CTC-41

Compiladores

Carlos Alberto Alonso Sanches

CTC-41

5) Análise sintática

Análise sintática ascendente (*bottom-up*) ou redutiva

Parser SLR(1)

A ferramenta *Yacc*

Análise sintática ascendente

- Na análise sintática ascendente (*bottom-up*):
  - A árvore de derivação é construída a partir dos *tokens*, (que são as folhas), chegando até a raiz (que é o símbolo inicial da gramática).
  - Em cada passo desta estratégia, um lado direito de uma produção (chamado de *handle*) é substituído por um não-terminal. Esta substituição recebe o nome de *redução*, e por isso a análise sintática ascendente também é chamada de redutiva.
  - Consequentemente, o reconhecimento é feito através da derivação mais à direita em ordem reversa.
- A sua implementação é realizada através de um autômato de pilha com o controle dirigido por uma tabela de análise.

Implementação de um *parser* redutivo

- Inicialmente, a fita de entrada do autômato contém a sentença a ser analisada seguida do símbolo $ (marcador de fim), e a pilha contém apenas $.
- O processo de reconhecimento consiste em transferir símbolos da fita de entrada para a pilha (operações *shift*), até que se tenha na pilha um lado direito de produção. Quando isso ocorre, este lado direito é substituído pelo lado esquerdo desta produção (operação *reduce*).
- Por isso, este *parser* também é chamado de *shift-reduce*.
- Para garantir um único estado inicial do autômato, geralmente é utilizada uma gramática aumentada, com a produção inicial $S' \rightarrow S$.
- No final da entrada, se a pilha contiver apenas $ S', a sentença analisada é aceita.

Exemplo 1

- Considere a gramática aumentada abaixo, para operações de adição:

  E′ → E
  E → E + n | n

- Ações do *parser* redutivo para a entrada **n + n** :

Passos	Pilha	Entrada	Ações
1	**$**	**n + n $**	**shift**
2	**$ n**	**+ n $**	**reduce E → n**
3	**$ E**	**+ n $**	**shift**
4	**$ E +**	**n $**	**shift**
5	**$ E + n**	**$**	**reduce E → E + n**
6	**$ E**	**$**	**reduce E′ → E**
7	**$ E′**	**$**	aceitação

- Os passos 3 e 6, embora tenham **E** no topo da pilha, possuem ações distintas. Motivo: diferentes *tokens* na entrada.
- *Parser* acompanha a derivação mais à direita da cadeia de entrada, embora os **reduce** ocorram na ordem reversa: **E′ ⇒ E ⇒ E + n ⇒ n + n**
- Quando a entrada é aceita, o conteúdo da pilha seguido do conteúdo da entrada sempre corresponde a uma forma sentencial da gramática.

Exemplo 2

- Considere a gramática aumentada abaixo, para listas:

```
S' → S
S  → [ L ] | a
L  → L ; S | S
```

- Ações do *parser* redutivo para a entrada `[ a ; a ]` :

Pilha	Entrada	Ações
$	[a ; a] $	shift
$ [	a ; a] $	shift
$ [a	; a] $	reduce S → a
$ [S	; a] $	reduce L → S
$ [L	; a] $	shift
$ [L ;	a] $	shift

Pilha	Entrada	Ações
$ [L ; a	] $	reduce S → a
$ [L ; S	] $	reduce L → L ; S
$ [L	] $	shift
$ [L]	$	reduce S → [L]
$ S	$	reduce S' → S
$ S'	$	aceitação

- Derivação mais à direita correspondente à análise:

S' ⇒ S ⇒ [L] ⇒ [L ; S] ⇒ [L ; a] ⇒ [S ; a] ⇒ [a ; a]

Parser LR(1)

