Análise léxica

Buferização da entrada

Prof. Edson Alves

Faculdade UnB Gama

Buferização

- Como a análise léxica é a única fase do compilador que lê os caracteres do programa fonte, um a um, ela pode concentrar uma parte significativa do tempo de execução do compilador
- Isto porque o acesso à entrada (em geral, um arquivo em disco) pode ser o gargalo, em termos de performance, do compilador
- A buferização consiste no uso de um ou mais vetores auxiliares (*buffers*), que permitem a leitura da entrada em blocos, de modo que o analisador léxico leia os caracteres a partir destes *buffers*, os quais são atualizados e preenchidos à medida do necessário
- Com a buferização os acessos aos disco são reduzidos e a leitura dos caracteres passa a ser feita em memória, com acessos consideravelmente mais rápidos

Estratégias para implementação de analisadores léxicos

Há três estratégias gerais para se implementar um analisador léxico, cada uma delas tratando a buferização de modo diferente. São elas, da mais simples para a mais complexa:

- Usar um gerador de analisador léxico, com uma entrada especificada a partir de expressões regulares. A buferização é tratada pelo próprio gerador
- Escrever o analisador léxico em alguma linguagem de programação convencional (C, C++, etc). A buferização fica atrelada aos mecanismos de I/O da linguagem
- Escrever o analisador em linguagem de montagem e tratar explicitamente a leitura da entrada e a buferização

Pares de *buffers*

- Na técnica de pares de *buffers*, um *buffer* (região contígua da memória) é dividido em duas metades, com N caracteres cada
- Em geral, N corresponde ao tamanho de um bloco do disco (por exemplo, 1024 ou 4096 caracteres)
- Cada metade do *buffer* é preenchida de uma única vez, por meio da chamada de uma função de leitura do sistema
- Caso restem na entrada menos do que N caracteres, é inserido um caractere especial no *buffer* para indicar o fim da entrada (em geral, o caractere `EOF` - *end of file*)
- Usando esta técnica, os tokens devem ser extraídos do *buffer*, sem o uso de chamadas individuais da rotina que lê um caractere da entrada

Dois ponteiros

- Os tokens podem ser extraídos do par de *buffers* por meio do uso de dois ponteiros L e R
- Uma cadeia de caracteres delimitada por este dois ponteiros é o lexema atual
- Inicialmente, L e R apontam para o primeiro caractere do próximo lexema a ser identificado
- O ponteiro R então avança até que o padrão de um token seja reconhecido
- Daí o lexema é processado e ambos ponteiros se movem para o primeiro caractere após o lexema
- Neste cenário, espaços em branco e comentários são padrões que não produzem tokens

Atualização dos *buffers* e o ponteiro R

- Se o ponteiro R tentar se deslocar para além do meio do *buffer*, será preciso preencher a metade direita com N novos caracteres antes deste avanço
- De forma semelhante, se R atingir a extremidade direita do *buffer*, a metade à esquerda deve ser devidamente atualizada
- Após esta atualização, R deve retornar para a primeira posição do *buffer*
- O uso de um par de *buffers* e dois ponteiros tem uma limitação clara: o lexema pode ter, no máxixmo, $2N$ caracteres
- O recuo de R, se necessário, também é limitado pela posição que L ocupa

Avanço de R em um par de *buffers*

1: **if** R está no fim da primeira metade **then**
2: Atualize a segunda metade com a leitura de N novos caracteres
3: $R \leftarrow R + 1$
4: **else if** R está no fim da segunda metade **then**
5: Atualize a primeira metade com a leitura de N novos caracteres
6: $R \leftarrow 0$ ▷ *Assuma que os índices de buffer começem em zero*
7: **else if** **then**
8: $R \leftarrow R + 1$

