UNIVERSIDADE FEDERAL DE ALAGOAS
INSTITUTO DE COMPUTAÇÃO
COMPILADORES - PROF. ROBERTA VILHENA

ATIVIDADE COMPLEMENTAR: DETALHAMENTO DE EXECUÇÃO DE UM COMPILADOR COM EXEMPLIFICAÇÃO PRÁTICA

Yanka Ribeiro, Victor Ferro, Ana Carolina Nesso e Guilherme Medeiros
Graduandos em Ciência da Computação

Sumário

MACEIÓ, AL
6 de janeiro de 2023

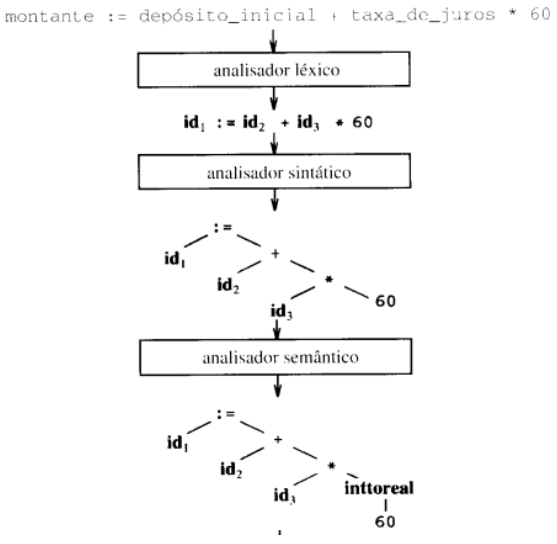
1. Princípios

Podemos definir um compilador como um software capaz de converter uma linguagem de alto nível em uma linguagem de baixo nível (linguagem de máquina) com o objetivo de tornar um programa executável, uma vez que, computadores são fluentes apenas na linguagem binária. Assim, o processo de compilação refere-se ao ato de conversão do código-fonte para o código-objeto.

Este processo pode ser dividido em duas partes: análise e síntese. Na parte de análise, o código-fonte é dividido em partes constituintes e uma representação intermediária é criada. Na síntese, esta representação intermediária é utilizada para a construção do código-objeto. Aqui daremos foco ao processo de análise do código-fonte.

Esta etapa é descrita em 3 fases:

- ➔ Análise linear ou léxica, onde o fluxo de caracteres existente no código original é lido da esquerda para a direita e combinado com um significado coletivo, assumindo uma nova estrutura denominada token.
- ➔ Análise hierárquica ou sintática, os tokens então são distribuídos hierarquicamente em frases gramaticais as quais o compilador utiliza para construir a árvore sintática.
- ➔ Análise semântica, aqui essa hierarquia de tokens é verificada para condizer com as especificações esperadas de um programa. Verificação de tipos, identificação de operadores e operandos são exemplos de ações semânticas.



1.1. Definição: Linguagem de programação

Uma linguagem de programação pode ser definida pela descrição da aparência de suas estruturas e o significado destas, sintaxe e semântica respectivamente. Para tanto, a notação utilizada é chamada de gramática livre de contexto ou BNF.

Além de especificar a sintaxe da linguagem, uma GLC pode ser usada como auxílio na tradução de programas e possui quatro componentes:

- ➔ Um conjunto de tokens, os símbolos terminais.
- ➔ Um conjunto de não-terminais.
- ➔ Um conjunto de produções, sendo uma produção: um não-terminal, uma seta e uma sequência de tokens e/ou não terminais.

➔ Uma designação a um dos não-terminais como o estado inicial.

Uma gramática deriva cadeias começando pelo estado inicial, e então substituindo repetidamente um não-terminal pelo direito de uma produção para aquele não-terminal. As cadeias de tokens que podem ser derivadas a partir do símbolo de partida formam a linguagem definida pela gramática.

$$
\begin{aligned}
&\textit{lista} \rightarrow \textit{lista} + \textit{dígito}\\
&\textit{lista} \rightarrow \textit{lista} - \textit{dígito}\\
&\textit{lista} \rightarrow \textit{dígito}\\
&\textit{dígito} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

1.2. Gerenciamento da tabela de símbolos

Uma outra função essencial de um compilador consiste no armazenamento e descrição de identificadores usados no código-fonte. Vários atributos sobre estes identificadores são coletados a fim de prover informações sobre memória, tipo, escopo além de argumentos e seus tipos de transmissão no caso de identificadores de procedimentos.

Para armazenar essas informações é utilizado uma tabela de símbolos. Cada linha da tabela refere-se a um identificador enquanto que as colunas dizem respeito aos atributos deste identificador. Esta estrutura de dados permite um acesso e manipulação rápida das informações durante todas as fases de execução do compilador.

