Compiladores II

Eduardo Ferreira dos Santos

Ciência da Computação
Centro Universitário de Brasília – UniCEUB

Março, 2017

Sumário

UniCEUB
Centro Universitário de Brasília

Pré-processamento

- Alguns outros programas podem ser necessários para a geração do executável;
- O pré-processador é responsável por coletar o programa fonte e, possivelmente, expandir macros em comandos na linguagem fonte.

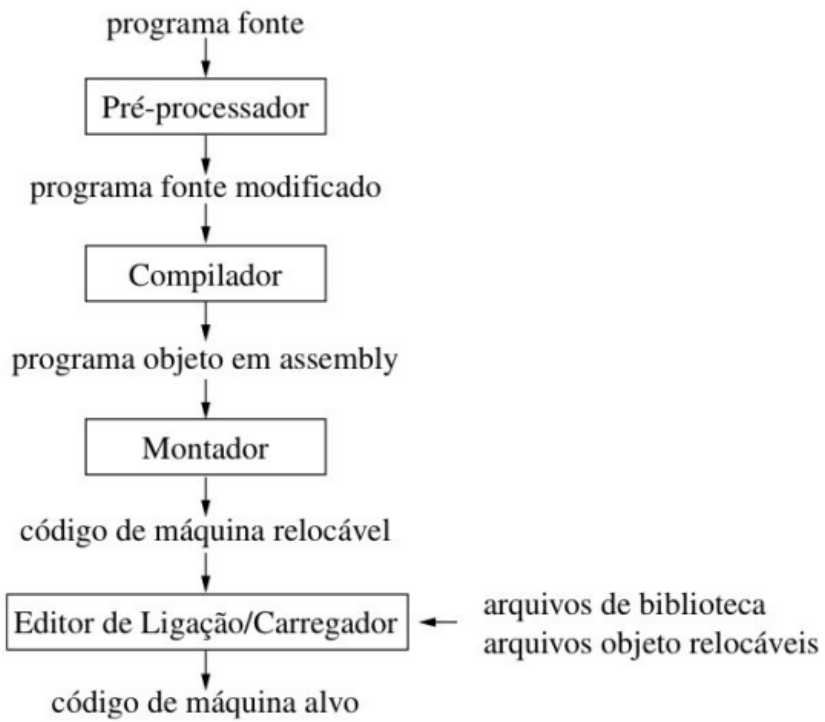

Figura 1.1: Um sistema de processamento de linguagem [Aho et al., 2007]

Modelo de análise e síntese

- A análise impõe um modelo gramatical para o código;
- Caso esteja sintaticamente mal formado ou semanticamente incorreto, deve informar qual é o erro;
- Gera também a tabela de símbolos, passada para a próxima etapa junto com a apresentação intermediária;
- A parte de síntese constrói o programa objeto a partir da tabela de símbolos e da representação intermediária;
- A compilação é organizada em fases, onde cada etapa transforma a representação anterior para a próxima camada.

