Compiladores

Análise Sintática (Parte 2)

Introdução

Após uma introdução à análise sintática e classificação de *parsers* em ascendentes e descendentes nas Notas de Aula sobre Análise Sintática (Parte 1), estas notas de aula seguem abordando a análise sintática apresentando estratégias de análise ascendente.

Análise Sintática Ascendente

Um analisador sintático ascendente (*bottom-up*) constrói a árvore sintática das folhas até a raiz e aceita mais gramáticas que o *top-down*, porém é mais difícil de implementar, sendo necessário o uso de ferramentas.

A cada passo de redução essa análise tenta reconhecer na entrada o lado direito de uma regra de produção (*handle*). O *handle* então é substituído pelo símbolo do lado esquerdo da regra de produção e esse processo é repetido até que toda a entrada seja reconhecida.

Esse tipo de análise é geralmente implementado usando a ***análise empilhar e reduzir***. Ela aceita um conjunto maior de gramáticas que as análises *top-down*. É geralmente usada em ferramentas como o `yacc` e `bison`.

Exemplo:
S → aABe
A → Abc | b

B → d

Entrada: `abbcde` → `aAbcde` → `aAde` → `aABe` → `S`

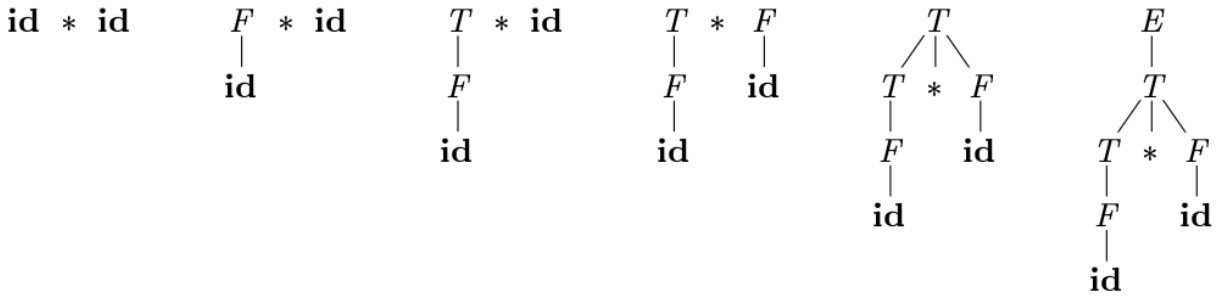


Figura 1. Exemplo de geração de árvore sintática ascendente. Fonte: Aho et al. 2007.

A partir da Figura 1, que exibe um exemplo de geração de árvore sintática derivada de forma ascendente partindo da expressão `id * id`, que regras de produção podemos identificar?
E → T
T → F | T * F
F → id

A gramática completa era:
E → E + T | T
T → T * F | F
F → (E) | id

Análise Empilhar/Reduzir

- Também conhecida como empilha/reduz ou *shift/reduce*.
- Técnica de implementação de analisadores *bottom-up* (também chamados de *ascendentes* ou *redutivos*).
- Nesta análise, usa-se uma pilha e uma entrada.
- A ideia é empilhar símbolos da entrada até que no topo da pilha apareça o *handle* (lado direito de uma regra). O *handle* então é substituído (reduzido) para o símbolo do lado esquerdo da regra de produção.

- As ações na pilha são somente empilhar, reduzir, aceitar ou erro.
- Esse processo é repetido até que a entrada esteja vazia e no topo da pilha esteja o símbolo inicial da gramática analisada.
- O símbolo $ será utilizado como delimitador/terminador. Também pode ser visto na literatura representado por outros símbolos, como λ e ε.

Outro exemplo:

E → E x E

E → E + E

E → (E)

E → id

Suponha que a entrada é `id + id x id`.

Pilha	**Entrada**	**Ação**
$	id + id x id$	empilhar
$ id	+ id x id$	reduzir
$ E	+ id x id$	empilhar
$ E +	id x id$	empilhar
$ E + id	x id$	reduzir
$ E + E	x id$	reduzir
$ E	x id$	empilhar
$ E x	id$	empilhar
$ E x id	$	reduzir
$ E x E	$	reduzir
$ E	$	aceita a entrada

Exercício: faça a análise empilhar/reduzir para

S → [L] | a

L → L, S | S

Entrada: `[a, a]`.

