

Linguagens Formais, Autômatos e Construção de Compiladores

Edson Luiz França Senne
Professor Adjunto
UNESP - Campus de Guaratinguetá
(www.feg.unesp.br/~elfsenne)

- ♦ Os objetivos do curso são:
 - Entender os fundamentos da computação (Linguagens Formais e Autômatos).
 - Estudar técnicas de implementação de linguagens de programação (Construção de Compiladores).

- ♦ Referências principais:
 - Introduction to automata theory, languages and computation (Hopcroft, J.E.; Ullman, J.D.)
 - Compiladores: princípios, técnicas e ferramentas (Aho, A.V.; Sethi, R.; Ullman, J.D.)

Linguagens Formais, Autômatos e Construção de Compiladores

♦ Outras referências:

- Formal languages and their relation to automata (Hopcroft, J.E.; Ullman, J.D.)
- Introduction to formal language theory (Harrison, M.A.)
- Fundamentals of the theory of computation (Greenlaw, R.; Hoover, H.J.)
- The theory of parsing, translation and compiling, vol. I: Parsing. (Aho, A.V.; Ullman, J.D.)
- The theory of parsing, translation and compiling, vol. II: Compiling. (Aho, A.V.; Ullman, J.D.)
- Compiler construction for digital computers (Gries, D.)

♦ Em português:

- Introdução à Teoria da Computação (Sipser, M.)
- Compiladores: Princípios e Práticas (Louden, K.C.)

Linguagens Formais, Autômatos e Construção de Compiladores

Estrutura do curso:

Parte 1 - Linguagens Formais e Autômatos

- 1.1 - Conceitos Fundamentais
- 1.2 - Autômatos Finitos e Linguagens Regulares
- 1.3 - Linguagens Livres de Contexto
- 1.4 - Máquinas de Turing
- 1.5 - Procedimentos Decisórios

Parte 2 - Construção de Compiladores

- 2.1 - Conceitos Fundamentais
- 2.2 - Análise Léxica
- 2.3 - Análise Sintática
- 2.4 - Organização de Memória em Tempo de Execução
- 2.5 - Organização de Tabelas de Símbolos
- 2.6 - Formas Internas do Programa Fonte
- 2.7 - Tradução Orientada pela Sintaxe
- 2.8 - Recuperação de Erros
- 2.9 - Intérpretes

Parte 1 - Linguagens Formais e Autômatos

1.1 - Conceitos Fundamentais

- ♦ Nesta parte do curso estaremos discutindo o poder e as limitações dos computadores, ou seja, o que os computadores **podem** e **não podem** fazer.
- ♦ Nesta discussão uma questão importante será: **De que forma a estrutura de uma máquina afeta seu poder de computação?**
- ♦ Muitos modelos de computação já foram propostos. Vamos nos concentrar em três classes de modelos de máquinas:
 - autômatos finitos
 - autômatos com pilha
 - modelos irrestritos (máquinas de Turing, linguagens de programação)
- ♦ Em paralelo e independentemente do desenvolvimento desses modelos de computação foi formalizada a noção de **gramática** e **linguagem**. Esse formalização resultou na definição de uma hierarquia de classes de linguagens definidas por gramáticas de complexidade crescente, tais como:
 - gramáticas regulares (ou lineares à direita)
 - gramáticas livres de contexto
 - gramáticas irrestritas

Conceitos Fundamentais

- ♦ Embora gramáticas e modelos de máquinas possam parecer muito diferentes, o processo de analisar sentenças de uma linguagem é muito semelhante ao processo de realizar uma computação. Vamos mostrar que as gramáticas regulares, livres de contexto e irrestritas são **equivalentes em poder computacional**, respectivamente, aos modelos de autômatos finitos, autômatos com pilha e máquinas de Turing. **Mera coincidência?**
- ♦ Existem dois sistemas principais para representação de linguagens:
 - o sistema gerador (gramáticas)
 - o sistema reconhecedor (autômatos)
- ♦ Nesta primeira parte do curso (**Linguagens Formais e Autômatos**) vamos estudar esses dois sistemas e discutir o poder e as limitações dos computadores dando ênfase às classes de linguagens importantes para a especificação de Linguagens de Programação.
- ♦ Na segunda parte do curso (**Construção de Compiladores**) vamos especificar a gramática de uma Linguagem de Programação e construir um reconhecedor (além de um gerador de código executável e um interpretador) para essa linguagem.

