Compiladores
(IF688)

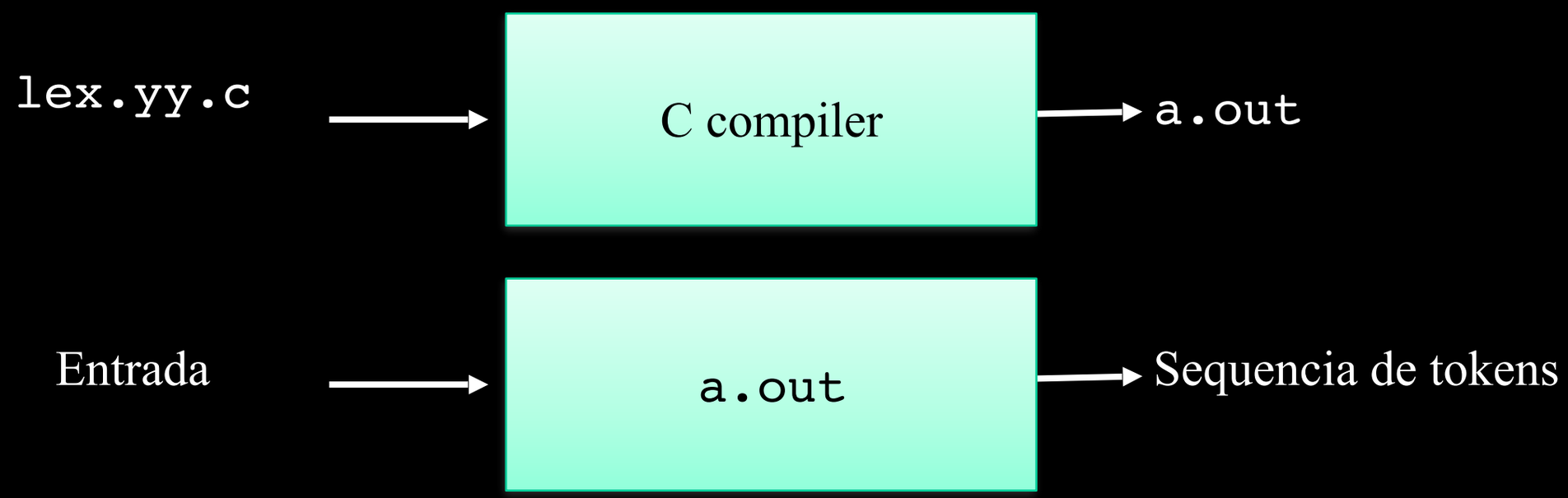
Leopoldo Teixeira
lmt@cin.ufpe.br | @leopoldomt

Lexer

- Em sua versão mais primitiva, um lexer apenas indica se o arquivo de entrada consiste de uma sequência de tokens válidos (reconhecedor)
- Tipicamente, o lexer coopera com o parser reconhecendo um token e retornando pro parser o objeto com tipo do token e valor

Lex


Programa fonte
lex.l
Compilador lex
(lex ou flex)
Lex.yy.c


lex.yy.c
C compiler
a.out
Entrada
a.out
Sequencia de tokens

Lex - implementação

- Transforma o programa de entrada em um programa em C que implementa um Autômato Finito
- Reconhece a linguagem regular expressa pelas expressões regulares definidas no arquivo

JFlex

- Gerador de analisadores léxicos escrito em Java para Java
- Implementado em cima de JLex
- Sintaxe baseada fortemente em lex
- Disponível em jflex.de

Adaptado parcialmente do material sobre JFlex de Fábio Mascarenhas (UFRJ)

Rodando JFlex

- Linha de comando
  - `java -jar jflex-1.6.1.jar <options> <inputfiles>`
- Ant task
- Maven plugin

Rodando JFlex

```
Reading "teste.jflex"
Constructing NFA : 40 states in NFA
Converting NFA to DFA :
..................
21 states before minimization,
17 states in minimized DFA
Writing code to "TesteLexer.java"
```