Pilha	Entrada	Ação
$	[a, a]$	empilhar
$[	a, a]$	empilhar
$[a	, a]$	reduzir
$[S	, a]$	reduzir
$[L	, a]$	empilhar
$[L,	a]$	empilhar
$[L, a	]$	reduzir
$[L, S	]$	reduzir
$[L	]$	empilhar
$[L]	$	reduzir
$S	$	aceita a entrada

Análise de Precedência de Operadores

A análise de precedência de operadores também é realizada de forma *bottom-up* e com ações de empilhar e reduzir. *Tokens* possuem relações de precedência definidas em uma tabela, chamada de ***tabela de ações***.

Existem três relações de precedência: <, >, =.

Sendo a e b símbolos terminais,

- a < b: a possui menor precedência que b
- a > b: a possui maior precedência que b
- a = b: a e b possuem a mesma precedência

A tabela indica as relações de precedência entre todos os terminais.

- Os terminais nas linhas representam os terminais no topo da pilha.
- Os terminais na coluna representam os terminais sobre o cabeçote de leitura.

Seja a o terminal mais ao topo da pilha e b o terminal sobre o cabeçote de leitura, então

1) Se a < b ou a = b, empilha
2) Se a > b, reduz

Exemplo:

E → E v E

E → E ^ E

E → id | (E)

Tabela:

Pilha	**Cabeçote**					
	id	**v**	**^**	**(**	**)**	**$**
id	N	>	>	N	>	>
v	<	>	<	<	>	>
^	<	>	>	<	>	>
(	<	<	<	<	=	N
)	N	>	>	N	>	>
$	<	<	<	<	N	Aceita

Observação: Nesta tabela, N significa nada, ou seja, pode ser ignorado. No entanto, se não existir relação de precedência entre o símbolo terminal mais ao topo da pilha e o símbolo da entrada, ocorre um erro sintático.

Verificar a validade da entrada `id v id ^ id`.

Pilha	**Relação**	**Entrada**	**Ação**
$	<	id v id ^ id$	empilhar
$id	>	v id ^ id$	reduzir
$E	<	v id ^ id$	empilhar
$E v	<	id ^ id$	empilhar
$E v id	>	^ id$	reduzir
$E v E	<	id$	empilhar
$E v E ^	<	$	empilhar
$E v E ^ id	>	$	reduzir

$E v E ^ E	>	$	reduzir
$E v E	>	$	reduzir
$E		$	aceita

Exercício:

Verificar a validade da entrada: `id ^ id v id`.

Pilha	**Relação**	**Entrada**	**Ação**
$	<	id ^ id v id$	empilhar
$id	>	^ id v id$	reduzir
$E	<	^ id v id$	empilhar
$E ^	<	id v id$	empilhar
$E ^ id	>	v id$	reduzir
$E ^ E	>	v id$	reduzir
$E	<	v id$	empilhar
$E v	<	id$	empilhar
$E v id	>	$	reduzir
$E v E	>	$	reduzir
$E		$	aceita

Até aqui, as análises foram feitas com base na tabela de precedência de operadores já fornecida. Mas como construir esta tabela?

As regras são as seguintes:

1) Se o operador O1 tiver maior precedência que o operador O2, então
   O1 (na pilha) > O2 (no cabeçote)
   O2 (na pilha) < O1 (no cabeçote)
2) Se O1 e O2 têm igual precedência e são associativos à esquerda, então
   O1 > O2
   O2 > O1
3) Se O1 e O2 têm igual precedência e são associativos à direita, então
   O1 < O2

O2 < O1

4) As relações entre os operadores e os demais *tokens* são pré-definidas. Para todo operador Φ, temos:

Φ < id	Φ < (	Φ >)	Φ > $
id > Φ	(< Φ	) > Φ	$ < Φ

5) As relações entre tokens que não são operadores também são fixas:

(< (	) >)	id >)	$ < (	
(=)	) > $	id > $	$ < id	(< id

Exercício: montar a tabela de precedência de operadores para a seguinte linguagem:
E → E + E | E * E | E ** E | (E) | id

Resposta:

** Tem maior precedência e é associativa à direita[1].

* Tem precedência intermediária e é associativa à esquerda.

