Um compilador simples de uma passagem

Um tradutor para expressões simples

Prof. Edson Alves

Faculdade UnB Gama

Sumário

Tradutor dirigido pela sintaxe

- Quando não há associação de atributos aos não-terminais, um tradutor dirigido pela sintaxe pode ser construído a partir da extensão de um analisador gramatical preditivo

Tradutor dirigido pela sintaxe

- Quando não há associação de atributos aos não-terminais, um tradutor dirigido pela sintaxe pode ser construído a partir da extensão de um analisador gramatical preditivo
- Para isso, inicialmente construa o analisador gramatical preditivo

Tradutor dirigido pela sintaxe

- Quando não há associação de atributos aos não-terminais, um tradutor dirigido pela sintaxe pode ser construído a partir da extensão de um analisador gramatical preditivo
- Para isso, inicialmente construa o analisador gramatical preditivo
- Em seguida, copie as ações sintáticas do tradutor nas posições adequadas no analisador gramatical preditivo

Tradutor dirigido pela sintaxe

- Quando não há associação de atributos aos não-terminais, um tradutor dirigido pela sintaxe pode ser construído a partir da extensão de um analisador gramatical preditivo
- Para isso, inicialmente construa o analisador gramatical preditivo
- Em seguida, copie as ações sintáticas do tradutor nas posições adequadas no analisador gramatical preditivo
- Se a gramática tiver uma ou mais produções recursivas à esquerda, é preciso modificar a gramatica para eliminar esta recursão antes de proceder com a construção do analisador gramatical preditivo

Transformação de produções recursivas à esquerda

Transformação de produção recursiva à esquerda

Seja $A \rightarrow A\alpha \mid A\beta \mid \gamma$ uma produção recursiva à esquerda. Esta produção equivale às produções recursivas à direta

$$A \rightarrow \gamma R$$
$$R \rightarrow \alpha R \mid \beta R \mid \epsilon$$

onde α e β é uma cadeia de terminais e não-terminais que não começam com A e nem terminam com R.

Exemplo de transformação de produção recursiva à esquerda

$$expr \rightarrow expr + digito$$
$$expr \rightarrow expr - digito$$
$$expr \rightarrow digito$$

Exemplo de transformação de produção recursiva à esquerda

$$expr \rightarrow expr + digito$$
$$expr \rightarrow expr - digito$$
$$expr \rightarrow digito$$

$$A = expr$$
$$\alpha = + \ digito$$
$$\beta = - \ digito$$
$$\gamma = digito$$
$$R = resto$$

Exemplo de transformação de produção recursiva à esquerda

$$expr \rightarrow expr + digito$$
$$expr \rightarrow expr - digito$$
$$expr \rightarrow digito$$

$$A = expr$$
$$\alpha = + \ digito$$
$$\beta = - \ digito$$
$$\gamma = digito$$
$$R = resto$$

$$expr \rightarrow digito \ resto$$
$$resto \rightarrow + \ digito \ resto$$
$$resto \rightarrow - \ digito \ resto$$
$$resto \rightarrow \epsilon$$

Esquema de tradução da gramática para expressões para a notação posfixa

$$
\begin{aligned}
expr &\rightarrow digito\ resto \\
\\
resto &\rightarrow +\ digito\ \{imprimir(\texttt{'+'})\}\ resto \\
resto &\rightarrow -\ digito\ \{imprimir(\texttt{'-'})\}\ resto \\
resto &\rightarrow \epsilon \\
\\
digito &\rightarrow 0\ \{imprimir(\texttt{'0'})\} \\
digito &\rightarrow 1\ \{imprimir(\texttt{'1'})\} \\
&\ldots \\
digito &\rightarrow 9\ \{imprimir(\texttt{'9'})\}
\end{aligned}
$$

Rotina de extração do próximo token em C++

```
#include <iostream>

using token = char;

token proximo_token()
{
    auto t = std::cin.get();

    return (token) t;
}
```