Fases

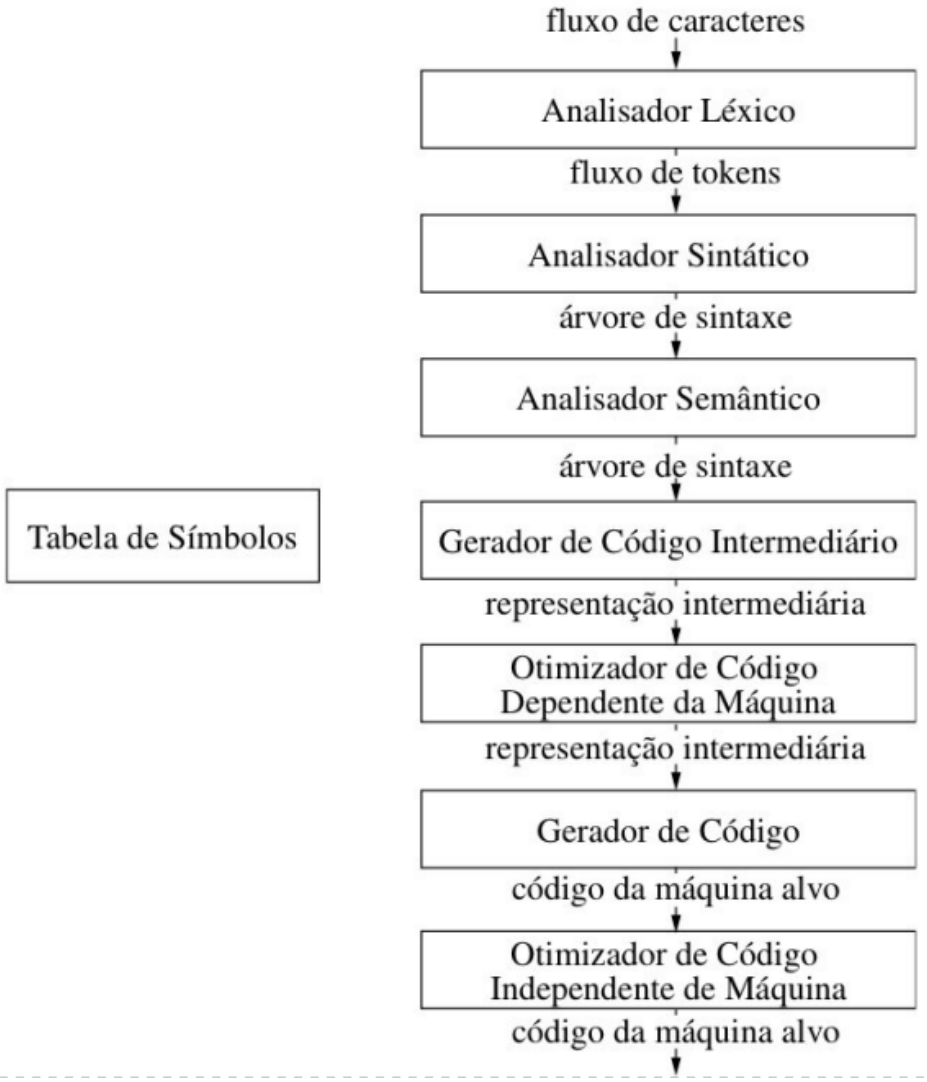

Figura 1.2: Fases do compilador [Aho et al., 2007]

Modelagem

- Ferramentas para descrever os algoritmos e as unidades léxicas utilizadas pelos compiladores:

  Autômatos Finitos Representação gráfica;
  Expressões Regulares Representação matemática.

- Ferramentas utilizadas para descrever a estrutura sintática:

  Gramáticas livres de contexto Representação matemática;
  Árvores Representação gráfica.