+ Tem menor precedência e é associativa

Pilha\Cabeçote	+	*	**	(	)	id	$
+	>	<	<	<	>	<	>
*	>	>	<	<	>	<	>
**	>	>	<	<	>	<	>
(	<	<	<	<	=	<	N
)	>	>	>	N	>	N	>
id	>	>	>	N	>	N	>
$	<	<	<	<	N	<	Aceita

Associatividade: define a ordem de precedência das operações quando não há parênteses. O operador ** é associativo à direita, isto é, x**y**z = x**(y**z).

Por exemplo, 4**2**3 = 4**(2**3). O resultado da operação é 4**8 = 65536. Se o operador fosse associativo à esquerda, (4**2)**3 = 16**3 = 4096, o que representa um valor completamente

diferente do correto.

Analisadores LR(k)

No primeiro texto sobre Análise Sintática, falou-se brevemente sobre alguns tipos de analisadores sintáticos. Nesta seção, são abordados os analisadores LR(*k*) e seus principais tipos.

Analisadores LR(*k*) (*Left to right with rightmost derivation*) são analisadores redutores eficientes que fazem a análise da esquerda para a direita e produzem uma derivação mais à direita considerando *k* símbolos no cabeçote de leitura.

Vantagens:

- São capazes de reconhecer praticamente qualquer estrutura definida por gramáticas livres de contexto.
- É o método mais geral e pode ser implementado com eficiência.
- Descobre erros sintáticos mais cedo.

Desvantagem:

- Usa uma tabela de análise complexa que deve ser gerada usando uma ferramenta.

Tipos:

- SLR (*Simple* LR): mais fácil de implementar, este analisador restringe as gramáticas.
- LR: mais poderoso.
- LALR (*Look Ahead* LR): intermediário, é utilizado, por exemplo, na ferramenta `yacc`.

SLR (Simple LR)

Análise do tipo empilhar/reduzir que usa uma pilha, uma entrada e uma tabela que indica a próxima ação/transição. Alguns autores tratam a tabela de ação/transição como duas tabelas, uma de ação (*action table*) e outra de transição (*goto table*).

Seja E_m o estado no topo da pilha e a_i o *token* sobre o cabeçote de leitura, o analisador

consulta a tabela.

Uma ação $[E_m, a_i] = X$, onde X pode ser

- Empilha. Ex.: empilhar "a_i E_x"
- Reduz N, onde N é o número da produção A→B: causa do desempilhamento de 2r

Símbolos onde r = |B|, e o empilhamento de "A_i E_y", onde E_y resulta da consulta à tabela de transição $[E_{m-r}, A]$

- Aceita
- Erro

Exemplo:

1) E→EvT
2) E→T
3) T→T^F
4) T→F
5) F→(E)
6) F→id

Pilha	**Símbolos terminais**						**Símbolos não terminais**		
	Ação						**Transição**		
	id	**v**	**^**	**(**	**)**	**$**	**E**	**T**	**F**
0	E5			E4			1	2	3
1		E6				AC			
2		R2	E7		R2	R2			
3		R4	R4		R4	R4			
4	E5			E4			8	2	3
5		R6	R6		R6	R6			
6	E5			E4				9	3
7	E5			E4					10

8		E6		E11					
9		R1	E7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Nesta tabela, ***AC*** significa ***aceita***.

