Atividade 04 - Expressões Constantes 1 - Análise Léxica

Andrei Formiga

December 01, 2025

Sumário

1. Introdução

Na Atividade 02 criamos o primeiro compilador da disciplina, um compilador para a linguagem de Constantes Inteiras. Em seguida, na Atividade 03 vimos como traduzir (manualmente) expressões aritméticas contendo apenas constantes para a linguagem *assembly* x86-64. Agora nosso objetivo é criar um compilador que traduzirá expressões aritméticas com operandos constantes para *assembly*.

A linguagem EC1 (Expressões Constantes 1) é um pouco mais complicada que a linguagem de Constantes Inteiras, e isso significa que o compilador para EC1 vai precisar fazer análise dos programas para poder gerar o código *assembly*.

A análise da sintaxe dos programas é normalmente feita em duas etapas nos compiladores: a análise léxica agrupa caracteres isolados em unidades chamadas de *tokens* que são similares às palavras da língua portuguesa. Em seguida, a análise sintática usa os *tokens* produzidos pela análise léxica para obter a estrutura sintática do programa de entrada.

Nesta atividade, o objetivo é realizar a análise léxica da linguagem EC1. As próximas atividades continuarão o processo de análise, interpretação e compilação da linguagem EC1.

2. A linguagem EC1 (Expressões Constantes 1)

Um programa na linguagem EC1 é uma expressão aritmética com operandos constantes e usando as quatro operações. Todas as operações devem ser escritas entre parênteses, então não vamos nos preocupar com precedência de operadores.

Alguns exemplos de programas na linguagem EC1:

```
333
(6 * 7)
(3 + (4 + (11 + 7)))
(33 + (912 * 11))
((427 / 7) + (11 * (231 + 5)))
```

A gramática para a linguagem EC1 é:

```
<programa> ::= <expressao>
<expressao> ::= (<expressao> <operador> <expressao>) | <literal-inteiro>
<operador> ::= + | - | * | /
<literal-inteiro> ::= <digito>+
<digito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Agora que a linguagem é mais complicada que apenas uma constante inteira, precisamos pensar em como analisar a estrutura do programa de entrada.

3. Análise léxica e sintática

O processo de analisar a estrutura de um programa de entrada para determinar seu significado (o que o programa faz) é normalmente chamado de análise sintática.

A análise sintática para linguagens computacionais é tradicionalmente dividida em duas etapas:

1. a análise léxica, que consiste em agrupar os caracteres individuais em *tokens* (similares às palavras da linguagem natural) e classificar esses *tokens* em categorias sintáticas
2. a análise sintática propriamente dita, que verifica a estrutura gramatical da entrada com base na sequência de *tokens* da análise léxica

Embora seja possível fazer a análise sintática em apenas uma etapa, a divisão em análise léxica e sintática torna todo o processo muito mais fácil. Não ter a análise léxica como processo separado é como ler um texto sem separação de palavras:

> OfundadordaminhafamíliafoiumcertoDamiãoCubas,quefloresceunaprimeirametadedo-séculoXVIII.Eratanoeirodeofício, naturaldoRiodeJaneiro,ondeteriamorridonapenúriaenaobscuridade,sesomenteexercesseatanoaria.

A base teórica para a análise léxica e análise sintática são as linguagens formais. Usamos as linguagens regulares na análise léxica, e as linguagens livres de contexto na análise sintática.

4. Análise léxica da linguagem EC1

A principal tarefa da análise léxica para a linguagem EC1 é agrupar os caracteres que formam as constantes inteiras. Um literal inteiro pode ser composto por um número arbitrário de dígitos, e a análise léxica deve reunir esses dígitos em um *token* do tipo constante.

Além das constantes, os outros tipos de *token* são relacionados à pontuação e operadores: `(`, `)`, `+`, `-`, `*` e `/`.

Ao gerar os *tokens*, a análise léxica classifica cada *token* gerado em uma das classes léxicas da linguagem. Para a linguagem EC1, as classes são:

- número
- pontuação: `(` e `)`
- operadores: `+`, `-`, `*` e `/`

A saída da análise léxica é uma sequência de *tokens*, cada um representado por alguma estrutura de dados que guarda as informações necessárias.

4.1. Estrutura de dados do token

As duas informações essenciais para guardar para cada *token* são o lexema que gerou o *token*, e a classe ou tipo do *token*. O lexema é a *string* da entrada que gerou o *token*. Por exemplo, a sequência de caracteres `1234` deve gerar um *token* do tipo número e com lexema `1234`.

Como vimos, a linguagem EC1 tem três tipos de *token*: números, pontuação e operadores. Para facilitar o uso dos *tokens* nas etapas posteriores do compilador, podemos criar um tipo separado para cada pontuação e cada operador; assim, ao invés de apenas um tipo pontuação, podemos ter um tipo para "parêntese esquerdo" e outro para "parêntese direito". Da mesma forma, podemos separar os operadores em quatro tipos. Alguns analisadores léxicos também incluem um tipo separado de *token* para sinalizar o final da entrada, muitas vezes chamado de um *token* EOF, da siga em inglês *End Of File*.

Além do lexema e do tipo do *token*, é comum guardar em cada um a posição em que ele ocorreu no arquivo de entrada. Isso é muito importante para o tratamento de erros no compilador; quando ocorre um erro no arquivo fonte, é necessário mostrar para o usuário qual o local do arquivo em que o erro foi encontrado.

A posição pode incluir apenas o deslocamento do caractere, mas pode incluir também o número de linha e coluna do caractere onde ocorre o erro. Mesmo que a estrutura do *token* não guarde o número de linha e coluna, é possível obter esses números a partir do deslocamento.

4.2. Exemplo de análise léxica de programa EC1

Para o seguinte programa EC1:

```
(33 + (912 * 11))
```

A saída do analisador léxico é a seguinte sequência de *tokens*, com cada *token* seguindo o formato `<tipo, lexema, posicao>`:

```
<ParenEsq, "(", 0>
<Numero, "33", 1>
<Soma, "+", 4>
<ParenEsq, "(", 6>
<Numero, "912", 7>
<Mult, "*", 11>
<Numero, "11", 13>
<ParenDir, ")", 15>
<ParenDir, ")", 16>
```

4.3. Erros léxicos

O analisador léxico deve gerar a sequência de *tokens* correspondente ao programa de entrada. O programa de entrada do compilador é fornecido pelo usuário, e esse usuário pode cometer erros. Se o programa de entrada incluir caracteres ou sequências de caracteres que não podem ser reconhecidas como um *token* da linguagem, isso é um *erro léxico*. Se houver um ou mais erros léxicos na entrada, o analisador léxico não tem como fazer a análise léxica completa e deve reportar o erro ao usuário.

O analisador léxico pode para o processo de análise léxica e informar o erro léxico assim que encontrar o primeiro erro léxico, ou pode tratar um erro léxico como um tipo de *token* específico e tentar continuar a análise. A segunda opção é útil pois pode reportar vários erros ao usuário de uma vez, mas pode ser um pouco mais difícil de implementar.

A posição guardada pelo analisador deve ser usada para reportar o erro, gerando mensagens como *Erro léxico na posição X*.

No caso da linguagem EC1, qualquer caractere fora do conjunto de parênteses, operadores e dígitos vai ser um erro léxico.

4.4. Comentários

Uma outra função comumente feita no analisador léxico é remover os comentários da entrada, já que normalmente os comentários não influenciam na semântica do programa (ou seja, o que o programa faz).

Na linguagem EC1 não temos uma definição de sintaxe para comentários, mas os grupos podem adicionar suporte a comentários (e decidir a sintaxe deles) de forma opcional. Poder escrever comentários será muito útil nas linguagens mais complexas que veremos em atividades futuras.

4.5. Uso do analisador léxico

A forma de usar o analisador léxico é geralmente por uma das seguintes funções:

1. Uma função para obter o próximo *token* (por exemplo `proximo_token`) que, quando chamada, retorna o próximo *token* da entrada; quando não houver mais *tokens* na entrada, essa função sinaliza o final da entrada por algum erro, exceção, ou retornando um *token* do tipo EOF
2. Uma função para obter todos os *tokens* da entrada de uma vez em uma lista ou vetor. Neste caso não é preciso ter um *token* específico para o final da entrada, já que todos os *tokens* são colocados em uma lista.

Em compiladores usados em produção, a entrada pode ser muito grande e ter todos os *tokens* na memória ao mesmo tempo pode criar um problema de desempenho no compilador. Como normalmente a análise sintática pode prosseguir apenas olhando o próximo *token*, uma função como `proximo_token` é suficiente.

Nos compiladores que faremos nessa disciplina isso não será um problema, portanto qualquer uma das duas interfaces vai funcionar.

5. Artefato para entrega

Cada grupo deve entregar um analisador léxico completo para a linguagem EC1 que recebe um arquivo de entrada e imprime a sequência de *tokens* da entrada. Além disso, o analisador deve possuir um conjunto de testes que verifique que o analisador funciona corretamente para expressões com diferentes tipos de espaços em branco, e que o analisador detecta e reporta os erros léxicos.

É possível usar a impressão da sequência de *tokens* como parte dos testes, por exemplo usando uma ferramenta como cram: https://bitheap.org/cram/°