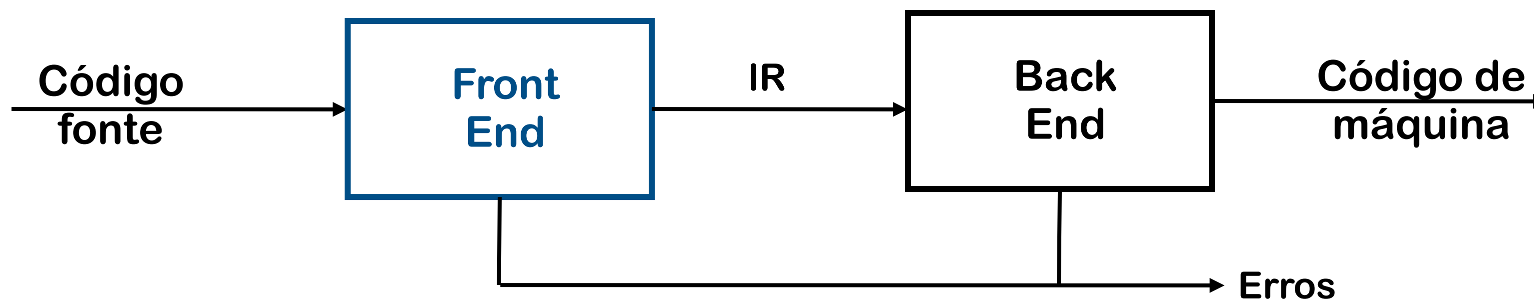
Análise Léxica

http://www.dcc.ufrj.br/~fabiom/comp

O Front End

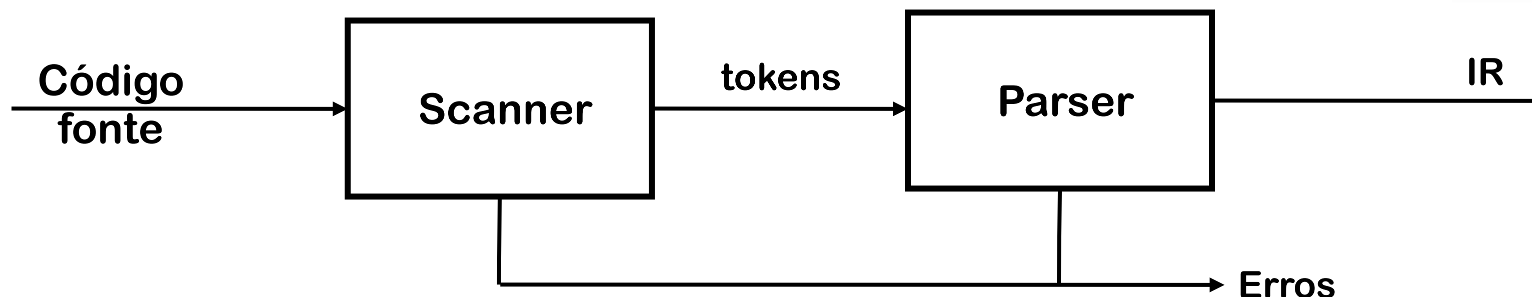


O propósito do front end é lidar com a linguagem de entrada

- Faz um teste de pertinência: código $\in$ linguagem?
- O programa é bem-formado (semanticamente) ?
- Constrói uma versão IR do código para o resto do compilador

O front end lida com forma (sintaxe) e significado (semântica)