Sentinelas

- O uso de um valor sentinela no fim de cada metade do *buffer* permite a redução dos testes para o avanço de R
- Além disso, o valor sentinela em outra posição do *buffer* indica o fim da entrada
- A redução do número de testes (de dois para um, na maioria dos casos) decorrente do uso de sentinelas leva a um ganho de performance do analisador léxico e, consequentemente, do compilador
- O valor sentinela (em geral, `EOF`) deve ser diferente de qualquer caractere válido da entrada, para evitar um encerramento prematuro da entrada, caso tal caractere faça parte da entrada

Atualização de R com o uso de sentinelas

1: $R \leftarrow R + 1$

2: **if** $R =$ `EOF` **then**
3: **if** R está no fim da primeira metade **then**
4: Atualize a segunda metade com a leitura de N novos caracteres
5: $R \leftarrow R + 1$
6: **else if** R está no fim da segunda metade **then**
7: Atualize a primeira metade com a leitura de N novos caracteres
8: $R \leftarrow 0$ ▷ *Assuma que os índices de buffer começem em zero*
9: **else** ▷ `EOF` *está no buffer, indicando o fim da entrada*
10: Finalize a análise léxica

Módulo buffer.h

```
1#ifndef BUFFER_H
2#define BUFFER_H
3
4const int N { 4 };
5
6class IOBuffer {
7public:
8    static IOBuffer& getInstance();
9
10    bool eof() const;
11    int tell() const;
12    void seek(int pos);
13
14    int get();
15    void unget();
```

Módulo buffer.h

```
17 private:
18     IOBuffer();
19
20     int pos, last_update;
21     char buffer[2*N + 2];
22
23     void update();
24 };
25
26 #endif
```

Módulo buffer.cpp

```cpp
#include <iostream>
#include "buffer.h"

using namespace std;

IOBuffer&
IOBuffer::getInstance()
{
    static IOBuffer buffer;
    return  buffer;
}

IOBuffer::IOBuffer() : pos(2*N), last_update(1)
{
    buffer[N] = buffer[2*N + 1] = EOF;
    update();
}
```

Módulo buffer.cpp

```
19 void IOBuffer::update()
20 {
21     ++pos;
22
23     if (buffer[pos] != EOF)
24         return;
25
26     if (pos == 2*N + 1)
27     {
28         pos = 0;
29
30         if (last_update == 1)
31         {
32             auto size = fread(buffer, sizeof(char), N, stdin);
33
34             if (size < N)
35                 buffer[size] = EOF;
36
37             last_update = 0;
38         }
39     }
```

Módulo buffer.cpp

```
40      else if (pos == N)
41      {
42          if (last_update == 0)
43          {
44              auto size = fread(buffer + N + 1, sizeof(char), N, stdin);
45
46              if (size < N)
47                  buffer[N + 1 + size] = EOF;
48
49              last_update = 1;
50          }
51
52          ++pos;
53      }
54  }
55
56  bool
57  IOBuffer::eof() const
58  {
59      return buffer[pos] == EOF;
60  }
```

Módulo buffer.cpp

```
62 int
63 IOBuffer::tell() const
64 {
65     return pos;
66 }
67
68 void
69 IOBuffer::seek(int p)
70 {
71     pos = p;             // Risco: salto arbritário (pode ir para posição desatualizada!)
72 }
73
74 int
75 IOBuffer::get()
76 {
77     auto c = buffer[pos];
78     update();
79
80     return c;
81 }
```

Módulo buffer.cpp

```
83 void
84 IOBuffer::unget()
85 {
86     --pos;              // Risco: o ponteiro pode regredir para áreas já atualizadas!
87
88     if (pos < 0)
89         pos = 2*N;
90     else if (pos == N)
91         --pos;
92 }
```

Referências

1. **AHO**, Alfred V, **SETHI**, Ravi, **ULLMAN**, Jeffrey D. *Compiladores: Princípios, Técnicas e Ferramentas*, LTC Editora, 1995.
2. GeeksForGeeks. Flex (Fast Lexical Analyzer Generator), acesso em 04/06/2022.