O analisador léxico é o responsável por detectar um identificador e representá-lo na tabela de símbolos, enquanto que nas fases remanescentes, as informações sobre os atributos são alteradas e/ou utilizadas de diferentes maneiras. Exemplo, a verificação de fases na fase semântica mencionada acima é feita utilizando a tabela de símbolos.

1.3. Detecção de erros

Cada etapa de análise é passível de reconhecer erros e lidar com estes caso existam, de tal forma que o processo de compilação possa continuar e o maior número de erros possa ser detectado. Assim sendo, na fase léxica são reconhecidos os erros de caracteres que não formam token de acordo com o estabelecido na linguagem. Quando o fluxo de tokens fere as regras de sintaxe estabelecidas, é gerado um erro de ordem sintática. Por fim, na análise semântica a construção de estruturas sintáticas incorretas é denunciada como um erro semântico.

2. Demonstração de execução

Diante do exposto acima, observaremos então os conceitos teóricos na prática implementando uma calculadora básica com 5 operações capaz de reconhecer uma

expressão aritmética válida ou não. A linguagem de programação escolhida para este ensaio foi Python 3.10.6.

2.1. Construindo a BNF

Objetivo: definir uma gramática livre de contexto contendo as produções esperadas pela calculadora.

```
expr → termo + expr | termo - expr | termo
termo → fator * termo | fator / termo | fator
fator → [0-9] | ( expr ) | $
```

Portanto, os símbolos terminais são + - * / ( ) 0 1 2 3 4 5 6 7 8 9 $, logo, os únicos caracteres que devem ser aceitos durante a análise léxica.

2.2. Analisador léxico

Com a gramática em mãos, podemos construir o analisador léxico e conferir a sintaxe do programa de exemplo. Todos os caracteres precisam ser símbolos terminais. Abaixo, temos o programa de entrada utilizado como exemplo. O símbolo $ delimita o final da string.

```
calculator > ≡ example.txt
  1   10 + 7 + 5*9 /$
```

E então a saída do analisador linear. A tabela de símbolos.

```
yanka@calculator > python3 main.py
------------------------------
~> LEXICAL

{'lexeme': '10', 'char_type': 'DIGIT'}
{'lexeme': '+', 'char_type': 'SUM-OP'}
{'lexeme': '7', 'char_type': 'DIGIT'}
{'lexeme': '+', 'char_type': 'SUM-OP'}
{'lexeme': '5', 'char_type': 'DIGIT'}
{'lexeme': '*', 'char_type': 'MUL-OP'}
{'lexeme': '9', 'char_type': 'DIGIT'}
{'lexeme': '/', 'char_type': 'DIV-OP'}
```

Para simular uma situação de erro nesta fase, foi introduzido um erro na entrada, demonstrado a seguir.

Input, com alteração no primeiro operador:

```
calculator > ≡ example.txt
  1   10 # 7 + 5*9 /$
```

Output:

```
yanka@calculator > python3 main.py
Traceback (most recent call last):
  File "/home/yanka/Desktop/UFAL/compiladores 22.1/calculator/main.py", line 9, in <module>
    main()
  File "/home/yanka/Desktop/UFAL/compiladores 22.1/calculator/main.py", line 5, in main
    lexical_analyzer(f)
  File "/home/yanka/Desktop/UFAL/compiladores 22.1/calculator/lexical.py", line 54, in lexical_analyzer
    raise Exception('\'' + char + '\' is not a valid character.')
Exception: '#' is not a valid character.
```

Percebe-se que o único erro reportado foi o de caráter léxico, mesmo a estrutura da divisão, no final da entrada, estando incompleta.

O código desta fase pode ser encontrado aqui.

2.3. Analisador sintático

Considerando a expressão de exemplo, 10 + 5 * 9, podemos dizer que há pelo menos duas interpretações: 10 + (5 * 9) ou (10 + 5) * 9. Assim, precisamos conhecer a precedência relativa dos operadores quando mais de um tipo deles estiver presente. Na aritmética ordinária, a multiplicação e a divisão têm maior precedência que a adição e a subtração. Portanto, dizer que * possui maior precedência que + é dizer que * captura seus operandos antes de + o fazer.