Conceitos fundamentais

Linguagens e Gramáticas

- ♦ Definição: Um alfabeto (ou vocabulário) Σ é um conjunto finito, não vazio, de símbolos.

- ♦ Definição: Uma palavra (ou cadeia de símbolos) sobre um alfabeto Σ é uma seqüência finita de símbolos de Σ .

x é uma cadeia $\Rightarrow x = a_1a_2...a_n$, $n \geq 0$, com $a_i \in \Sigma$ ($i = 1, \dots, n$)
se $n = 0$ então $x = \Lambda$ (cadeia vazia)

- ♦ Definição: O comprimento de uma cadeia x sobre um alfabeto Σ (representado por $|x|$) é o número de ocorrências de símbolos de Σ em x .

$\Sigma = \{0,1\}$ $|010| = 3$ $|\Lambda| = 0$

- ♦ Definição: Sejam x e y duas cadeias sobre Σ . A concatenação de x e y é definida como a cadeia xy .

$\Sigma = \{0,1\}$ $x = 010$ $y = 10$ $xy = 01010$ $yx = 10010$

- ♦ Definição: Sejam X e Y , conjuntos de cadeias sobre Σ . O produto de X e Y é definido por: $XY = \{xy \mid x \in X \text{ e } y \in Y\}$.

- ♦ Notação:

$X^0 = \{\Lambda\}$ $X^{i+1} = X^iX$ ($i \geq 0$)

O fecho transitivo e reflexivo (X^*) e o fecho transitivo (X^+) de um conjunto de cadeias X são, portanto:

$$X^* = \cup X^i \ (i \geq 0)$$

$$X^+ = \cup X^i \ (i \geq 1) = X^*X$$

Conceitos Fundamentais

- ♦ Definição: Uma linguagem L é um conjunto qualquer de cadeias sobre um alfabeto Σ , ou seja, $L \subseteq \Sigma^*$.
- ♦ Definição: Uma gramática é um quádrupla $G = (N, \Sigma, P, S)$ onde: N é um conjunto finito não vazio (de símbolos não terminais), Σ é um conjunto finito não vazio (de símbolos terminais) tal que $\Sigma \cap N = \emptyset$, $S \in N$ (símbolo inicial), e P (conjunto de produções ou regras de produção) é um conjunto de regras da forma $\alpha \rightarrow \beta$ onde $\alpha \in (N \cup \Sigma)^* N (N \cup \Sigma)^*$ e $\beta \in (N \cup \Sigma)^*$
- ♦ **Exemplo**:
 $G = (N, \Sigma, P, S)$ onde $N = \{A, S\}$, $\Sigma = \{a, b\}$ e
 $P = \{S \rightarrow ab, S \rightarrow aASb, S \rightarrow bSb, AS \rightarrow bSb, A \rightarrow \Lambda, aASAb \rightarrow aa\}$
- ♦ A gramática como definida acima é denominada gramática irrestrita. Mas existem outros tipos de gramáticas.
- ♦ Definição: $G = (N, \Sigma, P, S)$ com $V = N \cup \Sigma$ é gramática sensível ao contexto se toda produção de P é da forma:
 - a) $\alpha A \gamma \rightarrow \alpha \beta \gamma$ ($A \in N$; $\alpha, \gamma \in V^*$; $\beta \in V^+$), ou
 - b) $S \rightarrow \Lambda$ e se essa produção ocorre, então S não aparece no lado direito de qualquer produção.

