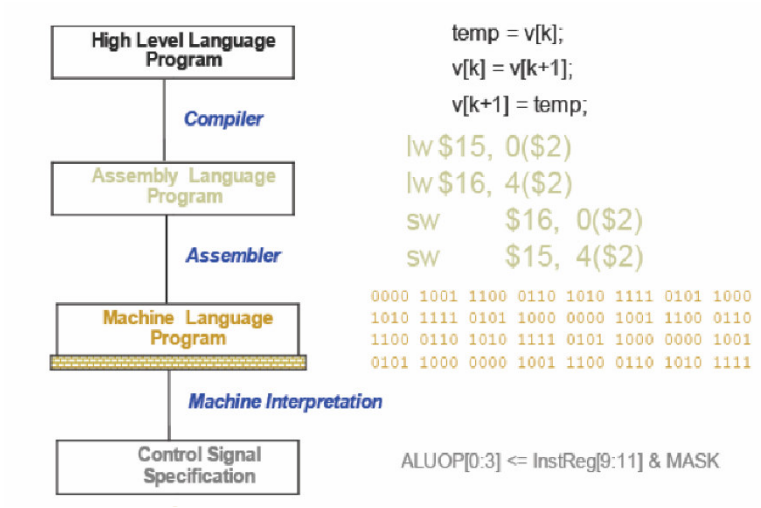
Compiladores

ED1 e ED2 Listas encadeadas, listas ordenadas, busca binária, arvores	ICC1 e ICC2 Arquivos, ponteiros, alocação dinamica, modularização, recursividade, matrizes	
	Compiladores	LM MOV, MUL, DIV, CMP, ...
LFA – gramáticas, automatos		

"*Quando se inventou o computador criou-se uma máquina a mais, quando se criou o compilador criou-se uma nova era tecnológica.*"



Um COMPUTADOR DIGITAL é uma **máquina** que pode resolver problemas executando uma série de **instruções,** é uma máquina programável. A seqüência de **instruções** que descrevem a maneira de se realizar uma determinada tarefa é chamada PROGRAMA. Tais instruções podem ser passadas para o computador em diversas linguagens, mas o processador entende apenas a **linguagem de máquina**, onde as instruções consistem de 0's e 1's , o que faz com que instruções em outras linguagens sejam traduzidas para a linguagem de máquina.

Podemos definir então linguagem de máquina como sendo o conjunto de instruções básicas que os circuitos eletrônicos de um determinado computador pode reconhecer e executar diretamente. Esta linguagem é uma linguagem primitiva (binária), e por isso é complicada para uso humano. Para facilitar a comunicação dos seres humanos com a máquina, utilizam-se linguagens de alto-nível, como C, C++, Delphi, Pascal, Fortran, Java ... Tais linguagens devem ser convertidas em linguagem de máquina para serem executadas. Temos dois métodos de conversão: tradução e interpretação.

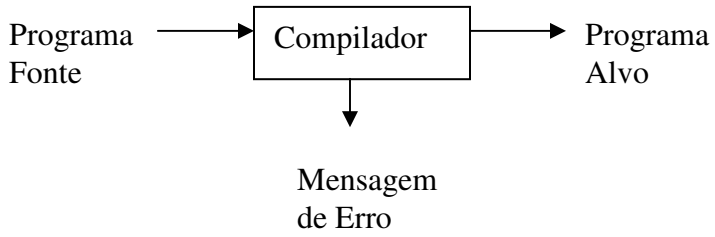
Na **tradução** (compilação), cada instrução do programa de alto nível é substituída por uma seqüência equivalente de instruções em linguagem de máquina. O programa resultante é composto inteiramente por instruções em linguagem de máquina.

Os tradutores são um sistema que aceita como entrada um programa escrito em uma linguagem de programação (programa fonte) e produz como resultado um programa equivalente em outra linguagem (programa alvo).

Os tradutores de linguagem de programação podem ser classificados em:

1) **Montadores (assemblers):** são aqueles tradutores que mapeiam instruções em linguagem simbólica (assembly) para instruções em linguagem de máquina, geralmente, numa relação de uma-para-uma, isto é, uma instrução de linguagem simbólica para uma instrução de máquina.
2) **Macro-assemblers**: são tradutores que mapeiam instruções em linguagem simbólica para linguagem de máquina, geralmente, numa relação de uma-para-várias. Muitas linguagens simbólicas possuem facilidades de definição de "macros" que são, na realidade, facilidades de expansão de texto em linguagem mnemônica. Um comando macro é traduzido para uma seqüência de comandos simbólicos antes de ser procedida a tradução efetiva para a linguagem de máquina.
3) **Compiladores**: são tradutores que mapeiam instruções em linguagem de alto nível para programas equivalentes em linguagem simbólica ou linguagem de máquina.
4) **Pré compiladores, pré-processadores ou filtros**: são tradutores que efetuam conversões entre duas linguagens de alto nível. Os pré-compiladores surgiram para facilitar a extensão de linguagens de alto nível existentes. As extensões são usadas, na maioria das vezes, para atender aplicações específicas, cujo objetivo é aprimorar o projeto e escrita de algoritmos. Por exemplo, existem pré-processadores FORTRAN, BASIC e COBOL estruturados que mapeiam programas em versões estruturadas dessas linguagens para programas em FORTRAN, BASIC e COBOL padrões.

Como importante parte do processo de tradução, o tradutor relata a seu usuário a presença de erros no programa fonte. Em geral, não será gerado nenhum código se algum erro for encontrado no programa fonte. Após a correção é necessária nova tradução. Exemplos de linguagens traduzidas são C e Pascal.



A **interpretação** é executada por um programa denominado interpretador. Um interpretador não gera nenhum tipo de código. O interpretador converte as instruções de um programa fonte para a forma binária (linguagem de máquina) e as executa imediatamente. Em geral, para executar cada ação possível existe um subprograma

(escrito na linguagem da máquina do computador hospedeiro), que recebe programas escritos em linguagem de alto nível como dados de entrada e efetua a execução examinando uma instrução de cada vez e executando a seqüência equivalente de instruções em linguagem de máquina. Assim, a interpretação de um programa é feita pela chamada daqueles subprogramas, em sequência apropriada. Estes tradutores estabelecem o conceito de maquinas virtuais, pois a execução do programa alvo está atrelada a essa ferramenta como se fosse o próprio computador.

Um computador é um conjunto de algoritmos e estruturas de dados capazes de armazenar e executar programas, não sendo necessariamente uma entidade física. Ele pode ser uma entidade lógica, construída via software. Neste caso, o computador é chamado de computador virtual ou máquina virtual.