Por conseguinte, 5 é capturado tanto por * em 10 + 5 * 9 quanto em 10 + 5 * 9, logo, as expressões são equivalentes, respectivamente, a 10 + (5 * 9) e (10 + 5) * 9. Considerando agora a gramática aqui exposta, notamos que qualquer expressão parentizada é fator e, portanto, com parênteses podemos desenvolver expressões que tenham níveis arbitrários de aninhamento.

```
associação à direita  (    (maior precedência)
associação à esquerda * /
associação à esquerda + -
```

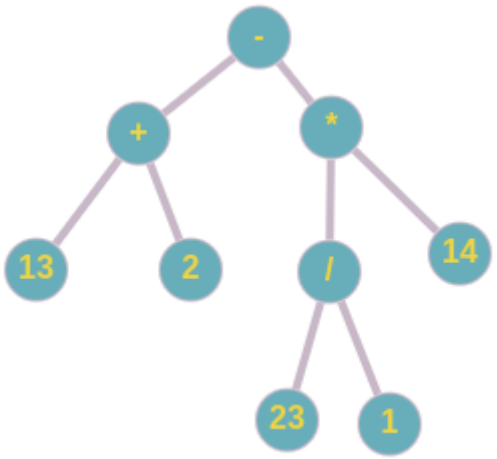
O objetivo desta fase então é construir uma árvore gramatical caso o programa de entrada esteja de acordo com as produções da gramática. Dado o exemplo, 13+2-23/1*14$, a saída agora será:

```
yanka@calculator > python3 main.py
------------------------------
~> LEXICAL

{'lexeme': '13', 'char_type': 'DIGIT'}
{'lexeme': '+', 'char_type': 'SUM-OP'}
{'lexeme': '2', 'char_type': 'DIGIT'}
{'lexeme': '-', 'char_type': 'MINUS-OP'}
{'lexeme': '23', 'char_type': 'DIGIT'}
{'lexeme': '/', 'char_type': 'DIV-OP'}
{'lexeme': '1', 'char_type': 'DIGIT'}
{'lexeme': '*', 'char_type': 'MUL-OP'}
{'lexeme': '14', 'char_type': 'DIGIT'}
------------------------------
~> PARSER

{'-': ({'+': ('13', '2')}, {'*': ({'/': ('23', '1')}, '14')})}
```

A árvore sintática na forma achatada que pode ser reescrita da seguinte forma:



Os erros previstos nesta fase são:

➔ Dois operadores juntos.

```
yanka@calculator > python3 main.py
------------------------------
~> LEXICAL

{'lexeme': '10', 'char_type': 'DIGIT'}
{'lexeme': '+', 'char_type': 'SUM-OP'}
{'lexeme': '+', 'char_type': 'SUM-OP'}
{'lexeme': '20', 'char_type': 'DIGIT'}
{'lexeme': '-', 'char_type': 'MINUS-OP'}
{'lexeme': '7', 'char_type': 'DIGIT'}
------------------------------
~> PARSER

Traceback (most recent call last):
  File "/home/yanka/Desktop/UFAL/compiladores 22.1/calculator/main.py", line 11, in <module>
    filename  = parserr(fpar)
  File "/home/yanka/Desktop/UFAL/compiladores 22.1/calculator/parserr.py", line 67, in parserr
    raise Exception('Cannot recognize pattern: Too many operators')
Exception: Cannot recognize pattern: Too many operators
```

➔ Operador no início ou final da cadeia

```
yanka@calculator > python3 main.py
------------------------------
~> LEXICAL

{'lexeme': '/', 'char_type': 'DIV-OP'}
{'lexeme': '2', 'char_type': 'DIGIT'}
{'lexeme': '+', 'char_type': 'SUM-OP'}
{'lexeme': '4', 'char_type': 'DIGIT'}
------------------------------
~> PARSER

Traceback (most recent call last):
  File "/home/yanka/Desktop/UFAL/compiladores 22.1/calculator/main.py", line 11, in <module>
    filename  = parserr(fpar)
  File "/home/yanka/Desktop/UFAL/compiladores 22.1/calculator/parserr.py", line 69, in parserr
    raise Exception('Cannot start with an arithmetic operator')
Exception: Cannot start with an arithmetic operator
```

```
yanka@calculator > python3 main.py
------------------------------
~> LEXICAL

{'lexeme': '5', 'char_type': 'DIGIT'}
{'lexeme': '+', 'char_type': 'SUM-OP'}
{'lexeme': '2', 'char_type': 'DIGIT'}
{'lexeme': '*', 'char_type': 'MUL-OP'}
------------------------------
~> PARSER

Traceback (most recent call last):
  File "/home/yanka/Desktop/UFAL/compiladores 22.1/calculator/main.py", line 11, in <module>
    filename  = parserr(fpar)
  File "/home/yanka/Desktop/UFAL/compila                                    y", line 74, in parserr
    raise Exception('Cannot end with an a
Exception: Cannot end with an arithmetic operator
```