Autômatos finitos

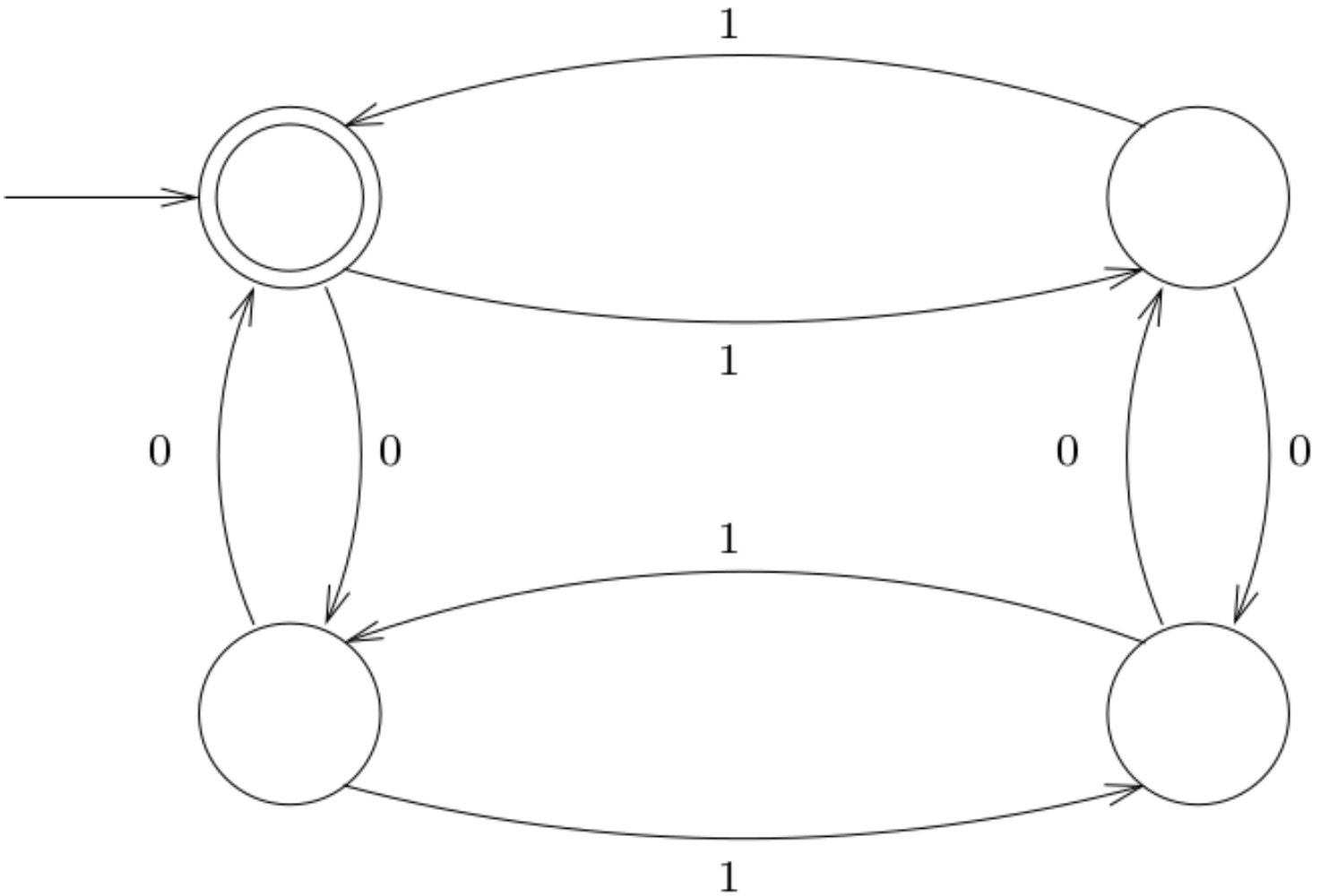

Figura 2.1: Um autômato [Pinto, 2016]

Expressões regulares

São expressões (sequências de símbolos), definidas recursivamente, que representam linguagens sobre um alfabeto Σ

Expressões Regulares

Expressão regular	representa a linguagem
$\emptyset$	vazia
ε	$\{\varepsilon\}$
$\boldsymbol{a}$	$\{a\}$, para cada $a \in \Sigma$
$(\boldsymbol{r} + \boldsymbol{s})$	$R \cup S$
$(\boldsymbol{r}\boldsymbol{s})$	RS
$(\boldsymbol{r}^*)$	R^*

onde $\boldsymbol{r}$ e $\boldsymbol{s}$ são expressões regulares representando as linguagens R e S

Figura 2.2: Algumas expressões regulares [Pinto, 2016]

Gramáticas livres de contexto

Mecanismo de formação de palavras (sentenças) a partir de substituição de variáveis [Pinto, 2016].

$$E \rightarrow \epsilon \tag{1}$$

$$E \rightarrow 0E1 \tag{2}$$

Exemplos de gramática [Pinto, 2016]