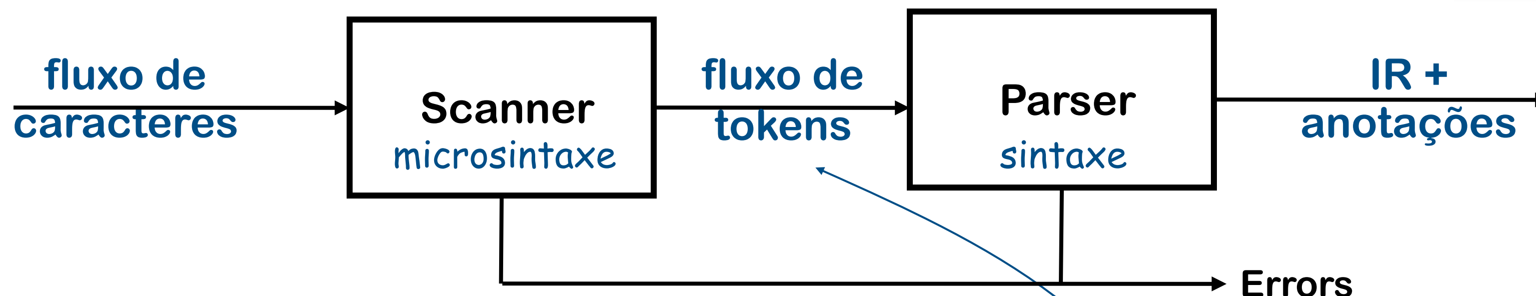
O Front End



Estratégia de Implementação

	Análise Léxica	Análise Sintática
Dar sintaxe	expressões regulares	gramáticas livres de contexto
Implementar reconhecedor	autômato finito determinístico	autômato de pilha
Fazer trabalho	Ações nas transições do autômato	

O Front End



Análise léxica é a única passada que toca todos os caracteres da entrada.

Por que separar o scanner e o parser?

- Scanner classifica palavras
- Parser constrói derivações gramaticais
- Análise sintática é mais difícil e mais lenta

Separação simplifica a implementação

- Scanners são simples
- Análise léxica leva a um parser menor e mais rápido

token é um par
<tipo, lexeme>

Visão Geral

O front end cuida da sintaxe

- Sintaxe de linguagens é especificada com categorias sintáticas, não com palavras
- Análise casa categorias sintáticas com uma gramática

Gramática de expressões

```
1. S     → expr

2. expr  → expr op termo
3.        | termo

4. termo → num
5.        | id

6. op    → +
7.        | –
```

S = S

T = { num, id, +, - }

N = { S, expr, termo, op }

P = {1, 2, 3, 4, 5, 6, 7}

categorias sintáticas

variáveis sintáticas

Visão Geral

Por que estudar construção de scanners?

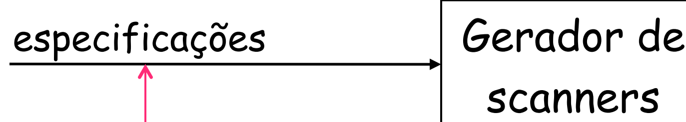
- Evitar escrever scanners à mão
- Aproveitar a teoria de linguagens formais

tempo de compilação

tabelas

Palavras como índices em uma tabela global (“interning”)

tempo de projeto



Especificações escritas com “expressões regulares”

Metas:

- Simplificar a especificação e implementação de scanners
- Entender as técnicas e tecnologias subjacentes

Operações de Conjuntos

Operação	Definição
União de L e M escrita $L \cup M$	$L \cup M = \{ s \mid s \in L \text{ ou } s \in M \}$
Concatenação de L e M escrita LM	$LM = \{ st \mid s \in L \text{ ae } t \in M \}$
Fecho de Kleene de L escrito L^*	$L^* = \cup_{0 \leq i \leq \infty} L^i$
Fecho positivo de L escrito L^+	$L^+ = \cup_{1 \leq i \leq \infty} L^i$

Expressões Regulares

Limitamos linguagens de programação de modo à estrutura léxica de uma palavra implicar qual a sua categoria sintática.

As regras que impõem esse mapeamento formam uma linguagem regular

Expressões regulares (REs) descrevem linguagens regulares

Expressão Regular (sobre alfabeto Σ)

- ε é uma RE denotando o conjunto {ε}
- Se a está em Σ, então a é uma RE denotando {a}
- Se x e y são REs denotando L(x) e L(y) então
  - x | y é uma RE denotando L(x) ∪ L(y)
  - xy é uma RE denotando L(x)L(y)
  - x^* é uma RE denotando L(x)*
  - x+ é uma RE denotando L(x)+

Precedência é

fecho > concatenação > alternativa

Expressões Regulares

Como esses operadores ajudam?

