Compiladores

Análise Sintática (Parte 1)

Introdução

Após a análise léxica, é realizada a análise sintática. Enquanto a ação da análise léxica é separar o código em conjuntos de símbolos chamados lexemas e produzir os *tokens* de acordo com as regras da gramática da linguagem, a análise sintática trata da forma com que os símbolos estão estruturados no código. Este assunto será abordado nessas notas de aula.

Análise Sintática (*Parsing*)

A análise sintática, também chamada de *parsing*, é a segunda análise realizada pelo compilador. Ela é realizada por uma ferramenta denominada ***analisador sintático*** (ou ***parser***).

É o *parser* que confere se a sequência de *tokens* produzida pelo *scanner* pode ter sido produzida pelas regras da gramática da linguagem.

> "O *parser* (analisador sintático) recebe o fluxo de *tokens* unidos da análise léxica e verifica se esse fluxo respeita a gramática da linguagem. Além disso, durante essa verificação, o parser constrói a árvore sintática" (DU BOIS, 2011).

Na análise sintática, começa a ser importante o **relacionamento dos lexemas**. Por exemplo, não seria detectado como um erro léxico declarar "`x int;`" (em vez de `int x;`), porém, na análise sintática o relacionamento entre os lexemas é verificado.

O analisador sintático reporta erros sintáticos, caso estes ocorram. Assim, o exemplo citado seria detectado como erro pelo analisador sintático.

Todas as linguagens possuem regras que descrevem a sintaxe de programas bem formados (DU BOIS, 2011). Linguagens de programação geralmente são descritas utilizando Gramáticas Livres de Contexto – GLC. Mas por quê? A hierarquia de Chomsky (Figura 1) mostra que as Gramáticas Regulares – GR são um subconjunto das GLC.

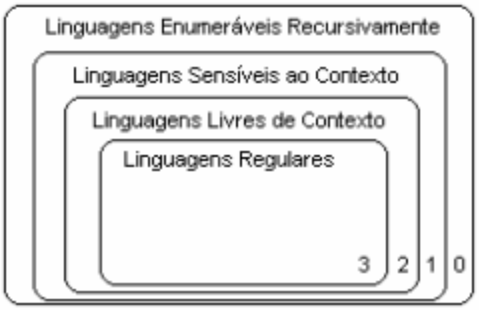

Figura 1. Hierarquia de Chomsky. Fonte: Prado, s.d.

Porém, GR aqui não são possíveis, pois não conseguem descrever toda estrutura sintática de linguagens de programação. As GR não conseguem expressar o balanceamento de blocos e parênteses, de acordo com Prado (s.d).

Conforme exemplo de Rigo (s.d.), uma soma do tipo "`a + b + c + ... + n`" pode ser reconhecida por meio de expressões regulares:

```
dígitos = [0-9]+
soma = (digitos "+")* dígitos
```

O analisador léxico substitui as abreviações antes de traduzir para um autômato finito:

```
soma = ([0-9]+ "+")* [0-9]
```

No entanto, a soma `(1 + (245 + 2))` não é reconhecida por qualquer linguagem somente com expressões regulares.

Tentativa:

```
dígitos = [0-9]+
soma = expr "+" expr
expr = "(" soma ")" | digitos
```

Sabendo que as abreviações precisam ser substituídas antes, ao tentarmos reconhecer algo como `(1 + (245 + 2))`, obtemos:

```
dígitos = [0-9]+
expr = "(" expr "+" expr ")" | dígitos
```

Acabamos de aumentar a regra, agora, como precisamos realizar as substituições antes, devemos continuar e substituir os "`exp`" internos:

```
dígitos = [0-9]+
expr = "(" ( "(" expr "+" expr ")" | digits ) "+" expr ")" | dígitos
```

Precisaríamos continuar substituindo e isso ocorreria infinitamente!