Conceitos Fundamentais

- ♦ Definição: $G = (N, \Sigma, P, S)$ é gramática livre de contexto se toda produção de P é da forma: $A \rightarrow \alpha$ ($A \in N$; $\alpha \in V^*$).

Exemplo: Gramática que representa o conjunto de inteiros

$I \rightarrow D$

$I \rightarrow ID$

$D \rightarrow 0$

...

$D \rightarrow 9$

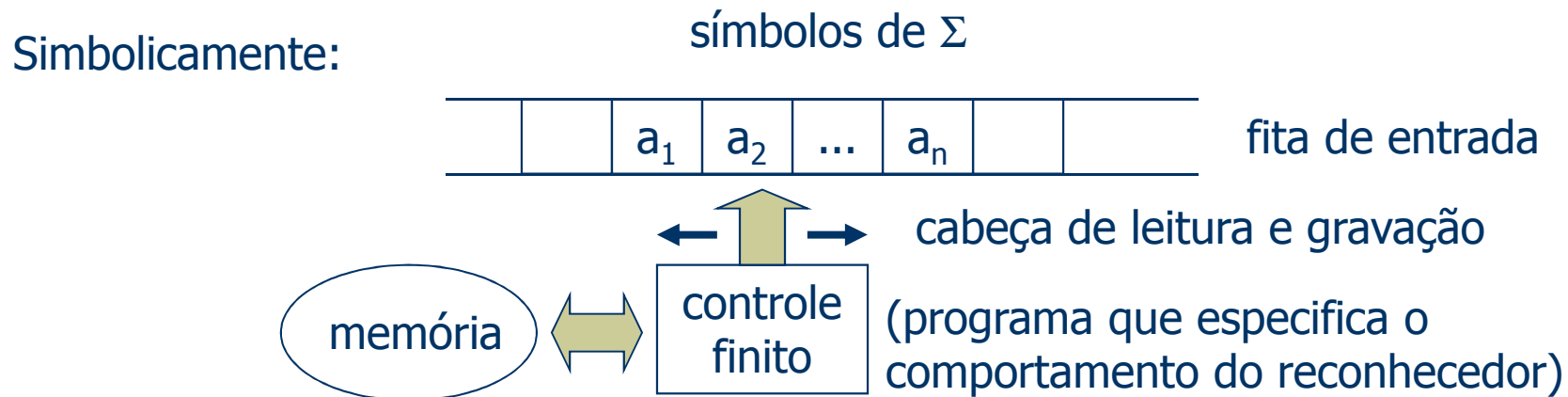
- ♦ Definição: $G = (N, \Sigma, P, S)$ é gramática regular se toda produção de P é de uma das formas:
 - a) $A \rightarrow aB$ ($A, B \in N$; $a \in \Sigma$)
 - b) $A \rightarrow a$
 - c) $S \rightarrow \Lambda$

Conceitos Fundamentais

- ♦ Definição: Seja uma gramática $G = (N, \Sigma, P, S)$ e seja $V = N \cup \Sigma$. Sejam $\alpha', \beta' \in V^*$. Dizemos que α' deriva diretamente β' ($\alpha' \Rightarrow \beta'$) se existem $\alpha_1, \alpha_2, \alpha$ e $\beta \in V^*$ tais que: $\alpha' = \alpha_1 \alpha \alpha_2$, $\beta' = \alpha_1 \beta \alpha_2$ e $\alpha \rightarrow \beta \in P$.
- ♦ Exemplo:
 $G = (N, \Sigma, P, S)$ onde $N = \{A, S\}$, $\Sigma = \{a, b\}$ e
 $P = \{S \rightarrow ab, S \rightarrow aASb, S \rightarrow bSb, AS \rightarrow bSb, A \rightarrow \Lambda, aASAb \rightarrow aa\}$
 $aASb \Rightarrow abSbb$
- ♦ Notação:
 \Rightarrow^n deriva em n passos
 \Rightarrow^* deriva em zero ou mais passos
 $S \Rightarrow aASb \Rightarrow abSbb \Rightarrow ababbb$ ($S \Rightarrow^3 ababbb$)
- ♦ Definição: Seja $G = (N, \Sigma, P, S)$ e $V = N \cup \Sigma$. O conjunto de formas sentenciais de G é definido por: $S(G) = \{\alpha \in V^* \mid S \Rightarrow^* \alpha\}$.
- ♦ Definição: Seja $G = (N, \Sigma, P, S)$ e $V = N \cup \Sigma$. A linguagem gerada por G é o conjunto $L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\} = \Sigma^* \cap S(G)$.
- ♦ Exemplo: Gramática que gera o conjunto $L = \{0^n 1^n \mid n \geq 1\}$
 $G = (N, \Sigma, P, S)$ com $N = \{S\}$, $\Sigma = \{0, 1\}$, $P = \{S \rightarrow 01, S \rightarrow 0S1\}$
 $S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 000111$

