Introdução à Software Básico: Estrutura e Gramática de Tradutores

Departamento de Ciência da Computação
Instituto de Ciências Exatas
Universidade de Brasília

Sumário

Conceitos Básicos

Da linguagem de alto-nível ao código objeto

Definição

Código Objeto:

- Também conhecido como Módulo Objeto, o código objeto é gerado pelo processo de tradução.
- De forma geral o código objeto é uma sequência de instruções usualmente em linguagem máquina, porém em alguns compiladores o código objeto é gerado em uma linguagem intermediária (como *Register transfer language*)
- O código objeto serve de entrada para o ligador.

Da linguagem de alto-nível ao código objeto

- Linguagens de alto nível são descritas por gramáticas.
- Uma gramática descreve a sintaxe de uma linguagem.
- Uma sintaxe descreve a forma dos **enunciados válidos**.
- Enunciados escritos pelo programador são comparados com as formas válidas descritas na gramática.
- Um enunciado em um programa fonte é composto por uma seqüência de **tokens**.
  - Os tokens são os blocos fundamentais em uma linguagem.
  - Um token é um string de um ou mais caracteres que possui um significado como grupo.
  - Um token pode ser uma palavra-chave, o nome de uma variável, um inteiro, um operador aritmético, etc.

Da linguagem de alto-nível ao código objeto

- A **análise léxica** é responsável por ler os enunciados no programa fonte, analisar uma seqüência de caracteres para reconhecer, formar e classificar os tokens.
- A parte (programa) responsável pela análise léxica é chamado de analisador léxico ou **scanner**
- Após a análise léxica, a **análise sintática** (ou **parsing**) verifica se cada enunciado é reconhecido como uma construção válida na linguagem.
- O **Parser** cria uma estrutura de dados, normalmente conhecida como **paser tree**, para verificar se a sintaxe de um determinado token esta correta.
- A **análise semântica** trata do inter-relacionamento entre partes distintas do programa
- Os últimos passos a serem executados são relacionados à geração do código objeto

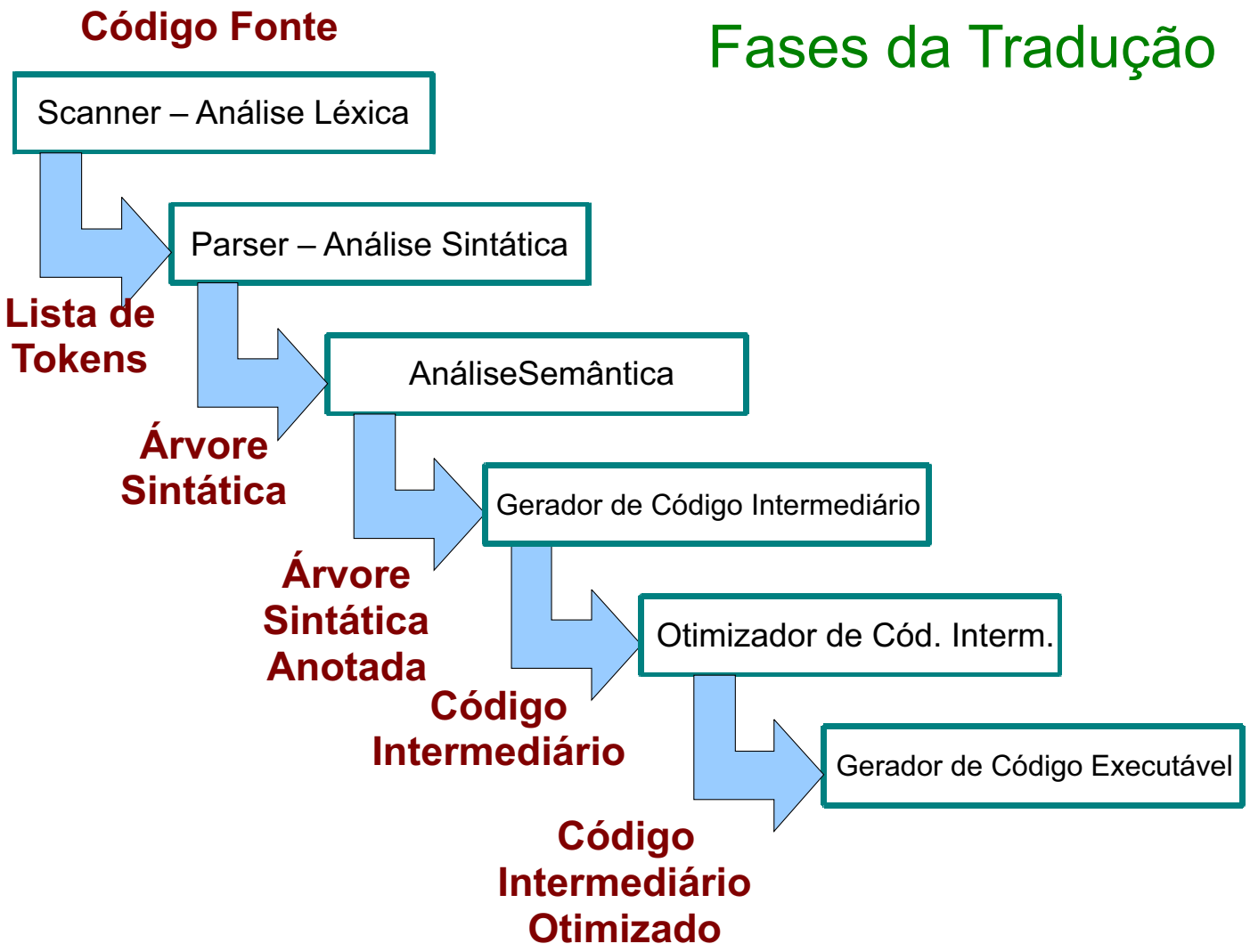


Figura: Fases da Tradução

Tradutores

- Como vimos na aula anterior, os tradutores são programas que convertem um programa fonte escrito em uma linguagem (fonte) para uma outra linguagem (alvo).

Figura: Fases da Tradução

- Para tanto, o tradutor deverá "entender" a linguagem fonte para então poder gerar a linguagem alvo

Tradutores

- Normalmente, o tradutor também está "preparado" para não aceitar determinados erros
- Caso um erro seja identificado, o tradutor interromperá o processo e emitirá uma mensagem de erro para que o programador possa consertá-lo.
- Sendo assim, um tradutor deve detectar erros feitos pelo programador, corrigir ou indicar o melhor caminho para a correção do programa, e só então traduzir para linguagem alvo.

Gramáticas

- Descrição formal da sintaxe de uma linguagem, ou seja, a forma como programas e enunciados são escritos nessa linguagem.
- A gramática não descreve a semântica (o significado) dos enunciados.
- Gramáticas são escritas usando-se diferentes notações.
- As regras gramaticais definem as construções da linguagem e podem ser descritas através de regras de produção
- Exemplo: Backus Naur Form (BNF) e W-grammar (Van Wijngaarden grammar).

Backus Naur Form

- BNF é uma das duas notações principais para gramáticas livres de contexto. Sendo a outra a W-grammar.
- Existem algumas extensões a BNF como Extended Backus–Naur Form (EBNF) and Augmented Backus–Naur Form (ABNF).
- BNF especifica que a regra de produção deve seguir o seguinte formato:
- $< symbol >= \_expression\_$
  - Onde $< symbol >$ é um símbolo não terminal, e $\_expression\_$ consiste de uma sequência de símbolos separados por operadores.

Backus Naur Form

- **símbolos terminais** são símbolos que fazem parte do código fonte como os operadores +,-,=,*,—,etc.
- **símbolos não terminais** são símbolos que são definidos por regras e devem ser substituídos por um determinado valor.

Backus Naur Form

- Como exemplo, tomemos as seguintes regras de produção que geram comandos de atribuição e comandos iterativos.
- Os símbolos não terminais estão delimitados por "<" e ">".
  - <comando> → <while> | <atrib> | ...
  - <while> → while <expr_bool> <comando>
  - <atrib> → <variável> = <expr_arit>
  - <expr_bool> → <expr_arit> < <expr_arit>
  - <expr_arit> → <expr_arit> + <termo> | <termo>
  - <termo> → <número> | <variável>
  - <variável> → I | J
  - <número> → 100

Exemplo de BNF

Algoritmo 1 Endereço de Correio

1: <Endereço> = <nome><detalhes><CEP>
2: <nome> = <pessoal><sobrenome> “ , ”<opt-suffixo> | <pessoal><nome>
3: <pessoal> = <pimeiro-nome> | <inicial>“ . ”
4: <detalhes> = <rua><número><opt-apto><cidade><estado>
5: <CEP> = <némero-CEP>
6: <opt-suffix-part> = Sr|Sra|“ ”
7: <opt-apto> = <número-apto> |“ ”

Exemplo de BNF

Endereço de Correio

- O exemplo anterior indica que um Endereço é formado por três partes: nome, detalhes e CEP.
- O nome é formado por: pessoal, sobrenome o símbolo “ , ”e um sufixo. Ou pode ser formado por uma parte pessoal e uma parte nome (indicando recursão, para pessoas como mais de um nome).
- O nome é formado ou pelo primeiro nome ou pela inicial seguida de ponto.
- o sufixo é formado ou por Sr, ou por Sra ou pode ser deixado em branco.