Rotina de tratamento de erro e declaração de *lookahead* em C++

```
12 token lookahead;
13
14 void erro()
15 {
16     std::cerr << "\nErro de sintaxe! lookahead = " << lookahead << '\n';
17     exit(-1);
18 }
```

Rotina de reconhecimento de tokens em C++

```
20 void reconhecer(token t)
21 {
22     if (lookahead == t)
23         lookahead = proximo_token();
24     else
25         erro();
26 }
```

Rotina associada ao não-terminal *expr* em C++

```
52 void expr()
53 {
54     digito();
55     resto();
56 }
```

Rotina associada ao não-terminal *digito* em C++

```
28 void digito()
29 {
30     if (isdigit(lookahead))
31     {
32         std::cout.put(lookahead);
33         reconhecer(lookahead);
34     } else
35     {
36         erro();
37     }
38 }
```

Rotina associada ao não-terminal *resto* em C++

```
40 void resto()
41 {
42     if (lookahead == '+' or lookahead == '-')
43     {
44         auto c = lookahead;
45         reconhecer(c);
46         digito();
47         std::cout.put(c);
48         resto();
49     }
50 }
```

Rotina principal do tradutor em C++

```
58 int main()
59 {
60     lookahead = proximo_token();
61
62     expr();
63
64     std::cout.put('\n');
65
66     return 0;
67 }
```

Scanners

- Em uma dada gramática, as sentenças de uma linguagem são compostas por cadeias de tokens

Scanners

- Em uma dada gramática, as sentenças de uma linguagem são compostas por cadeias de tokens
- A sequência de caracteres que compõem um único token é denominada lexema

Scanners

- Em uma dada gramática, as sentenças de uma linguagem são compostas por cadeias de tokens
- A sequência de caracteres que compõem um único token é denominada lexema
- Um *scanner* (ou analisador léxico) processa a entrada para produzir uma sequência de tokens

Scanners

- Em uma dada gramática, as sentenças de uma linguagem são compostas por cadeias de tokens
- A sequência de caracteres que compõem um único token é denominada lexema
- Um *scanner* (ou analisador léxico) processa a entrada para produzir uma sequência de tokens
- Dentre as diferentes tarefas que um *scanner* pode realizar estão: remoção de espaços em branco e comentários, identificação de constantes, identificadores e palavras-chave

Remoção de espaços em branco e comentários

- No fluxo de entrada, a presença de outros caracteres que não fazem parte da gramática pode levar a erros no tradutor

Remoção de espaços em branco e comentários

- No fluxo de entrada, a presença de outros caracteres que não fazem parte da gramática pode levar a erros no tradutor
- Várias linguagens permite a presença de "espaços em branco" (espaço em branco, nova linha, tabulação, etc) entre os tokens

Remoção de espaços em branco e comentários

- No fluxo de entrada, a presença de outros caracteres que não fazem parte da gramática pode levar a erros no tradutor
- Várias linguagens permite a presença de "espaços em branco" (espaço em branco, nova linha, tabulação, etc) entre os tokens
- Os espaços em branco podem ser tratados de duas maneiras:

Remoção de espaços em branco e comentários

- No fluxo de entrada, a presença de outros caracteres que não fazem parte da gramática pode levar a erros no tradutor
- Várias linguagens permite a presença de "espaços em branco" (espaço em branco, nova linha, tabulação, etc) entre os tokens
- Os espaços em branco podem ser tratados de duas maneiras:
  1. a gramática deve ser alterada para contemplar os espaços (localização, quantidade, etc), o que traz dificuldades para a especificação da gramática e para a implementação do *scanner*

Remoção de espaços em branco e comentários

- No fluxo de entrada, a presença de outros caracteres que não fazem parte da gramática pode levar a erros no tradutor
- Várias linguagens permite a presença de "espaços em branco" (espaço em branco, nova linha, tabulação, etc) entre os tokens
- Os espaços em branco podem ser tratados de duas maneiras:
  1. a gramática deve ser alterada para contemplar os espaços (localização, quantidade, etc), o que traz dificuldades para a especificação da gramática e para a implementação do *scanner*
  2. o *scanner* simplesmente ignora os espaços em branco (solução mais comum)