Os passos para interpretação são:

- Obter próximo comando
- Determinar que ações devem ser executadas
- Executar estas ações

Pode-se dizer que uma instrução de uma linguagem interpretada é uma chamada de rotina. Quando erro é encontrado somente aquele comando é retraduzido de forma interativa. Ex. Haskell

O código intermediário necessita de uma posterior retradução na forma de compilação, montagem ou mesmo interpretação. Ex.: o compilador da máquina "Pascal Concorrente" gerava, a partir de um programa escrito em uma linguagem chamada "Pascal Concorrente", um código em pseudo linguagem de montagem. Esse código, para ser executado, necessitava de uma interpretação. O interpretador era escrito na linguagem de montagem da máquina na qual se desejava executar o programa.

Vantagens da compilação em relação à interpretação

a. A execução é mais rápida, pois não necessitam de qualquer tradução durante a execução.
b. Economia de memória na execução, pois não requerem a carga de maquina virtual na memória.
c. Maior controle sobre o código gerado
d. Pode efetuar otimização de código
e. Geram um programa objeto bem mais eficiente, pois esse interage diretamente com o computador.

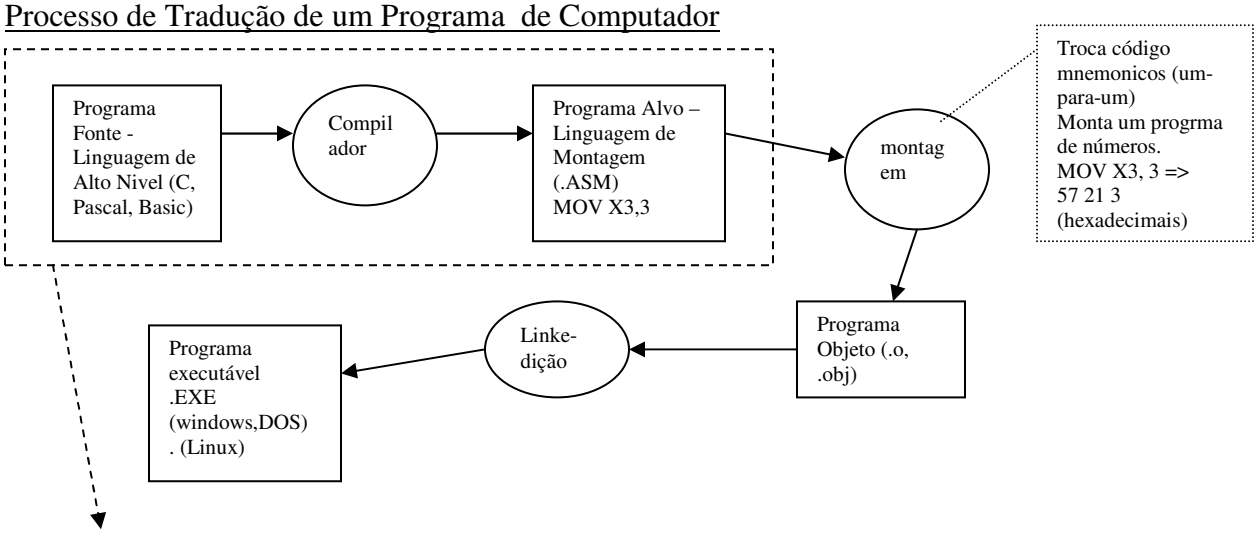
Desvantagens da compilação em relação à interpretação

a. Inadequada para usuários novatos, pois cada vez que se comete um erro deve-se compilar o programa inteiro, já o interpretador, ao perceber erros durante a execução do programa, pausa, o erro então deve ser corrigido, sem perda da sua mansa de testes.
b. Perde-se a referência com o código fonte, em caso de erro é difícil apontar a consequência do erro.
c. Construção mais complexa
d. Com a geração do código intermediário pode-se gerar programas com excelente grau de portabilidade, desenvolvendo-se máquinas virtuais para diferentes plataformas.

O Processo de Compilação

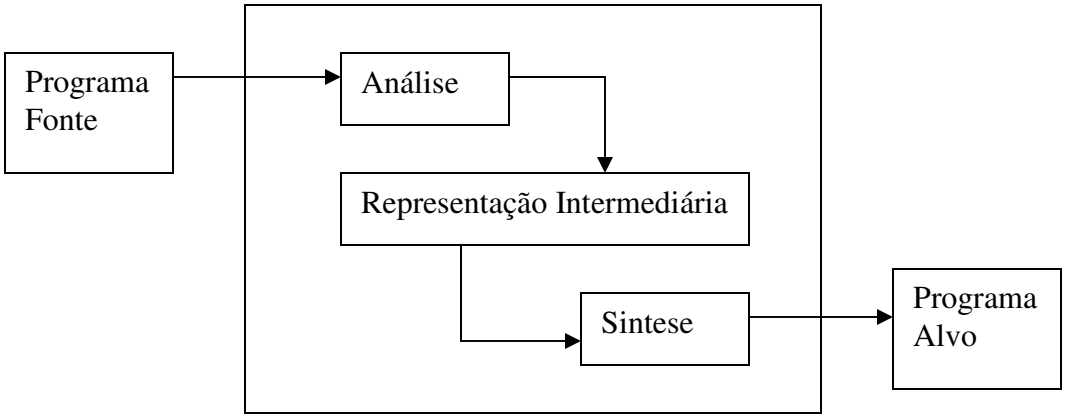
Um compilador é basicamente um programa que traduz um texto de programa escrito em alguma linguagem denominada *linguagem fonte* (normalmente de alto nível) para uma outra linguagem denominada *linguagem objeto* (normalmente de baixo nível).

Processo de Tradução de um Programa de Computador



Parte que iremos estudar

Existem duas partes na compilação: a análise e a síntese. A parte de análise divide o programa fonte nas partes constituintes e cria uma representação intermediária do mesmo. A parte de síntese constrói o programa alvo desejado, a partir da representação intermediária. Das duas, a síntese requer as técnicas mais especializadas.



Análise

Durante a fase de análise o compilador "lê" o texto escrito em linguagem fonte e verifica se está escrito de acordo com as construções da linguagem. São armazenadas internamente ao compilador uma representação da gramática que descreve a forma das construções válidas e uma representação das regras semânticas. Erros detectados na análise devem ser reportados ao programador. Durante a fase de análise é feita a geração de uma representação intermediária. Na compilação, a análise consiste em três fases (tipos de análise).

