Compiladores (CC3001) — Aula 3: Geradores de Analisadores Lexicais

Mário Florido

DCC/FCUP

2024

[dcc] FACULDADE DE CIÊNCIAS UNIVERSIDADE DO PORTO

Esta aula

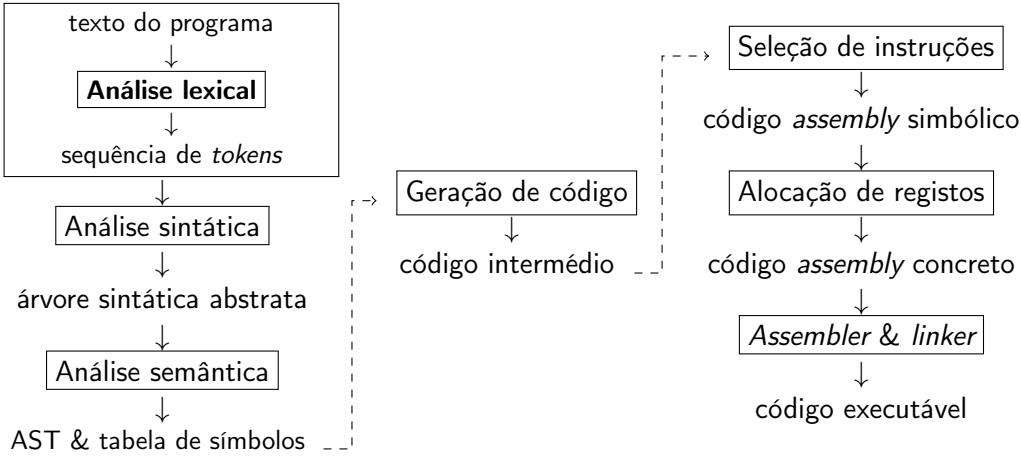
Geradores de analisadores lexicais

Alex

Flex

Escrever regras

Re-lembrar: fases dum compilador



Geradores de analisadores lexicais

Alex

Flex

Escrever regras

Na aula passada vimos:

- como usar expressões regulares para descrever *tokens*
- como transformar uma expressão regular num autómato não-determinístico (NFA)
- como transformar um NFA num autómato determinístico (DFA)

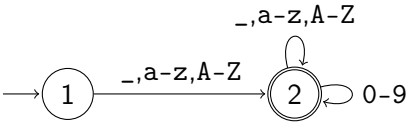
Vamos agora ver:

- como implementar um analisador lexical a partir do DFA
- *Alex* e *Flex*: dois geradores automáticos para analisadores lexicais

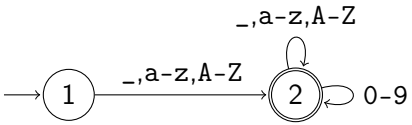
A expressão regular para identificadores na linguagem C ou Java:

```
[_a-zA-Z][_a-zA-Z0-9]*
```

Neste caso podemos obter diretamente um DFA equivalente:



(Em geral: temos de obter primeiro o NFA.)



$$\text{símbolos } \Sigma = \{\text{carateres ASCII}\}$$

$$\text{estados } Q = \{1, 2\}$$

$$\text{estado inicial } q_0 = 1$$

$$\text{estados finais } F = \{2\}$$

$$\begin{aligned}\text{transições } \delta = \{ &(1, \_, 2),\ (1, \mathtt{a}, 2),\ (1, \mathtt{b}, 2), \ldots, (1, \mathtt{z}, 2), (1, \mathtt{A}, 2),\ (1, \mathtt{B}, 2), \ldots, (1, \mathtt{Z}, 2),\\ &(2, \_, 2),\ (2, \mathtt{a}, 2),\ (2, \mathtt{b}, 2), \ldots, (2, \mathtt{z}, 2), (2, \mathtt{A}, 2),\ (2, \mathtt{B}, 2), \ldots, (2, \mathtt{Z}, 2),\\ &(2, 0, 2),\ (2, 1, 2), \ldots, (2, 9, 2)\ \}\end{aligned}$$