Conceitos Fundamentais

- ♦ Um reconhecedor de uma linguagem L é a representação de um procedimento que, quando apresentado a uma cadeia qualquer:
 - pára e responde "sim", após um número finito de passos, caso a cadeia pertença à linguagem L , ou
 - pára e responde "não" ou não pára, caso a cadeia não pertença a L .



- ♦ Uma configuração do reconhecedor é uma descrição de:
 - o estado do controle finito
 - o conteúdo da fita de entrada e a posição da cabeça
 - o conteúdo da memória

Conceitos Fundamentais

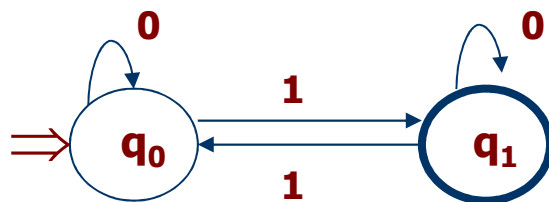
Um reconhecedor aceita (ou reconhece) uma cadeia w se:

- partindo de uma configuração inicial (isto é, controle finito num estado inicial, fita de entrada contendo w , cabeça posicionada no símbolo mais à esquerda de w , memória com conteúdo inicial),
- faz uma seqüência finita de "movimentos", e
- termina numa configuração final (controle finito num estado final e tendo sido lidos todos os símbolos de w).

A linguagem aceita (ou definida, ou reconhecida) por um reconhecedor R é:

$$L(R) = \{w \in \Sigma^* \mid R \text{ aceita } w\}$$

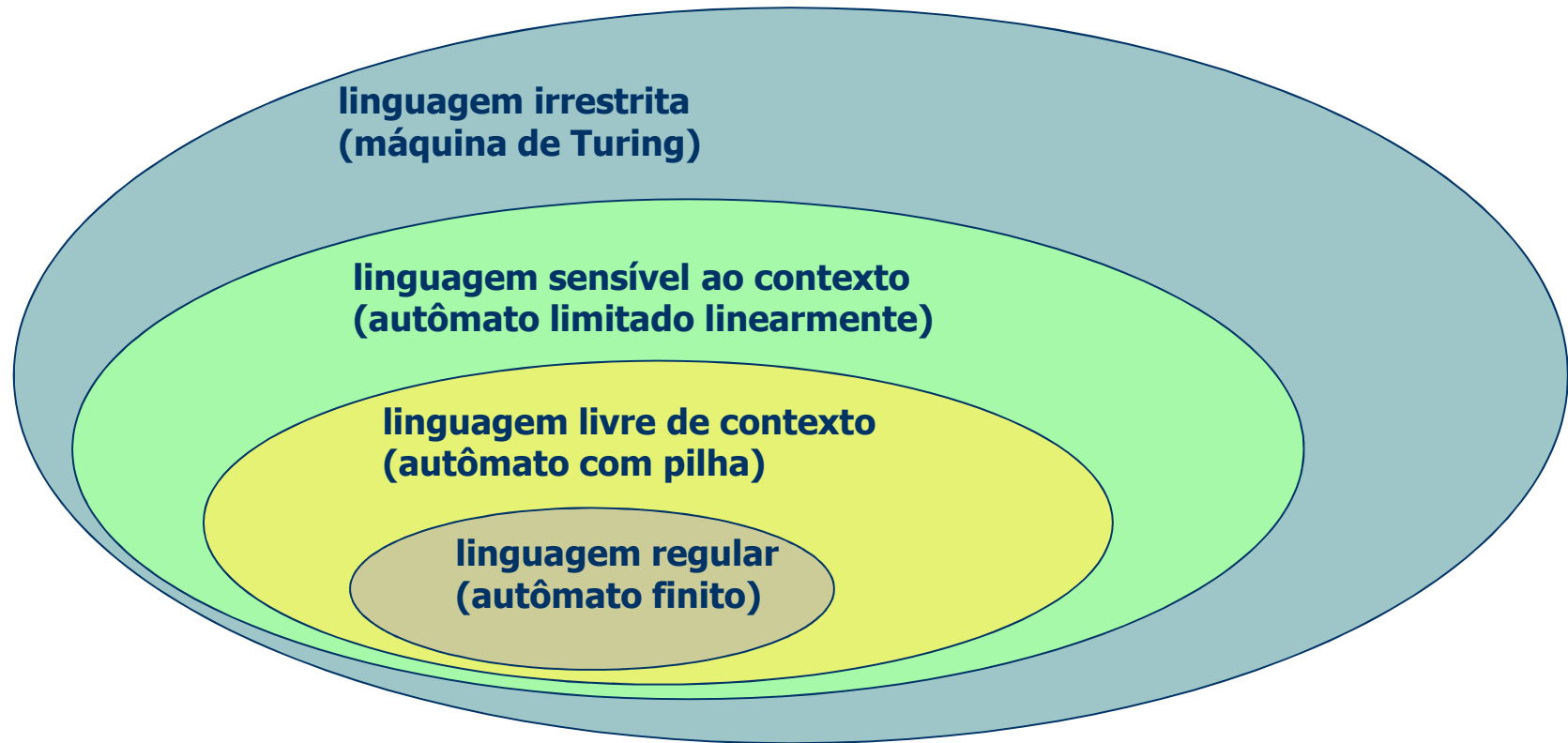
Exemplo:



Qual é a linguagem aceita por este reconhecedor?

Conceitos Fundamentais

Hierarquia de Chomsky



Conceitos Fundamentais

- ♦ Vamos considerar alguns problemas para ilustrar quatro conceitos fundamentais da teoria da computação:
 - a noção de **linguagem** (aceita ou reconhecida) por um autômato;
 - o conceito de **configuração** (ou estado) de um autômato;
 - a noção de que **restrições estruturais** limitam o poder de computação;
 - o fato (surpreendente?) de que certas computações são **intrinsecamente impossíveis**.
- ♦ Para isso, vamos considerar a linguagem de programação C⁻. Um programa C⁻ será sempre da forma:

```
void main() {  
    int s = 0;    // s é a única variável  
    while (1) {  
        <corpo>  
    }  
}
```

Conceitos Fundamentais

- ♦ O **<corpo>** pode conter somente:

```
s = <constante inteira>;
```

```
switch (s ou getchar()) {  
    case <constante inteira>:  
        ...  
        break;  
    ...  
    default: ...  
}
```

<constante inteira> representa um valor numérico (por exemplo, 1) ou um caractere (por exemplo, 'a')

- ♦ Entrada: fluxo único de caracteres (lido uma única vez, caractere a caractere)
- ♦ Um programa C não escreve saída alguma. O programa pára por meio de: **exit(Sim)** (responde "SIM") ou **exit(Não)** (responde "NÃO").
- ♦ Problema 1: Descobrir, em um arquivo de texto (cadeias de caracteres), se existe a palavra "onda". Vamos escrever em C o programa **Detecta** que pára por meio de **exit(Sim)** se o arquivo de entrada contém a palavra "onda" e pára por meio de **exit(Não)**, se o arquivo de entrada não contém a palavra "onda".