1. Análise Léxica (lê palavra)

   Também chamada de esquadrinhamento (scanning). É a fase da análise linear, na qual um fluxo de caracteres constituindo um programa é lido da esquerda para a direita e agrupado em tokens, que são seqüências de caracteres tendo um significado coletivo. Tokens são palavras válidas, conhecidos também como lexemas. Exemplos são: WHILE, x1, 23. Também são detectados nesta fase erros léxicos, exemplo: !epa não é um lexema válido em C. É considerada apenas a relação dos caracteres entre si para a formação do lexema. O relacionamento dos lexemas entre si fica a cargo da análise sintática.

2. Análise Sintática (lê frase)

   Ou parsing. É uma análise hierárquica, também chamada de análise gramatical. Envolve o agrupamento dos tokens do programa fonte em frases gramaticais, que são usadas pelo compilador, a fim de sintetizar a saída. Usualmente, as frases gramaticais do programa fonte são representadas por uma árvore gramatical.

3. Análise Semântica (verifica validade, sentido)

   Verifica os erros semânticos no programa fonte e captura informações de tipo para a fase subseqüente de geração de código. Faz verificação quanto à compatibilidade de tipos, se um identificador não foi declarado, etc...

Representação Intermediária

Durante o processo de análise, o computador pode gerar uma representação intermediária do programa fonte.

Vantagens:

- Facilita o trabalho de portar programas para máquinas de arquitetura diferentes.
- Algumas formas de representação intermediária são passíveis de otimização, facilitando e antecipando o trabalho de otimização do código objeto que é feita durante o processo de síntese.

Síntese

Durante a fase de síntese a representação intermediaria pode ser otimizada e é posteriormente traduzida em linguagem objeto. A síntese pode envolver ainda a otimização do código gerado em linguagem objeto.

Características de Algumas Linguagens que Dificultam o Processo de Análise

Em PL/I as palavras chaves não são palavras reservadas. THEN pode significar parte do comando IF-THEN-ELSE, ou pode ser um identificador. A construção abaixo é válida em PL/I

IF THEN THEN THEN=ELSE ELSE ELSE=THEN

Em geral, os tradutores de linguagem de programação (compiladores, interpretadores) são programas bastante complexos. Porém, devido à experiência acumulada ao longo dos anos e, principalmente, ao desenvolvimento de teorias relacionadas às tarefas de análise e síntese de programas, existe um consenso sobre a estrutura básica desses processadores.

Independentemente da linguagem a ser traduzida ou do programa objeto a ser gerado, o tradutor, de um modo geral, compõem-se de funções padronizadas, que compreendem a análise do programa fonte e a posterior síntese para a derivação do código objeto.

O processo de tradução é, comumente, estruturado em fases, conforme a figura abaixo, no qual cada fase se comunica com a seguinte através de uma linguagem intermediaria adequada, transformando o programa fonte de uma representação para a outra. Na prática (ponto de vista de implementação), seguidamente, a distinção entre as fases não é muito clara, podendo até, ter-se algumas das fases agrupadas e a representação intermediária entre as mesmas não precisando ser explicitamente construída.

As Fases de Análise

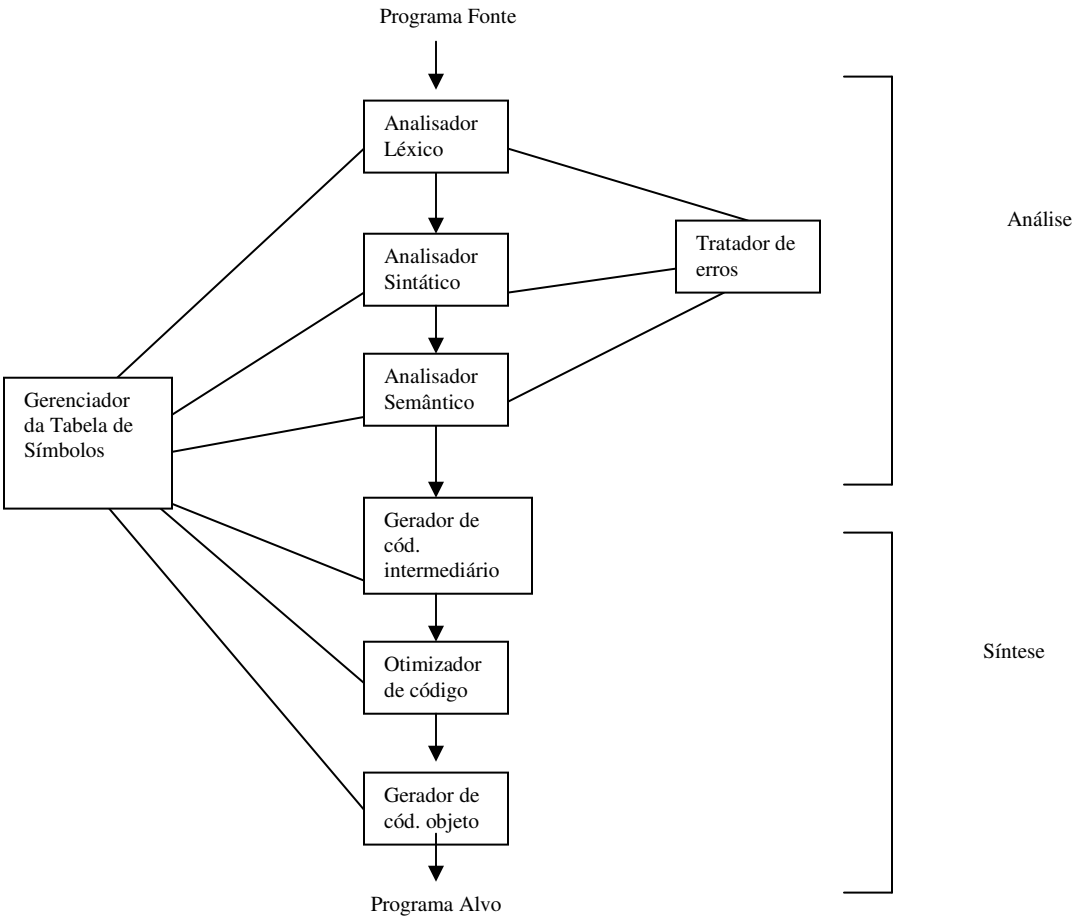
Á medida que a tradução progride, a representação interna do compilador para o programa fonte muda. Vamos exemplificar essas representações considerando a tradução dos seguintes enunciados:

Enunciado 1:

```
montante := depósito_inicial + taxa_de_juros * 60
```

Enunciado 2:

```
WHILE I<100 do I:=J+I;
```



1. Análise Léxica

O objetivo principal desta fase é identificar seqüências de caracteres que constituem unidades léxicas (“tokens”). O analisador léxico lê o código fonte, caractere a caractere verificando se os caracteres lidos pertencem ao alfabeto da linguagem, identificando as unidades léxicas (tokens), e desprezando comentários e espaços em branco. Os tokens constituem classes de símbolos tais como palavras reservadas, delimitadores, identificadores, etc., e podem ser representados, internamente, através do próprio símbolo (como no caso dos delimitadores e das palavras reservadas) ou por um par ordenado, no qual o primeiro elemento indica a classe do símbolo, e o segundo, um índice para uma área onde o próprio símbolo foi armazenado (por exemplo, um identificador e sua entrada numa tabela de identificadores). Além da identificação de tokens, o analisador léxico, em geral, inicia a construção da Tabela de Símbolos e envia mensagens de erro caso identifique unidades léxicas não aceitas pela linguagem em questão.

A saída do analisador léxico é uma cadeia de tokens que é passada para a próxima fase, a Análise Sintática. Em geral, o Analisador Léxico é implementado como uma subrotina que funciona sob o comando do Analisador Sintático.

Após a Análise Léxica dos enunciados 1 e 2 poderia-se ter a seguinte cadeia de tokens:
Enunciado 1:

```
id1 := id2 + id3 * 60
```

Enunciado 2:

```
[while,][id,7][<,][cte,100][do,][id,7][:=,][id,12][+,][id,7][;,]
```

onde, palavras reservadas, operadores e delimitadores são representados pelo próprios símbolos, e identificadores de variáveis e constantes numéricas são representados por um par [classe do símbolo, índice de tabela]

2. Analise Sintática

O analisador léxico vê o programa como um fluxo de caracteres e gera uma lista de tokens. O analisador sintático vê o programa fonte como uma lista de tokens.
A fase de análise sintática tem por função verificar se a estrutura gramatical do programa está correta (isto é, se essa estrutura foi formada usando as regras gramaticais da linguagem).

O Analisador Sintático identifica seqüências de símbolos que constituem estruturas sintáticas (por exemplo, expressões, comandos), através de uma varredura ou "parsing" da representação interna (cadeia de tokens) do programa fonte. O Analisador Sintático produz (explicita ou implicitamente) uma estrutura em árvore, chamada árvore de derivação, que exibe a estrutura sintática do texto fonte, resultante da aplicação das regras gramaticais da linguagem. Em geral, a árvore de derivação não é produzida explicitamente, mas sua construção está implícita nas chamadas das rotinas recursivas que executam a análise sintática. Em muitos compiladores, a representação interna do programa resultante da análise sintática não é a árvore de derivação completa do texto fonte, mas uma árvore compactada (árvore de sintaxe) que visa a eliminar redundâncias e elementos supérfluos. Essa estrutura objetiva facilitar a geração do código que é a fase seguinte à análise. O analisador sintático é executado através de chamadas recursivas.

Outra função dos reconhecedores sintáticos é a detecção de erros de sintaxe, identificando clara e objetivamente a posição e o tipo de erro ocorrido. Mesmo que erros tenham sido encontrados, o Analisador Sintático deve tentar recuperá-los prosseguindo a análise do texto restante.

Muitas vezes, o Analisador Sintático opera conjuntamente com o Analisador Semântico, cuja principal atividade é determinar se as estruturas sintáticas analisadas fazem sentido, ou seja, verificar se um identificador declarado como variável é usado como tal; se existe compatibilidade entre operandos e operadores em expressões; etc. Por exemplo, em Pascal, o comando while tem a seguinte sintaxe:

while <expressão> do <comando>;

a estrutura <expressão> deve apresentar-se sintaticamente correta, e sua avaliação deve retornar um valor do tipo lógico. Isto é, a aplicação de operadores (relacionais/lógicos) sobre os operandos (constantes/variáveis) deve resultar num valor do tipo lógico (verdadeiro/falso).

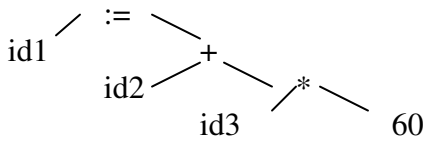
As regras gramaticais que definem as construções da linguagem podem ser descritas através de produções (regras que produzem, geram) cujos elementos incluem *símbolos terminais* (aqueles que fazem parte do código fonte) e *símbolos não-terminais*

(aqueles que geram outras regras). No exemplo que segue, as produções são apresentadas na *Forma Normal de Backus*.

O exemplo abaixo mostra produções que geram comandos de atribuição e comandos iterativos. Os terminais aparecem em negrito e os símbolos não terminais aparecem delimitados por "<" e ">". Os comandos while e de atribuição podem ser definidos (parcialmente) pelas seguintes produções:

<comando> → <while> | <atrib> | ...
<while> → **while** <expr_bool> **do** <comando>
<atrib> → <variavel> := <expr_arit>
<expr_bool> → <expr_arit> < <expr_arit>
<expr_arit> → <expr_arit> + <termo> | <termo>
<termo> → <numero> | <variavel>
<variavel> → I | J
<numero> → 100

Para o nosso enunciado 1 seria formada a seguinte árvore de derivação:



3. Análise Semântica

Tem por finalidade verificar se as estruturas do programa irão fazer sentido durante a execução, ou seja, verificar se um identificar declarado como variável é usado como tal. Verifica erros do tipo: variável declarada mais não utilizada, variável utilizada e não declarada, incompatibilidade de tipos, etc..

As fases até aqui descritas constituem módulos que executam tarefas analíticas. As fases seguintes trabalham para construir o código objeto: geração de código intermediário, otimização e geração de código objeto.

As Fases de Síntese

4. Gerador de Código Intermediário

Esta fase utiliza a representação interna produzida pelo Analisador Sintático e gera como saída uma sequência de código. Esse código pode, eventualmente, ser o código objeto final, mas, na maioria das vezes, constitui-se num código intermediário, pois a tradução de código fonte para objeto em mais de um passo apresenta algumas vantagens:

1. Possibilita a otimização de código intermediário gerando código objeto final mais eficiente
2. Resolve gradualmente as dificuldades da passagem de código fonte para código objeto (alto nível para baixo nível), já que o código fonte pode ser

visto como um texto condensado que “explode”em inúmeras instruções elementares de baixo nível.

