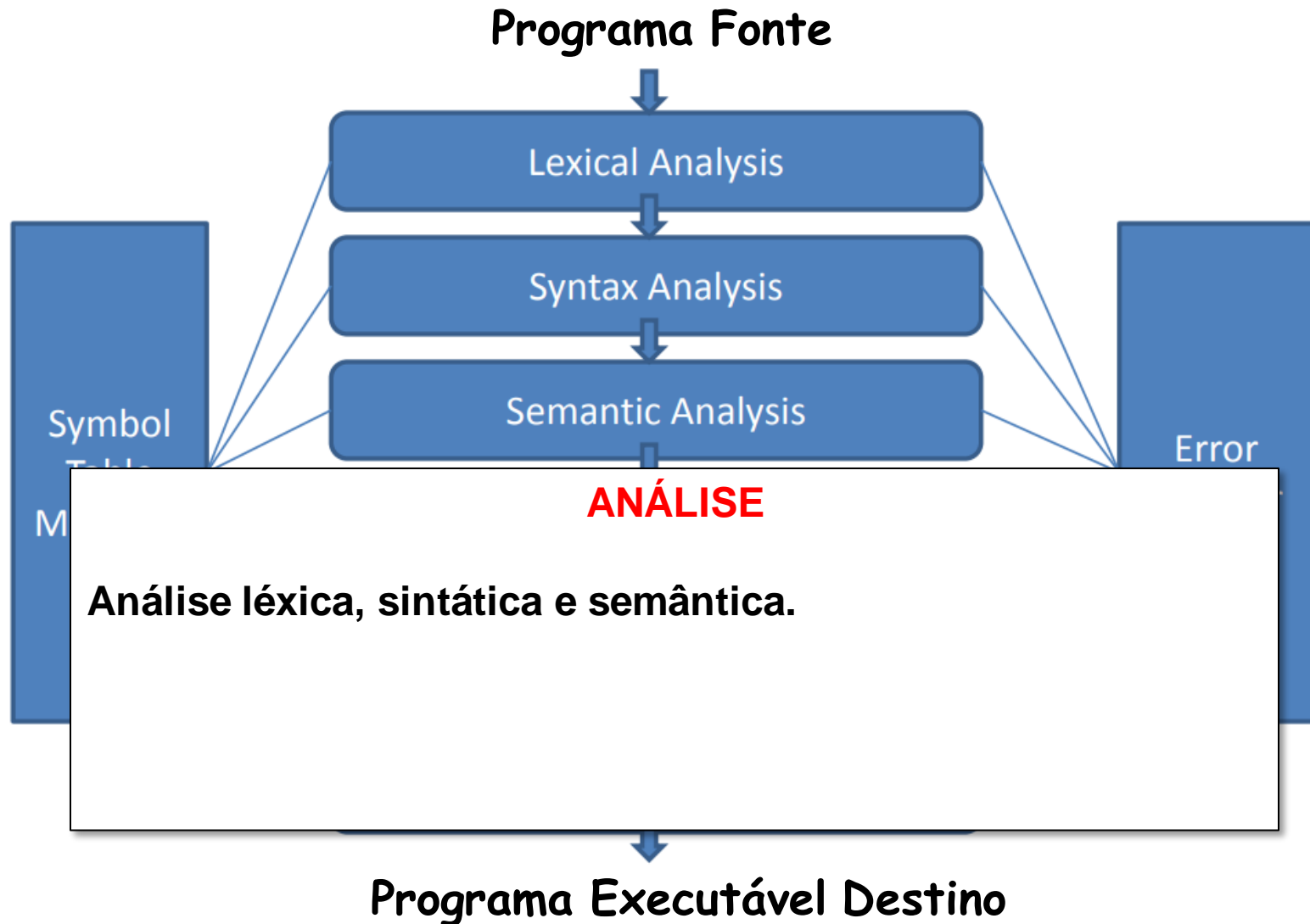
Deteção e relato de erros

- Compilação só ocorre após resolver erros gerados pela análise léxica, sintática e semântica.
- Se o código não gera tokens, a estrutura é violada e o erro detectado pela análise léxica.
- Se a sintaxe é violada, o erro é detectado pela análise sintática.
- Na análise semântica, o compilador tenta detectar construções com estrutura sintática correta.