Conceitos Fundamentais

```
void main() {
    int s = 0;
    while (1) {
        switch(s) {
            case 0: // significa que procuramos por um 'o'
                switch(getchar()) {
                    case 'o': s = 1; break;
                    case EOF: exit(Não); break;
                    default: break;
                }
                break;
            case 1: // significa que o caractere anterior foi um 'o'
                switch(getchar()) {
                    case 'o': s = 1; break;
                    case 'n': s = 2; break;
                    case EOF: exit(Não); break;
                    default: s = 0; break; // recomeçar
                }
                break;
        }
    }
}
```

Conceitos Fundamentais

```
case 2: // significa: os caracteres anteriores foram 'on'
    switch(getchar()) {
        case 'o': s = 1; break;
        case 'd': s = 3; break;
        case EOF: exit(Não); break;
        default: s = 0; break; // recomeçar
    }
    break;
case 3: // significa: os caracteres anteriores foram 'ond'
    switch(getchar()) {
        case 'o': s = 1; break;
        case 'a': exit(Sim); break;
        case EOF: exit(Não); break;
        default: s = 0; break; // recomeçar
    }
} // end switch(s)
} // end while
} // end main
```

Este problema ilustra o conceito de **linguagem reconhecida** (ou aceita):

$L(\text{Detecta}) = \{ \text{strings para os quais o programa responde "Sim"} \}$

Conceitos Fundamentais

- ♦ Problema 2: Determinar, para um conjunto de strings binários, a paridade de cada string (a paridade de um string é par se existe um número par de 1's no string; caso contrário, a paridade será ímpar).

Vamos escrever em C o programa **Paridade** cuja linguagem reconhecida consiste de todos os strings binários com paridade par.

```
void main() {
    int s = 0;
    while (1) {
        switch(s) {
            // s == 0 significa: número par de 1's
            case 0:
                switch(getchar()) {
                    case '1': s = 1; break;
                    case EOF: exit(1); break;
                    default: break;
                }
            case 1:
                // s == 1 significa: número ímpar de 1's
                // ... (código não mostrado) ...
        }
    }
}
```

Conceitos Fundamentais

```
// s == 1 significa: número impar de 1's
case 1:
    switch(getchar()) {
        case '1': s = 0; break;
        case EOF: exit(Não); break;
        default: break;
    }
} // end switch(s)
} // end while
} // end main
```

- ♦ Suponha que desejamos interromper a execução de um programa C⁺⁺. Vamos imaginar que a interrupção ocorre sempre no início do **while(1)**. Que informações precisamos salvar para que seja possível retomar a execução do programa? Isto ilustra o conceito de **configuração**.
- ♦ Imagine que a entrada para o programa Paridade é 01101 e que o programa é interrompido após ter lido 011. Para recomençar a execução desse programa tudo o que precisamos saber é:
 - valor de $s = 0$
 - entrada restante = 01

(0, 01) é uma forma de representar uma configuração do programa Paridade.

Conceitos Fundamentais

- ♦ Problema 3: Como vimos, um programa C^{--} é bastante limitado. As restrições estruturais de um programa C^{--} afetam as computações que podem ser feitas? Por exemplo, podemos escrever em C^{--} um programa **Compara**, que determina se a entrada contém mesmo número de 0's e de 1's?
- ♦ Vamos mostrar que é impossível escrever o programa **Compara** em C^{--} .
- ♦ Vamos admitir que o programa **Compara** existe. Para este programa, teremos os seguintes fatos:
- ♦ Fato 1: Existe um limite sobre o número de valores distintos atribuídos a s .
Demonstração: Como o programa tem um tamanho fixo, existe somente um número finito de atribuições de um valor constante à variável s (expressões não são permitidas em C^{--}). Seja $n \in \{ 1, 2, 3, \dots \}$ o limite superior sobre o número de possíveis valores distintos atribuídos à variável s .
- ♦ Fato 2: O programa deve ler toda a entrada antes de parar e dar uma resposta.
Demonstração: Trivial!
- ♦ Fato 3: Após toda a entrada ter sido lida, se o valor de s se repetir no ponto de inspeção (início do `while(1)`), então o programa estará num loop infinito.
Demonstração: No <corpo> do programa, o valor atribuído a s é função do seu valor atual e dos caracteres lidos em um passo. Se não existem mais caracteres para serem lidos, então somente o valor atual de s afeta seu próximo valor.