Vejamos um exemplo:

S → S; S

S → id = E

S → print (L)

E → id

E → num

E → E + E

E → (S, E)

L → E

L → L, E

Um código na forma a seguir pertence à linguagem definida pela gramática acima? (adaptado de Rigo s.d.)

```
id = num; id = id + (id = num + num, id)
```

S → S; S

S → id = E; id = E

S → id = num; id = E

S → id = num; id = E + E

S → id = num; id = id + E

S → id = num; id = id + (S, E)

S → id = num; id = id + (id = E, E)

S → id = num; id = id + (id = E + E, E)

S → id = num; id = id + (id = num + E, E)

S → id = num; id = id + (id = num + num, E)

S → id = num; id = id + (id = num + num, id)

Pertence!

Exemplo de código:

```
a = 7;
b = c + (d = 5 + 6, d)
```

Eles pertencem a esta linguagem?

No exemplo acima, as derivações ocorreram da esquerda para a direita, ou seja, o símbolo não terminal mais à esquerda foi substituído. Quando os símbolos mais à esquerda são substituídos, diz-se que foi utilizada a derivação *left-most*; quando as derivações ocorrem sempre iniciando pelo símbolo mais à direita, diz-se que esta é uma derivação *right-most*. Das duas formas, chega-se a árvores sintáticas equivalentes.

Outro exemplo:
Supondo que uma gramática é definida pelas regras a seguir, deve-se indicar uma sequência de derivações capaz de gerar a cadeia "`a * (a + a)`". O símbolo inicial é "`E`".

E → E + T | T

T → T * F | F

F → (E) | a

Resposta:

E →

T →

T * F →

F * F →

a * F →

a * (E) →

a * (E + T) →

a * (T + T) →

a * (F + T) →

a * (a + T) →

a * (a + F) →

a * (a + a)

Outra ordem possível:

E →

T →

T * F →

T * (E) →

T * (E + T) →

T * (E + F) →

T * (E + a) →

T * (T + a) →

T * (F + a) →

F * (F + a) →

a * (F + a) →

a * (a + a)

Ambas gerarão árvores sintáticas equivalentes, pois as regras aplicadas são sempre as mesmas para cada símbolo, mudando apenas a ordem de aplicação.

Em algumas gramáticas, pode acontecer de duas ou mais derivações obterem o mesmo resultado a partir da aplicação de regras diferentes. Neste caso, diz-se que essas gramáticas são ambíguas. Segundo Aho (2007), a gramática não pode ser ambígua para ocorrer a correta implementação do *parser*, pois não seria possível dizer quais foram as regras que geraram a sentença e, consequentemente, qual é a árvore de derivação.

A gramática a seguir é ambígua: A → A + A | A * A | a

Aqui estão duas formas de atingir a sentença `a + a * a`:

1) A → A + A
→ a + A
→ a + A * A
→ a + a * A
→ a + a * a

2) A → A * A
→ A * a
→ A + A * a
→ a + A * a
→ a + a * a

Notação BNF

Uma notação bastante utilizada para representar GLC é o ***Formalismo de Backus-Naur*** (***Backus-Naur Form*** – ***BNF***) ou ***Forma Normal de Backus***. Nela usa-se um símbolo à esquerda seguido de `::=` e uma expressão com símbolos à direita. Em alguns exemplos, o símbolo `::=` pode aparecer substituído por →. Na literatura os dois símbolos são muito encontrados.

Exemplos (adaptados de Kohler & Yu, 2020):

```
Compiladores ::= "Compiladores é " PalavraEmotiva
PalavraEmotiva ::= "incrível" | "terrível"

Letra ::= "a" | "b" | "c" | ... | "z"
Palavra ::= Letra | Letra Palavra

Digito ::= 0 | 1 | ... | 9
NumeroNatural ::= Digito NumeroNatural
```

Na notação utilizada até aqui em regras gramaticais, a diferença está no uso de ::= no lugar da seta (→).