Expressões Regulares (sobre um alfabeto Σ)

- ε é uma RE denotando o conjunto $\{\varepsilon\}$
- Se a está em Σ, então a é uma RE denotando {a}
  - → qualquer palavra é uma RE
- Se x e y são REs denotando L(x) e L(y) então
  - x |y é uma RE denotando $L(x) \cup L(y)$
    - → listas de palavras podem ser escritas como REs (w_0 | w_1 | … | w_n)
  - xy é uma RE denotando L(x)L(y)
  - x^* é uma RE denotando $L(x)^*$
    - → podemos usar concatenação e fecho para escrever REs mais concisas (fatoração) e para especificar conjuntos infinitos usando uma descrição finita

Exemplos de Expressões Regulares

Identificadores:

Letra → (a|b|c| … |z|A|B|C| … |Z)

Dígito → (0|1|2| … |9)

Identificador → Letra (Letra | Dígito)*

atalho para

(a|b|c| … |z|A|B|C| … |Z) ((a|b|c| … |z|A|B|C| … |Z) | (0|1|2| … |9))*

Números:

Inteiro → (+|-|ε) (0| (1|2|3| … |9)(Dígito *))

Decimal → Inteiro . Dígito *

Real → (Inteiro | Decimal) E (+|-|ε) Dígito *

Complexo → (Real , Real)

os sublinhados são letras do alfabeto de entrada

Números podem ser bem mais complicados!

Usar nomes não implica recursão (não é CFG)

Expressões Regulares

Pra que então?

Usamos expressões regulares para especificar o mapa de palavras para tokens do analisador léxico

Usando resultados de teoria de autômatos e de algoritmos podemos automatizar a construção de reconhecedores eficientes a partir de REs

As técnicas automáticas geram scanners eficientes

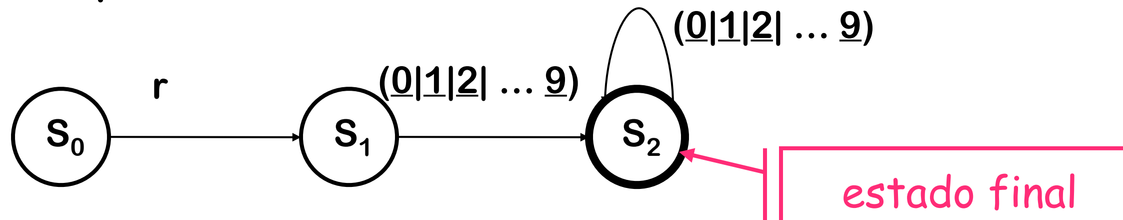
Exemplo

Considere o problema de reconhecer o nome de um registrador

Registrador → r (0|1|2| … | 9) (0|1|2| … | 9)*

- Permite registradores com qualquer número
- Exige pelo menos um dígito

RE corresponde ao reconhecedor (ou DFA)



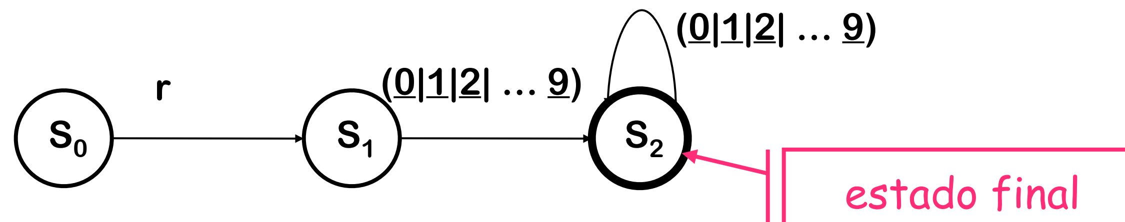
Reconhecedor para Registrador

Transições em outras entradas vão para um estado de erro s_e

Exemplo (continuação)

Operação de um DFA

- Comece no estado S_0 e faça transições em cada caractere da entrada
- DFA aceita palavra x se só se x o deixa em um estado final (S_2)



Reconhecedor para Registrador

Logo,

- r17 leva ele por s_0, s_1, s_2 e é aceito
- r leva ele por s_0, s_1 e falha
- a leva ele direto pra s_e

Exemplo (continuação)

Reconhecedor precisa ser convertido em código

```
Char ← próx. caractere
State ← s0

while (Char ≠ EOF)
    State ← δ(State,Char)
    Char ← próx. caractere

if (State é final)
    then sucesso
    else  falha
```