- Um caso importante de *parser* redutivo é o LR(1): a entrada é processada da esquerda para a direita (**L**), o reconhecimento é através da derivação mais à direita (**R**), e é utilizado um único *token* à frente (**1**).
- Um ponto fundamental é a ação a ser executada, definida através de consulta a uma tabela de análise. Para isso, cada elemento da pilha será um par (X_i, E_j), onde X_i é um símbolo da gramática e E_j é um estado do autômato.
- Sejam E_m o estado no topo da pilha e a_i o próximo *token*. O *parser* LR(1) consulta a posição $[E_m, a_i]$ da tabela de análise e realiza determinadas ações de acordo com o seu conteúdo:
  - **s** E_x: empilha o par (a_i, E_x).
  - **r n**, onde n identifica a produção $A \rightarrow \alpha$: desempilha $|\alpha|$ pares e empilha o par $(A, [E_m, A])$, onde $[E_m, A]$ é uma posição desta mesma tabela.
  - **acpt**: a cadeia de entrada é reconhecida como válida.
  - **error**: é identificado um erro sintático.

Exemplo de tabela LR(1)

Gramática:

1) `E → E + T`
2) `E → T`
3) `T → T * F`
4) `T → F`
5) `F → ( E )`
6) `F → id`

Para terminais e `$`: **ações** (*shift*, *reduce*, *acpt*)

Espaços vagos: erros

Para não-terminais: **transições de estados**

Estados	Ação						Transição		
	`id`	`+`	`*`	`(`	`)`	`$`	`E`	`T`	`F`
0	s 5			s 4			1	2	3
1		s 6				acpt			
2		r 2	s 7		r 2	r 2			
3		r 4	r 4		r 4	r 4			
4	s 5			s 4			8	2	3
5		r 6	r 6		r 6	r 6			
6	s 5			s 4				9	3
7	s 5			s 4					10
8		s 6			s 11				
9		r 1	s 7		r 1	r 1			
10		r 3	r 3		r 3	r 3			
11		r 5	r 5		r 5	r 5			

Tabelas de análise SLR(1)

- Veremos a construção do caso mais básico de tabelas LR(1), chamado de SLR(1) ou LR(1) simples.
- As gramáticas reconhecidas por tabelas SLR(1) não são ambíguas.
- No entanto, há várias gramáticas não ambíguas que não podem ser reconhecidas por tabelas SLR(1), exigindo um método mais sofisticado, conhecido como LALR(1), que é o mais utilizado nos geradores de *parsers*.
- Fases da construção da tabela SLR(1) de uma gramática *G*:
  - Embora não seja sempre necessário, incluir uma nova produção $S' \rightarrow S$, onde S é o símbolo inicial de *G*, para indicar ao *parser* o momento de terminar a análise e aceitar a entrada;
  - Definir a coleção canônica *C* de itens LR(0) de *G*;
  - Com as funções auxiliares Fecho e IrPara, definir os estados e as transições de um AFD;
  - A partir deste AFD, construir a tabela SLR(1).

Itens LR(0)