Especificando

```
código Java a incluir

%%

opções e declarações

%%

regras léxicas
```

Código Java a incluir

- O texto antes do primeiro %% é incluído '*as is*' no topo da classe gerada
  - antes da declaração da classe
- Normalmente são colocados declarações de pacote e imports de classes
- Também pode definir comentários javadoc, se não incluir, JFlex gera um automaticamente

Opções

- **`%class Nome`**
  - faz com que a classe do scanner gerado seja nomeada de acordo com o argumento, no caso acima, salvará o arquivo em Nome.java
- **`%implements %extends %public %final %abstract`**
  - auto-explicativos
- **`%unicode`**
  - define o conjunto de caracteres a ser utilizado.
- **`%cup ou %cup2`**
  - ativa modo de compatibilidade com o gerador de parsers CUP

Opções

- **`%line`** e **`%column`**
  - ativa contagem de linhas e colunas, facilitando depuração
- **`%{...}`**
  - código dentro destes blocos é incluído dentro da classe do scanner gerado

Opções

- **`%init{… %init}`**
  - inclui código dentro do construtor
- **`%function getToken`**
  - renomeia o método que pega próximo token
- **`%type Token`**
  - define o tipo de retorno do método que pega próximo token

Declarações

- **`padrao = r`**
  - associa a expressão regular r com o nome ‘padrao’
  - similar à definição de uma macro, onde aparecer padrao substituiremos por r

Notação

expressão	significado
`a`	caractere ‘a’
`[abc]`	a’, ‘b’, ‘c’
`[a-d]`	a’, ‘b’, ‘c’, ‘d’
`[^ab]`	qualquer caractere, exceto ‘a’ e ‘b’
`.`	qualquer caractere exceto quebra de linha (\n)
`x | y`	união
`xy`	concatenação
`x*`	fecho de kleene
`x+`	fecho positivo
`x?`	opcional
`!x`	negação
`~x`	todos os caracteres até x (inclusive)
`a {n}`	equivalente com: a concatenado n vezes
`a {n,m}`	a concatenado pelo menos n e no máximo m vezes

Regras Léxicas

- **`exp {ação}`**
  - a ação dentro das chaves é disparada ao reconhecer a expressão do lado esquerdo
  - expressamos a ação em código Java, que é copiado para o Scanner
  - métodos auxiliares permitem pegar valores como o lexema — yytext()

Regras Léxicas

```
exp1 |
exp2 |
exp3 |
 {ação}
```

==

```
exp1 {ação}
exp2 {ação}
exp3 {ação}
```

Exemplos simples

Especificações Heterogêneas

- O lexer não tem conhecimento do programa
- Lê tokens sem ciência da estrutura sintática
- Não tem como saber se o token faz sentido ou não naquela parte do programa
  - por ex.: “**`123+-/4if`**”

Especificações Heterogêneas

- No entanto, o nível léxico da linguagem pode não ser uniforme

- Em HTML, as regras léxicas no interior de tags são diferentes das regras fora da tag

  - **`<foo src="bar">foo src="bar"`**...

Estados

- Podemos tratar estas diferenças em regras como a combinação de várias linguagens
- Cada linguagem tem a sua especificação homogênea e separamos estas especificações por meio de estados
- Dependendo do estado atual do lexer, certas regras são ativadas ou não

Estados

```
<ESTADO1> {
   …regras…
}

<ESTADO2> {
   …outras regras…
}
```

Estados

- Mecanismo para definir pré-condições para aplicações das regras
- O estado YYINITIAL sempre existe
- Regras fora de blocos de estado valem para qualquer estado

Estados

- Declarados na seção de opções e declarações com a diretiva **`%state`**
- Para mudarmos de estado usamos a função **`yybegin`** na ação de algum token
- Não existe **`yyend`**. Voltamos ao estado inicial chamando **`yybegin(YYINITIAL)`**