Outro exemplo, mais próximo de uma linguagem de programação (adaptado de http://professor.pucgoias.edu.br/SiteDocente/admin/arquivosUpload/17389/material/Aulao.pdf):

```
<comando> ::= <while> | <atrib> | ...
```

```
<while> ::= while <expBool> do <comando>
<atrib> ::= <variável> := <expArit>
<expBool> ::= <expArit> < <expArit>
<expArit> ::= <expArit> + <termo> | <termo>
<termo> ::= <número> | <variável>
<variável> ::= I | J
<número> ::= 100
```

Como seria a regra <doWhile>?

```
<doWhile> ::= do <comando> while <expBool>
```

Tipos de *Parsers* (baseado em Du Bois, 2011)

Existem dois tipos de analisadores sintáticos que diferem na forma como a árvore sintática é construída. São eles o analisador *bottom-up* (ou ascendente) e o analisador *top-down* (ou descendente). Em geral, métodos de análise que reconheçam todas as GLC são considerados custosos. Costuma-se trabalhar com subclasses de GLC suficientemente expressivas para representar a maioria das linguagens de programação, tais como LL(k) e LR(k). As mais comuns são as gramáticas LL(1) e LR(1).

- LL(1): *Left-to-right scan, Left-most derivation, 1 token look-ahead* (Varredura da esquerda para a direita, derivação mais à esquerda, antecipação de 1 *token*).
    - Esta gramática não pode ter recursão à esquerda (ex: A → Aa | b), não pode ser ambígua (exemplo na sequência do material) e deve eliminar indecisões inserindo símbolo(s) não terminal(is) quando uma derivação possuir mais de uma alternativa para um não terminal com um prefixo comum.
    - É utilizada na análise sintática descendente (*top-down*) preditiva tabular.
- LR(1): *Left-to-right scan, Right-most derivation, 1 token look-ahead* (Varredura da esquerda para a direita, derivação mais à direita, antecipação de 1 *token*).
    - É utilizada na análise sintática ascendente (*bottom-up*).
    - Segundo Aho et al. (2007), uma gramática LR nunca pode ser ambígua.

Analisador top-down

Constrói a árvore sintática de cima para baixo, ou seja, da raiz em direção às folhas. É o mais fácil de se implementar manualmente, porém reconhece uma classe mais restrita de gramáticas. Utilizado por exemplo em analisadores de compiladores didáticos.

Pode-se criar um *parser*

- Descendente recursivo
- Recursivo preditivo
- Preditivo tabular

Parser descendente recursivo

- Técnica mais simples para implementação de *parsers*.
- Cada símbolo não terminal da gramática vira uma função lógica.
- Essas funções são combinadas usando os operadores lógicos "E" e "OU, representados respectivamente por "&&" e "||". Exemplo:

S → N ID | A B

N → ...

A → ...

B → ...

Seria implementado como:

```
boolean S() {
      return (N() && consomeToken(ID)) || (A() && B());
}
boolean N() {
      ...
}
boolean A() {
      ...
}
```

Se a produção for recursiva, a função também será. Exemplo:

S → (...)

N → N A

Seria implementado como:

```
boolean N() {
      return (N() && A());
}
```

Os símbolos terminais são consumidos da esquerda para a direita.

Os *parsers* utilizados hoje pelos compiladores GCC e Clang, por exemplo, são descendentes recursivos. O GCC já utilizou um *parser* gerado automaticamente pelo `yacc`, mas agora usa um *parser* escrito manualmente. O Clang também usa um *parser* desenvolvido manualmente, único para C, Objective C, C++ e Objective C++:

> '*Clang is the "C Language Family Front-end", which means we intend to support the most popular members of the C family. We are convinced that the right parsing technology for this class of languages is a hand-built recursive-descent parser. Because it is plain C code, recursive descent makes it very easy for new developers to understand the code, it easily supports ad-hoc rules and other strange hacks required by C*' (CLANG, s.d.).

Tradução:

> *Clang é o "front-end da família de linguagens C", o que significa que pretendemos oferecer suporte aos membros mais populares da família C. Estamos convencidos de que a tecnologia de análise correta para essa classe de linguagens é um analisador descendente recursivo construído à mão. Por ser um código C simples, a descendência recursiva torna muito fácil para novos desenvolvedores entender o código, suporta facilmente regras ad-hoc e outros hacks estranhos exigidos por C.*

Parsers escritos manualmente aprimoram a recuperação de erros, as mensagens de erro e a *performance* (GCC, 2008).