Árvores

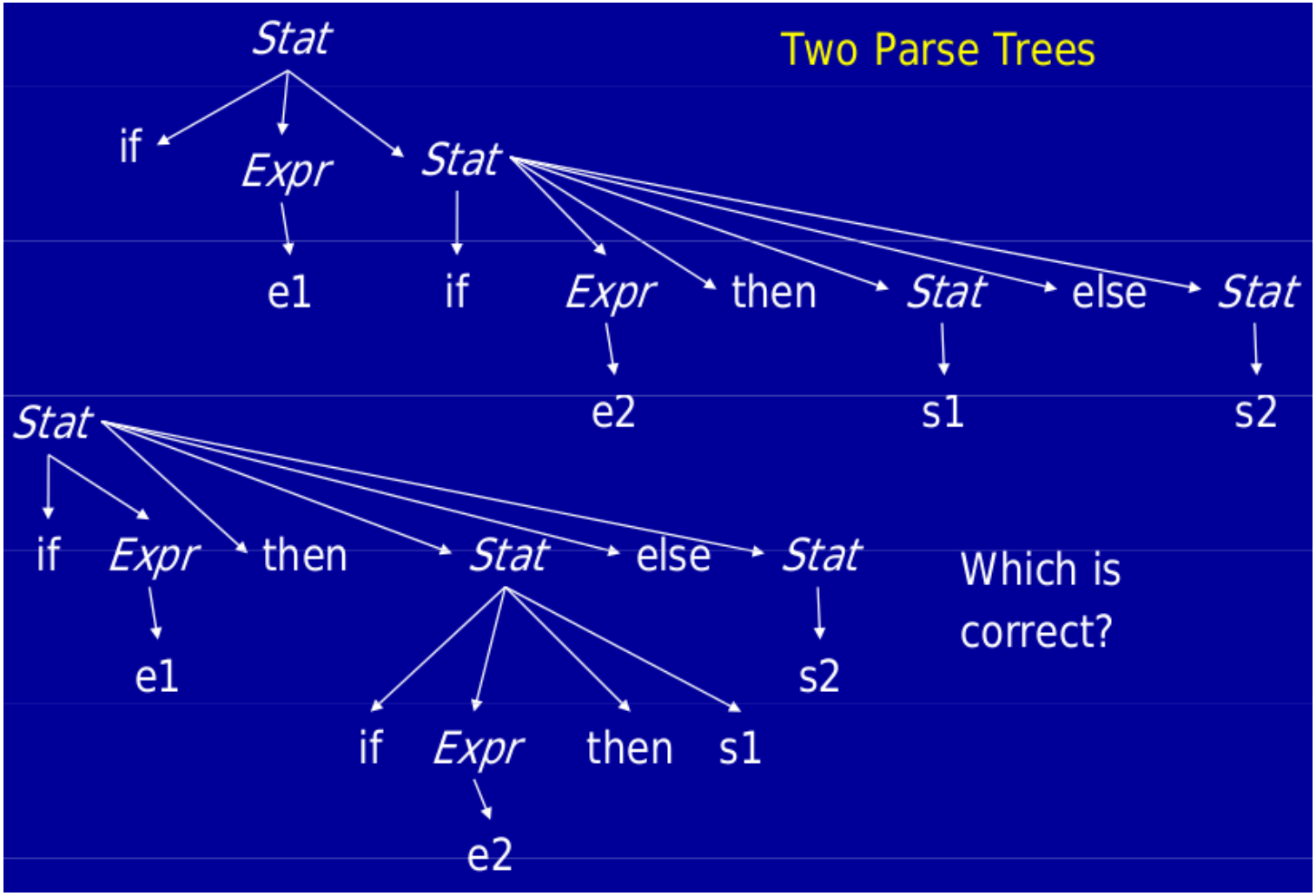

Figura 2.3: Árvore de parsing [Amarasinghe and Rinard, 2010]

Análise Léxica

- Primeiros passos: reconhecer palavras

This is a sentence.

- Análise léxica não é trivial.

ist his a se nte nce

- A análise léxica divide o programa em palavras ou tokens

if x == y then z = 1; else z = 2;

Parsing

- Uma vez que as palavras sejam entendidas, o próxima passo é entender a estrutura;
- *Parsing* = Diagramar sentenças
- A diagramação é organizada através de uma árvore.