➔ Desbalanceamento de parênteses

```
yanka@calculator > python3 main.py
------------------------------
~> LEXICAL

{'lexeme': '(', 'char_type': 'OPEN-PAR'}
{'lexeme': '9', 'char_type': 'DIGIT'}
{'lexeme': '*', 'char_type': 'MUL-OP'}
{'lexeme': '2', 'char_type': 'DIGIT'}
{'lexeme': ')', 'char_type': 'CLOSE-PAR'}
{'lexeme': '-', 'char_type': 'MINUS-OP'}
{'lexeme': '5', 'char_type': 'DIGIT'}
{'lexeme': ')', 'char_type': 'CLOSE-PAR'}
{'lexeme': ')', 'char_type': 'CLOSE-PAR'}
------------------------------
~> PARSER

Traceback (most recent call last):
  File "/home/yanka/Desktop/UFAL/compiladores 22.1/calculator/main.py", line 11, in <module>
    filename  = parserr(fpar)
  File "/home/yanka/Desktop/UFAL/compiladores 22.1/calculator/parserr.py", line 77, in parserr
    raise Exception('No opening parenthesis left')
Exception: No opening parenthesis left
```

```
yanka@calculator > python3 main.py
------------------------------
~> LEXICAL

{'lexeme': '(', 'char_type': 'OPEN-PAR'}
{'lexeme': '(', 'char_type': 'OPEN-PAR'}
{'lexeme': '13', 'char_type': 'DIGIT'}
{'lexeme': '+', 'char_type': 'SUM-OP'}
{'lexeme': '2', 'char_type': 'DIGIT'}
{'lexeme': ')', 'char_type': 'CLOSE-PAR'}
{'lexeme': '-', 'char_type': 'MINUS-OP'}
{'lexeme': '(', 'char_type': 'OPEN-PAR'}
{'lexeme': '23', 'char_type': 'DIGIT'}
{'lexeme': '/', 'char_type': 'DIV-OP'}
{'lexeme': '1', 'char_type': 'DIGIT'}
{'lexeme': ')', 'char_type': 'CLOSE-PAR'}
------------------------------
~> PARSER

Traceback (most recent call last):
  File "/home/yanka/Desktop/UFAL/compiladores 22.1/calculator/main.py", line 11, in <module>
    filename  = parserr(fpar)
  File "/home/yanka/Desktop/UFAL/compiladores 22.1/calculator/parserr.py", line 80, in parserr
    raise Exception('Too many opening parenthesis left')
Exception: Too many opening parenthesis left
```

O código desta fase pode ser encontrado aqui.

2.4. Analisador semântico

Nesta fase, nosso objetivo é testar a execução da calculadora e apresentar o resultado da expressão, um número real.

```
yanka@calculator > python3 main.py
------------------------------
~> LEXICAL

{'lexeme': '5', 'char_type': 'DIGIT'}
{'lexeme': '+', 'char_type': 'SUM-OP'}
{'lexeme': '2', 'char_type': 'DIGIT'}
{'lexeme': '*', 'char_type': 'MUL-OP'}
{'lexeme': '8', 'char_type': 'DIGIT'}
------------------------------
~> PARSER

{'+': ['5', {'*': ['2', '8']}]}
------------------------------
~> SEMANTIC

Valor da expressão = 21.0
```

O código desta fase pode ser encontrado aqui.

3. Possíveis erros e melhorias de código

- ➔ Utilizar conceitos de programação orientada a objetos na criação dos tokens (durante a fase linear da análise).
- ➔ Incluir suporte para operações com números flutuantes. Double, float, e etc.

➔ Incluir suporte para operações aritméticas intermediárias. Exponenciação, fatorial, e etc.

4. Bibliografia

V. AHO, Alfred; SETHI, Ravi; D. ULLMAN, Jeffrey; *Compiladores: Princípios, Técnicas e Ferramentas*. Rio de Janeiro, RJ: LTC Livros Técnicos e Científicos Editora S.A., 1995, *E-book (351 p.)*. ISBN-10 8588639246. Disponível em: https://www.scribd.com/document/394038196/Livro-Compiladores-pdf. Acesso em 04 jan. 2023.

NORVELL, Theodore. *Parsing Expressions by Recursive Descent*. Faculty of Engineering and Applied Science - Memorial University of Newfoundland, 1999. Disponível em: https://www.engr.mun.ca/~theo/Misc/exp_parsing.htm. Acesso em: 5 jan. 2023.

Disponível: https://github.com/Sukhrobjon/Binary-Expression-Tree/blob/master/binary_expression_tree.py. Acesso em: 6 jan 2023