Análise empilhar/reduzir para a entrada `id ^ id v id`:

Pilha	**Entrada**	**Ação**	**Comentário**
0	id ^ id v id$	E5	Observa-se a linha 0 e a coluna id, que resulta em E5 (empilhar). Linha 0 porque é o que há no topo da pilha, id porque este é o símbolo atual no cabeçote de leitura. Portanto, empilha-se "id 5".
0 id 5	^ id v id$	R6	Observa-se a linha 5 e a coluna ^, que resulta em R6 (reduzir à regra 6, que é F→id). A linha da próxima ação é buscada pela próxima linha na pilha (0) com o símbolo que substituirá o *handle* (F). 0 com F resulta em 3.
0 F 3	^ id v id$	R4	Observa-se a linha 3 e a coluna ^, que resulta em R4 (reduzir à regra 4, que é T→F). A linha da próxima ação é buscada pela próxima linha na pilha (0) com o símbolo que substituirá o *handle* (T). 0 com T resulta em 2.
0 T 2	^ id v id$	E7	Observa-se a linha 2 e a coluna ^, que resulta em E7 (empilhar). Linha 2 porque é o que há no topo da pilha, ^ porque este é o símbolo atual no cabeçote de leitura. Portanto, empilha-se "^ 7".
0 T 2^7	id v id$	E5	Observa-se a linha 7 e a coluna id, que resulta em E5 (empilhar). Linha 7 porque é o que há no topo da pilha, id porque este é o símbolo atual no cabeçote de leitura. Portanto, empilha-se "id 5".
0 T 2 ^ 7 id 5	v id$	R6	Observa-se a linha 5 e a coluna v, que resulta em R6 (reduzir à regra 6, que é F→id). A linha da próxima ação é buscada pela próxima linha na pilha (7) com o símbolo que substituirá o *handle* (F). 7 com F resulta em 10.
0 T 2 ^ 7 F 10	v id$	R3	Observa-se a linha 10 e a coluna v, que resulta em R3 (reduzir à regra 3, que é T→T^F). A linha da próxima ação é buscada pela próxima linha na pilha (0) com o símbolo que substituirá o *handle* (T). 0 com T resulta em 2.
0 T 2	v id$	R2	Observa-se a linha 2 e a coluna v, que resulta em R2 (reduzir à regra 2, que é E→T). A linha da próxima ação é buscada pela próxima linha na pilha (0) com o símbolo que substituirá o *handle* (E). 0 com T resulta em 1.
0 E 1	v id$	E6	Observa-se a linha 1 e a coluna v, que resulta em E6 (empilhar). Linha 1 porque é o que há no topo da pilha, v porque este é o símbolo atual no cabeçote de leitura. Portanto, empilha-se "v 6".
0 E 1 v 6	id$	E5	Observa-se a linha 6 e a coluna id, que resulta em E5 (empilhar). Linha 6 porque é o que há no topo da pilha, id porque este é o símbolo atual no cabeçote de leitura. Portanto, empilha-se "id 5".

0 E 1 v 6 id 5	$	R6	Observa-se a linha 5 e a coluna $, que resulta em R6 (reduzir à regra 6, que é F→id). A linha da próxima ação é buscada pela próxima linha na pilha (6) com o símbolo que substituirá o *handle* (F). 6 com F resulta em 3.
0 E 1 v 6 F 3	$	R4	Observa-se a linha 3 e a coluna $, que resulta em R4 (reduzir à regra 4, que é T→F). A linha da próxima ação é buscada pela próxima linha na pilha (6) com o símbolo que substituirá o *handle* (T). 6 com T resulta em 9.
0 E 1 v 6 T 9	$	R1	Observa-se a linha 9 e a coluna $, que resulta em R1 (reduzir à regra 1, que é E→EvT). A linha da próxima ação é buscada pela próxima linha na pilha (0) com o símbolo que substituirá o *handle* (E). 0 com E resulta em 1.
0 E 1	$	Aceita	Observa-se a linha 1 e a coluna $, que resulta em AC (aceita a entrada).

Exercício: Dadas as regras a seguir e a tabela, reproduza a análise empilhar/reduzir para a entrada "`bab`".