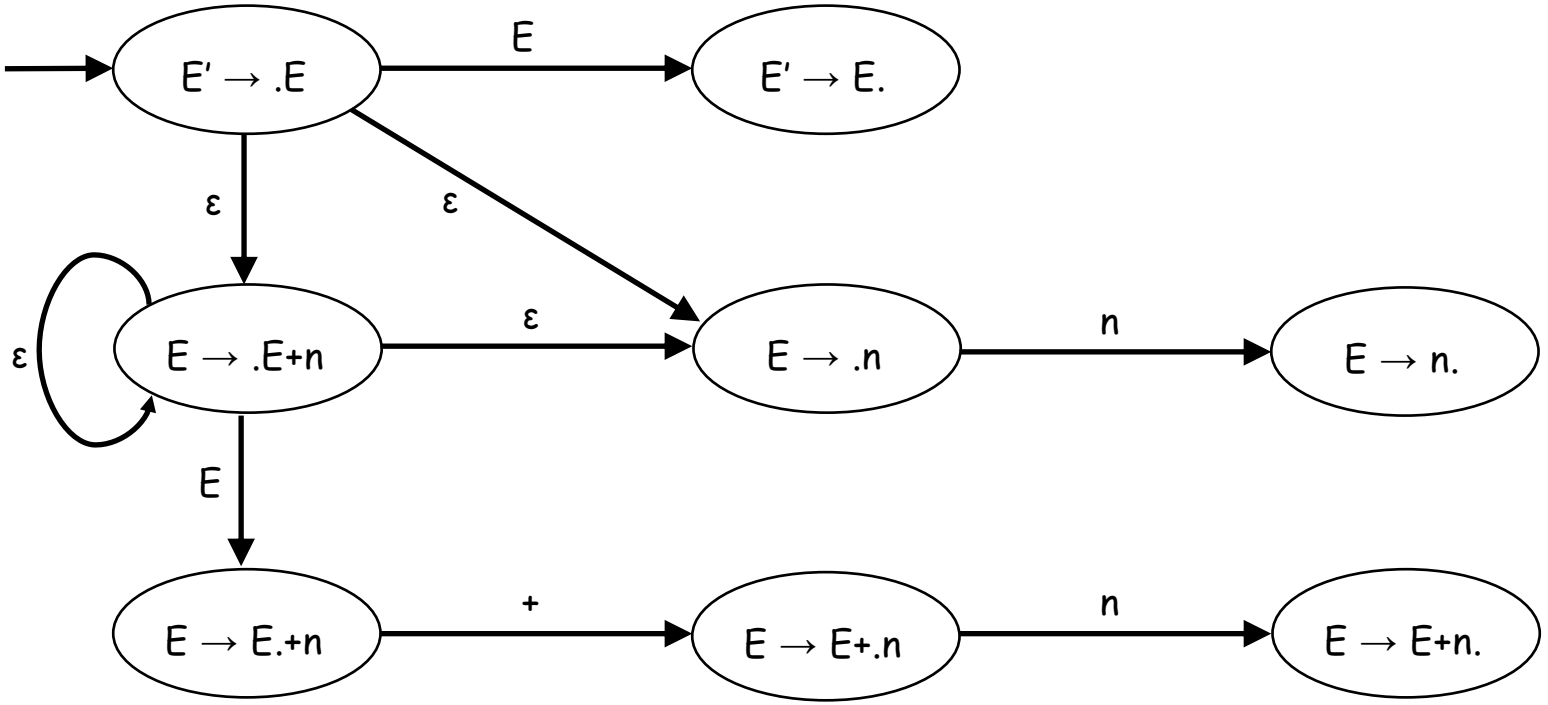
- Um item LR(0) para uma gramática *G* é uma produção de *G* com um ponto em alguma das suas posições no lado direito.
- Por exemplo, a produção $A \rightarrow XYZ$ tem quatro itens:
  - $A \rightarrow .XYZ$
  - $A \rightarrow X.YZ$
  - $A \rightarrow XY.Z$
  - $A \rightarrow XYZ.$
- A produção $A \rightarrow \varepsilon$ tem apenas o item $A \rightarrow .$
- Um item LR(0) indica a parcela já reconhecida de uma produção em um determinado momento da sua análise sintática.
- Considerando cada item LR(0) como um estado, é possível construir um AFND com transições a partir de símbolos da gramática e ε-transições, como veremos a seguir.

Exemplo

- Considere a gramática aumentada abaixo, para operações de adição:

```
E' → E
E  → E + n | n
```

- Os estados do AFND são obtidos a partir dos itens LR(0).
- As transições deste AFND são obtidas com símbolos ou ε-transições.