Esqueleto de reconhecedor

δ	r	0,1,2,3,4, 5,6,7,8,9	Outros
s_0	s_1	s_e	s_e
s_1	s_e	s_2	s_e
s_2	s_e	s_2	s_e
s_e	s_e	s_e	s_e

Tabela codificando RE

custo O(1) por caractere (ou transição)

Exemplo (continuação)

Podemos adicionar "ações" a cada transição

```
Char ← próx. caractere
State ← s0
while (Char ≠ EOF)
    Next ← δ(State,Char)
    Act  ← α(State,Char)

    executa ação Act
    State ← Next
    Char ← próx. caractere

if (State é final)
    then sucesso
    else  falha
```

Esqueleto de reconhecedor

δ α	r	0,1,2,3,4, 5,6,7,8,9	Outros
s_0	s_1 início	s_e erro	s_e erro
s_1	s_e erro	s_2 soma	s_e erro
s_2	s_e erro	s_2 soma	s_e erro
s_e	s_e erro	s_e erro	s_e erro

Tabela codificando RE

Ação típica é capturar o lexeme

E pra uma especificação mais precisa?

r Dígito Dígito* aceita qualquer número

- Aceita r00000
- Aceita r99999
- Se quisermos limitar a r0 para r31 ?

Escreva uma RE mais precisa

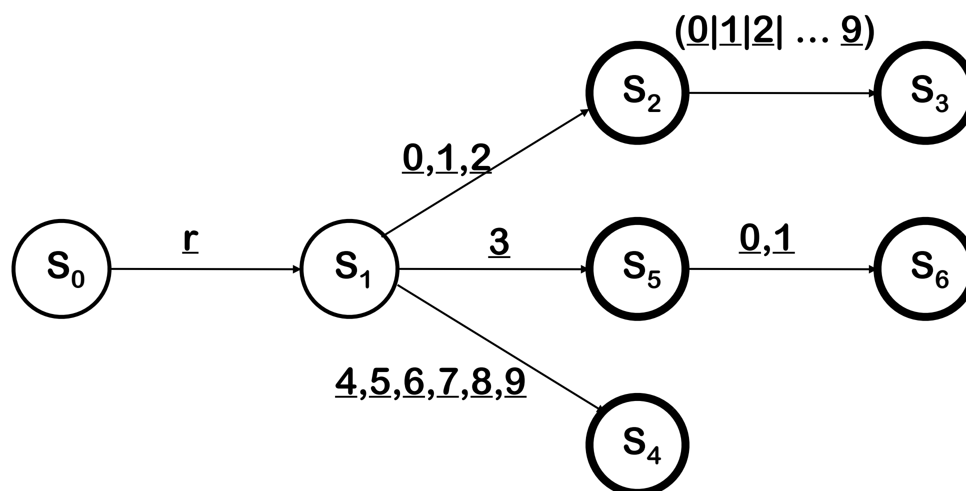
  - Reg → r ((0|1|2) (Dígito | ε) | (4|5|6|7|8|9) | (3|30|31))
  - Reg → r0|r1|r2| … |r31
  - Produz um DFA mais complexo

- DFA tem mais estados
- DFA tem mesmo custo por transição (ou por caractere)
- DFA tem mesma implementação básica

Registradores mais restritos (continuação)

A DFA para

Reg → r ((0|1|2) (Dígito | ε) | (4|5|6|7|8|9) | (3|30|31))



- Aceita um conjunto mais restrito de registradores
- Mesmo conjunto de ações, mais estados

Registradores mais restritos (continuação)

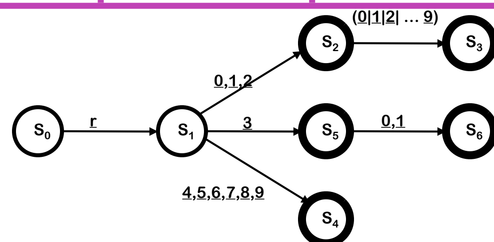
δ	r	0,1	2	3	4-9	Outros
s_0	s_1	s_e	s_e	s_e	s_e	s_e
s_1	s_e	s_2	s_2	s_5	s_4	s_e
s_2	s_e	s_3	s_3	s_3	s_3	s_e
s_3	s_e	s_e	s_e	s_e	s_e	s_e
s_4	s_e	s_e	s_e	s_e	s_e	s_e
s_5	s_e	s_6	s_e	s_e	s_e	s_e
s_6	s_e	s_e	s_e	s_e	s_e	s_e
s_e	s_e	s_e	s_e	s_e	s_e	s_e