1) S -> AA
2) A -> aA
3) A -> b

Pilha	**Símbolos terminais**			**Símbolos não terminais**	
	Ação			**Transição**	
	a	**b**	**$**	**S**	**A**
0	E3	E4		1	2
1			AC		
2	E3	E4			5
3	E3	E4			6
4	R3	R3	R3	8	
5		R6	R1		
6	R2	R2	R2		

Resposta:

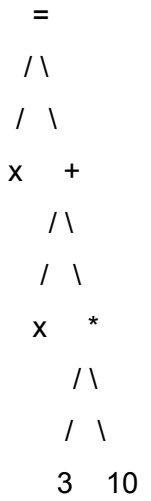
Pilha	**Entrada**	**Ação**
0	bab$	E4
0 b4	ab$	R3
0 A2	ab$	E3

0 A2 a3	b$	E4
0 A2 a3 b4	$	R3
0 A2 a3 A6	$	R2
0 A2 A5	$	R1
0 S1	$	Aceita

Observação: alguns autores iniciam a contagem de linhas para fins de geração de sequência de regras e de tabela de ação/transição iniciando por 0.

Geração da Árvore Sintática

Nesta etapa, o compilador gera um grafo conexo e acíclico, ou seja, uma árvore sintática abstrata. A árvore sintática abstrata representa as estruturas sintáticas de cadeias de símbolos do programa de acordo com a sua gramática formal. Todos os nós são valorados sem derivações.



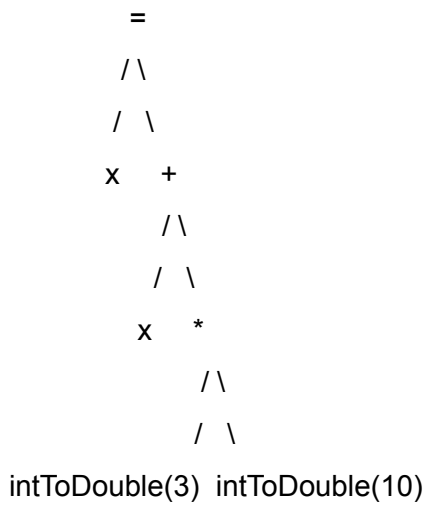
O passo seguinte é verificar se a árvore faz sentido, podendo inclusive aplicar correções. Isso é feito através da análise semântica. Isso envolve fazer algumas verificações como:

- `x` tem tipo correto?
- `x` é `double`?
- `x` é `float`?
- `x` é `int`?
- `x` é `unsigned int`?

- `x` é `string`?

Nesta etapa também é possível ocorrerem correções. `3` pode virar `3.0` e `10` pode virar `10.0`, caso `x` seja uma variável do tipo numérica de ponto flutuante.

Dessa forma, a árvore resultante seria:



A partir deste ponto, considera-se pronta a árvore sintática abstrata. Ela também é utilizada na próxima etapa, a geração de código intermediário. Após a geração, ou mesmo durante esta geração, otimizações podem ser realizadas. Em seguida, faz-se a geração do código alvo.

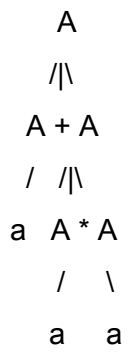
```
LOAD ...
MOV  ...
MULT ...
(...)
```

Voltando ao exemplo de gramáticas ambíguas, com os conhecimentos sobre a representação sintática em árvore, torna-se fácil compreender que as árvores de gramáticas ambíguas são diferentes.

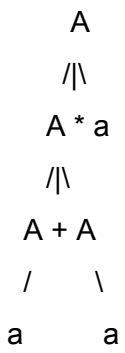
1) $A \rightarrow A + A$
   $\rightarrow a + A$
   $\rightarrow a + A * A$

2) $A \rightarrow A * A$
   $\rightarrow A * a$
   $\rightarrow A + A * a$

→ a + a * A
→ a + a * a



→ a + A * a
→ a + a * a



Gramáticas LL e LR não podem ser ambíguas.

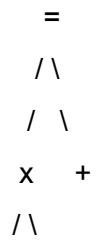
Agora, com uma breve introdução sobre o funcionamento de um compilador, considere a instrução abaixo e responda:

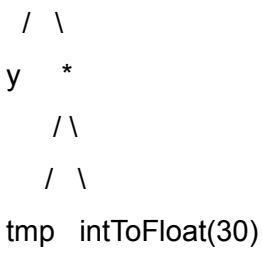
```
x = y + tmp * 30
```

1) Quais são os *tokens* da expressão?