Estrutura Genérica dos Tradutores

- As fases dos tradutores podem ser divididas em dois grupos.
- Fases de **ANÁLISE**: Divide o programa fonte nas partes constituintes e cria uma representação intermediária estruturada
- Fases de **SÍNTESE**: Constrói o programa objeto desejado, a partir da representação intermediária.

Estrutura Genérica dos Tradutores

- Os tradutores de linguagens de programação, em geral, são bastante complexos.
- Mesmo assim, existe um consenso sobre a estrutura básica que um tradutor deve ter.
- Essa estrutura é independente da linguagem a ser traduzida ou do programa objeto a ser gerado.

Estrutura Genérica dos Tradutores

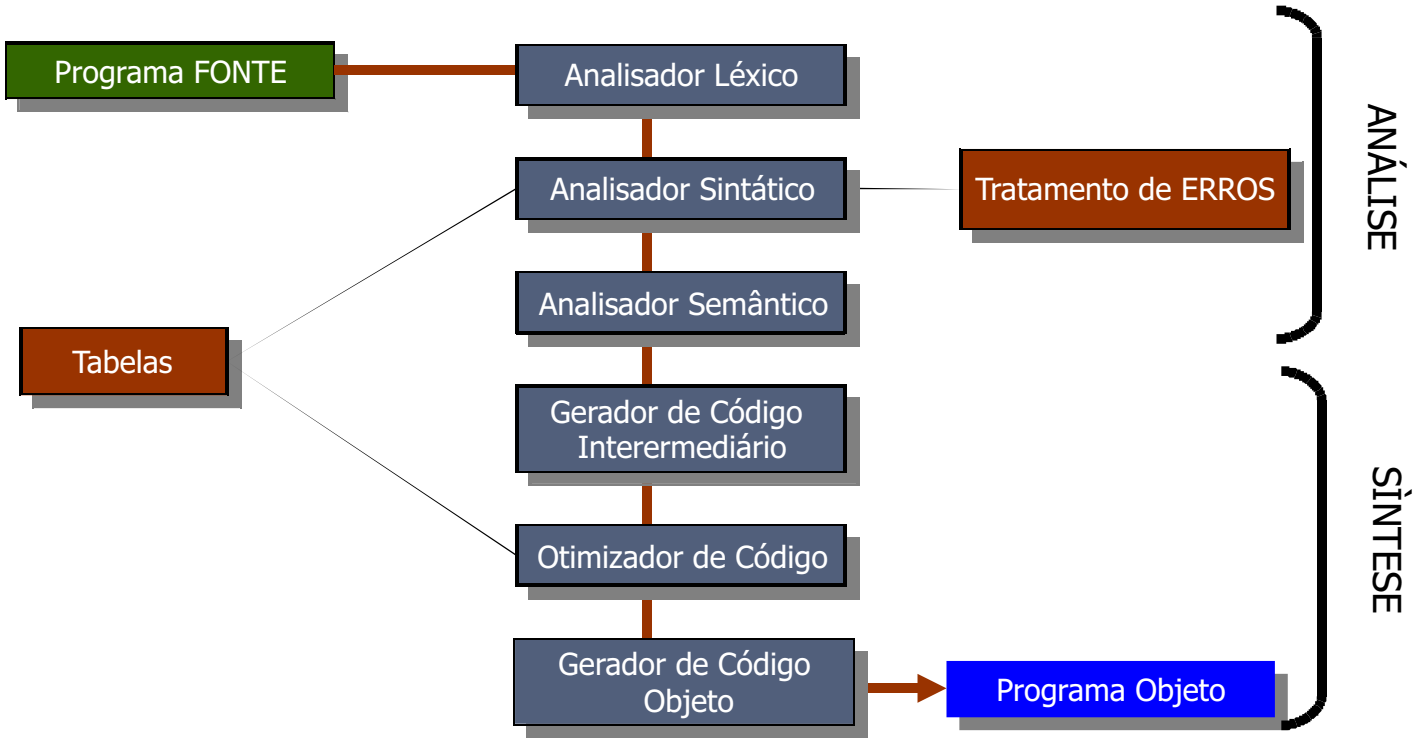


Figura: Estrutura Genérica dos Tradutores

Análise Léxica

- O principal objetivo do analisador léxico é identificar seqüências de caracteres que constituem unidades léxicas, que são chamadas de tokens.
- Token: é uma unidade do código que estamos traduzindo (compilando).
  - um literal,
  - uma função,
  - um sinal qualquer,
  - um abre-parênteses ou um fecha-parênteses,
  - um ponto e vírgula, uma palavra reservada,
  - enfim, qualquer parte do código original que faça algum sentido na linguagem fonte.

Análise Léxica

- O programa fonte é encarado como uma seqüência de caracteres que deverão ser agrupados e identificados como:
  - palavras reservadas da linguagem (em C, por exemplo, *main*, *int*, *for*),
  - constantes (123, $0\text{x}1F$, $'A'$),
  - identificadores (*myvar*, *Str1*), etc.
- Para isso, o analisador léxico (ou scanner) lê o programa fonte, caractere por caractere, verificando os caracteres lidos, identificando os tokens, e desprezando comentários e espaços em branco desnecessários

Exemplo de Análise Léxica

- Exp = (A + B) * 1.5;
- Os tokens (itens léxicos) contidos nesta expressão, são: 'Exp', '=', '(', 'A', '+', 'B', ')', '*', '1.5' , ';'
- Os itens léxicos a serem reconhecidos pelo analisador léxico são determinados pela gramática da linguagem-fonte.
- Deste modo, caso um item léxico não seja definido por esta gramática, um erro léxico é gerado.
- Por exemplo, suponhamos que uma linguagem só suporte valores inteiros. Então o valor "1.5" iria ocasionar um erro, que deverá ser tratado.

Análise Léxica

- Além da identificação de tokens, o analisador léxico inicia a construção da Tabela de Símbolos, e também envia mensagens de erro caso identifique unidades léxicas não aceitas pela linguagem.
  - Tabela de símbolos: estrutura usada para armazenar informações sobre declarações de variáveis, declarações dos procedimentos ou subrotinas, etc.
- A saída do analisador léxico é uma cadeia de tokens que é passada para a próxima fase do compilador, a Análise Sintática
- Geralmente, o Analisador Léxico é implementado como uma subrotina que funciona sob o comando do Analisador Sintático

Análise Léxica

- O scanner normalmente é um procedimento chamada pelo parser quando ele necessita um novo token
- Cada chamada ao scanner produz o próximo token no programa fonte
- O parser é responsável por armazenar qualquer token que seja necessário para análise futura
- O scanner precisa conhecer características dependentes da linguagem como a interpretação a ser dada a espaços e saltos de linha