Conceitos Fundamentais

Portanto, se a entrada já foi exaurida, a configuração do programa é dada apenas pelo valor de s . Assim, se um valor de s se repete, o programa repete uma mesma configuração e portanto está num ciclo repetindo para sempre a mesma sequência de valores de s .

- ♦ Fato 4: Após exaurir a entrada, se o programa faz mais de n iterações adicionais, então o programa nunca irá parar.
- ♦ Demonstração: Como existem, no máximo, n valores distintos de s , se o programa faz mais de n iterações, um valor de s deve se repetir e, portanto, podemos aplicar o Fato 3.
- ♦ Fato 5: O mesmo valor de s não pode se repetir sem a leitura de, pelo menos, um caractere de entrada. Do contrário, o programa nunca irá parar.
- ♦ Demonstração: Consequência do Fato 3!
- ♦ Considere que uma entrada t é apresentada ao programa **Compara**. Vamos observar os valores de s no ponto de inspeção. Em geral, o valor de s pode mudar sem que algum caractere de entrada tenha sido lido (mas não pode mudar mais do que $n-1$ vezes, pelo Fato 5). Além disso, durante um ciclo, o programa pode ler mais do que um caractere, pois pode haver várias (mas apenas um número finito de) chamadas à função **getchar()**. Seja, então, k o número máximo de caracteres que podem ser lidos num ciclo.

Conceitos Fundamentais

Seja: $t = \underbrace{00\dots0}_{kn} \underbrace{11\dots1}_{kn}$. O programa **Compara** deve dizer **Sim** para essa entrada.

Pelo Fato 2, sabemos que todos os caracteres têm que ser lidos antes que o programa possa parar. Seja: $s_0, s_1, s_2, \dots, s_m$ a sequência de valores de s no ponto de inspeção após todos os caracteres **0** terem sido processados (ou seja, s_i é o valor de s , no ponto de inspeção, na i -ésima iteração). Como existem kn 0's, o ponto de inspeção será alcançado pelo menos n vezes, ou seja: $m \geq n$. Como existem $m+1$ valores na sequência, existirão pelo menos dois valores iguais de s (digamos s_j e s_k). Seja p o número de caracteres lidos antes do passo j e q o número de caracteres lidos entre os passos j e k . Observe que $q \geq 1$, pois se $q = 0$, o programa estaria repetindo uma configuração e, pelo Fato 5, estaria num loop infinito. Portanto, o mesmo valor de s ocorre após a leitura de p ou de $p+q$ caracteres **0**. Então, se removermos q ocorrências de **0** em t , o programa não irá notar a diferença, ou seja, ainda irá responder **Sim** para essa entrada mais curta. Mas, como essa entrada mais curta não tem o mesmo número de 0's e 1's, o programa não poderia dizer **Sim**.

- ♦ Note que, nesta argumentação, não usamos nada além do fato de que **Compara** é um programa C^{--} . Logo, a linguagem C^{--} não é suficientemente poderosa para reconhecer a linguagem constituída por strings que contêm o mesmo número de 0's e 1's. Este exemplo, portanto, nos dá a idéia de que **restrições estruturais** limitam o poder de computação.

Conceitos Fundamentais

- ◆ Problema 4. O problema da parada

A linguagem C⁻ é um exemplo de modelo computacional. Os problemas 1 e 2 são exemplos de computações possíveis de serem realizadas por este modelo, e o problema 3 é um exemplo de computação impossível de ser realizada com este modelo.