Remoção de espaços em branco e comentários

- No fluxo de entrada, a presença de outros caracteres que não fazem parte da gramática pode levar a erros no tradutor
- Várias linguagens permite a presença de "espaços em branco" (espaço em branco, nova linha, tabulação, etc) entre os tokens
- Os espaços em branco podem ser tratados de duas maneiras:
  1. a gramática deve ser alterada para contemplar os espaços (localização, quantidade, etc), o que traz dificuldades para a especificação da gramática e para a implementação do *scanner*
  2. o *scanner* simplesmente ignora os espaços em branco (solução mais comum)
- O *scanner* também pode ignorar o comentários, de modo que estes possa ser tratados como espaços em branco

Identificação de constantes inteiras

- Constantes inteiras são sequências de dígitos

Identificação de constantes inteiras

- Constantes inteiras são sequências de dígitos
- As constantes podem ser inseridas na gramática da linguagem por meio de produções, ou sua identificação pode ser delegada para o analisador léxico, que irá criar tokens para estas constantes

Identificação de constantes inteiras

- Constantes inteiras são sequências de dígitos
- As constantes podem ser inseridas na gramática da linguagem por meio de produções, ou sua identificação pode ser delegada para o analisador léxico, que irá criar tokens para estas constantes
- A segunda alternativa permite tratar constantes inteiras como unidades autônomas durante a tradução

Identificação de constantes inteiras

- Constantes inteiras são sequências de dígitos
- As constantes podem ser inseridas na gramática da linguagem por meio de produções, ou sua identificação pode ser delegada para o analisador léxico, que irá criar tokens para estas constantes
- A segunda alternativa permite tratar constantes inteiras como unidades autônomas durante a tradução
- Para cada constante inteira, o *scanner* gerará um token e um atributo, sendo o token um identificador de constantes inteiras (por exemplo, `num`) e o atributo o valor inteiro da constante

Identificação de constantes inteiras

- Constantes inteiras são sequências de dígitos
- As constantes podem ser inseridas na gramática da linguagem por meio de produções, ou sua identificação pode ser delegada para o analisador léxico, que irá criar tokens para estas constantes
- A segunda alternativa permite tratar constantes inteiras como unidades autônomas durante a tradução
- Para cada constante inteira, o *scanner* gerará um token e um atributo, sendo o token um identificador de constantes inteiras (por exemplo, `num`) e o atributo o valor inteiro da constante
- Por exemplo, a entrada `3 + 14 + 15` seria transformada na sequência de tokens

  `<num, 3>` `<+,>` `<num, 14>` `<+,>` `<num, 15>`

  onde o par `<x, y>` indica que o token `x` tem atributo `y`

Reconhecimento de identificadores e palavras-chave

- As linguagens de programação utilizam identificadores para nomear variáveis, vetores, funções e outros elementos

Reconhecimento de identificadores e palavras-chave

- As linguagens de programação utilizam identificadores para nomear variáveis, vetores, funções e outros elementos
- As gramáticas das linguagens, em geral, tratam os identificadores como tokens

Reconhecimento de identificadores e palavras-chave

- As linguagens de programação utilizam identificadores para nomear variáveis, vetores, funções e outros elementos
- As gramáticas das linguagens, em geral, tratam os identificadores como tokens
- Os analisadores gramaticais (*parsers*) destas gramáticas esperam um mesmo token (por exemplo, **id**) sempre que um identificador aparece na entrada

Reconhecimento de identificadores e palavras-chave

- As linguagens de programação utilizam identificadores para nomear variáveis, vetores, funções e outros elementos
- As gramáticas das linguagens, em geral, tratam os identificadores como tokens
- Os analisadores gramaticais (*parsers*) destas gramáticas esperam um mesmo token (por exemplo, **id**) sempre que um identificador aparece na entrada
- Por exemplo, a expressão `x = x + y;` deve ser convertida pelo *scanner* para

id = id + id ;