Podemos pensar nessa representação intermediária como um programa para uma máquina abstrata. Essa representação intermediária deveria possuir duas propriedades importantes: ser fácil de produzir e fácil de traduzir no programa alvo. A geração de código intermediário pode estar estruturalmente distribuída nas fases anteriores (análise sintática e semântica) ou mesmo não existir (tradução direta para código objeto), no caso de tradutores bem simples.

A grande diferença entre o código intermediário e o código objeto final é que o intermediário não especifica detalhes tais como quais registradores serão usados, quais endereços de memória serão referenciados, etc...

Por exemplo, para o enunciado 2, o gerador de código intermediário, poderia produzir a seguinte seqüência de instruções:

```
L0   if I < 100  goto  L1
     goto L2
L1   TEMP := J + I
     I := TEMP
     goto L0
L2   . . .
```

Há vários tipos de código intermediário: quádruplas, triplas, notação polonesa pós-fixada, etc. A linguagem intermediária do exemplo acima é chamada “código de três endereços”, pois cada instrução tem no máximo três operandos. Já o enunciado 1 poderia ser expresso no código de três endereços como:

```
temp1:= inttoreal (60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

Esta forma intermediária possui várias propriedades. Primeiro, cada instrução de três endereços possui, no máximo, um operador, além do de atribuição. Então, ao gerar essas instruções, o compilador precisa decidir sobre a ordem em que as mesmas devam ser realizadas; a multiplicação precede a adição no programa fonte do enunciado. Segundo, o compilador precisa gerar um nome temporário para receber o valor computado em cada instrução. Terceiro, algumas instruções de três endereços possuem menos do que três operandos, por exemplo, a primeira e a última instruções acima.

5. Otimização de Código

A fase de otimização tenta melhorar o código intermediário, de tal forma que venha resultar um código de máquina mais rápido em tempo de execução e otimizado em termos de espaço de memória. Algumas otimizações são triviais. Por exemplo, o algoritmo que gera o código intermediário do enunciado 1, após a análise semântica, usando uma instrução para cada operador na representação em árvore, ainda que exista

uma maneira melhor de se realizar a mesma computação, usando-se as duas instruções, ficaria:

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

Não existe nada de errado com esse algoritmo simples, dado que o problema pode ser corrigido durante a fase de otimização de código. Ou seja, o compilador pode deduzir que a conversão de 60, da representação de inteiro para a de real, pode ser realizada uma vez, e para todo o sempre, em tempo de compilação, e, por conseguinte, a operação inttoreal pode ser eliminada. Além do mais, temp3 é usada para transmitir seu valor a id1 uma única vez. Torna-se seguro, então, substituir temp3 por id1 e isso feito torna o ultimo enunciado do código acima desnecessário.

Já para o enunciado 2 o seguinte código otimizado poderia ser obtido:

```
L0   if I < 100  goto  L2
     I := J+I
     goto L0
L2   . . .
```

Existe uma grande variação na quantidade de otimizações de código que cada compilador executa. Naqueles que mais a realizam, chamados de “compiladores otimizantes”, uma porção significativa de seus tempos é gasta nessa fase. Entretanto, existem otimizações simples que melhoram significativamente o tempo de execução do programa alvo, sem alongar o tempo de compilação.

6. Geração de Código

A fase final do compilador é a geração do código alvo, consistindo normalmente de código de máquina relocável ou código de montagem. Esta fase tem como objetivos: produção de código objeto, reserva de memória para constantes e variáveis, seleção de registradores. É a fase mais difícil, pois requer uma seleção cuidadosa das instruções e dos registradores da máquina alvo a fim de produzir código objeto eficiente. Existem tradutores que possuem mais uma fase para realizar a otimização do código objeto, isto é, otimização do código dependente de máquina.

As instruções intermediárias são, cada uma, traduzidas numa seqüência de instruções de maquina que realizam a mesma tarefa. Um aspecto crucial é a atribuição das variáveis aos registradores.

Por exemplo, para o enunciado 1, usando-se os registradores 1 e 2, a tradução do código:

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

poderia se tornar

```
MOVF      id3, R2
```

```
MULF      #60.0, R2
MOVF      id2, R1
ADDF      R2, R1
MOVF      R1, id1
```

O primeiro e o segundo operandos de cada instrução especificam o emissor e o receptor, respectivamente. O F em cada instrução nos informa que as instruções lidam com números em ponto flutuante. Esse código copia o conteúdo do endereço id3 no registrador 2 e, então, multiplica-o pela constante 60.0. O # significa que o 60.0 deve ser tratado como uma constante. A terceira instrução copia o conteúdo de id2 no registrador 1 e o adiciona ao valor previamente computado no registrador 2. Finalmente, o valor no registrador 1 é copiado para o endereço de id1.

Para o enunciado 2 obter-se-ia o código objeto final abaixo, código este baseado na linguagem simbólica de um microcomputador PC 8086

