Compiladores - Análise Léxica

Fabio Mascarenhas – 2018.1

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

Introdução

- Primeiro passo do front-end: reconhecer *tokens*
- Tokens são as palavras do programa
- O analisador léxico transforma o programa de uma sequência de caracteres sem nenhuma estrutura para uma sequência de tokens

```
if x == y then
  z = 1;
else
  z = 2;
```

```
if| |x| |==| |y| |then|\n  |z| |=| |1|;|\n|else|\n  |z| |=| |2|;|<EOF>
```

Tipo do token

- Em português:
  - substantivo, verbo, adjetivo...
- Em uma linguagem de programação:
  - identificador, numeral, if, while, (, ;, *indentação*, ...

Tipo do token

- Cada tipo corresponde a um *conjunto de strings*

- Identificador: *strings de letras ou dígitos, começadas por letra*

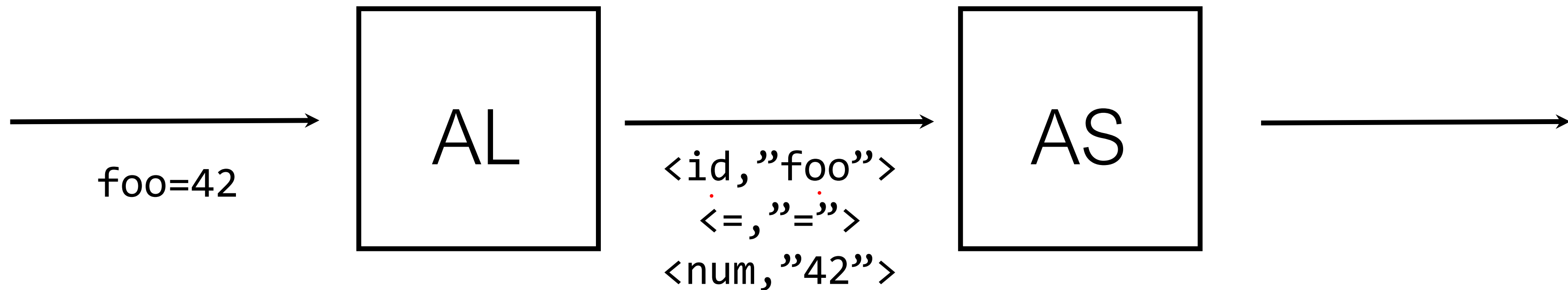
- Numeral: *strings de dígitos*

- Espaço em branco: *uma string de brancos, quebras de linha, tabs, ou comentários*

- while: *a string* while

Análise léxica

- Classificar substrings do programa de acordo com seu tipo
- Fornecer esses tokens (par tipo e substring) ao analisador sintático



Exemplo

- Para o código abaixo, conte quantos tokens de cada tipo ele tem

```
x = 0;\nwhile (x < 10) {\n\tx++;\n}\n
```

Tipos: id, espaço, num, while, outros

Exemplo

- Para o código abaixo, conte quantos tokens de cada tipo ele tem

```
x = 0;\nwhile (x < 10) {\n\tx++;\n}\n
```

Tipos: id (3), espaço (10), num (2), while (1), outros (9)

Ambiguidade

- A análise léxica de linguagens modernas é bem simples, mas historicamente esse não é o caso

- Em FORTRAN, espaços em branco *dentro de um token* também são ignorados

  - VAR1 e VAR 1 são o mesmo token

  - DO5I=1,25 são 7 tokens: “DO”, “5”, “I”, ‘=’, “1”, “,”, “25”

  - Já DO5I=1.25 são 3 tokens: “DO5I”, “=”, “1.25”

Ambiguidade

- As palavras-chave de PL/1 não são reservadas
  - IF ELSE THEN THEN = ELSE; ELSE ELSE = THEN
- Mas mesmo linguagens modernas têm ambiguidades léxicas
  - ==, ++, +=
  - Templates C++/Generics Java: List<List<Foo>> vs foo >> 2;
- O analisador léxico precisa manter um “lookahead” para saber onde um token começa e outro termina

Linguagens regulares

- Um tipo de token é um conjunto de strings
- Outro nome para conjunto de strings é *linguagem*
- Geralmente os conjuntos de strings que caracterizam os tipos de tokens de linguagens de programação são *linguagens regulares*
- Em linguagens formais, uma *linguagem regular* é qualquer conjunto de strings que pode ser expresso usando uma *expressão regular*
- Logo, o fato dos tipos de tokens serem linguagens regulares dá uma notação conveniente para especificarmos como classificar os tokens!