Reconhecimento de identificadores e palavras-chave

- As linguagens de programação utilizam identificadores para nomear variáveis, vetores, funções e outros elementos
- As gramáticas das linguagens, em geral, tratam os identificadores como tokens
- Os analisadores gramaticais (*parsers*) destas gramáticas esperam um mesmo token (por exemplo, **id**) sempre que um identificador aparece na entrada
- Por exemplo, a expressão `x = x + y;` deve ser convertida pelo *scanner* para

  id = **id** + **id**;

- Na análise sintática, é útil saber que as duas primeiras ocorrências de **id** se referem ao lexema `x`, enquanto que a última se refere ao lexema `y`

Reconhecimento de identificadores e palavras-chave

- Uma tabela de símbolos pode ser usada para determinar se um dado lexema já foi encontrado ou não

Reconhecimento de identificadores e palavras-chave

- Uma tabela de símbolos pode ser usada para determinar se um dado lexema já foi encontrado ou não
- Na primeira ocorrência o lexema é armazenado na tabela de símbolos e também nela, e em todas as demais ocorrências, o lexema se torna o atributo do token **id**

Reconhecimento de identificadores e palavras-chave

- Uma tabela de símbolos pode ser usada para determinar se um dado lexema já foi encontrado ou não
- Na primeira ocorrência o lexema é armazenado na tabela de símbolos e também nela, e em todas as demais ocorrências, o lexema se torna o atributo do token **id**
- As palavras-chave da linguagem são cadeias fixas de caracteres usadas como pontuação ou para identificar determinadas construções

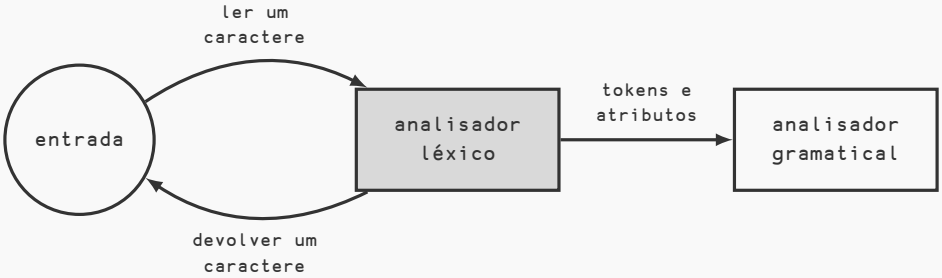
Reconhecimento de identificadores e palavras-chave

- Uma tabela de símbolos pode ser usada para determinar se um dado lexema já foi encontrado ou não
- Na primeira ocorrência o lexema é armazenado na tabela de símbolos e também nela, e em todas as demais ocorrências, o lexema se torna o atributo do token **id**
- As palavras-chave da linguagem são cadeias fixas de caracteres usadas como pontuação ou para identificar determinadas construções
- Em geral, as palavras-chave seguem a mesma regra de formação dos identificadores

Reconhecimento de identificadores e palavras-chave

- Uma tabela de símbolos pode ser usada para determinar se um dado lexema já foi encontrado ou não
- Na primeira ocorrência o lexema é armazenado na tabela de símbolos e também nela, e em todas as demais ocorrências, o lexema se torna o atributo do token **id**
- As palavras-chave da linguagem são cadeias fixas de caracteres usadas como pontuação ou para identificar determinadas construções
- Em geral, as palavras-chave seguem a mesma regra de formação dos identificadores
- Se as palavras-chave forem reservadas, isto é, não puderem ser usadas como identificadores, a situação fica facilitada: um lexema só será um identificador caso não seja uma palavra-chave

Interface para um analisador léxico



Produtor e consumidor

- O analisador léxico e o analisador gramatical formam um par produtor-consumidor

Produtor e consumidor

- O analisador léxico e o analisador gramatical formam um par produtor-consumidor
- O analisador léxico produz tokens; o analisador gramatical os consome

Produtor e consumidor

- O analisador léxico e o analisador gramatical formam um par produtor-consumidor
- O analisador léxico produz tokens; o analisador gramatical os consome
- A interação entre ambos depende do *buffer* que armazena os tokens produzidos: o *scanner* não pode gerar novos tokens se o *buffer* está cheio, o *parser* não pode prosseguir se o *buffer* estiver vazio