Essa tabela roda no mesmo esqueleto de reconhecedor

Tabela codificando RE para especificação mais precisa de registradores

Registradores mais restritos (continuação)

Estado Ação	r	0,1	2	3	4,5,6 7,8,9	outro
0	1 início	e	e	e	e	e
1	e	2 soma	2 soma	5 soma	4 soma	e
2	e	3 soma	3 soma	3 soma	3 soma	e sai
3,4	e	e	e	e	e	e sai
5	e	6 soma	e	e	e	e sai
6	e	e	e	e	e	x sai
e	e	e	e	e	e	e



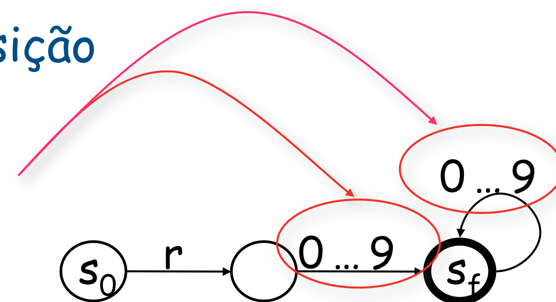
Scanners de Tabela

Estratégia comum é simular um DFA

- Tabela + Esqueleto de Scanner
  - Até agora usamos um esqueleto simplificado

```
state ← s0 ;
while (state ≠ exit) do
   char ← NextChar( )              // lê próximo caractere
   state ← δ(state,char);          // faz a transição
```

- Na prática, o esqueleto é mais complicado
  - Classes de caracteres para comprimir a tabela
  - Construção do lexeme
  - Reconhecer subexpressões
    - → Prática é combinar todas as REs em um DFA único
    - → Reconhecer palavras sem precisar encontrar fim de arquivo



Scanners de Tabela

Classes de Caracteres

- Agrupar caracteres pela suas ações no DFA
  - Combinar columas idênticas na tabela de transição, δ
  - Indexar δ por classe encolhe a tabela

```
state ← s0 ;
while (state ≠ exit) do
   char ← NextChar( )          // ler próximo caractere
   cat ← CharCat(char)         // classificar caractere
   state ← δ(state,cat)        // faz a transição
```

- Funciona bem em ASCII
  - conjunto compacto e caracteres de tamanho fixo (um byte)
  - número de valores limitado
- Mais trabalhoso com Unicode (UTF-8)

Scanners de Tabela

Construção do lexeme

- Scanner produz categoria sintática
  - Compiladores precisam do lexeme (palavra) também

```
state ← s0
lexeme ← “”
while (state ≠ exit) do
   char ← NextChar( )              // próximo caractere
   lexeme ← lexeme + char          // concatena no lexeme
   cat ← CharCat(char)             // classifica caractere
   state ← δ(state,cat)            // faz a transição
```

- Problema simples, mas não trivial
  - Lexeme pode ter tamanho arbitrário, solução ingênua pode ser quadrática

Scanners de Tabela

Escolher uma categoria com uma RE ambígua

- Queremos um DFA, então combinamos as REs em um só
  - Alguns lexemes pertencem ao RE de mais de uma categoria
    - → Identificadores vs palavras chave
    - → Queremos codificá-los no RE e reconhecê-los
  - Scanner tem que escolher categoria para estados finais ambíguos
    - → Solução tradicional: prioridade na ordem de REs (primeira vence)

Solução Alternativa (e bem popular)

- Construir tabela de palavras chave e juntar palavras chave com identificadores no DFA
- Faz sentido se
  - Scanner vai jogar todos os identificadores na tabela
  - Scanner é escrito à mão
- Senão, deixa o DFA cuidar disso (custo $O(1)$ por caractere)