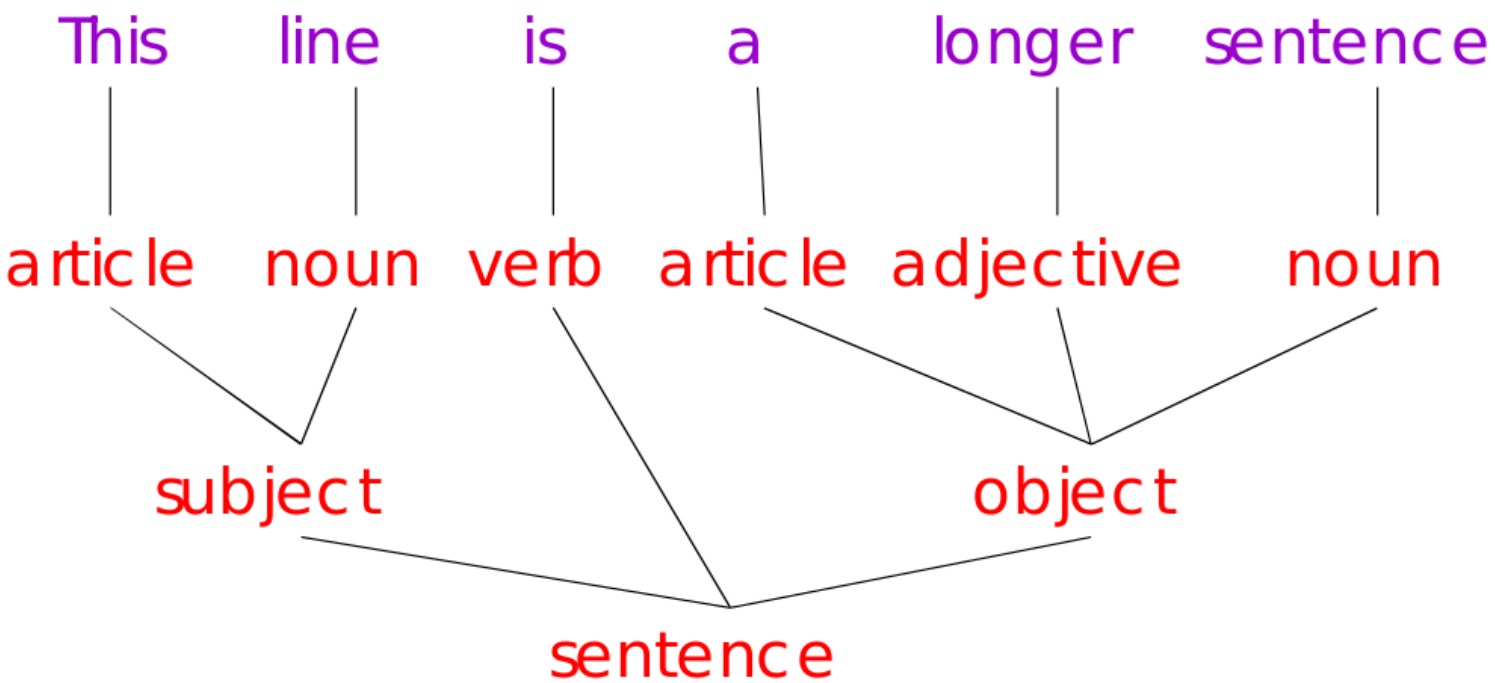

Figura 2.4: Diagramação através de árvore [Schwarz et al., 2016]

Parsing de programas

- Considere o programa:

$$\text{if } x == y \text{ then } z = 1; \text{ else } z = 2;$$

- Realizando o *parsing*:

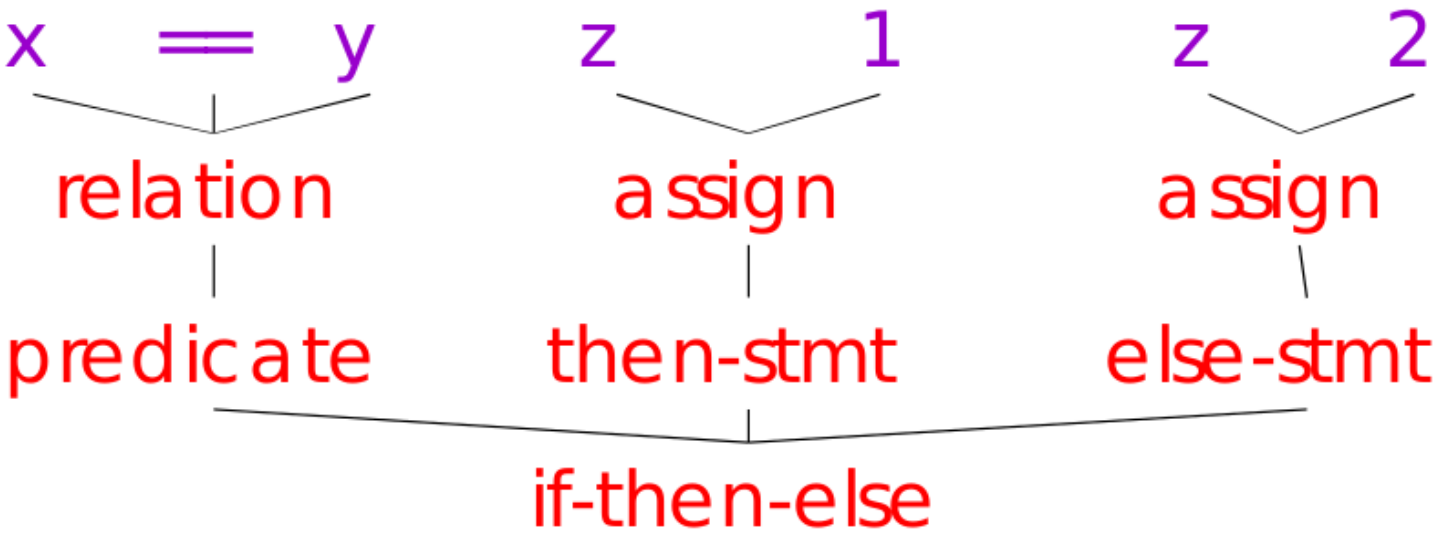

Figura 2.5: *Parsing* de um programa [Schwarz et al., 2016]

Análise semântica

- Uma vez que a sentença esteja construída, podemos entender seu significado;
- Os compiladores realizam uma análise limitada para detectar inconsistências;
- Ex.: O Alexandre disse que o Arthur fez sua tarefa:
  - Qual é o significado do conector sua?
  - A tarefa se refere a Alexandre ou Arthur?
- Complicando: O Alexandre disse que o Alexandre deixou sua tarefa em casa?
  - Quantos Alexandres existem?
  - Qual deles esqueceu a tarefa?