```
L0   MOV  AX,I
     CMP  AX,100
     JGE       L2
     MOV  AX,J
     MOV  BX,I
     ADD  BX
     MOV  I,AX
     JMP  L0
L2   . . .
```

A seguir iremos comentar sobre dois módulos que não constituem uma fase do compilador no sentido das já discutidas até o momento, são eles: Gerencia de Tabelas de Simbolos e Atendimento a Erros.

Gerência de Tabelas de Símbolos

Compreende um conjunto de tabelas e rotinas associadas que são utilizadas por quase todas as fases do tradutor.

Algumas das tabelas usadas são fixas para cada linguagem, por exemplo, a tabela de palavras reservadas, tabelas de delimitadores, etc. Entretanto, a estrutura que possui importância fundamental é aquela que é montada durante a análise do programa fonte, quando são coletadas informações sobre os seus diversos atributos. Esses atributos podem providenciar informações sobre a memória reservada para o identificador, seu tipo, escopo (onde é válido no programa) e ainda informações sobre:

- declarações de variáveis;
- declarações dos procedimentos e subrotinas;
- parâmetros de subrotinas; etc.

Essas informações são armazenadas na Tabela de Símbolos (às vezes chamada de tabela de nomes ou lista de identificadores). Uma tabela de símbolos é uma estrutura de dados contendo um registro para cada identificador, com os campos contendo os atributos do identificador. A cada ocorrência de um identificador no programa fonte, a tabela é

acessada, e o identificador é procurado na tabela. Quando encontrado, as informações associadas a ele são comparadas com as informações obtidas no programa fonte, sendo que qualquer nova informação é inserida na tabela.

Os dados a serem coletados e armazenados na tabela de símbolos dependem da linguagem, do projeto do tradutor, do programa objeto a ser gerado. Entretanto, os atributos mais comumente registrados são:

- para variáveis: classe(var), tipo, endereço no texto, precisão, tamanho;
- parâmetros formais: classe (par), tipo, mecanismo de passagem;
- procedimentos/subrotinas: classe (proc), número de parâmetros.

A tabela de símbolos deve ser estruturada de uma forma tal que permita rápida inserção e extração de informações, porém deve ser tão compacta quanto possível.

Atendimento a Erros

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

É fundamental que o tradutor prossiga na tradução, após a detecção de erros, de modo que o texto seja totalmente analisado.

As fases de análise sintática e semântica tratam usualmente de uma ampla fatia dos erros detectáveis pelo compilador. A fase de análise léxica pode detectá-los quando os caracteres remanescentes na entrada não formem qualquer token da linguagem. Os erros, onde o fluxo de tokens viole as regras estruturais (sintaxe) da linguagem, são determinados pela fase de análise sintática. Durante a análise semântica, o compilador tenta detectar as construções que possuam a estrutura sintática correta, sem nenhuma preocupação com o significado da operação envolvida, como, por exemplo, ao tentarmos adicionar dois identificadores, um dos quais seja um nome de um array e o outro o nome de um procedimento.

Análise Léxica

A análise léxica é a primeira fase do compilador. A função do analisador léxico, também denominado scanner, é fazer a leitura do programa fonte, caractere a caractere, e traduzi-lo para uma sequência de *símbolos léxicos*, também chamados de tokens, que o parser utiliza para a análise sintática. Essa interação é comumente implementada fazendo-se com que o analisador léxico seja uma sub-rotina ou uma co-rotina do parser. Ao receber um comando "obter próximo token",o analisador léxico lê os caracteres de entrada até que possa identificar o próximo token

Exemplos de símbolos léxicos são as palavras reservadas, os identificadores, as constantes e os operadores da linguagem. Durante o processo de análise léxica, são desprezados caracteres não significativos como espaços em branco e comentários. Além de reconhecer os símbolos léxicos, o analisador também realiza outras funções, como

armazenar alguns desses símbolos (tipicamente identificadores e constantes) em tabelas internas e indicar a ocorrência de erros léxicos. A seqüência de *tokens* produzida (reconhecida) pelo analisador léxico é utilizada como entrada pelo módulo seguinte do compilador, o analisador sintático. É interessante observar que o mesmo programa fonte é visto pelos analisadores léxico e sintático como sentenças de linguagens diferentes. Para o analisador léxico, o programa fonte é uma seqüência de palavras de uma *linguagem regular*. Para o analisador sintático, essa seqüência de *tokens* constitui uma sentença de uma *linguagem livre de contexto*.

Como o analisador léxico é a parte do compilador que lê o texto-fonte, também pode realizar algumas tarefas secundárias ao nível da interface com o usuário. Uma delas é a de remover do programa fonte os comentários e espaços em branco, os últimos sob a forma de espaços, tabulações e caracteres de avanço de linha. Uma outra é a de correlacionar as mensagens de erro do compilador com o programa fonte. Por exemplo, o analisador léxico pode controlar o número de caracteres examinados, de tal forma que um número de linha possa ser relacionado a uma mensagem de erro associadas ao mesmo. Se a linguagem fonte suporta algumas funções sob a forma de macros pré-processadas as mesmas também podem ser implementadas na medida em que a análise léxica vá se desenvolvendo.

Algumas vezes, os analisadores léxicos são divididos em duas fases em cascata, a primeira chamada de "varredura" (scanning) e a segunda de "análise léxica". O scanner é responsável por realizar tarefas simples, enquanto o analisador léxico propriamente dito realiza as tarefas mais complexas. Por exemplo, um compilador Fortran pode usar um scanner para eliminar os espaços da entrada.

Uma forma simples de se construir um analisador léxico é escrever um diagrama que ilustre a estrutura de tokens da linguagem fonte e então introduzi-lo manualmente num programa que os localize. Analisadores léxicos eficientes podem ser construídos dessa forma.

Temas da Analise Léxica

Existem várias razões para se dividir a fase de análise de compiladores em análise léxica e análise gramatical (parsing)

1. Um projeto mais simples talvez seja a consideração mais importante. A separação das analises léxica e sintática freqüentemente nos permite simplificar um ou outra dessas fases. Por exemplo, um parser que incorpore as convenções para comentários e espaços em branco é significativamente mais complexo do que um que assuma que os mesmos já tenham sido removidos pelo analisador léxico. Se estivermos projetando uma nova linguagem, separar as convenções léxicas das sintáticas pode levar a um projeto global de linguagem mais claro.
2. A eficiência do compilador é melhorada. Um analisador léxico separado nos permite construir um processador especializado e potencialmente mais eficiente para a tarefa. Uma grande quantidade de tempo é gasta lendo-se o programa fonte e particionando-o em tokens. Técnicas de **buferização** especializadas para a leitura de caracteres e o processamento de tokens podem acelerar significativamente o desempenho de um compilador.

3. A portabilidade do compilador é realçada. As peculiaridades do alfabeto de entrada e outras anomalias específicas de dispositivos podem ser restringidas ao analisador léxico. A representação de símbolos especiais ou não padrão, pode ser isolada no analisador léxico.

Tokens, Padrões, Lexemas

Quando se fala sobre a análise léxica, usamos os termos "*token*", "padrão"e "lexema" com significados específicos. Em geral, existe um conjunto de cadeias de entrada para as quais o mesmo *token* é produzido como saída. Esse conjunto de cadeias é descrito por uma regra chamada de um *padrão* associado ao *token* de entrada. O padrão é dito *reconhecer* cada cadeia do conjunto. Um lexema é um conjunto de caracteres no programa fonte que é reconhecido pelo padrão de algum *token*. Por exemplo, no enunciado:

```
const  pi = 3.14;
```

a subcadeia `pi` é um lexema para o token "identificador".

Tratamos os *tokens* como símbolos terminais na gramática para a linguagem fonte, usando nomes em negrito para representá-los. Os lexemas reconhecidos pelo padrão do *token* representam cadeias de caracteres no programa fonte, e podem receber um tratamento conjunto, como instâncias de uma mesma unidade léxica (por exemplo, instâncias de identificadores, números, etc.).

Na maioria das linguagens de programação, as seguintes construções são tratadas como *tokens*: palavras-chave, operadores, identificadores, constantes, literais, cadeias e símbolos de pontuação, como parênteses, vírgulas e ponto e vírgulas. No exemplo acima, quando a seqüência de caracteres `pi` aparece no programa fonte, um *token* representando um identificador é repassado ao *parser*. O repasse de um *token* é freqüentemente implementado transmitindo-se um inteiro associado ao *token*.

Um padrão é uma regra que descreve o conjunto de lexemas que podem representar um *token* particular nos programas fonte. O padrão para o *token* **const** na tabela abaixo é exatamente a singela cadeia `const`, que soletra a palavra chave. O padrão para o *token* **relação** é o conjunto de todos os seis operadores relacionais. Para descrever precisamente os padrões para *tokens* mais complexos, como **id** (identificador) e **num** (número), usa se a notação de expressões regulares.

Exemplo de *Tokens*:

TOKEN	LEXEMAS EXEMPLO	DESCRIÇÃO INFORMAL DO PADRÃO
Const	`Const`	Const
If	`If`	If
Relação	`<,<=,=,<>,>,>=`	`< ou<= ou = ou <> ou >ou >=`
Id	`pi,contador, D2`	letras seguida por letras e/ou dígitos
Num	`3.1416, 0, 6.02E23`	qualquer constante numérica
Literal	`"conteúdo da memória"`	quaisquer caracteres entre aspas, exceto aspas

Tokens

A função do analisador léxico é ler uma seqüência de caracteres que constitui um programa fonte e coletar, dessa seqüência, os *tokens* (palavras de uma linguagem regular) que constituem o programa. Os *tokens* ou símbolos léxicos são as unidades básicas do texto do programa. Cada *token* é representado internamente por três informações:

- CLASSE: o tipo do *token* reconhecido. Exemplos de classes são: identificadores, constantes numéricas, cadeias de caracteres, palavras reservadas, operadores e separadores.
- VALOR: depende da classe. Para *tokens* da classe constante inteira, por exemplo, o valor do *token* pode ser o número inteiro representado pela constante. Para *tokens* da classe identificador, o valor pode ser a seqüência de caracteres, lida no programa fonte, que representa o identificador, ou o apontador para a entrada de uma tabela que contém essa seqüência de caracteres. Algumas classes de *tokens*, como palavras reservadas, não têm valor associado. Nesta função os *tokens* podem ser divididos em dois grupos:
  - Token Simples: não tem valor associado(como as palavras reservadas, operadores e delimitadores) porque a classe do *token* o descreve totalmente. Esses *tokens* correspondem a elementos fixos da linguagem.
  - Token com Argumento: têm valor associado. Correspondem aos elementos da linguagem definidos pelo programador como, por exemplo, identificadores, constantes numéricas e cadeias de caracteres.
- POSIÇÃO: local do texto fonte onde ocorreu o token. Essa informação é utilizada, principalmente, para indicar o local de erros.

Exemplo: `WHILE I < 100 do I := J + I;`