Expressões regulares

- Assim como uma expressão aritmética denota um número (por exemplo, “2+3*4” denota o número 14, uma expressão regular denota uma linguagem regular
    - Por exemplo, `a0+` denota a linguagem { “a0”, “a00”, “a000”, ... }
- Podemos explorar expressões regulares usando a função `lex.RE.findAll` das notas de aula executáveis
- Ela recebe uma expressão regular e uma string e retorna todas as ocorrências daquela expressão regular na string
    - `findAll(“a0+”, “a0 fooa000bar a005”) => [“a0”, “a000”, “a00”]`

Caracteres e classes

- *Caracteres* e *classes de caracteres* são o tipo mais simples de expressão regular
- Denotam conjuntos de cadeias de um único caractere
- A expressão a denota o conjunto { "a" }, a expressão x o conjunto { "x" }
- A expressão `.` é especial e denota o *conjunto alfabeto* (conjunto de todos os caracteres)
- Uma *classe* `[abx]` denota o conjunto { "a", "b", "x" }
- Uma classe `[ab-fx]` denota { "a", "b", "c", "d", "e", "f", "x" }
- Uma classe `[^ab-fx]` denota o *conjunto complemento* da classe "[ab-fx]" em relação ao alfabeto

Concatenação ou justaposição

- A concatenação ou justaposição de expressões regulares denota um conjunto com cadeias de vários caracteres, onde cada caractere da cadeia vem de uma das expressões concatenadas

- `[a-z][0-9]` denota o conjunto { “a0”, “a1”, ..., “a9”, “b0”, ..., “b9”, ..., “z9” }

- `while` denota o conjunto { “while” }

- `[wW][hH][iI][lL][eE]` denota o conjunto { “while”, “While”, “wHile”, “WHile”, ... }

- `...` denota o conjunto de todas as cadeias de três caracteres (incluindo espaços!)

Repetição

- O operador + denota a *repetição* de um caractere ou classe de caracteres
  - `[a-z]+` denota o conjunto { “a”, “aa”, “aaa”, ..., “b”, “bb”, ..., “aba”, ... }, ou seja, cadeias formadas de caracteres entre *a* e *z*
  - `[a-z][0-9]+` denota o conjunto { “a0”, “a123”, “d25”, ... }, ou seja, cadeias formadas por um caractere de *a* a *z* seguidas por um ou mais dígitos
- O operador * é uma repetição que permite *zero* caracteres ao invés de ao menos 1
  - `[a-z][0-9]*` denota o conjunto acima, mais o conjunto { “a”, “b”, ... “z” }
  - `\”[^\”]*\”` denota o conjunto de cadeias de quaisquer caracteres entre aspas duplas, exceto as próprias aspas duplas, e inclui a cadeia “”

União e opcional

- Uma barra (|) em uma expressão regular denota a união dos conjuntos das expressões à esquerda e à direita da barra

  - `[a-zA-Z_][a-zA-Z0-9_]*|[0-9]+` é a união do conjunto denotado por `[a-zA-Z_][a-zA-Z0-9_]*` com o conjunto denotado por `[0-9]*`

- O operador ? denota o conjunto denotado pela expressão que ele modifica, mais a cadeia vazia

  - `[0-9]+([.][0-9]+)?` denota o conjunto de todos as sequências de dígitos, mais o conjunto das sequências de dígitos seguidas por um ponto e outra sequência de dígitos