`x`	<identificador, x>
`=`	<operador, =>
`y`	<identificador, y>
`+`	<operador, +>
`tmp`	<identificador, tmp>
`*`	<operador, *>
`30`	<valor_numérico, 30>

2) Como fica a árvore sintática considerando que as variáveis foram declaradas como float?





Há diversos tipos de *parsers* e a apresentação de todos demandaria uma disciplina completa sobre isso. Para expandir os conhecimentos sobre os tipos de *parsers* existentes, recomenda-se a leitura do capítulo 4 do livro de Aho *et al*. (2007).

Ferramentas geradoras de analisadores sintáticos:

- `antlr`
- `happy`
- `pegen`
- `yacc`
- `javacc`
- `bison`

PARA PENSAR

Supondo que exista uma linguagem X e seu compilador ABC v. 1.0. Está sendo construído o compilador ABC v. 1.1, dispondo de novas funcionalidades e de estruturas de otimização, gerando um código alvo mais rápido.

Como o código do novo compilador será compilado?

Conclusão

Analisadores sintáticos ascendentes, como o de empilhar e reduzir (*shift/reduce*), constroem a árvore sintática das folhas para a raiz, aceitando um conjunto mais amplo de gramáticas, embora sejam mais complexos de implementar.

A análise de precedência de operadores é a análise em que os *tokens* possuem relações de precedência definidas em uma tabela de ações. Os exemplos vistos permitem demonstrar a verificação da validade de entradas específicas utilizando esta tabela, além de mostrar como construir uma tabela de precedência de operadores para diferentes gramáticas.

As notas de aula também exploraram os analisadores LR(k), que são eficientes para reconhecer estruturas definidas por GLC, com destaque para os tipos SLR, LR e LALR, cada um com suas vantagens e complexidades. Sobre este tópico, exemplos práticos demonstraram o uso das tabelas de ação e transição para a análise de entradas específicas.

Por fim, abordou-se a geração da árvore sintática e a análise semântica subsequente, verificando a correção e aplicando possíveis correções. A transformação de expressões em árvores sintáticas abstratas demonstra como a análise semântica garante a coerência dos tipos de dados antes da geração de código intermediário e otimizações.

Exercícios

1) A estratégia de análise sintática por empilhar/reduzir é baseada na técnica de reconhecimento de sentenças por construção ascendente (*bottom-up*). Nessa estratégia, símbolos terminais da sentenças são lidos um a um; a cada símbolo lido, o analisador decide se prossegue com a leitura (empilha) ou se é possível aplicar uma produção aos símbolos previamente lidos para substituí-los por um símbolo não terminal da gramática (reduz). O procedimento conclui com sucesso se toda a sentença foi lida e apenas o símbolo sentencial resulta da aplicação de todas as reduções. Nesse sentido, analise as afirmações a seguir sobre analisadores de precedência de operadores e marque-as com V ou F:
    a) Seja x o terminal mais ao topo da pilha e y o primeiro terminal da cadeia de entrada sendo analisada. Se x > y, então procura-se o lado direito do *handle* na pilha e o substitui pelo seu lado esquerdo.
    b) Para identificar os *handles* (substituições), são utilizadas as relações de precedência existentes entre os símbolos terminais (operandos e operadores) e não terminais (variáveis) em uma tabela sintática ou de precedência.

c) Na análise de precedência de operadores, existem dois momentos nos quais o analisador pode descobrir erros sintáticos. Um deles pode ocorrer na consulta à matriz de precedência, caso não exista relação de precedência entre o terminal mais ao topo da pilha e o símbolo da entrada. A outra possibilidade pode ocorrer quando o analisador supõe a existência de um *handle* no topo da pilha, mas não existe produção com o lado direito correspondente.

2) Assumindo que a expressão `resposta = 9 * (8 - (4 ^ 2)) / 3` foi utilizado em um código para a linguagem Python:
    a) Desenhe a árvore sintática para a expressão.
    b) Qual é o resultado da expressão?

Referências

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

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.

DU BOIS, André Rauber. Notas de Aula sobre Compiladores. 13 out. 2011.