Parser recursivo preditivo (baseado em Mascarenhas, 2018)

- Uma alternativa que simplifica o *parser* descendente recursivo.
- É possível para várias gramáticas.
- Não tenta alternativas até uma ser bem sucedida, mas usa um *lookahead* na entrada para prever qual alternativa ele deve seguir.
- Quanto mais *tokens* puder analisar antecipadamente, melhor é o *parser*.

Parser preditivo tabular

- *Parser* que não utiliza recursão.
- Utiliza uma pilha explícita.
- É mais rápido.
- Mais difícil de implementar, pois necessita de uma tabela de transição (também chamada de *tabela M*)
- Existem algoritmos para gerar essa tabela automaticamente.

O esquema gráfico que ilustra o *parser* preditivo tabular está presente na Figura 2.

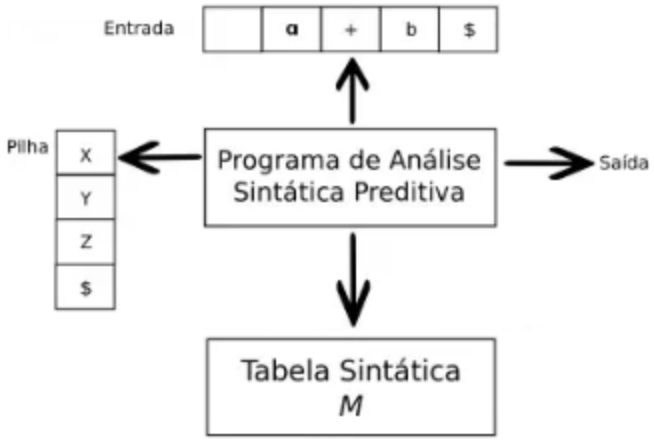

Figura 2. Esquema de funcionamento do *parser* preditivo tabular.

Para criar a tabela M a partir de uma gramática LL(1), é necessário identificar os conjuntos FIRST e FOLLOW desta gramática.

- O conjunto FIRST de um não terminal contém todos os terminais possíveis após sua derivação.

- As regras para a elaboração do conjunto FIRST, retirada de Compiladores (s.d) e presentes na Figura 3, são:
    a. Se X é um símbolo terminal, então FIRST(X) = {X].
    b. Se X é um símbolo não terminal e X → $Y_1Y_2...Y_k$ é uma produção para k>=1, então acrescente α ao FIRST(X) se, para algum i, α estiver em FIRST(Y_i), e ε estiver em todos os FIRST(Y_1), …, FIRST(Y_{i-1}); ou seja, $Y_1...Y_{i-1}$ → ε. Se ε está em FIRST(Y_j) para todo j=1, 2, …, k, então ε está em FIRST(X).
    c. Se X → e é uma produção de X então ε está em FIRST(X).

1. Se X é um símbolo terminal, então First(X)= {X}
2. Se X é um símbolo não-terminal e X → $Y_1Y_2..Y_k$ é uma produção para k ≥ 1, então acrescente α ao First(X) se, para algum i, α estiver em First(Y_i), e ε estiver em todos os First(Y_1),...,First(Y_{i-1}); ou seja, $Y_1...Y_{i-1}$ → ε. Se ε está em First(Y_j) para todo j=1, 2, …, k então ε está em First(X).
3. Se X → e é uma produção de X então ε está em First(X).

Figura 3. Regras para elaboração do conjunto FIRST. Fonte: COMPILADORES, s.d.

- O conjunto FOLLOW de um não terminal contém os símbolos terminais que o sucedem do lado direito da(s) regra(s) de produção, ou seja, todos os símbolos terminais que podem aparecer imediatamente à direita do não terminal em uma forma sentencial.
- As regras para a elaboração do conjunto FOLLOW, retirada de Compiladores (s.d) e presentes na Figura 4, são:
    a. Coloque $ (símbolo de final de cadeia) em FOLLOW(S), onde S é o símbolo inicial da gramática.
    b. Se houver uma produção A → αBβ, então tudo no FIRST(β) exceto ε está em FOLLOW(B).
    c. Se houver uma produção A → αB, ou uma produção A → αBβ onde FIRST(β) contém ε, então inclua FOLLOW(A) ao FOLLOW(B).