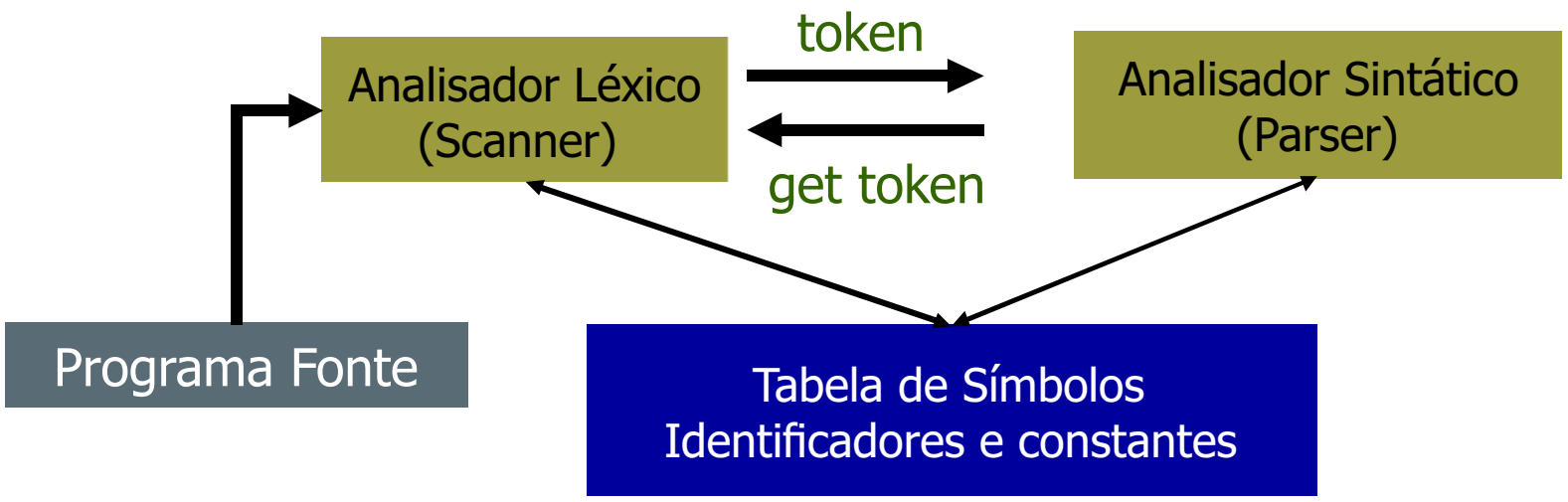


Figura: Scanner e Parser

Análise Sintática

- A função da análise sintática é verificar se a estrutura gramatical do programa está correta, ou seja, se essa estrutura foi formada de acordo com as regras gramaticais da linguagem.
- Em outras palavras, a análise sintática é responsável por reconhecer e validar expressões de diversos tipos – declarações, expressões aritméticas, construções de controle de execução, etc
- Essa validação é feita através de uma varredura, ou parsing, da representação interna (da cadeia de tokens), do programa fonte.

Análise Sintática

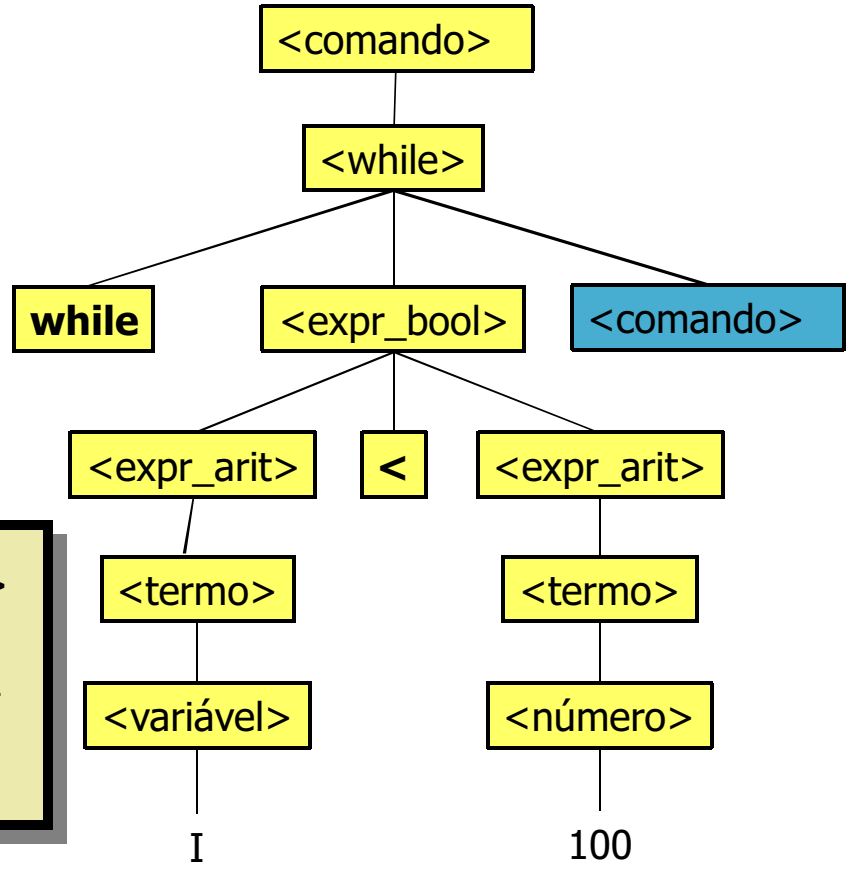
- O Analisador Sintático produz explícita ou implicitamente uma estrutura em árvore, que é chamada de **Árvore de Derivação** ou **Árvore de Sintaxe**.
- A árvore de derivação exibe a estrutura sintática do programa fonte, que é resultado da aplicação das regras gramaticais da linguagem.

Nota

em muitos compiladores, a representação interna do programa resultante da análise sintática não é a árvore de derivação completa do programa fonte, mas sim, uma árvore compacta, eliminando redundâncias e elementos que existem para facilitar a programação (açucar sintático), com o objetivo facilitar a geração de código.

Exemplo de uma árvore de derivação

```
while(I < 100)
<comando>
```




```
<comando>   → <while > | <atrib> | ...
<while>     → while  <expr_bool> <comando>
<atrib>     → <variável> = <expr_arit>
<expr_bool> → <expr_arit> < <expr_arit>
<expr_arit> → <expr_arit> + <termo>| <termo>
<termo>     → <número> | <variável>
<variável>  → I | J
<número>    →100
```

Figura: Exemplo de uma árvore de derivação do comando: while(I<100)

Exemplo de uma árvore de derivação

Comando *FOR*

Seguindo a gramática utilizada no exemplo do comando *while()*, vamos montar a árvore de derivação para o comando *for()*.

Exemplo de uma árvore de derivação

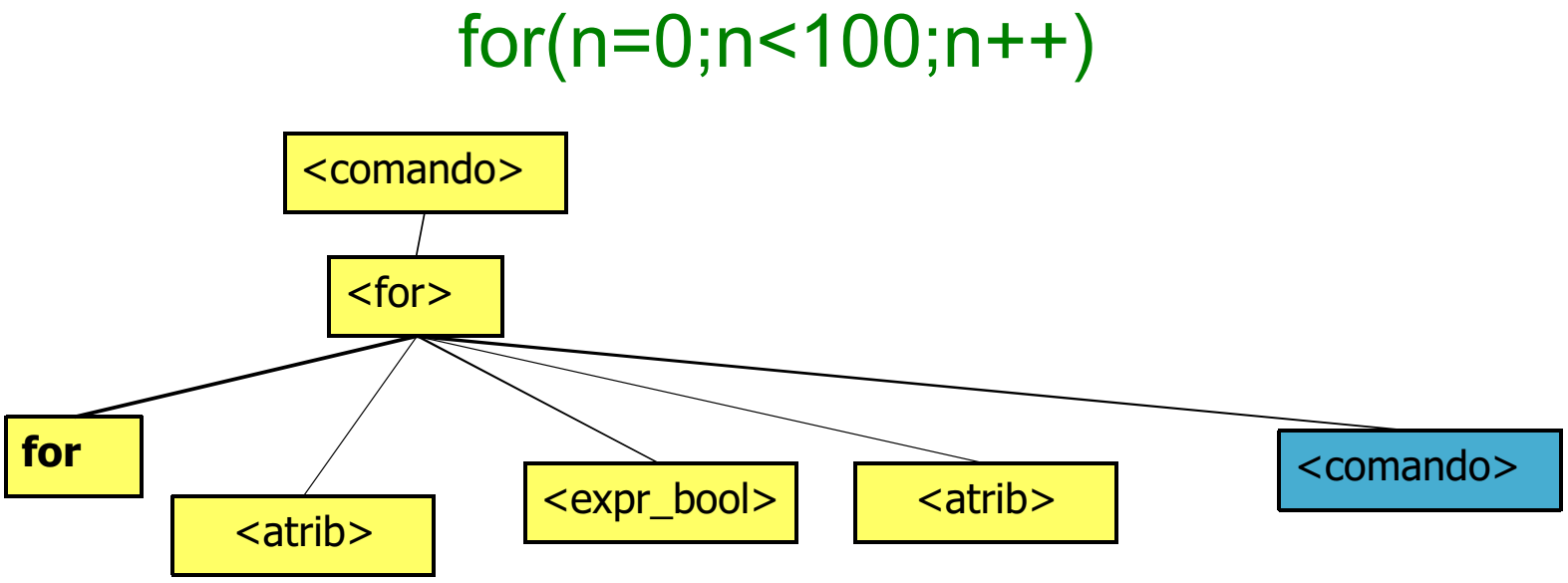


Figura: Exemplo de uma árvore de derivação do comando: $for(n = 0; n < 100; n + +)$

Exemplo de uma árvore de derivação

for(n=0;n<100;n++)