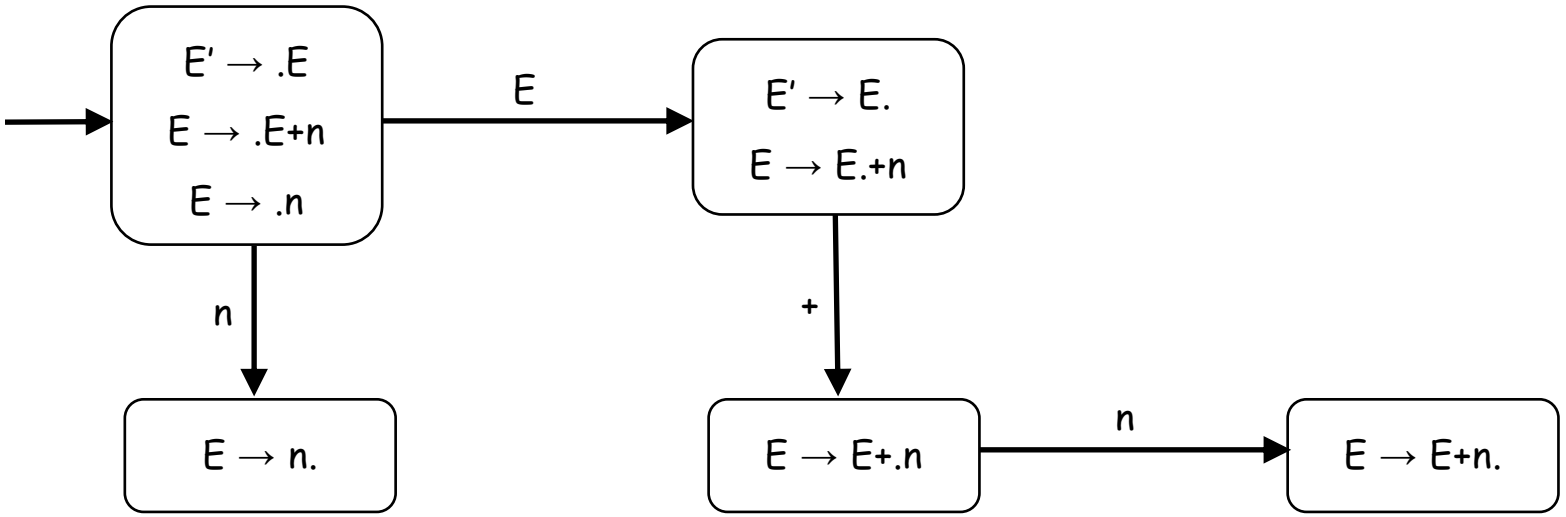


Exemplo (continuação)

- Considere a mesma gramática anterior:

```
E'  →  E
E   →  E + n  |  n
```

- Podemos encontrar um AFD equivalente ao AFND anterior:



- A seguir, veremos como é possível obter este AFD diretamente, com o uso das funções Fecho e IrPara.

Função Fecho

- Seja I um conjunto de itens LR(0) para uma gramática G.
- A função Fecho(I) é definida da seguinte forma:
  - Todo item de I pertence a Fecho(I).
  - Se $A \rightarrow \alpha.X\beta$ está em Fecho(I) e $X \rightarrow \gamma$ é produção de G, então adicione $X \rightarrow .\gamma$ a Fecho(I).
- Como exemplo, considere a gramática abaixo:
  - `S → a | [ L ]`
    `L → L ; S | S`
  - Itens LR(0) = {`S→.a, S→a., S→.[L], S→[.L], S→[L.], S→[L].,`
    `L→.L;S, L→L.;S, L→L;.S, L→L;S., L→.S, L→S.`}
  - Fecho({`S→[.L]`}) = {`S→[.L], L→.L;S, L→.S, S→.a, S→.[L]`}
  - Fecho({`L→.L;S`}) = {`L→.L;S, L→.S, S→.a, S→.[L]`}
  - Fecho({`L→L;.S`}) = {`L→L;.S, S→.a, S→.[L]`}
  - Fecho({`L→.S`}) = {`L→.S, S→.a, S→.[L]`}
  - Para os demais itens, o fecho contém apenas o próprio item.
- Esta função será utilizada para definir os estados do AFD.

Função IrPara

- Seja I um conjunto de itens LR(0) para uma gramática G, e X um símbolo de G.
- A função IrPara(I,X) é definida como o fecho de todos os itens $A \rightarrow \alpha X.\beta$ tais que $A \rightarrow \alpha.X\beta$ estejam em I.
- Como exemplo, considere a mesma gramática anterior:
  - `S → a | [ L ]`
    `L → L ; S | S`
  - IrPara({`S→.a`}, `a`) = {`S→a.`}
  - IrPara({`S→a.`}, `[`) = ϕ
  - IrPara({`S→.[L]`}, `[`) = {`S→[.L]`, `L→.L;S`, `L→.S`, `S→.a`, `S→.[L]`}
  - IrPara({`S→[L.]`, `L→L.;S`}, `;`) = {`L→L;.S`, `S→.a`, `S→.[L]`}
  - IrPara({`S→[L.]`, `L→L.;S`}, `]`) = {`S→[L].`}
  - E assim por diante...
- Esta função será utilizada para definir as transições do AFD a partir de cada símbolo X.