```
[while,][id, 7][<,][cte,13][do,][id,7][:=,][id, 12][+,][id,7][;,]
```

Para simplificar, os tokens estão representados por pares (omitiu-se a posição). Identificadores e constantes numéricas estão representados pelo par `[classe do token, indice tabela].` As classes para palavras reservadas constituem-se em abreviações dessas, não sendo necessário passar seus valores para o analisador sintático. Para delimitadores e operadores, a classe é o próprio valor do *token*. Usualmente, os compiladores representam a classe de um *token* por um número inteiro para tornar a representação mais compacta. Neste texto, empregou-se uma representação simbólica para ajudar a compreensão.

Tabela de Símbolos

A tabela de símbolos é uma estrutura de dados gerada pelo compilador com o objetivo de armazenar informações sobre os nomes (identificadores de variáveis, de parâmetros, de funções, de procedimentos, etc.) definidos pelo programa fonte. A Tabela de Símbolos associa atributos (tais como tipo, escopo, limites no caso de vetores e

números de parâmetros no caso de funções) aos nomes definidos pelo programador. Em geral, a Tabela de Símbolos começa a ser construída durante a analise léxica, quando os identificadores são reconhecidos. Na primeira vez que um identificador é encontrado, o analisador léxico armazena-o na tabela, mas talvez não tenha ainda condições de associar atributos a esse identificador. Às vezes, essa tarefa so pode ser executada durante as fases de análise sintática e semântica. Toda vez que um identificador é reconhecido no programa fonte, a Tabela de Símbolos é consultada, a fim de verificar se o nome já está registrado; caso não esteja, é feita sua inserção na tabela. É importante que o compilador possa variar dinamicamente o tamanho da Tabela de Símbolos. Se o tamanho for fixo, este deve ser escolhido grande suficiente para permitir a análise de qualquer programa fonte que se apresente.

Existem vários modos de organizar e acessar tabelas de símbolos. Os mais comuns são através de listas lineares, árvores binárias e tabelas hash. Lista linear é o mecanismo mais simples, mas seu desempenho é pobre quando o número de consulta elevado. Tabelas hash têm melhor desempenho, mas exigem mais memória e esforço de programação.

Cada entrada na Tabela de Símbolos está relacionada com a declaração de um nome. As entradas podem não ser uniforme para classes distintas de identificadores. Por exemplo, entradas para identificadores de funções requerem registro do número de parâmetros, enquanto entradas para identificadores de matrizes requerem registro dos limites inferior e superior de cada dimensão. Nesses casos, pode-se ter parte da entrada uniforme e usar ponteiros para registros com informações adicionais.

O armazenamento dos nomes pode ser feito diretamente na tabela ou em uma área distinta. No primeiro caso, temos um desperdício de memória pela diversidade de tamanho dos identificadores; no segundo, a recuperação de nomes é ligeiramente mais demorada.

Especificação de um Analisador Léxico

Conforme já referido, os analisadores léxicos são, usualmente, especificados através de notações para a descrição de linguagens regulares tais como autômatos finitos, expressões regulares ou gramáticas regulares. A especificação de um analisador léxico descreve o conjunto dos *tokens* que formam a linguagem. Também fazem parte dessa especificação a seqüência de caracteres que podem aparecer entre *tokens* e devem ser ignoradas tais como espaços em branco e comentários.

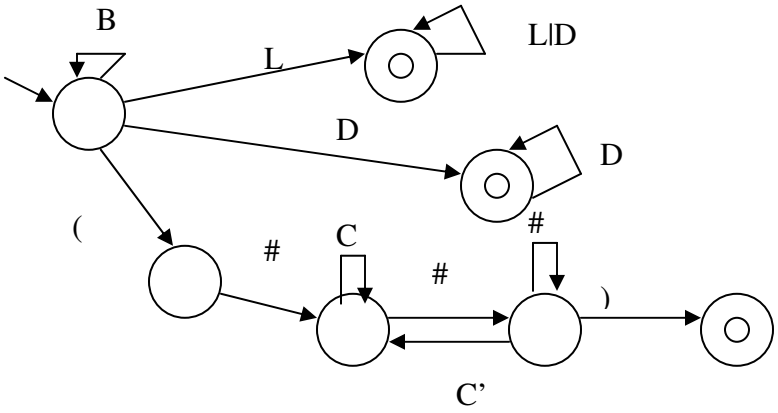
Na maioria das linguagens de programação, as palavras reservadas da linguagem são casos particulares dos identificadores e seguem a mesma notação desses. Logo, as especificações de analisadores léxicos, usualmente, não expressam, explicitamente, o reconhecimento de palavras reservadas. Essas palavras são armazenadas em uma tabela interna, que é examinada cada vez que um identificador é reconhecido. Se o identificador ocorre na tabela, então trata-se de uma palavra reservada. Caso contrário, trata-se de um identificador.

Exemplo: *Reconhecimento de tokens*

Considere a linguagem simples cujos *tokens* são os seguintes:

1. Identificadores são formados por uma letra seguida, opcionalmente, por uma ou mais letras e/ou digitos. L(L|D)*
1. Números inteiros formados por um ou mais digitos. D^+
2. Comentários delimitados por (# e #) . (# (L|D)* #)

Espaços em branco (B) são ignorados. Iremos representar letra por L e dígito por D. C representa qualquer caracter diferente de # e C' representa quanlquer caractere diferente de # ou)



O analisador léxico pode constituir um passo individual do compilador. Porém, em geral, o analisador léxico e o analisador sintático formam um único passo. Nesse caso, o analisador léxico atua como uma subrotina que é chamada pelo analisador sintático sempre que este necessita de mais um *token*. Os motivos que levam a dividir (conceitualmente) a análise léxica e sintática são os seguintes:

- Modularização e simplificação do projeto do compilador;
- Os tokens podem ser descritos utilizando-se notações simples, tais como expressões regulares, enquanto a estrutura sintática de comandos e expressões das linguagens de programação requer uma notação mais expressiva, como as gramáticas livres de contexto;
- Os reconhecedores construídos a partir da descrição dos tokens (através de gramáticas regulares, expressões regulares ou autômatos finitos) são mais eficientes e compactos do que os reconhecedores construídos a partir das gramáticas livres de contexto.

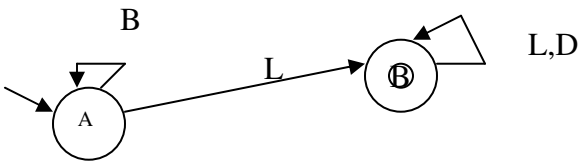
Implementação

A implementação de analisadores léxicos é feita, em geral, através de uma tabela de transição, a qual indica a passagem de um estado a outro pela leitura de um determinado caractere. Essa tabela e o correspondente programa de controle podem ser gerados automaticamente com o uso de um gerador de analisadores léxicos, tal como o LEX. No caso de gerador, o projetista especifica os tokens a serem reconhecidos através de expressões regulares e, a partir dessas, o LEX gera um programa correspondente.

Outra alternativa para implementação de um analisador léxico é a construção de um programa que simula o funcionamento do autômato correspondente, conforme é mostrado a seguir.

- Utiliza a TTE
- TTE e o correspondente programa de controle podem ser gerados automaticamente com o uso de um gerador de analisadores léxicos, tal como o LEX.
- Outra alternativa: construir o programa que simula o funcionamento do autômato correspondente.

Ex.: Identificadores:




```
Begin
     A: c:= próximo caracter
          Se c for letra então
               Lexema := Lexema + c;
               Goto B;
          Fim então
          Senão erro;
     B: c := próximo caracter
          Se c for letra ou digito então
               Lexema := Lexema + c;
               Goto B;
          Fim então
          Senão devolver lexema;
FIM
```

Erros Léxicos

Poucos erros são distinguíveis somente no nível léxico, uma vez que um analisador léxico possui uma visão muito local do programa fonte. Se a cadeia `fi` for encontrada pela primeira vez num programa C, no contexto

```
fi ( a = = f(x)) )...
```

um analisador léxico não poderá dizer se `fi` é a palavra chave `if` incorretamente grafada ou um identificador de função não declarada. Como `fi` é um identificador válido, o analisador léxico precisa retornar o token identificador e deixar alguma fase posterior do compilador tratar o eventual erro.

Mas, suponhamos que emerja uma situação na qual o analisador léxico seja incapaz de prosseguir, porque nenhum dos padrões reconheça um prefixo na entrada remanescente. Talvez a estratégia mais simples de recuperação seja a da "modalidade pânico". Removemos sucessivos caracteres da entrada remanescente até que o analisador

léxico possa encontrar um token bem formado. Essa técnica de recuperação pode ocasionalmente confundir o parser, mas num ambiente de computação interativo pode ser razoavelmente adequada.

Outras possíveis ações de recuperação de erros são:

1. remover um caractere estranho
2. inserir um caractere ausente
3. substituir um caractere incorreto por um correto
4. transpor dois caracteres adjacentes

Transformações de erro como essas podem ser experimentadas numa tentativa de consertar a entrada. A mais simple de tais estratégias é a de verificar se um prefixo de entrada remanescente pode ser transformado num lexema válido através de uma única transformação. Essa estratégia assume que a maioria dos erros léxicos seja resultante de um único erro de transformação, uma suposição usualmente confirmada na prática, embora nem sempre.

Uma forma de se encontrar erros num programa é computar o número mínimo de transformações de erros requeridas para tornar um programa erado num que seja sintaticamente bem formado. Dizemos que um programa errado possui k erros se a menor seqüência de transformação de erros que irá mapeá-lo em algum programa válido possui comprimento k. A correção de erros de distância mínima é uma conveniente ferramenta teórica de longo alcance, mas que não é geralmente usada por ser custosa demais de implementar. Entretanto, uns poucos compiladores experimentais têm usado o critério da distância mínima para realizar correções localizadas.