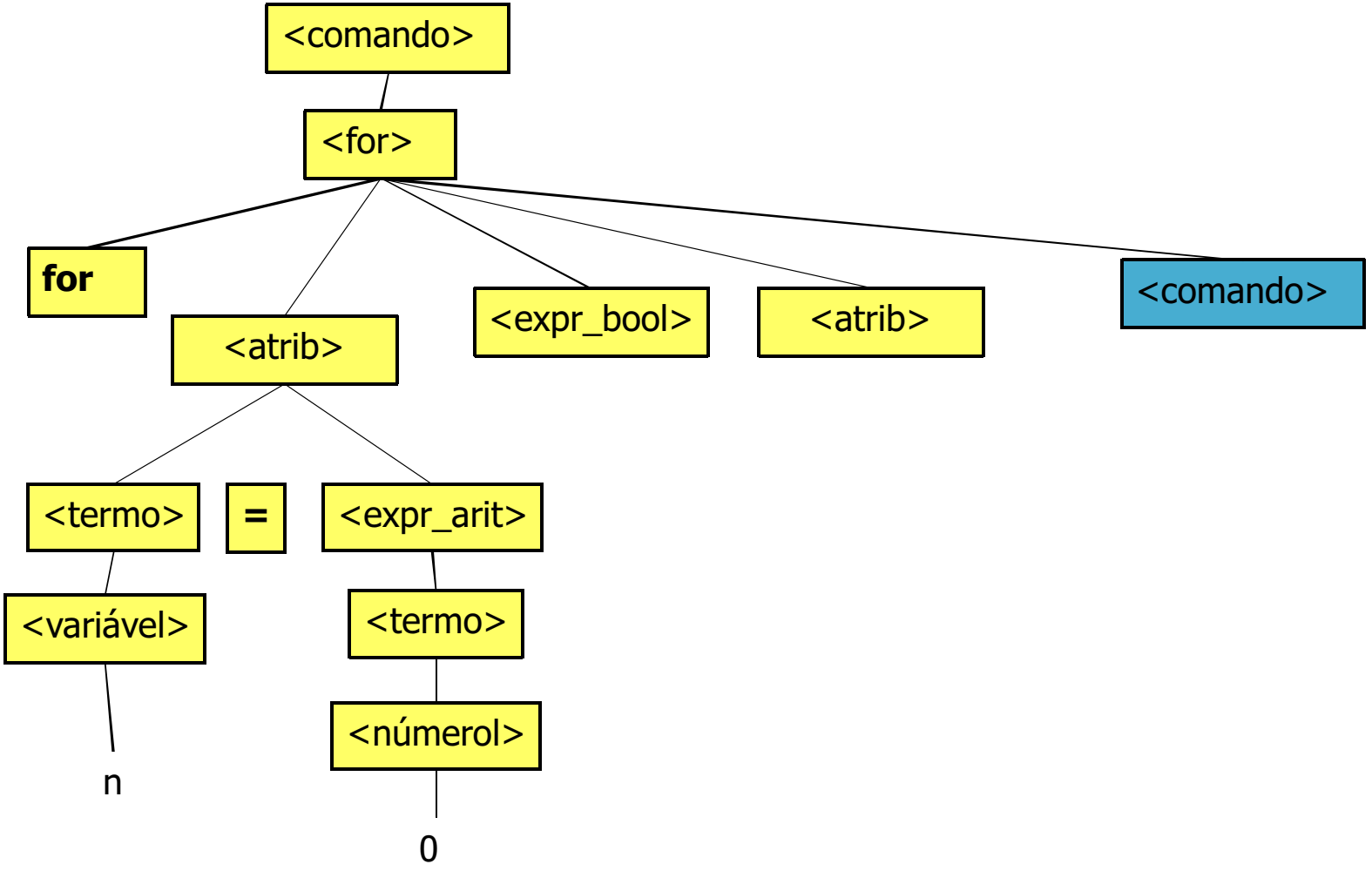


Figura: Exemplo de uma árvore de derivação do comando: $for(n = 0; n < 100; n++)$

Exemplo de uma árvore de derivação

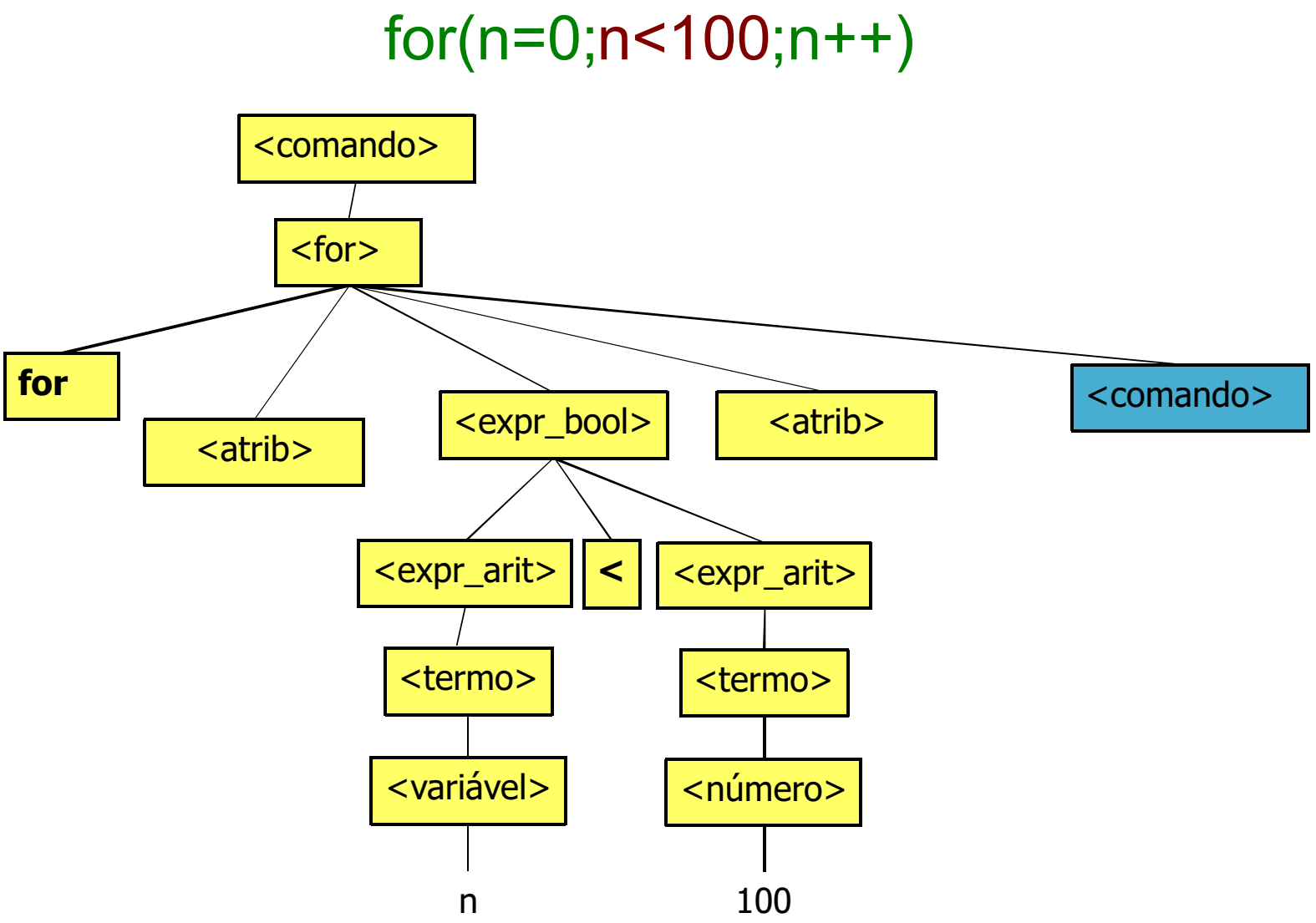


Figura: Exemplo de uma árvore de derivação do comando: $for(n = 0; n < 100; n++)$