Error
Handler

Código Executável


Análise





Análise Léxica

- Converte as linhas de caracteres de programas fonte ou páginas web em uma sequência de símbolos léxicos – tokens.
- O programa que faz a análise léxica é também conhecido como lexer ou scanner.
- Gerador de análise léxica mais conhecido é o LEX, escrito em 1975 e que se tornou padrão em sistema Unix.
 - Lex processa entrada especificando o analisador léxico;
 - Gera código fonte implementando o lexer na linguagem C.



```
%%  
    /** Rules section **/  
  
    /* [0-9]+ matches a string of one or more digits */  
    [0-9]+ {  
        /* yytext is a string containing the matched text. */  
        printf("Saw an integer: %s\n", yytext);  
    }  
  
    .|\n { /* Ignore all other characters. */ }  
  
%%
```

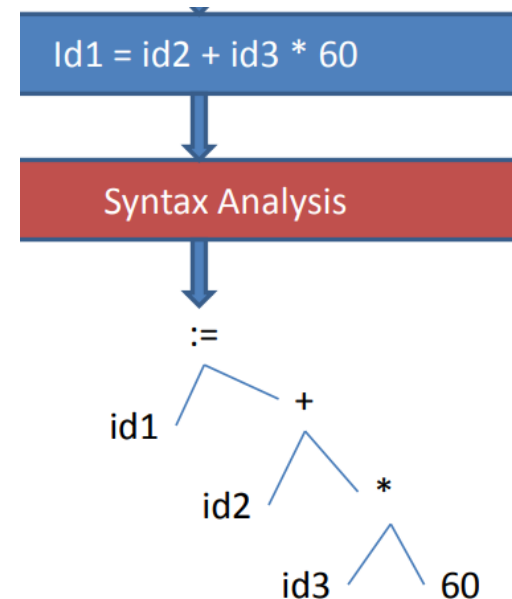
abc123z. !&*2gj6

Saw an integer: 123
Saw an integer: 2
Saw an integer: 6



Análise Sintática

- Análise de string de símbolos que adere à regras gramaticais.
- Análise formal de uma sentença em seus constituintes.
- O programa que faz análise sintática é também conhecido como parser.
- Yacc (Yeat Another Compiler-Compiler) é um gerador de parser conhecido do mundo Unix, que opera junto com o Lex.
- Original em C mas já foi reescrito:
 - Pascal
 - Java
 - Python
 - Ruby
 - Go
 - Lisp
 - Erlang, etc.





Análise Semântica

- **Verifica se há significado em uma sequência de tokens.**
- **Determina tipo dos valores e sua interação com as expressões.**
- **Identifica declarações e escopos e acrescenta informações complementares à tabela de símbolos.**
- **Inspeciona identificadores e literais para checar se os resultados são compatíveis com a definição da linguagem.**
- **Exemplos típicos de erros semânticos são:**
 - Uma variável não declarada;
 - Uma multiplicação entre tipos de dados diferentes;
 - Atribuição de um literal para outro tipo, como um inteiro em uma string ou vice-versa.