Implementação de um DFA

```
#define STATES      3       // estados 0, 1, 2
#define SYMBOLS     128     // códigos ASCII
const int delta[STATES][SYMBOLS] = {
    /* símbolos          .....   a, b, ..., A, B, ... */
    /* estado 0 */   { 0, ..., 0, 0, ..., 0, 0, ... 0 },
    /* estado 1 */   { 0, ..., 2, 2, ..., 2, 2, ... 0 },
    /* estado 2 */   { 0, ..., 2, 2, ..., 2, 2, ... 0 } };
```

- Representamos estados por inteiros e símbolos por códigos ASCII
- Uma tabela estática de transições: `delta`$[s][x] = \delta(s, x)$
- O estado 0 representa um erro (i.e. ausência de transições)
- O estado 2 é o unico estado final (aceitação)

Pseudo-código para o ciclo principal:

```
int state = 1;          // estado inicial
int c;                  // próximo símbolo
while ((c = getchar())!= EOF) {
  state = delta[state][c];
  if (state == 0)
      return ERROR;  // rejeite
  if (FINAL(state))
      return ACCEPT;  // aceite
}
return ERROR; // fim do input
```

Esta aula

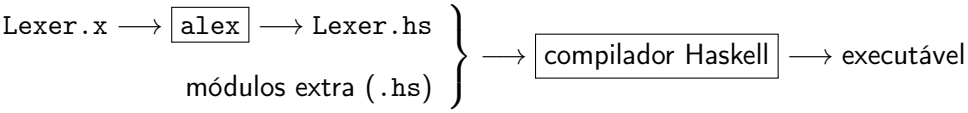
Geradores de analisadores lexicais

Alex

Flex

Escrever regras

- Um programa gerador de analisadores lexicais
- Recebe uma descrição do analisador (e.g. `Lexer.x`):
  - regras com expressões regulares
  - ações escritas em Haskell e definições auxiliares
- Produz um ficheiro de código Haskell (`Lexer.hs`)
  - Exporta tipos e uma função para análise lexical
  - Pode ser compilado juntamente com outros módulos


Lexer.x
alex
Lexer.hs
módulos extra (.hs)
compilador Haskell
executável

Exemplo de um ficheiro *Alex*

```
{
module Lexer where
}
%wrapper "basic"
$alpha = [_a-zA-Z]
$digit = [0-9]

tokens :-
$white+                  ; -- ignorar carateres brancos
if                       { \_ -> IF }
$alpha($alpha|$digit)*   { \s -> ID s }
$digit+                  { \s -> NUM (read s) }
$digit+"."$digit+        { \s -> REAL (read s) }

{
data Token = IF | ID String | NUM Int | REAL Double
}
```

Exemplo de um ficheiro *Alex* (cont.)

- As partes entre chavetas { } são código Haskell
- Restantes: *diretivas*, *definições* e *padrões*
- Cada regra tem a forma

  padrão { *ação em Haskell* }
- Cada padrão é uma expressão regular
- Cada ação é uma função de tipo `String -> Token`
- Um bloco de definições Haskell no final (opcional)

Formato de regras *Alex*

```
{
declaração de módulo e imports
}
%wrapper "..."
definições de padrões

tokens :-
padrão 1          { ação 1  }
padrão 2          { ação 2  }
⋮

{
definições auxiliares
}
```

- O Alex permite diferentes *interfaces* para analisadores
- No exemplo anterior usamos a interface básica:

  `%wrapper "basic"`
- A interface `basic` gera um analisador que retorna a lista de *tokens*

  `alexScanTokens :: String -> [Token]`
- Outras interfaces permitem mais flexibilidade, e.g.:

  `%wrapper "posn"`

  permite associar a posição a cada *token* (linha e coluna) para reportar erros.