Produtor e consumidor

- O analisador léxico e o analisador gramatical formam um par produtor-consumidor
- O analisador léxico produz tokens; o analisador gramatical os consome
- A interação entre ambos depende do *buffer* que armazena os tokens produzidos: o *scanner* não pode gerar novos tokens se o *buffer* está cheio, o *parser* não pode prosseguir se o *buffer* estiver vazio
- Em geral, o *buffer* armazena um único token

Produtor e consumidor

- O analisador léxico e o analisador gramatical formam um par produtor-consumidor
- O analisador léxico produz tokens; o analisador gramatical os consome
- A interação entre ambos depende do *buffer* que armazena os tokens produzidos: o *scanner* não pode gerar novos tokens se o *buffer* está cheio, o *parser* não pode prosseguir se o *buffer* estiver vazio
- Em geral, o *buffer* armazena um único token
- Neste caso, o *parser* pode requisitar ao *scanner*, por demanda, a produção de novos tokens

Implementação da identificação de constantes inteiras

- Para que as constantes inteiras possam ser devidamente identificadas no código do *scanner*, é preciso que elas façam parte da gramática

Implementação da identificação de constantes inteiras

- Para que as constantes inteiras possam ser devidamente identificadas no código do *scanner*, é preciso que elas façam parte da gramática
- Por exemplo, a produção do não terminal $fator$

$$fator \rightarrow \textbf{digito} \mid (expr)$$

pode ser modificada para

$$fator \rightarrow (expr) \mid \textbf{num} \quad \{imprimir(\textbf{num}.valor)\}$$

Implementação da identificação de constantes inteiras

- Para que as constantes inteiras possam ser devidamente identificadas no código do *scanner*, é preciso que elas façam parte da gramática
- Por exemplo, a produção do não terminal $fator$

$$fator \rightarrow \textbf{digito} \mid (expr)$$

pode ser modificada para

$$fator \rightarrow (expr) \mid \textbf{num} \quad \{imprimir(\textbf{num}.valor)\}$$

- Em relação à implementação, um token deve ser identificador por um par contendo o identificador do token e o seu atributo

Exemplo de implementação do terminal $fator$ em C++

```
1 using token_t = std::pair<int, int>;
2
3 // NUM deve ter um valor diferente de qualquer caractere da tabela ASCII
4 const int NUM { 256 };
5
6 void fator()
7 {
8     auto [token, valor] = lookahead;
9
10    if (token == '(') {
11        reconhecer('(');
12        expr();
13        reconhecer(')');
14    } else if (token == NUM) {
15        reconhecer(NUM);
16        std::cout << valor;
17    } else
18        erro();
19 }
```

Exemplo de implementação de um scanner de constantes inteiras em C++

```
1 #include <bits/stdc++.h>
2
3 using token_t = std::pair<int, int>;
4 const int NUM = 256, NONE = -1;
5
6 token_t scanner()
7 {
8     while (not std::cin.eof())
9     {
10        auto c = std::cin.get();
11
12        if (isspace(c))
13            continue;
```

Exemplo de implementação de um scanner de constantes inteiras em C++

```
15          if (isdigit(c))
16          {
17              int valor = c - '0';
18
19              while (not std::cin.eof() and (c = std::cin.get(), isdigit(c)))
20                  valor = 10*valor + (c - '0');
21
22              std::cin.unget();
23
24              return { NUM, valor };
25          } else
26              return { c, NONE };
27      }
28
29      return { EOF, NONE };
30  }
```

Exemplo de implementação de um scanner de constantes inteiras em C++

```
32 int main()
33 {
34     while (true)
35     {
36         auto [lookahead, valor] = scanner();
37 
38         if (lookahead == EOF)
39             break;
40         else if (lookahead == NUM)
41             std::cout << "Número lido: " << valor << '\n';
42         else
43             std::cout << "Token lido: " << (char) lookahead << '\n';
44     }
45 
46     return 0;
47 }
```