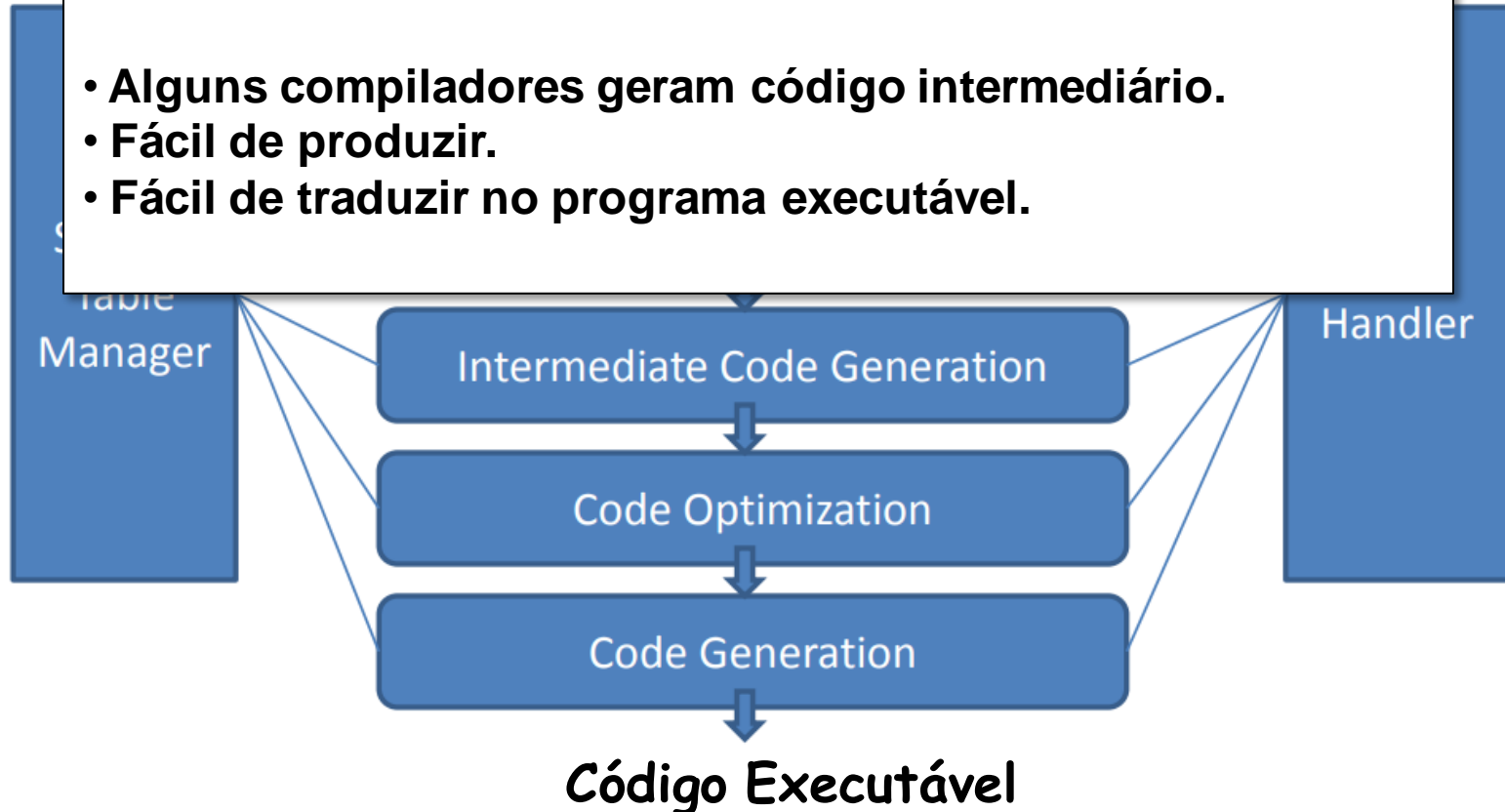
Geração Código Intermediário

Programa Fonte

Intermediate Code Generation

Representação Intermediária do programa fonte.

- Alguns compiladores geram código intermediário.
- Fácil de produzir.
- Fácil de traduzir no programa executável.





Otimização de Código

Programa Fonte



Code Optimization

Tenta melhorar código intermediário.

- Resulta em código de máquina mais rápido.
- Otimização melhora o tempo de execução do programa.
- Mas não diminui a velocidade da compilação.



Code Optimization



Code Generation



Código Executável



Geração de Código

Programa Fonte



Code Generation

Fase final que gera o código executável na máquina alvo.

- Consiste de:
 - código de máquina relocável
 - ou código assembly
- Localizações de memória são selecionadas para cada variável usada no programa.

Code Generation



Código Executável

Aspectos da Compilação



Compiladores & Interpretadores fazem a ponte entre o domínio da linguagem de programação (LP) e o domínio da execução.



Linguagem de Programação

- **Características da Linguagem**
 - **Tipos de Dados**
 - **Estrutura de Dados**
 - **Regras de Escopo**
 - **Estruturas de Controle**
- **Alocação de Memória**
 - **Alocação Estática**
 - **Alocação Dinâmica**



Tipo de Dado

- **Especificação de:**
 - **Valores legais para variável do tipo**
 - **Operações legais nos valores legais do tipo**
- **Verificar legalidade das operações pelo tipo dos seus operandos.**
- **Usar operações de conversão de tipos sempre que possível.**
- **Usar sequência de instruções apropriada.**

```
var
  x,y : real;
  i,j : integer;
Begin
  y := 10;
  x := y + i;
  Type conversion of i is needed.
  i : integer;
  a,b : real;
  a := b + i;
```