Usando a interface

```
%wrapper "posn"
```

- Cada ação passa a ter tipo

  ```
  AlexPosn -> String -> Token
  ```

- A posição de cada *token* é passada como um valor `AlexPosn`

  ```
  data AlexPosn = AlexPn !Int  -- absolute character offset
                         !Int  -- line number
                         !Int  -- column number
  ```

- Permite guardar a posição de cada `Token` para reportar em caso de erro

Exemplo com posições

```
{
module Lexer where
}
%wrapper "posn"
$alpha = [_a-zA-Z]
$digit = [0-9]

tokens :-
$white+                  ; -- ignorar carateres brancos
if                       { \pos _ -> IF pos }
$alpha($alpha|$digit)*   { \pos s -> ID pos s }
$digit+                  { \pos s -> NUM pos (read s) }
$digit+"."$digit+        { \pos s -> REAL pos (read s) }

{
data Token = ID AlexPosn String | NUM AlexPosn Int | REAL AlexPosn Double
}
```

Usar o analisador gerado

```
module Main where
import Lexer
-- importar o tipos e a função
-- alexScanTokens :: String -> [Token]

main = do
   txt <- getContents
   print (alexScanTokens txt)
```

Esta aula

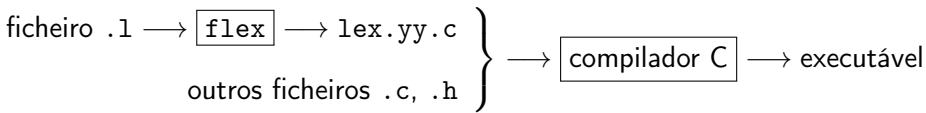
Geradores de analisadores lexicais

Alex

Flex

Escrever regras

- *Flex* é um gerador de analisadores lexicais para C
- Recebe a descrição do analisador (e.g. `lexer.l`)
  - regras com expressões regulares
  - ações escritas em C e definições auxiliares
- Produz um ficheiro de código C (`lex.yy.c`)
- Pode ser compilado juntamente com outros módulos


ficheiro .l
flex
lex.yy.c
outros ficheiros .c, .h
compilador C
executável

Exemplo de um ficheiro *Flex*

```
%{
#include "tokens.h"
%}
%option noyywrap
alpha         [_a-zA-Z]
digit         [0-9]
%%

[ \t\n\r]+                    /* skip whitespace */
if                            { return IF; }
{alpha}({alpha}|{digit})*     { return ID; }
{digit}+                      { yynum = atoi(yytext); return NUM; }
{digit}+"."{digit}+           { yyreal = atof(yytext); return REAL; }
<<EOF>>                       { return EOF; /* end of input */ }
```

```
#include "tokens.h"

int main(void) {
  int token;
  while ((token = yylex()) != EOF) {
    switch (token) {
      case IF: printf("IF "); break;
      case ID: printf("ID(%s) ", yytext); break;
      case NUM: printf("NUM(%d) ", yynum); break;
      case REAL: printf("REAL(%f) ", yyreal); break;
    }
  }
```

- A função `yylex()` retorna o próximo *token*
- Os *tokens* são representados por inteiros definidos num ficheiro *header*
- A variável global `yytext` contém os carateres do *token* atual
- Os valores de cada *token* são guardados em variáveis de estado (e.g. globais)

```
%{
declarações
%}
definições
%%

padrão 1          { ação 1  }
padrão 2          { ação 2  }
⋮

%%

código C auxiliar (opcional)
```

- O *Flex* pode contar a posição no ficheiro usando

  `%option yylineno`
- A variável global `yylineno` contém o número de linha
- É responsabilidade do programador guardar essa posição à medida que vai obtendo *tokens*

Esta aula

Geradores de analisadores lexicais

Alex

Flex

Escrever regras

O *Flex*, *Alex* e outros geradores deste género usam dois critérios para desambiguar regras:

- Longest match: Escolher a regra que consome o *maior comprimento* de carateres de *input*
- First match: Em caso de empate no comprimento, escolher a regra que ocorre primeiro

Recomendação: colocar padrões mais específicos antes dos mais gerais (e.g. palavras reservadas antes de identificadores).

Alex User Guide: `https://www.haskell.org/alex/doc/html/index.html`

Flex Manual: `https://westes.github.io/flex/manual/`