Coleção canônica de itens LR(0)

- Seja uma gramática aumentada *G*, com símbolo inicial *S* e produção acrescentada $S' \rightarrow S$.
- Algoritmo para obtenção da sua coleção canônica *C* de itens LR(0):

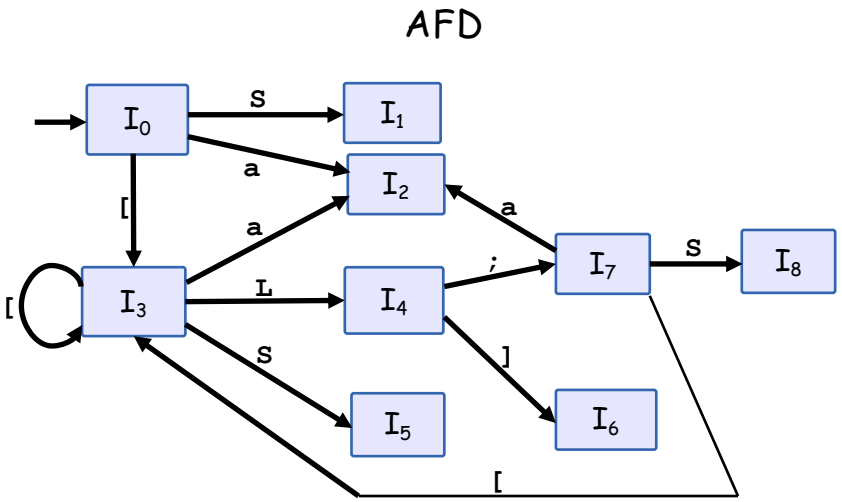
```
ColeçãoCanônica(G) {
    C = {I0 = Fecho({S' → .S})};
    repeat
        for each (I ∈ C and X ∈ G) do
                if (IrPara(I,X) ≠ φ)
                        C = C U IrPara(I,X);
    until não haja mais conjuntos para serem incluídos em C;
    return C;
}
```

Exemplo

- Considere a gramática aumentada abaixo, para listas:

```
S'  →  S
S   →  [ L ]  |  a
L   →  L ; S  |  S
```

- Conjuntos de itens LR(0) da coleção C:
  - I_0 = Fecho({S' → .S}) = {S' → .S, S → .a, S → .[L]}
  - IrPara(I_0, S) = {S' → S.} = I_1
  - IrPara(I_0, a) = {S → a.} = I_2
  - IrPara(I_0, [) = {S → [.L], L → .L;S, L → .S, S → .a, S → .[L]} = I_3
  - IrPara(I_3, L) = {S → [L.], L → L.;S} = I_4
  - IrPara(I_3, S) = {L → S.} = I_5
  - IrPara(I_3, a) = I_2
  - IrPara(I_3, [) = I_3
  - IrPara(I_4,]) = {S → [L].} = I_6
  - IrPara(I_4, ;) = {L → L;.S, S → .a, S → .[L]} = I_7
  - IrPara(I_7, S) = {L → L;S.} = I_8
  - IrPara(I_7, a) = I_2
  - IrPara(I_7, [) = I_3



Construção da tabela SLR(1)