O que devemos acrescentar à linguagem C⁻ para torná-la um modelo mais poderoso (capaz de resolver o problema 3, por exemplo)? É fácil escrever um programa **Compara** na linguagem C.

Um compilador é um programa bem sofisticado: recebe como entrada um programa e produz como saída um outro programa (equivalente ao de entrada, mas escrito em código executável). Os compiladores podem detectar erros de sintaxe, indicar possíveis erros semânticos (por exemplo, usar "=" onde normalmente se esperaria um "==") e, até mesmo, modificar o programa de saída para otimizar o código resultante em termos de uso de memória ou de tempo de execução. Quão "inteligente" pode ser um compilador?

Um compilador seria capaz de detectar que para valores negativos de entrada, o programa ao lado entra num loop infinito?

```
void main() {  
    int a, i = 0;  
    scanf("%d", &a);  
    while (i > a)  
        i = a + 1;  
    printf("%d", i);  
}
```

Conceitos Fundamentais

- ♦ Um compilador pode responder à questão geral: "Um programa P irá parar quando submetido a uma entrada E qualquer?"

Vamos supor que sim. E que a parte do programa compilador que responde a esta questão está codificada na função **H**. Como, tanto um programa, como sua entrada, são simplesmente cadeias de texto (strings), a função **H** poderia ser especificada como:

int H(char *P, char *E);

H(P, E) == 1 se o string P é um programa válido e pára quando submetido à entrada E;

== 0 caso contrário (ou o programa P é inválido, ou ele não pára quando submetido à entrada E).

Vamos admitir que a função **H** funciona para qualquer par de strings (ou seja, sempre dá uma resposta 1 ou 0). Por exemplo:

char *P = "void main(){ int a, i = 0; scanf("%d", &a); while (i > a) i=a+1; printf("%d",i); }";

H(P, "1") retorna 1

H(P, "-1") retorna 0

Vamos imaginar uma função **mscanf** que lê um string qualquer e aloca a memória necessária para armazenar o string lido. Vamos admitir que não existem restrições de memória de modo que a função **mscanf** sempre retorna um valor.

Conceitos Fundamentais

- Imagine então o seguinte programa:

```
<inserir aqui o código da função H>
```

```
void main() {  
    char *prog;  
    mscanf("%s", &prog);  
    if ( H(prog, prog) == 1 )  
        while(1); // loop infinito  
}
```

suponha que esse programa é salvo no arquivo **diagonal.c**

- O que acontece se executarmos o programa **diagonal** tendo como entrada o arquivo **diagonal.c** ? Depende do valor da função H:
H(diagonal, diagonal) retorna 0 - Isso significa que o programa diagonal com entrada diagonal não pára jamais. Como diagonal é um programa válido, para que esse programa não páre jamais é necessário que a avaliação da função H(diagonal, diagonal) retorne 1. Isso é uma contradição!
H(diagonal, diagonal) retorna 1 - Isso significa que o programa diagonal com entrada diagonal pára. Isso somente irá acontecer se a avaliação da função H(diagonal, diagonal) retornar 0. Isso também é uma contradição!
Logo, qualquer que seja o valor retornado pela função H haverá uma contradição. Conclusão: não é possível escrever a função **H**.

Conceitos Fundamentais

- ♦ A função **H** não pode ser codificada na linguagem C. Seria essa função programável em alguma outra linguagem?

Vamos mostrar que não!

Vamos mostrar que não pode existir uma linguagem capaz de codificar corretamente a função **H**. Em outras palavras, que não é possível escrever um programa capaz de resolver o problema da parada. Com isso estaremos mostrando que **nem todos os problemas são solúveis** por computadores, ou seja, que certas computações são **intrinsecamente impossíveis**.

- ♦ O problema da parada tem uma importância prática: Não é possível escrever um compilador que, submetido a um programa qualquer, seja capaz de detectar se tal programa pára ou não pára.