Scanners de Tabela

Analisando uma Sequência de Palavras

- Scanners reais não procuram por uma palavra apenas
  - Queremos que o scanner ache todas as palavras da entrada, em ordem
  - Deve retornar uma palavra por vez
  - Solução sintática: exigir delimitadores
    - → Espaço, tab, pontuação, ...
    - → Mas queremos forçar espaços em todo lugar?
  - Solução típica
    - → Executar DFA até erro ou EOF, retornar para estado de aceitação
- Scanner deve retornar token, não verdadeiro ou falso
  - Token é par <tipo do token, lexeme>
  - Use um mapa do estado do DFA pra tipo do token

Scanners de Tabela

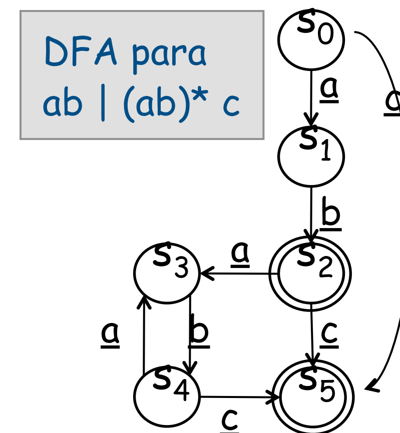
Tratando uma Sequência de Palavras

```
// reconhecer palavras
state ← s0
lexeme ← ""
limpa pilha
push (erro)
while (state ≠ se) do
   char ← NextChar( )
   lexeme ← lexeme + char
   if state ∈ SA
     then limpa pilha
   push (state)
   cat ← CharCat(char)
   state ← δ(state,cat)
end;
```

```
// limpar estado final
while (state ∉ SA and state ≠ erro) do
   state ← pop()
   trunca lexeme
   volta entrada um caractere
end;
// report the results
if (state ∈ SA )
   then return <PoS(state), lexeme>
   else  return erro
```

Evitando voltar demais

- Algumas REs podem ser quadráticas
  - Considere ab | (ab)* c e seu DFA
  - Entrada "abababababc"
    - → $s_0, s_1, s_2, s_3, s_4, s_3, s_4, s_3, s_4, s_5$
  - Entrada "abababab"
    - → $s_0, s_1, s_2, s_3, s_4, s_3, s_4, s_3, s_4$ volta 6 caracteres
    - → $s_0, s_1, s_2, s_3, s_4, s_3, s_4$ volta 4 caracteres
    - → s_0, s_1, s_2, s_3, s_4 volta 2 caracteres
    - → s_0, s_1, s_2



- Esse comportamento pode ser prevenido
  - O scanner pode lembrar caminhos que falharam para entradas particulares
  - Modificação simples cria o "scanner guloso"

Scanner Guloso

```
// reconhecer palavras
state ← s0
lexeme ← ""
limpa pilha
push (erro,erro)

while (state ≠ se) do
   char ← NextChar( )
   InputPos ← InputPos + 1
   lexeme ← lexeme + char

   if Falha[state,InputPos]
      then break;

   if state ∈ SA

     then limpa pilha

   push (state,InputPos)
   cat ← CharCat(char)

   state ← δ(state,cat)

end

// limpa estado final
while (state ∉ SA and state ≠ erro) do

    Falha[state,InputPos] ← true
    ⟨state,InputPos⟩ ← pop()

    trunca lexeme
    volta entrada um caractere
end

// reporta resultados
if (state ∈ SA )

    then return <PoS(state), lexeme>
    else return erro

InitScanner()
   InputPos ← 0
   for state s no DFA do
       for i ← 0 to |input| do
            Falha[s,i] ← false

       end;
   end;
```

Scanner Guloso

- Usa array de bits Falha para becos sem saída
  - Inicializa InputPos e Falha em InitScanner()
  - Falha requer espaço $\propto$ |tamanho da entrada|
- Evita comportamento quadrático
  - Produz scanner eficiente
  - Sua linguagem pode causar comportamento quadrático?
    - → Se sim, a solução é barata
    - → Senão, você pode encontrar o problema em outras aplicações da tecnologia de scanners (busca em strings)

Scanners de Tabela vs Scanners Diretos

Scanners de tabela usam muita indexação

- Ler próximo caractere
- Classifica
- Acha próximo estado
- Volta pro início

```
state ← s0 ;
while (state ≠ exit) do
   char ← NextChar( )
   cat ← CharCat(char )
   state ← δ(state,cat);
```