Exemplo de uma árvore de derivação

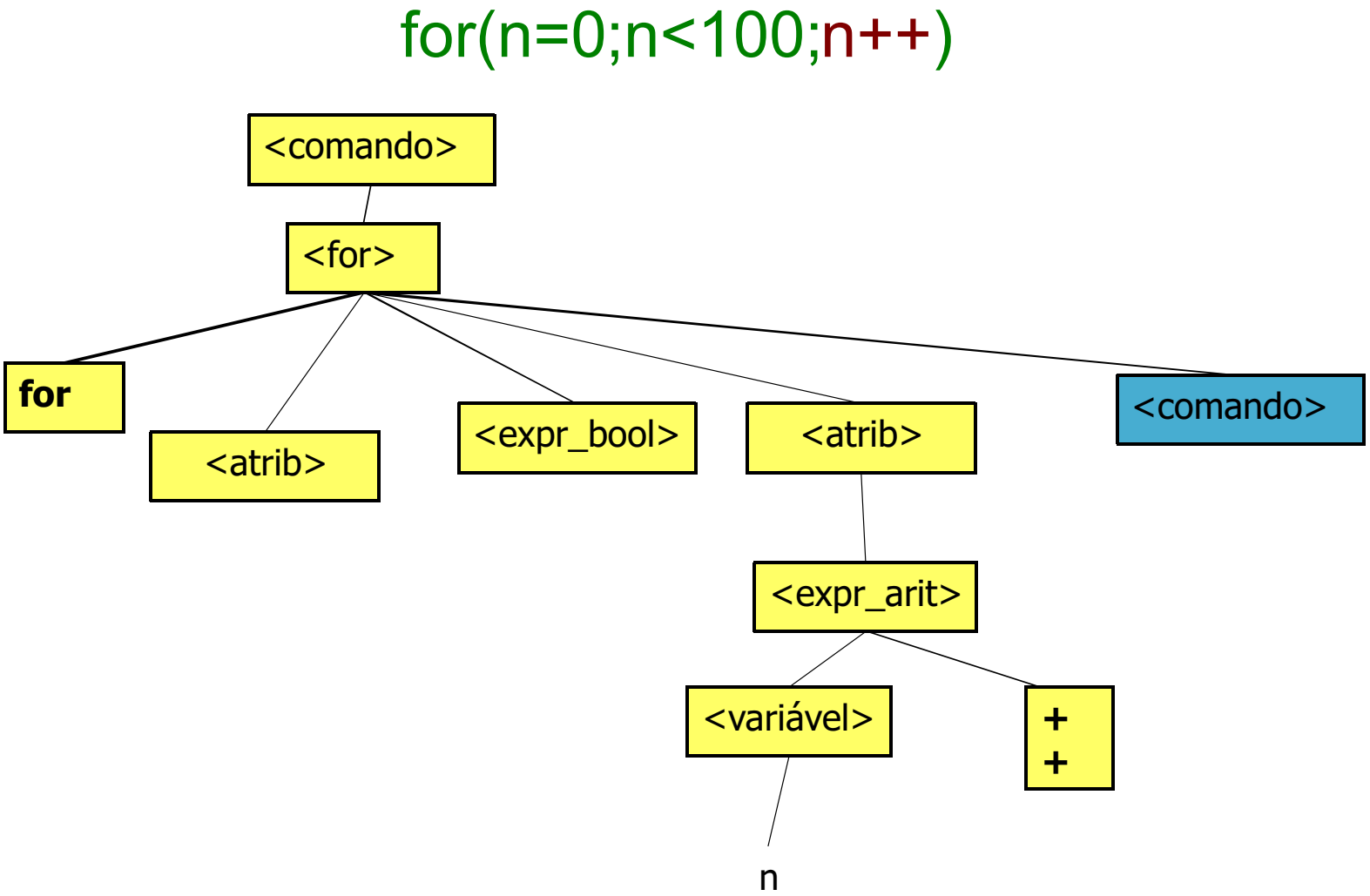


Figura: Exemplo de uma árvore de derivação do comando: $for(n = 0; n < 100; n++)$

Análise Semântica

- Verificar se uma expressão obedece às regras de formação de uma dada gramática.
- Porém, seria muito difícil expressar através de gramáticas algumas regras usuais em linguagem de programação
  - Exemplo:
  - todas as variáveis devem ser declaradas, e situações onde o contexto em que ocorre a expressão ou o tipo da variável deve ser verificado
- O objetivo da análise semântica é trabalhar nesse nível de inter-relacionamento entre partes distintas do programa, e não na sintaxe de cada expressão (isso é feito pelo parser).

Análise Semântica

- As tarefas básicas desempenhadas durante a análise semântica incluem:
  - verificação de tipos,
  - verificação do fluxo de controle, e
  - verificação da unicidade da declaração de variáveis.
- Entrada: programa na forma de uma árvore de derivação.
- Saída: árvore anotada ou semanticamente correta.

Análise Semântica

- Considere o seguinte exemplo de código em C:

Algoritmo 2 Erro Semântico

```
1: int modulo(int a, float b)
2: { return a%b; }
```

Análise Semântica

- A tentativa de compilar esse código irá gerar um erro detectado pelo analisador semântico, mais especificamente pelas regras de verificação de tipos.
- Será indicando que o operador módulo % não pode ter um operador real.
- No compilador gcc, essa mensagem é
  In function ‘modulo’:
  ...:invalid operands to binary %

Análise Semântica

- Em alguns casos, o compilador realiza a conversão automática de um tipo para outro que seja adequado à aplicação do operador.
- Por exemplo, na expressão em C
  $a = x -' 0';$
- A constante do tipo caráter '0' é automaticamente convertida para inteiro para compor corretamente a expressão aritmética na qual ela toma parte;
- Todo *char* em uma expressão é convertido pelo compilador para um *int*. Esse procedimento de conversão de tipo é denominado **coerção** (cast).

Análise Semântica

- Em outras situações, a conversão deve ser indicada explicitamente pelo programador através do operador de molde, com o nome do tipo entre parênteses na frente da expressão cujo resultado deseja-se converter.
  - Por exemplo, um programa com as declarações:

Algoritmo 3 Erro Semântico 2

```
int a;
int *p;
a = p;
```

Análise Semântica

- geraria a seguinte mensagem do compilador:
  warning: assignment makes integer from pointer without a cast
- Porém, se o programador indicar que sabe que está fazendo uma conversão "'forçada" através do operador de molde, então nenhuma mensagem é gerada.
  a = (int) p;

Análise Semântica

- Outro exemplo de erro detectado pela análise semântica, neste caso pela verificação de fluxo de controle, é ilustrado pelo código

Algoritmo 4 Erro Semântico 3

```
void f2(int j, int k) {
        if (j == k)
                break;
        else
                continue; }
```

Análise Semântica

- Nesse caso, o compilador gera as mensagens:
  In function ‘f2’:
  ...:break statement not within loop or switch
  ...:continue statement not within a loop

Análise Semântica

- A verificação de unicidade detecta situações tais como duplicação em declarações de variáveis, de componentes de estruturas e em rótulos do programa.
- A compilação do seguinte código

Algoritmo 5 Erro Semântico 4

```
void f3(int k) {
  struct {
    int a;
    float a;
  } x;
  float x;
  switch (k) {
  case 0x31: x.a = k;
  case '1': x = x.a;
  }
}
```

Análise Semântica

- O exemplo anterior gera os seguintes erros:
  In function ‘f3’:
  ...:duplicate member ’a’
  ...:previous declaration of ’x’
  ...:duplicate case value