1. Coloque $ (símbolo de final de cadeia) em Follow(S), onde S é o símbolo inicial da gramática
2. se houver uma produção A → αBβ, então tudo no First(β) exceto ε está em Follow(B).
3. se houver uma produção A → αB, ou uma produção A → αBβ onde Firts(β) contém ε , então inclua Follow(A) ao Follow(B).

Figura 4. Regras para elaboração do conjunto FOLLOW. Fonte: COMPILADORES, s.d.

Suponha a gramática a seguir:

SEQ → DECL COMMAND

COMMAND → ; SEQ | ε

DECL → id

Quem são FIRST e FOLLOW?

FIRST(SEQ) = {id}

FIRST(COMMAND) = {;, ε}

FIRST(DECL) = {id}

FOLLOW(SEQ) = {$}

FOLLOW(COMMAND) = {$}

FOLLOW(DECL) = {;, $}

Observação: o símbolo $ é usado para denotar o final da cadeia.

A tabela M é preenchida de acordo com o conjunto FIRST:

- Cada coluna é um símbolo terminal
- Cada linha é um símbolo não terminal

- Para cada não terminal cujo conjunto FIRST possui vazio (ε), coloca-se também o conjunto FOLLOW.

	id	**;**	**$**
SEQ	SEQ→DECL COMMAND		
COMMAND		COMMAND→;SEQ	COMMAND → ε
DECL	DECL→id		

Um exemplo prático para a entrada id; id$.

PILHA	**ENTRADA**	**AÇÃO**
$SEQ	id; id$	SEQ → DECL COMMAND
$COMMAND DECL	id; id$	DECL → id
$COMMAND id	id; id$	CONSOME id
$COMMAND	; id$	COMMAND → ; SEQ
$SEQ ;	; id$	CONSOME ;
$SEQ	id$	SEQ → DECL COMMAND
$COMMAND DECL	id$	DECL → id
$COMMAND id	id$	CONSOME id
$COMMAND	$	COMMAND → ε
$	$	CONSOME $

Exercício: dada a gramática abaixo, identifique os conjuntos FIRST e FOLLOW, crie a tabela M e reconheça id + id * id$.

E → T E'

E' → + T E' | ε

T → F T'

T' → * F T' | ε

F → - F | id

FIRST(E) = {-, id}

FIRST(E') = {+, ε}

FIRST(T) = {-, id}

FIRST(T') = {*, ε}

FIRST(F) = {-, id}

FOLLOW(E) = {$}

Nunca há nada sucedendo um E do lado direito da regra, portanto, coloca-se apenas o símbolo de final de cadeia.

FOLLOW(E') = {$}

Nunca há nada sucedendo um E' do lado direito da regra, portanto, coloca-se apenas o símbolo de final de cadeia.

FOLLOW(T) = {+, $}

T E' poderia ser derivado para T + T E', ou seja, após um T pode vir um "+".

FOLLOW(T') = {+, $}

E' poderia ser derivado para + T E', depois para + F T' E', depois para + F T' + T E'.

FOLLOW(F) = {*, +, $}

E' poderia ser derivado para + T E', depois para + F T' E', depois para + F * F T' E', depois + F * F ε E', e + F * F ε + T E'

Só neste exemplo já temos F com símbolos e * e + depois dele.

Observação: a derivação ocorrerá sempre a partir do símbolo mais à esquerda, aqui apenas foi demonstrado qual poderia ser o próximo símbolo consumido.