Estratégia alternativa: codificação direta

- Codificar estado na posição do programa
  - Cada estado tem seu código separado
- Testes locais de transição e saltos diretos
- Gera "código espaguete"
- Mais eficiente que estratégia por tabela
  - Menos acessos à memória, pode ter mais saltos

Scanners de Tabela vs Scanners Diretos

Custo de Busca em Tabela

- Cada busca em CharCat ou δ envolve um cálculo de endereço e uma operação de memória
  - CharCat(char) vira

    $@CharCat_0 + char \times w$ — w é sizeof(el. de CharCat)

  - δ(state,cat) vira

    $@\delta_0 + (state \times cols + cat) \times w$ — cols é no. de colunas em δ

    w é sizeof(el. de δ)

- As referências para CharCat e δ viram múltiplas ops
- Certo overhead por caractere
- Evite as buscas nas tabelas e o scanner será mais rápido

Scanners Diretos

Um scanner direto para r Dígito Dígito*

```
start: accept ← se
       lexeme ← “”
       count ← 0
       goto s0

s0:    char ← NextChar
       lexeme ← lexeme + char
       count++
       if (char = ‘r’)
          then goto s1
          else goto sout

s1:    char ← NextChar
       lexeme ← lexeme + char
       count++
       if (‘0’ ≤ char ≤ ‘9’)
          then goto s2
          else goto sout

s2:    char ← NextChar
       lexeme ← lexeme + char
       count ← 0
       accept ← s2
       if (‘0’ ≤ char ≤ ‘9’)
          then goto s2
          else goto sout

sout:  if (accept ≠ se)
         then begin
              for i ← 1 to count
                  RollBack()
              report sucesso
              end
         else report falha
```

Menos operações de memória

Não precisa de CharCat

Usa muitas estratégias para teste e salto

Scanners Diretos

Um scanner direto para r Dígito Dígito

start: accept ← s_e
lexeme ← ""
count ← 0
goto s_0

s_0: char ← NextChar
lexeme ← lexeme + char
count++
if (char = 'r')
then goto s_1
else goto s_{out}

s_1: char ← NextChar
lexeme ← lexeme + char
count++
if ('0' ≤ char ≤ '9')
then goto s_2
else goto s_{out}

s_2: char ← NextChar
lexeme ← lexeme + char
count ← 1
accept ← s_2
if ('0' ≤ char ≤ '9')
then goto s_2
else goto s_{out}

s_{out}: if (accept ≠ s_e)
then begin
end
else report failure

Se o teste pra sair do estado tem muitos casos o scanner pode condirerar outros esquemas

- Consultar tabela
- Busca binária

Scanners Feitos à Mão

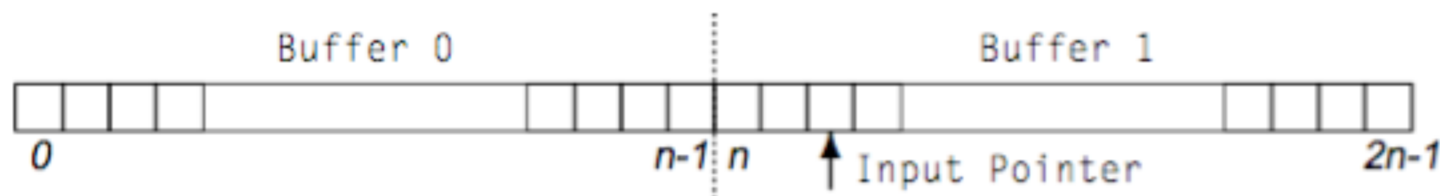
Muitos (a maioria?) dos compiladores modernos usa scanners feitos à mão

- Ainda assim é bom começar de um DFA pois facilita o projeto
- Evita as limitações de uma ferramenta
  - Computar o valor de números
    - Em (F|JF)LEX, muitos usam rotinas de conversão string -> número
    - Pode-se usar truques para computar valor à medida que caracteres são lidos (scanner da calculadora)
  - Combinar estados similares
  - Não montar um lexema em casos não necessários
- Escrever scanners é divertido
  - Compactos, compreensíveis, fáceis de depurar, fáceis de se escrever testes automáticos
  - Não se empolgue (p. ex., hash perfeito para palavras chave)