Ambiguidade

- As linguagens de programação definem regras para evitar ambiguidades:

Listing 1: [Schwarz et al., 2016]

```
{
  int Alexandre = 3;
  {
    int Alexandre = 4;
    cout << Alexandre;
  }
}
```

- O código imprime como resultado 4;
- A definição mais de dentro é utilizada.

Otimização

- A otimização do código depende da capacidade de controlar a saída para todas as possíveis saídas;
- Caso a afirmação seja válida, dizemos que a saída é ótima;
- Requisitos da otimização [Aho et al., 2007]:
  - *A otimização precisa ser correta, ou seja, preservar a semântica do programa compilado;*
  - *A otimização precisa melhorar o desempenho de muitos programas;*
  - *O tempo de compilação precisa continuar razoável;*
  - *O esforço de engenharia empregado precisa ser administrável.*
- Importância da corretude;
- Conceito de eficência;
- Novo paradigma: consumo de energia!

Execução eficiente

- Do alto para o baixo nível [Amarasinghe and Rinard, 2010]:
  - Mapeamento puro e simples do programa para o *assembly* normalmente gera uma execução ineficiente;
  - Quanto maior o nível de abstração, mais ineficiente.
- As abstrações para alto nível só são úteis se forem eficientes;
- Um bom compilador fornece a possibilidade de escrever em alto nível de abstração com a performance de instruções de baixo nível.

Execução

- A fase de análise divide o programa e impõe uma estrutura gramatical;
- Se houver algum erro, deve fornecer mensagens esclarecedoras;
- A fase de síntese constrói o programa objeto usando:
  - Representação intermediária;
  - Tabela de símbolos.
- Normalmente a compilação é dividida em fases, como descrito na Figura 6.

Análise léxica

- Normalmente a compilação se inicia pela análise léxica;
- Analisador léxico: lê o fluxo de caracteres e agrupa em sequências significativas;
- As sequências significativas são chamadas lexemas;
- Para cada lexema, o analisador léxico produz como saída um token;

$$\langle nome\_token, valor\_atributo \rangle \tag{3}$$

- O token é enviado para a etapa seguinte, a análise sintática.

Exemplo

$$position = initial + rate * 60 \tag{4}$$

position Mapeado para o token $\langle id, 1 \rangle$:

- id Símbolo abstrato que significa identificador;
- 1 Entrada na tabela de símbolos onde está *position*.

= Mapeado para o token $\langle = \rangle$. Por não exigir um valor de atributo, omitimos o segundo componente;

initial Mapeado para o token $\langle id, 2 \rangle$;

\+ Mapeado para o token $\langle + \rangle$;

rate Mapeado para o token $\langle id, 3 \rangle$;

* Mapeado para o token $\langle * \rangle$;

60 Mapeado para o token $\langle 60 \rangle$[1].

[1] Para o lexema 60 deveria haver um token como $\langle numero, 4 \rangle$

Atribuição

- Após a análise léxica, executa-se o comando de atribuição;
- O comando gera as substituições apontadas no exemplo 4;
- Após a substituição, obtemos o resultado 5:

$$\langle id, 1\rangle\langle =\rangle\langle id, 2\rangle\langle +\rangle\langle id, 3\rangle\langle *\rangle\langle 60\rangle \tag{5}$$

- É possível perceber que alguns tokens são símbolos abstratos;
- Os símbolos representam operadores.

Análise sintática

- Analisador sintático: utiliza os primeiros componentes dos tokens para gerar uma representação de árvore;
- A árvore representa a estrutura gramatical dos tokens;
- Representação da árvore de sintaxe:
  - Cada nó interior representa uma operação;
  - Filhos dos nós representam os argumentos da operação.
- As próximas fases do compilador utilizam a estrutura gramatical;
- Gerar o programa fonte;
- Gerar o programa objeto.