	id	**+**	**-**	*****	**$**
E	E→TE’		E→TE’		
E’		E’→+TE’			E’→ε
T	T→FT’		T→FT’		
T’		T’→ε		T’→*FT’	T’→ε
F	F→id		F→-F		

PILHA	ENTRADA	AÇÃO
$E	id + id * id$	E → TE'
$E'T	id + id * id$	T → FT'
$E'T'F	id + id * id$	F → id
$E'T'id	id + id * id$	CONSOME id
$E'T'	+ id * id$	T' → ε
$E'	+ id * id$	E' → +TE'
$E'T+	+ id * id$	CONSOME +
$E'T	id * id$	T → FT'
$E'T'F	id * id$	F → id
$E'T'id	id * id$	CONSOME id
$E'T'	* id$	T' → *FT'
$E'T'F*	* id$	CONSOME *
$E'T'F	id$	F → id
$E'T'id	id$	CONSOME id
$E'T'	$	T' → ε
$E'	$	E' → ε
$	$	CONSOME $

Outra forma de identificar se a gramática é LL(1) é verificar se para cada entrada na tabela existe apenas uma produção possível. Isso significa que as sentenças geradas pela gramática podem ser analisadas da esquerda (*left-to-right*) para a direita produzindo uma derivação mais à esquerda (*left-most*) lendo apenas um símbolo de entrada (1).

Conclusão

A Análise Sintática é o processo responsável por verificar a estrutura dos *tokens* gerados na análise léxica. Através do uso de *parsers*, é possível garantir que o código fonte segue as regras da gramática da linguagem de programação utilizada.

Existem diferentes tipos de *parsers*, como os descendentes e ascendentes, cada um com suas características e complexidades de implementação. Neste material, foram explicados somente

os *parsers* descendentes. Os *parsers* descendente recursivo e preditivo, são mais fáceis de implementar manualmente e são adequados para gramáticas LL(1). Já os *parsers* ascendentes, como o LR(1), são capazes de lidar com um conjunto mais amplo de gramáticas, porém são mais complexos de implementar.

Exercícios

1) Dada a gramática abaixo, identifique os conjuntos FIRST e FOLLOW, crie a tabela M e reconheça -(-id(id))$. O símbolo inicial da gramática é A.

A → -A | (A) | CB

B → -A | ε

C → idD

D → (A) | ε

2) Dada a gramática abaixo, cujo símbolo inicial é C, e a sequência de *tokens*: x x y y x, faça o que se pede:

C → X Y

X → x | x X

Y → y | y Y

   a) Verifique se a sequência de *tokens* pertence à linguagem definida pela gramática fornecida.
   b) Se a sequência não pertencer à gramática, identifique o ponto onde o erro sintático ocorreu e explique o motivo do erro.

Referências

AHO, Alfred V. et al. Compilers: principles, techniques, & tools. Pearson Education India, 2007.

CLANG. Clang - Features and Goals. Disponível em: https://clang.llvm.org/features.html. Acesso em: 26 mai. 2024.

COMPILADORES. Disponível em: http://professor.pucgoias.edu.br/SiteDocente/admin/arquivosUpload/17389/material/Aulao.pdf. Acesso em: 26 mai. 2026.

DU BOIS, André Rauber. Notas de Aula sobre Compiladores. 18 ago. 2011.

DU BOIS, André Rauber. Notas de Aula sobre Compiladores. 01 set. 2011.

GCC. New_C_Parser. GCC Wiki. Editada pela última vez em 2008-01-10 19:38:45. Disponível em: https://gcc.gnu.org/wiki/New_C_Parser. Acesso em: 26. mai. 2024.

MASCARENHAS, Fábio. Compiladores – Análise Preditiva. 2018-1. Disponível em: https://dcc.ufrj.br/~fabiom/comp/08Preditiva.pdf. Acesso em: 26 mai. 2024.

PRADO, Simone das Graças Domingues. Linguagens Livres de Contexto. s.d. Disponível em: http://wwwp.fc.unesp.br/~simonedp/zipados/TC03.pdf. Acesso em: 26 abr. 2024.

RIGO, Sandro. Análise Sintática. s.d. Disponível em: https://www.ic.unicamp.br/~sandro/cursos/mc910/slides/cap3-parser.pdf. Acesso em: 26 mai. 2024.

KOHLER, Eddie; YU, Minlan. BNF Grammars. CS 61 2020. Disponível em: https://cs61.seas.harvard.edu/site/2020/BNFGrammars/. Acesso em: 26 mai. 2026.