Estrutura de Dados

- LP permite declarar Data Structure (DS).
- Compilador referencia elementos DS na memória.
- Pode ser complexo.
- Usuário pode definir DS.
- Requer mapeamento de combinação de DS.
- Dois tipos de mapeamento:
 - Array reference
 - Field of Record

```
Program example (input,output);
  type
    employee = record
      name : array [1..10] of character;
      sex : character;
      id : integer
    end;
    weekday = (mon,tue,wed,thur,fri,sat,sun);
  var
    info : array [1..500] of employee;
    today : weekday;
    i,j : integer;
  begin {main program}
    today := mon;
    info[i].id := j;
    if today = tue then....
  end
```



Regras de Escopo

- Determina a acessibilidade de variável declarada em blocos diferentes do programa.
- Para determinar a acessibilidade da variável, o compilador realiza operações:
 - Análise de escopo
 - Resolução de Nomes

```
A{  
    x,y,z : integer;  
    B{  
        g : real;  
        C{  
            h,z : real;  
        }C  
    }B  
    D{  
        i,j : integer;  
    }D  
}A
```

Block	Accessibility of Variables	
	Local	Non-Local
A	xA,yA,zA	
B	gB	xA,yA,zA
C	hC, zC	xA,yA,gB
D	iD, jD	xA, yA, zA



Estruturas de Controle

- Coleção de comandos da linguagem para alterar o fluxo de execução.
- Inclui:
 - Transferência condicional
 - Execução condicional
 - Controle iterativo
 - Chamada de procedure
- Compilador deve garantir a não violação da semântica.

```
Eg: for i = 1 to 100 do
      begin
          lab1 : if i = 10 then
                .....
                .....
      End
```

Não se pode
transferir
controle
para *lab1* de
fora do loop!



Alocação de Memória



- **Determina os requisitos de memória para representar os itens de dados**
- **Determina a alocação de memória para implementar tempo de vida e escopo dos itens de dados**
- **Determina o mapeamento de memória para acessar o valor de itens de dados não escalares**
- **Associa os atributos do endereço de memória do item de dados e o endereço da área de memória**
- **Alocação estática e dinâmica de memória**
- **Alocação e acesso de arrays**

Alocação Estática de Memória



- **Memória é alocada para variável antes da execução iniciar.**
- **Realizada durante compilação.**
- **Na execução, nenhuma alocação/dealocação é realizada.**
- **Alocação para variável existe mesmo se unidade do programa não está ativa**
- **Variável permanece alocada permanentemente.**
- **Vantagem da simplicidade e acesso mais rápido.**
- **Desvantagem do desperdício de memória.**
- **Exemplo: Fortran.**

Memory
Code(A)
Data (A)
Code (B)
Data (B)
Code (C)
Data (C)

Alocação Dinâmica de Memória



- Memória é alocada no instante da execução.
- Realizada durante execução.
- Alocação/dealocação é realizada somente na execução.
- Alocação para variável ocorre somente se unidade do programa está ativa.
- Estado da variável varia entre alocada e livre.
- Vantagem de recursão suportar dinamicamente o tamanho do DS.
- Menos desperdício de memória.
- Exemplo: PL/I, Pascal, Java, C#.

Memory:
Only A is
active

Code(A)

Code (B)

Code (C)

Data (A)

Memory:
A calls B.
A & B is
active

Code(A)

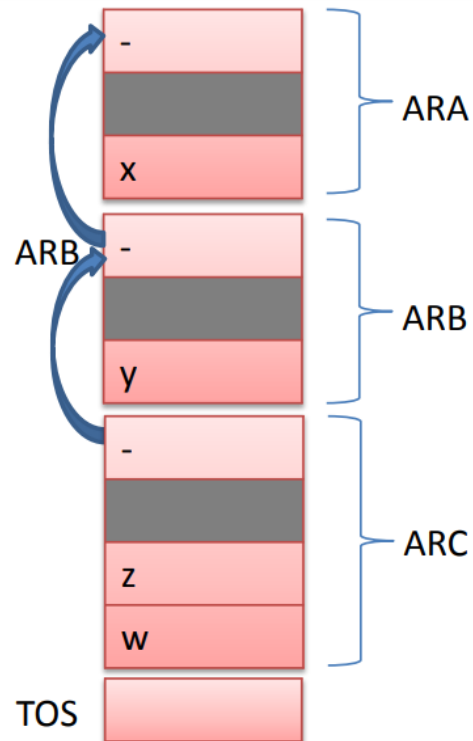
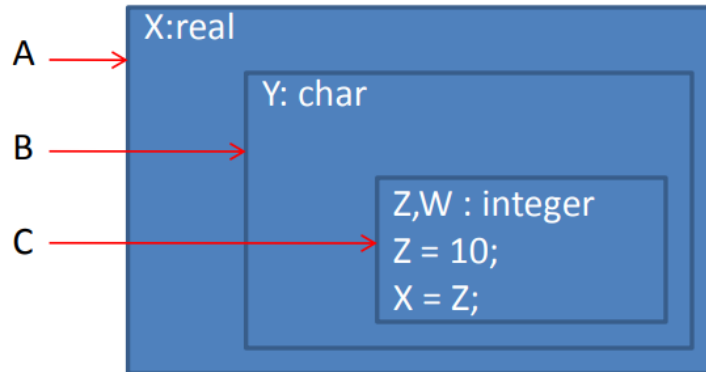
Code (B)