Estrutura Genérica dos Tradutores

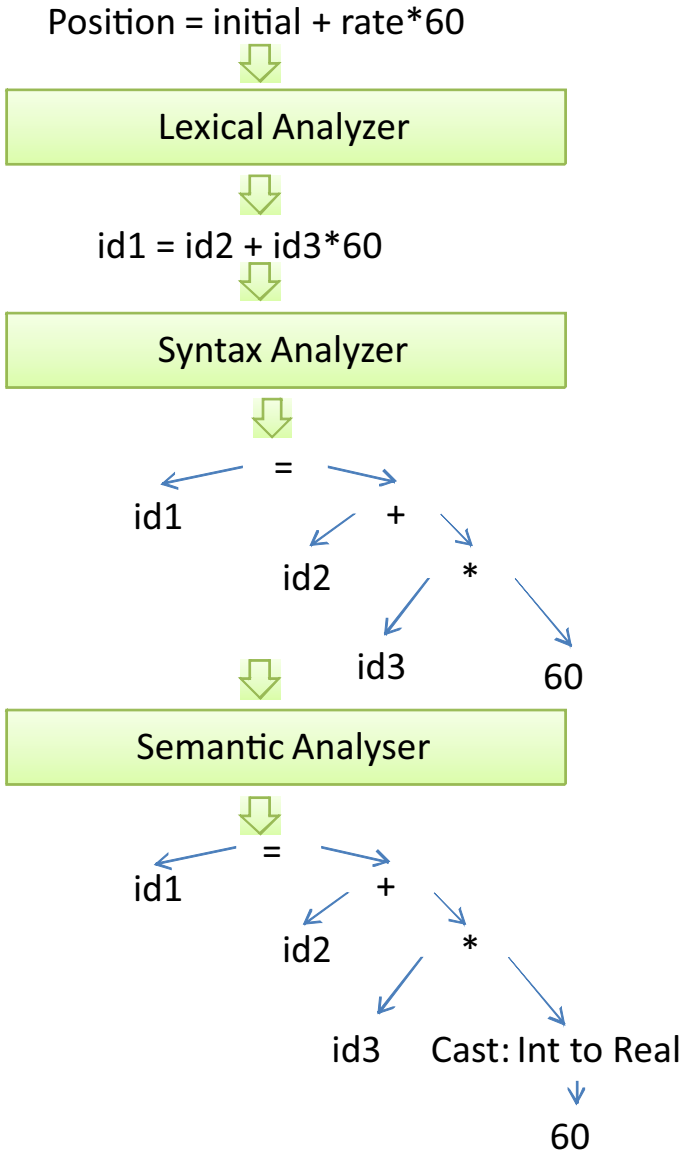


Figura: Exemplo

Estrutura Genérica dos Tradutores

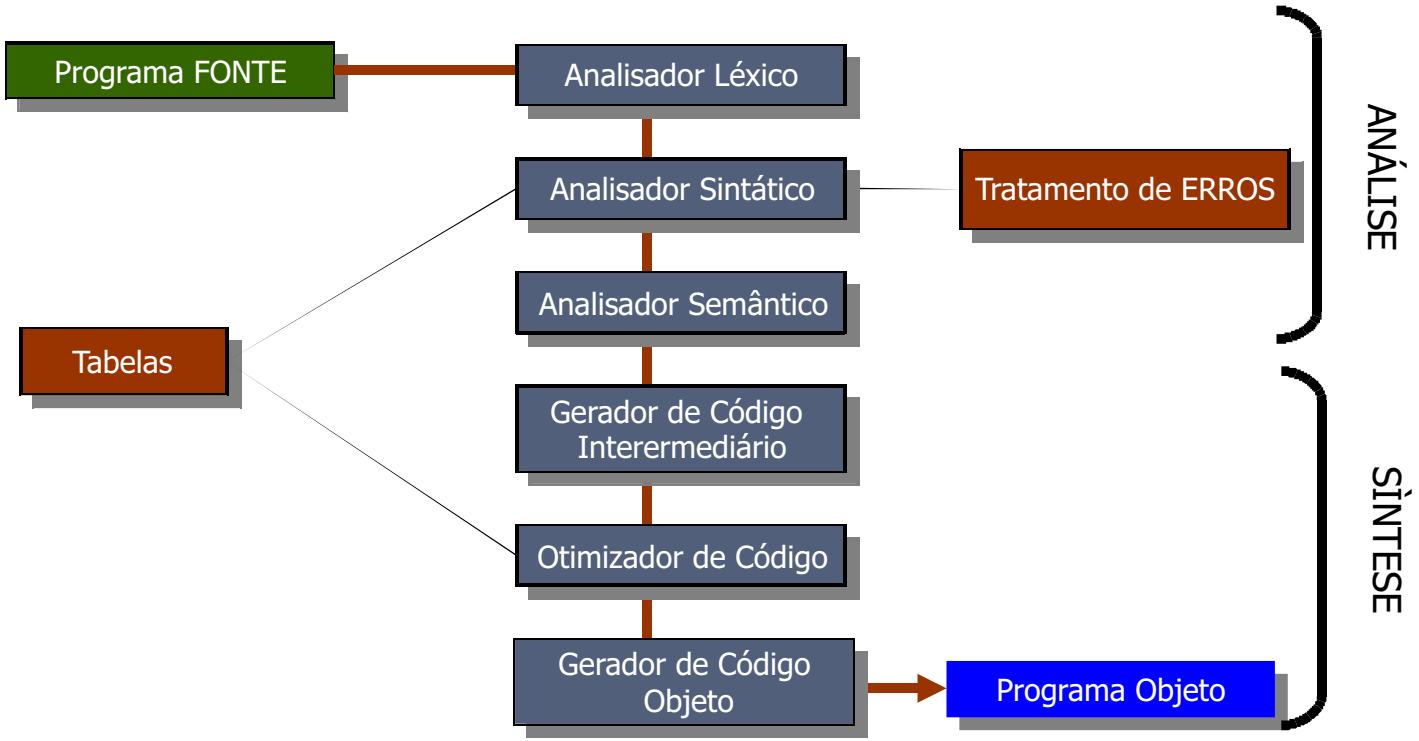


Figura: Estrutura Genérica dos Tradutores

Geração do Código Intermediário

- A partir da representação interna, árvore de derivação, produzida pelo analisador Sintático é gerado uma seqüência de **código objeto final** ou, como ocorre na maioria das vezes, gera o **código intermediário**
- A linguagem utilizada para a geração de um código em formato intermediário entre a linguagem de alto nível e a linguagem assembly deve representar, de forma independente do processador para o qual o programa será gerado, todas as expressões do programa original.
- Duas formas usuais para esse tipo de representação são:
  - código de três endereços, e a
  - notação posfixa

Código de três endereços

- O código de três endereços é composto por uma seqüência de instruções envolvendo operações binárias ou unárias e uma atribuição.
- O nome três endereços está associado à especificação, em uma instrução, de no máximo três variáveis:
  - duas para os operadores binários e uma para o resultado.

Código de três endereços

- Assim, expressões envolvendo diversas operações são decompostas nesse código em uma série de instruções, eventualmente com a utilização de variáveis temporárias introduzidas na tradução.
- Dessa forma, obtém-se um código mais próximo da estrutura da linguagem assembly e, conseqüentemente, de mais fácil conversão para a linguagem-alvo.
- Uma possível especificação de uma linguagem de três endereços envolve quatro tipos básicos de instruções:
  - expressões com atribuição, desvios, invocação de rotinas e acesso indexado e indireto.

Instruções de atribuição

- Instruções de atribuição são aquelas nas quais o resultado de uma operação é armazenado na variável especificada à esquerda do operador de atribuição, aqui denotado por ":=".
  - Há três formas para esse tipo de instrução:
    - $x := y\ op\ z$
    - $x := op\ y$
    - $x := y$
- A variável recebe o resultado de uma operação binária, unária, ou através de uma simples cópia de valores de uma variável para outra
  - Exemplo com atribuição:
    - $a = b + c * d;$
  - seria traduzida nesse formato para as instruções:
    - $\_t1 = c * d$
    - $a = b + \_t1$

Instruções de Desvio

- Instruções de desvio tem formas básicas:
  - Uma instrução de desvio incondicional: $goto\ L$
    - onde L é um rótulo simbólico que identifica uma linha do código.
  - A outra forma de desvio é o desvio condicional, $if\ x\ opr\ y\ goto\ L$
    - onde opr é um operador relacional de comparação

Código de três endereços

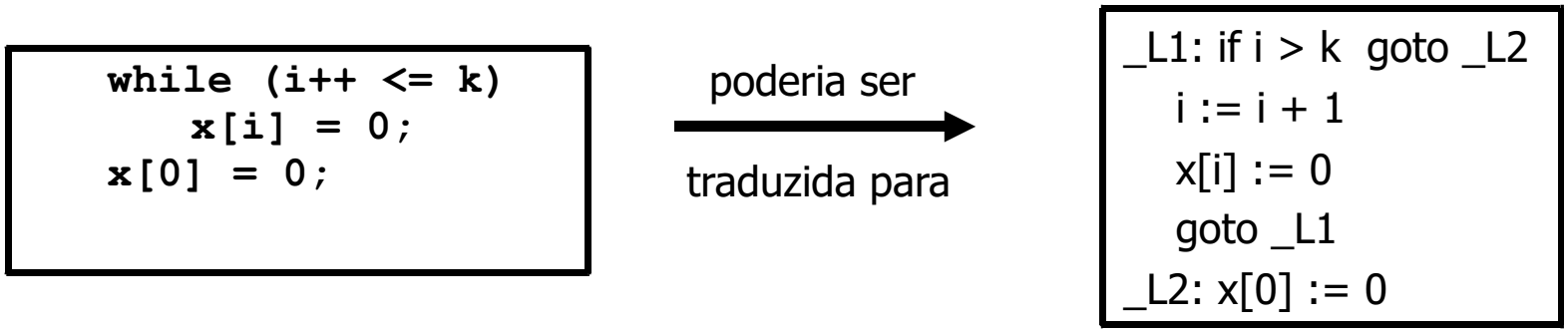


Figura: Instruções de Desvio

Invocação de Rotinas

1. Os argumentos do procedimento são "registrados" (empilhados) em memória mediante a instrução **param**;
2. Após a definição dos argumentos, a instrução **call** completa a invocação da rotina. A instrução **return** indica o fim de execução de uma rotina.

Opcionalmente, a instrução **return** pode especificar um valor de retorno, que pode ser atribuído na linguagem intermediária a uma variável.