Scanners Feitos à Mão

Bufferização da entrada

- Estratégia comum é usar um buffer duplo (e circular) e limitar a quantidade de retrocesso




```
NextChar:
   Char ← Buffer[Input]
   Input ← Input+1 mod 2n
   if (Input mod n = 0)
      then begin
         encha Buffer[Input:Input+n-1]
         Fim ← (Input+n) mod 2n
      end;
   return Char
```

```
Rollback:
   if (Input = Fim)
       then erro no retrocesso
   Input = (Input-1) mod 2n

Init:
   Input ← 0
   Fim ← 0
   encha Buffer[0:n]
```

Construindo Scanners

Conclusão

- Toda essa tecnologia permite automatizar a construção de scanners
- Projetista escreve expressões regulares
- Gerador de scanners constrói autômato e escreve o código do scanner (de tabela ou direto)
- Isso produz scanners rápidos e robustos

Para a maioria dos recursos linguagens modernas, isso funciona

- Pense duas vezes antes de introduzir um recurso que dificulta a vida de um scanner baseado em autômatos
- As que nós vimos (espaços em branco opcionais, palavras chave não reservadas) não se provaram úteis ou duradouras

Claro que nem tudo cabe numa linguagem regular...

JFlex

- O gerador de scanners que vocês usarão no trabalho
- Baixe seguindo o link na página da disciplina
- O arquivo inclui scripts para executar ele tanto em Linux (jflex) quanto Windows (jflex.bat); use-os!

```
$ jflex scanner_spec.lex
Reading "scanner_spec"
Constructing NFA : 36 states in NFA
Converting NFA to DFA :
............
14 states before minimization, 5 states in minimized DFA
Writing code to "Scanner.java"
```

Especificando um Scanner

- Arquivo de especificação:

```
código Java (fica fora da classe do scanner)
%%
opções e declarações
%%
regras do scanner
```

- Código Java normalmente são **import** de pacotes que você pretende referenciar no código do scanner
- Opções controlam como é o scanner gerado
- Regras são expressões regulares e as ações que o scanner executa quando reconhece uma delas

Opções

- **%class Foo**
  - Gera uma classe pro scanner com nome Foo (em um arquivo Foo.java)
- **%line** e **%column**
  - Ativa contagem automática de linhas e colunas, respectivamente (acessadas pelas variáveis **yyline** e **yycolumn**); útil para mensagens de erro
- **%{ ... %}**
  - Inclui código Java **dentro** da classe do scanner
- **%init{ ... %init}**
  - Inclui código Java dentro do **construtor** da classe do scanner
- **nome = regexp**
  - Define uma macro que pode ser referenciada pelas regras do scanner com **{nome}**
- **%function getToken**
  - Define o nome do método que executa o scanner
- **%type Token** (ou **%int**)
  - Define o tipo de retorno do método que executa o scanner como **Token**

Expressões JFlex

Expressão	Significado
a	Caractere 'a'
"foo"	Cadeia "foo"
[abc]	'a', 'b' ou 'c'
[a-d]	'a', 'b', 'c' ou 'd'
[^ab]	Qualquer caractere exceto 'a' e 'b'
.	Qualquer caractere exceto \n
x | y	Expressão x ou y
xy	Concatenação
x*	Fecho de Kleene
x+	Fecho positivo
x?	Opcional
!x	Negação
~x	Tudo até x (inclusive)

Regras e Ações

- Regras têm o formato

    regexp { código Java }

- O código Java é copiado para dentro do método do scanner
- Para pegar o valor do lexeme usa-se o método **yytext()**
- Lembre sempre de retornar ao final do código, ou o scanner continua rodando!
- Regra especial **<<EOF>>** casa com o final do arquivo

Exemplo

```
%%

%class CalcLex
%function getToken
%type void

digito = [0-9]

%%

{digito}+  { Calc.tokval = Integer.parseInt(yytext());
             Calc.token = Calc.integer;
             return; }

"+"|"-"|"*"|"/"|";"|"("|")"  { Calc.token = yytext().charAt(0); return; }

[ \n\t]    { }

<<EOF>>    { Calc.token = 0; return; }

.          { Calc.calcError("Illegal character "+yytext()); }
```