Code (C)

Data (A)

Data (B)

Alocação Dinâmica de Memória



Extended Stack Model

Escopo determinado por Pointers

AR: Activation Record

ARB: Activation Record Block

Alocação Dinâmica de Memória



Array allocation & Access

- $A[5,10]$. 2D array arranged column wise.
- Lower bound is 1.
- Address of element $A[s1,s2]$ is determined by formula:

$$Ad.A[s1,s2] = Ad.A[1,1] + \{ (s2-1) \times n + (s1-1) \} \times k$$

- n is number of rows.
- k is size, number of words required by each element.

$A[1,1]$
-
-
$A[5,1]$
$A[1,2]$
-
$A[5,2]$
-
-
$A[1,10]$
-
$A[5,10]$



Interpretador

- **Evita o overhead da compilação**
- **Desvantagem de ser caro em termos de CPU**
- **Ciclo do Processo:**
 - Busca a instrução.
 - Analisa o “statement”.
 - Determina seu significado.
 - Executa o significado do “statement”.

Compilador x Interpretador



COMPILADOR

- Durante compilação, análise do statement é seguida de geração de código.
- Compilador converte em exe somente uma vez.
- Infraestrutura desenvolvida devagar, de uma só vez.
- Erros não são fáceis de resolver e depurar.

INTERPRETADOR

- Durante interpretação, análise é seguida de ações para implementação.
- Não se pode rodar um programa sem interpretação.
- Desenvolvimento repetitivo mais rápido.
- Erros podem ser resolvidos facilmente.

Compilador x Interpretador



COMPILADOR

- Melhor para linguagens estáticas.
- O compilador é necessário apenas uma vez.
- O compilador é carregado apenas na primeira vez.
- Menor custo a longo prazo.

INTERPRETADOR

- Melhor para linguagens dinâmicas.
- Cada vez que programa é executado, o interpretador é necessário.
- O interpretador é necessário em cada carga.
- Mais caro.



Porque usar Interpretador?

- **Simplicidade.**
- **Eficiência.**
- **Preferido durante desenvolvimento do programa.**
- **Idem quando programas não são executados frequentemente.**
- **Bom para escrever programas para editor, interface do usuário ou desenvolvimento sistema operacional.**

Componentes



- **Tabela de Símbolos**: guarda informações referentes às entidades presentes no programa.
- **Data Store**: contem valor dos itens de dados declarados.
- Rotinas de **manipulação de dados**: conjunto de rotinas para cada ação legal de manipulação de dados .
- **Vantagens**:
 - Significado do “statement” tem implementação simples;
 - Evita geração de instruções em código de máquina;
 - O interpretador é codificado em linguagem de alto nível.



Interpretadores Puros

- Programa é mantido no formato fonte por toda a interpretação.
- Tem a desvantagem de overhead substancial na interpretação do statement.
- Elimina maior parte da análise na interpretação, exceto análise de tipo.



Interpretadores Impuros

- Realizam processamento intermediário do programa fonte, reduzindo overhead da análise durante a interpretação
- Pré-processador converte programa para código intermediário que é usado durante a interpretação
- Código intermediário pode ser analisado mais eficientemente que o programa fonte
- Interpretação fica mais rápida
- Por gerar código intermediário, tem a desvantagem de ter que pré-processar o programa inteiro após quaisquer modificações
- Acarreta um overhead fixo no início da interpretação
- Elimina a maior parte da análise durante a interpretação

Aspectos da Compilação



Compiladores & Interpretadores fazem a ponte entre o domínio da linguagem de programação (LP) e o domínio da execução.