Código de três endereços

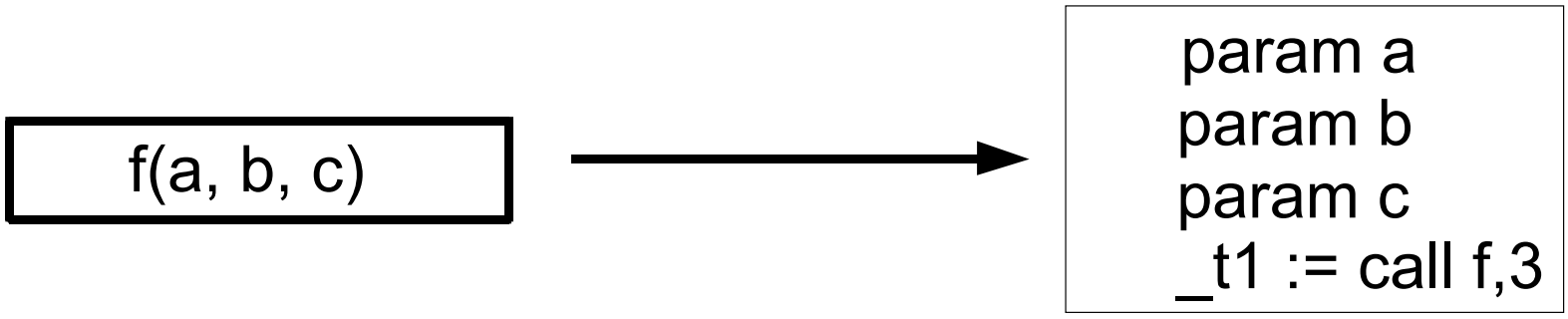


Figura: Exemplo, no caso de chamarmos uma função com retorno de valor não utilizado.

Acesso Indexado e Indireto

- O último tipo de instrução para códigos de três endereços refere-se aos modos de endereçamento **indexado e indireto**.
  - Para atribuições indexadas, as duas formas básicas são
    - $x := y[i]$
    - $x[i] := y$
  - As atribuições associadas ao modo indireto permitem a manipulação de endereços e seus conteúdos. As instruções em formato intermediário também utilizam um formato próximo àquele da linguagem C:
    - $x := \&y$
    - $w := *x$
    - $*x := z$

Representação Interna

- A representação interna das instruções em códigos de três endereços dá-se na forma de armazenamento em tabelas com quatro ou três colunas.
- Na abordagem que utiliza quádruplas (as tabelas com quatro colunas), cada instrução é representada por uma linha na tabela com a especificação do operador, do primeiro argumento, do segundo argumento e do resultado.
- Por exemplo, a tradução da expressão

`a=b+c*d;` ⟶

	operador	arg 1	arg 2	resultado
1	*	c	d	_t1
2	+	b	_t1	a

- Para algumas instruções, como aquelas envolvendo operadores unários ou desvio incondicional, algumas das colunas estariam vazias.

Representação Interna

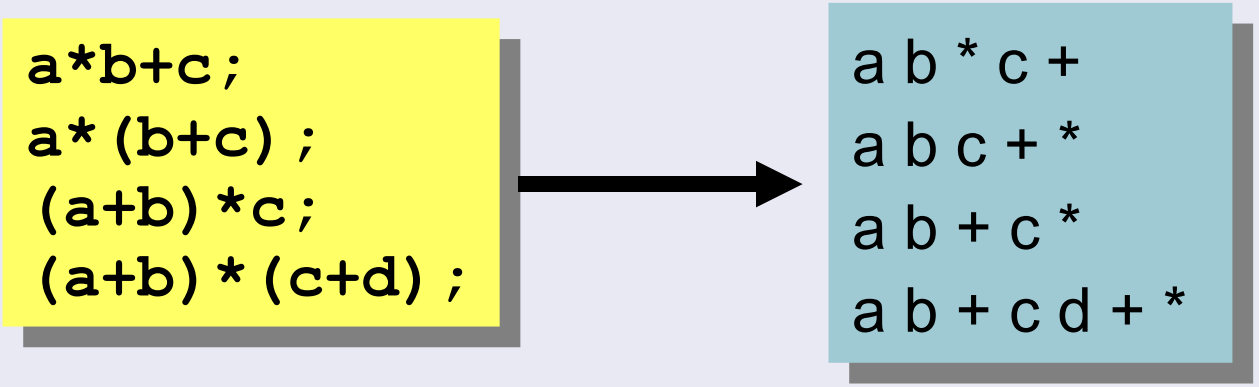
- Na outra forma de representação, por triplas, evita a necessidade de manter nomes de variáveis temporárias ao fazer referência às linhas da própria tabela no lugar dos argumentos.
  - Nesse caso, apenas três colunas são necessárias, uma vez que o resultado está sempre implicitamente associado à linha da tabela.
  - No mesmo exemplo apresentado para a representação interna por quádruplas, a representação por triplas seria

```
a=b+c*d;
```

	operador	arg 1	arg 2
1	*	c	d
2	+	b	(1)

Notação posfixa

- A notação tradicional para expressões aritméticas, que representa uma operação binária na forma x+y, ou seja, com o operador entre seus dois operandos, é conhecida como notação infixa.
- Uma notação alternativa para esse tipo de expressão é a notação posfixa, também conhecida como notação polonesa reversa, na qual o operador é expresso após seus operandos.
  - O nome deriva da nacionalidade de Jan Lukasiewicz quem inventou nos anos 20 a notação polonesa,
- O atrativo da notação posfixa é que ela dispensa o uso de parênteses. Por exemplo:



Notação posfixa

- Instruções de desvio em código intermediário usando a notação posfixa assumem a forma
  - $L\ jump$
  - $x\ y\ L\ jcc$
- para desvios incondicionais e condicionais, respectivamente.
- No caso de um desvio condicional, a condição a ser avaliada envolvendo x e y é expressa na parte jcc da própria instrução.
- Assim, jcc pode ser uma instrução entre
- **jeq** (desvio ocorre se x e y forem iguais),
- **jne** (se diferentes),
- **jlt** (se x menor que y),
- **jle** (se x menor ou igual a y),
- **jgt** (se x maior que y) ou
- **jge** (se x maior ou igual a y).

Notação posfixa

- Expressões em formato intermediário usando a notação posfixa podem ser eficientemente avaliadas em máquinas baseadas em pilhas, também conhecidas como máquinas de zero endereços.
- Nesse tipo de máquinas, operandos são explicitamente introduzidos e retirados do topo da pilha por instruções $push$ e pop, respectivamente.
- Além disso, a aplicação de um operador retira do topo da pilha seus operandos e retorna ao topo da pilha o resultado de sua aplicação.
- Por exemplo, a avaliação da expressão $a * (b + c)$ em uma máquina baseada em pilha poderia ser traduzida para o código $\rightarrow$ a b c $+$ *
  - $push\ a$
  - $push\ b$
  - $push\ c$
  - add
  - $mult$

Notação posfixa

Exercício

Escreva uma função em C (expr) que avalie uma expressão em notação pósfixa (notação Polonesa). A entrada é feita pela linha de comando, onde cada operador ou operando é um argumento separado.

- Exemplo: expr $2\ 3\ 4\ +\ *\ \rightarrow$ equivale a $2 * (3 + 4)$

Algoritmo 6 Exercício

```
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv) {
        int i; double vlr;
        for (i = 1; i < argc; ++i) {
                switch (argv[i][0]){
                case '\0':
                        printf("Linha de argumento vazia");
                        exit(0);
                        break;
        case '0':case '1':case '2':case '3':case '4':case '5':case '6':case '7':case '8':case '9':
        /* neste caso o argumento eh numerico */
        vlr=atoi(argv[i]);
                 /* fazer alguma coisa */
                        break;
                case '+':
                        /* fazer alguma coisa */
                        break;
                case '-':
                        /* fazer alguma coisa */
                        break;
                case '*':
                        /* fazer alguma coisa */
                        break;
                case '/':
                        /* fazer alguma coisa */
                        break;
                default:
                        printf("Operador %s desconhecido\n", argv[i] );
                        exit(0);
                }
        }
    /* mostre o resultado da expressao aqui */
        return 0; }
```

Otimizador de Código

- Otimização dependente da máquina
  - Algumas máquinas contém registradores que podem ser usados para o armazenamento de operandos
  - Se não há registradores em quantidade suficiente para armazenar todos os operandos
    - Necessário escolher que valor substituir quando for necessário atribuir um novo valor ao registrador
    - Deve ser escolhido o valor que não será necessário pelo maior período de tempo
  - Ao usar os registradores o compilador deve levar em consideração o fluxo de execução do programa

Otimizador de Código

- Otimização dependente da máquina
  - Eliminação de sub-expressões que aparecem em mais de um ponto no programa e que computam o mesmo valor
  - O compilador pode calcular a sub-expressão apenas uma vez e usa o valor calculado onde for necessário
  - Remoção de valores que não são modificados entre interações de um loop
  - Valores podem ser calculados antes da entrada no loop

Geração de Código Objeto

- Objetiva produzir o código objeto, reservar memória para constantes e variáveis, registradores, etc.
- Esta é considerada a fase mais difícil, pois depende da máquina alvo.
- O arquivo do código objeto possui formato dependente do Sistema Operacional