Árvore de sintaxe

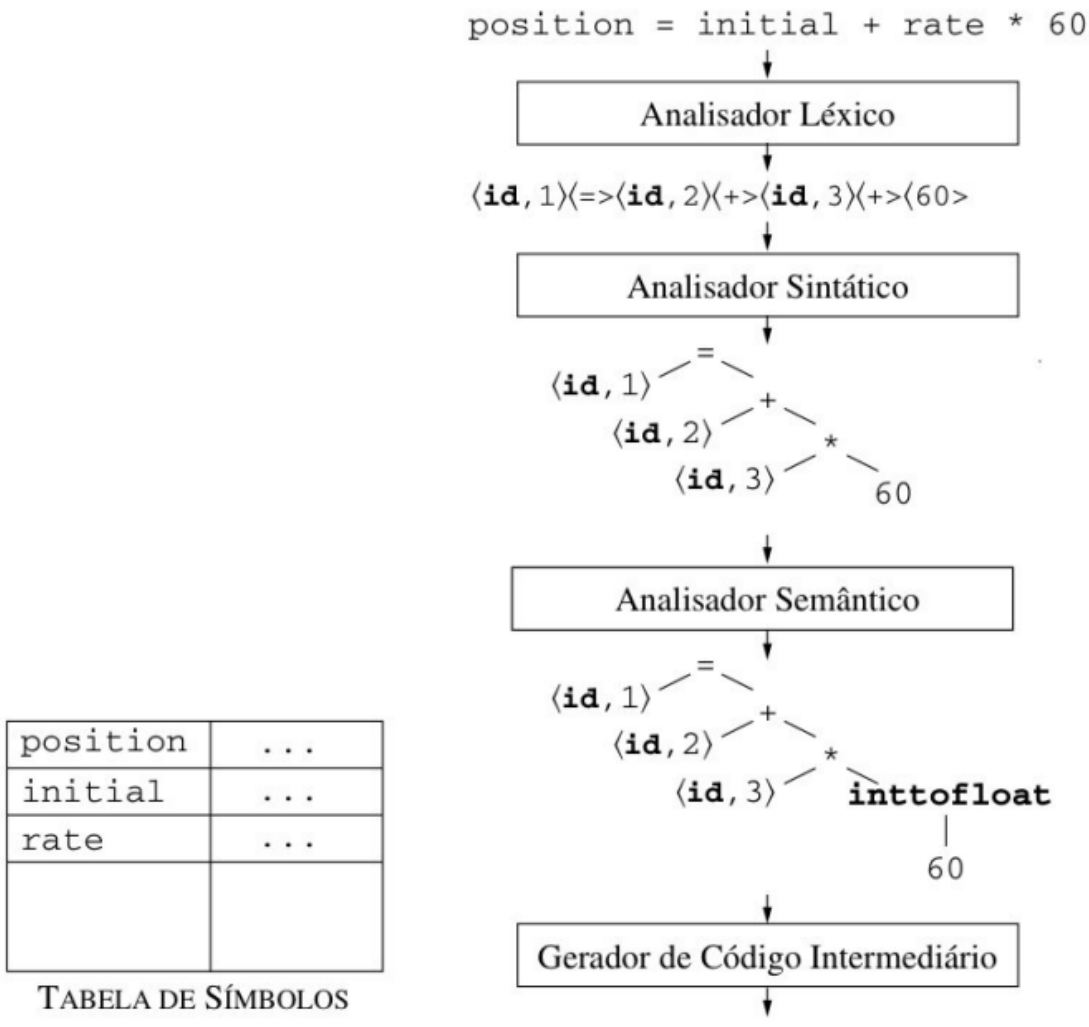

Figura 3.1: Tradução de uma instrução para o exemplo 4 [Amarasinghe and Rinard, 2010]

Análise semântica

- Utiliza a árvore de sintaxe e a tabela de símbolos para verificar a consistência semântica;
- Relaciona a semântica do programa fonte com a definição da linguagem;
- Reúne informações sobre tipos e salva na árvore de sintaxe;
- Realiza a verificação de tipos;
- Permite algumas conversões de tipo chamada coerções:
  - Aplicar um operador aritmético binário a um par de inteiros;
  - Ao aplicar um operador a um ponto flutuante e a um inteiro, esse último pode ser convertido para ponto flutuante;
  - Ex.: Na figura 26 aparece o operador inttofloat.