Estados

```
%state TAG
%%
<YYINITIAL> {
   [<] {yybegin(TAG); return new Token('<');}
   [^< ]+ {return new Token(WORD, yytext());}
}

<TAG> {
   [>] {yybegin(YYINITIAL); return new Token('>');}
   …outras regras…
}
```

Outros usos

- Estados são úteis mesmo em linguagens com especificações homogêneas
- Podemos tratar partes tradicionalmente espinhosas da linguagem com estados próprios
  - Por ex.: strings e comentários
- Definir o que é permitido no interior de strings e comentários sem afetar o resto da especificação

```
/* foo /* bar */ e agora? */
```

Comentários aninhados

- Em algumas linguagens de programação, comentários consistem de qualquer texto entre `/*` e `*/`
- Ou seja, o primeiro `*/` que aparecer dentro de um comentário o encerra
- No entanto, existem linguagens que permitem aninhamento de comentários

Comentários aninhados

- Em **`/* foo /* bar */ e agora? */`**, dependendo da especificação, o comentário encerra logo após **`bar`**
- O que vem após o primeiro **`*/`** seria tokenizado normalmente, sem considerar como comentário
- Se a linguagem permite aninhar comentários, o que está acima seria visto como um único comentário
- Não podemos expressar aninhamento com expressões regulares (lema do bombeamento), mas conseguimos alcançar este resultado com estados e ações em JFlex

Comentários aninhados

- A solução seria um estado dedicado a comentários
- Ao entrarmos no estado de comentários, inicializamos um contador de nível de aninhamento
- Neste estado, a cada `/*` encontrado, o nível é incrementado em 1
- Cada `*/` encontrado decrementa o nível em 1
- Quando nível chega a 0 voltamos a **`YYINITIAL`**
- Todos os demais caracteres são ignorados

Indentação

- Ideia semelhante pode ser aplicada para tokenizar a estrutura de blocos de Python
- Não há tokens para delimitar início/fim de blocos
- Pilha de indentação indica quantidade de espaços naquele nível
- Espaços em branco empilham nível de indentação caso número de espaços seja maior que o topo

Mudando estado fora do lexer

- A mudança de estado pode ser controlada na fase de análise sintática
- O analisador sintático pode mudar o estado do analisador léxico dependendo da parte do programa que estiver analisando

Outras ferramentas:

JavaCC, SableCC, ANTLR, etc.

JavaCC

```
PARSER_BEGIN(MyParser)
   class MyParser {}
PARSER_END(MyParser)

/* For the regular expressions on the right, the token on the left will be returned: */
TOKEN : {
    < IF: "if" >
  | < #DIGIT: ["0"-"9"] >
  | < ID: ["a"-"z"] (["a"-"z"]|<DIGIT>)* >
  | < NUM: (<DIGIT>)+  >
  | < REAL: ( (<DIGIT>)+ "." (<DIGIT>)* ) |
            ( (<DIGIT>)* "." (<DIGIT>)+ )>
}

/* The regular expressions here will be skipped during lexical analysis: */
SKIP : {
    <"--" (["a"-"z"])* ("\n" | "\r" | "\r\n")>
  |  " "
  |  "\t"
  |  "\n"
}

/* If we have a substring that does not match any of the regular expressions in TOKEN or SKIP,
   JavaCC will automatically throw an error. */
void Start() :
{}
{  ( <IF> | <ID> | <NUM> | <REAL> )*  }
```

Adaptado de Andrew Appel. ***Modern Compiler Implementation in Java***

SableCC

```
Helpers
    digit = ['0'..'9'];
Tokens
    if = 'if';
    id = ['a'..'z'](['a'..'z'] | (digit))*;
    number = digit+;
    real =  ((digit)+ '.' (digit)*) |
            ((digit)* '.' (digit)+);
    whitespace = (' ' | '\t' | '\n')+;
    comments = ('--' ['a'..'z']* '\n');
Ignored Tokens
    whitespace,
    comments;
```

Adaptado de Andrew Appel. ***Modern Compiler Implementation in Java***

Exemplo

Infixa — Pós-Fixa