Estrutura Genérica dos Tradutores

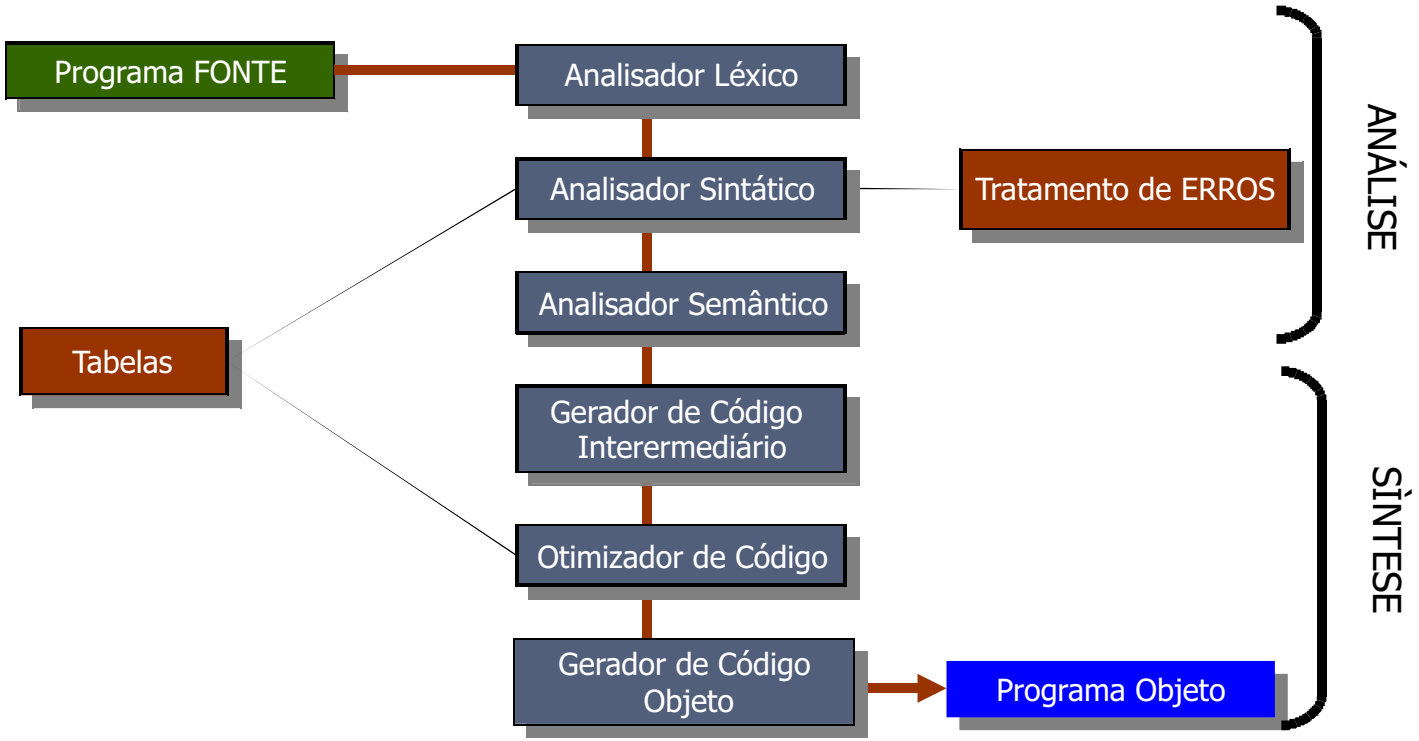


Figura: Estrutura Genérica dos Tradutores

Gerência de Tabelas

- compreende um conjunto de tabelas e rotinas associadas que são utilizadas por quase todas as fases do tradutor.
- Algumas tabelas usadas são fixas para cada linguagem, tais como: a tabela de palavras reservadas, tabela de delimitadores, etc.
- Entretanto, a estrutura de maior importância é aquela que é montada durante a análise do programa fonte, com informações sobre: declarações de variáveis, declarações dos procedimentos ou subrotinas , parâmetros, etc.
- Essas informações são armazenadas na Tabela de Símbolos. A cada ocorrência de um identificador no programa fonte, a tabela é acessada, e o identificador é procurado na tabela.
- Caso encontrado, as informações associadas a ele são comparadas com as informações obtidas no programa fonte, sendo que qualquer nova informação é inserida na tabela.

Atendimento a Erros

- Tem por objetivo "tratar erros" que são detectados em todas as fases de análise do programa fonte.
- Qualquer fase analítica deve prosseguir em sua análise, mesmo que erros tenham sido detectados.
- Isto pode ser realizado através de mecanismos de recuperação de erros, encarregados de re-sincronizar a fase com o exato ponto do texto fonte em análise.
- A perda do sincronismo faria a análise prosseguir de forma errada, propagando o efeito do erro.

Estrutura Genérica dos Tradutores

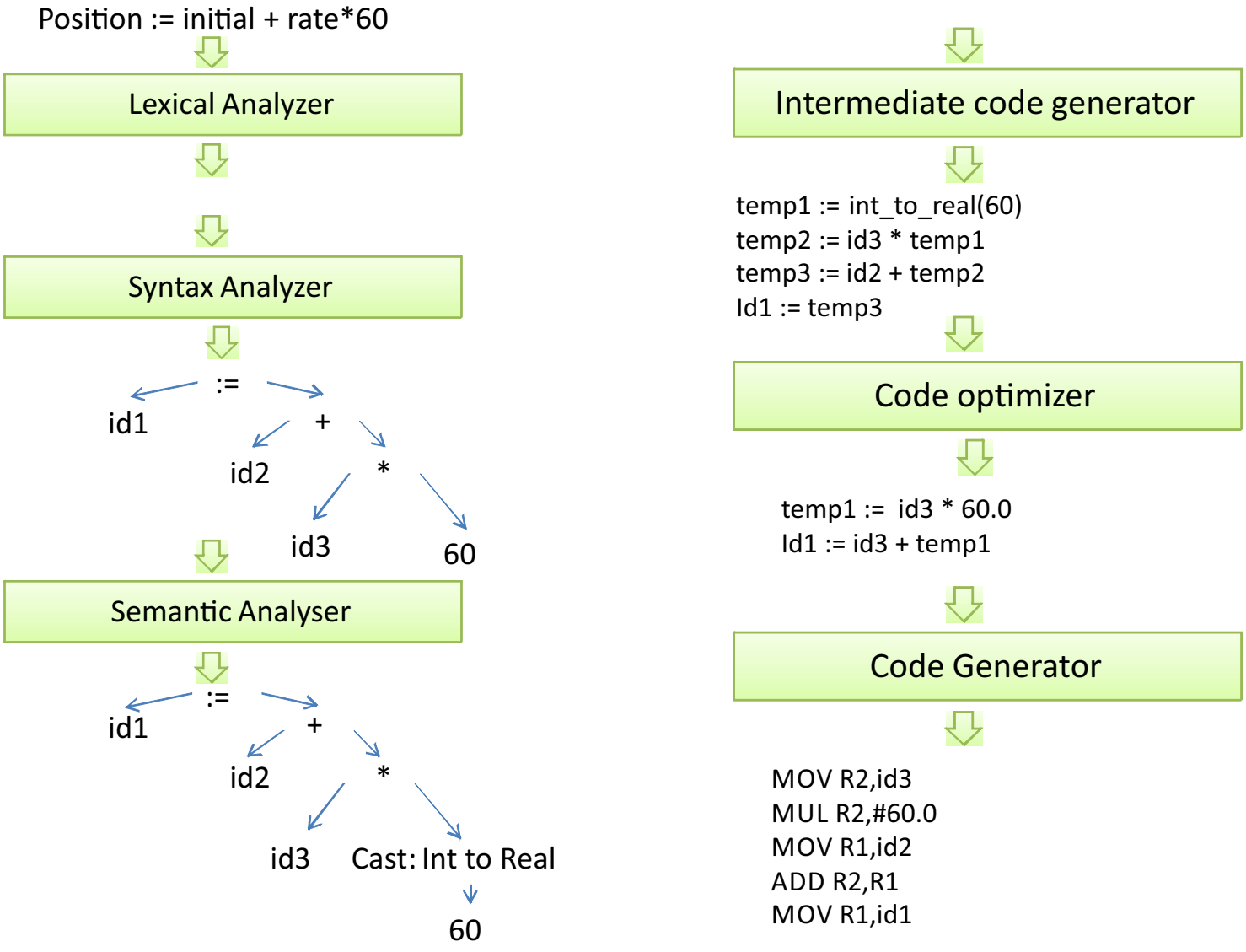


Figura: Exemplo

Próxima Aula

Montador