Código intermediário

- O compilador pode produzir uma ou mais representações intermediárias;
- Após as análises sintática e semântica, o compilador normalmente gera uma representação mais próxima da linguagem de máquina;
- Objetivos da representação de baixo nível:
  1. Ser facilmente produzida;
  2. Ser facilmente traduzida para a máquina alvo.

Tradução e atribuição

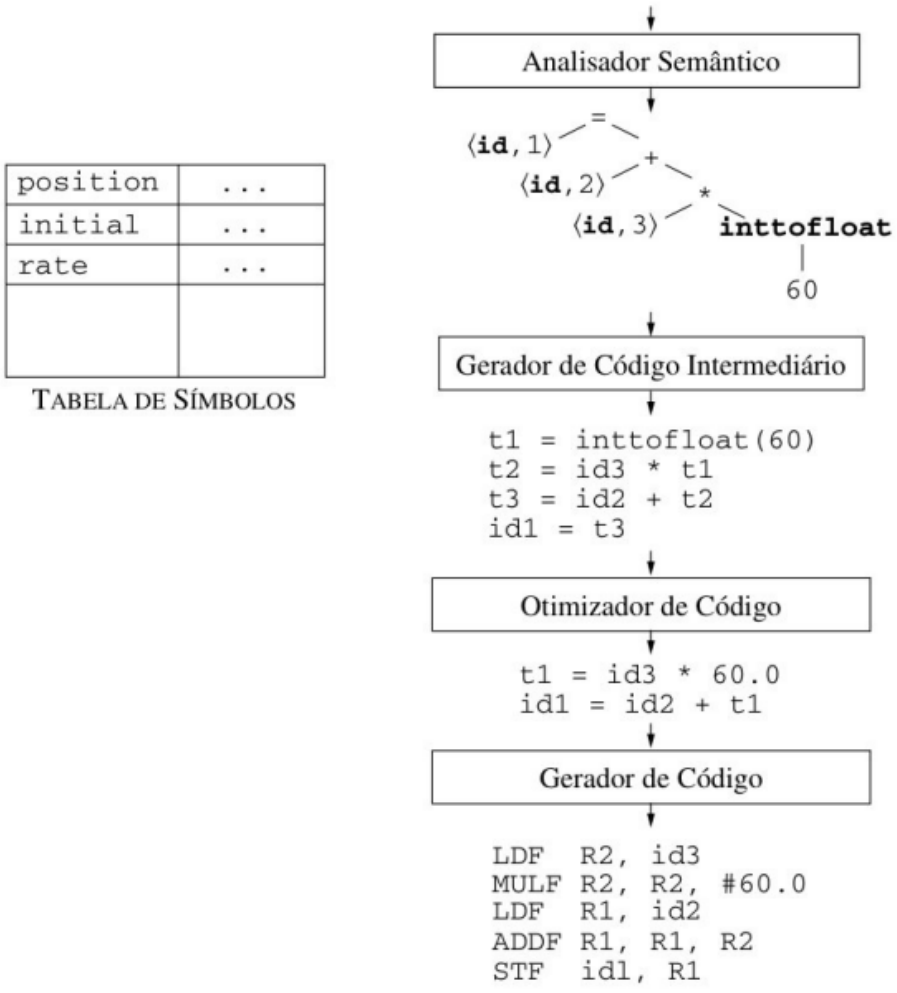

Figura 3.2: Tradução e atribuição para o exemplo 4 [Amarasinghe and Rinard, 2010]

Três endereços

- Representação em código de três endereços:

$$t1 = inttofloat(60)$$
$$t2 = id3 * t1$$
$$t3 = id2 + t2$$
$$id1 = t3$$

1. Cada instrução de atribuição tem no máximo um operador do lado direito;
   - Determinam a ordem de realização das operações.
2. O compilador precisa guardar o valor computador pela instrução;
   - Geração de nome intermediário.
3. Algumas instruções podem ter menos de três operandos.

OBRIGADO!!!
PERGUNTAS???


Aho, A., Lam, M., Sethi, R., and Ullman, J. (2007).
Compiladores–Princıpios Técnicas e Ferramentas.
Pearson, 2a. edition.


Amarasinghe, S. and Rinard, M. (2010).
Computer language engineering.
Disponível em `http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-035-computer-language-engineering-spring-2010/` Acessado em 02/08/2016.


Pinto, G. (2016).
Notas de aula do Prof. Guilherme Pinto.


Schwarz, K., Papadakis, H., and Mittal, R. (2016).
Compilers.
Disponível em `http://web.stanford.edu/class/cs143/` Acessado em 30/09/2016.