- Seja $C = \{I_0, I_1, ..., I_n\}$ a coleção canônica de G.
- As linhas da tabela, que representam os estados, serão indexadas de 0 a n, onde 0 é o estado inicial.
- O conteúdo da linha i é construído a partir do conjunto I_i:
  - Regras para as ações (coluna é indexada por um terminal ou por $):
    - Se IrPara(I_i, a) = I_j, então Ação[i,a] = "s j"
    - Se $A \rightarrow \alpha.$ está em I_i, então, para todo a em Follow(A), Ação[i,a] = "r n", onde n identifica a produção $A \rightarrow \alpha$
    - Se $S' \rightarrow S.$ está em I_i, então Ação[i,$] = "acpt"
  - Regras para as transições (coluna é indexada por um não-terminal):
    - Se IrPara(I_i, A) = I_j, então Transição[i,A] = "j"
- Ações não definidas na tabela correspondem a situações de erro.
- Se houver situações conflitantes na tabela, então a gramática G não é SLR(1).

Exemplo 1

Gramática equivalente à aumentada:

1) S → E
2) E → E + T
3) E → E - T
4) E → T
5) T → T * F
6) T → F
7) F → n
8) F → (E)

	Ação							Transição			
	n	(	+	-	*	)	$	S	E	T	F
0	s 5	s 6						1	2	3	4
1							acpt				
2	r 1	r 1	s 7	s 8	r 1	r 1	r 1				
3	r 4	r 4	r 4	r 4	s 9	r 4	r 4				
4	r 6	r 6	r 6	r 6	r 6	r 6	r 6				
5	r 7	r 7	r 7	r 7	r 7	r 7	r 7				
6	s 5	s 6							10	3	4
7	s 5	s 6								11	4
8	s 5	s 6								12	4
9	s 5	s6									13
10			s 7	s 8		s 14					
11	r 2	r 2	r 2	r 2	s 9	r 2	r 2				
12	r 3	r 3	r 3	r 3	s 9	r 3	r 3				
13	r 5	r 5	r 5	r 5	r 5	r 5	r 5				
14	r 8	r 8	r 8	r 8	r 8	r 8	r 8				

Exemplo 2

Gramática
(não aumentada):

1) **E → E + n**
2) **E → n**

	Ação			Transição
	n	**+**	**$**	**E**
0	s 2			1
1		s 3	acpt	
2		r 2	r 2	
3	s 4			
4		r 1	r 1	

Exercício: com esta tabela, simule a análise sintática de **n + n + n**

Ambiguidades

- Se houver ambiguidade entre **shift** e **reduce** numa mesma posição da tabela, geralmente **shift** é escolhido.
- Se houver ambiguidade entre **reduce** e **reduce**, o melhor será reelaborar a gramática.
- Exemplo de gramática ambígua:
  1) **S → I**
  2) **S → outro**
  3) **I → if S**
  4) **I → if S else S**

Neste caso, dar preferência ao **shift** significa empilhar o **else** para que fique aninhado com o último **if**.

	Ação				Transição	
	if	**else**	**outro**	**$**	**S**	**I**
0	s 4		s 3		1	2
1				acpt		
2		r 1		r 1		
3		r 2		r 2		
4	s 4		s 3		5	2
5		s 6		r 3		
6	s 4		s 3		7	2
7		r 4		r 4		

Recuperação de erros

- Um erro é detectado quando houver acesso a uma **ação** vazia da tabela: não há detecção de erro nas posições de **transição**.
- Não é preciso se preocupar com as reduções: sempre estarão corretas. Por isso, o *parser* SLR(1) pode realizar várias reduções antes de anunciar um erro, mas nunca empilhará um *token* incorreto.
- Recuperação em "modo pânico":
  - Limpar a pilha até encontrar um estado s com transição para um determinado não-terminal A. Pode haver mais de uma escolha para A, que geralmente representa trechos maiores do programa: expressões, declarações, blocos, etc.
  - Descartar a entrada até encontrar um *token* que possa seguir A.
  - Empilhar o par (A, [s,A]) e continuar a análise.
- Recuperação em nível de frase:
  - De acordo com a posição da tabela em que o erro é detectado, e com base no uso da linguagem e na propensão de erro nessa situação, construir um procedimento de recuperação específico.

Parser LALR(1)

- A maioria dos compiladores utiliza um *parser* LALR(1), que possui as seguintes características básicas:
  - Dada uma gramática, são encontrados seus itens LR(1). Cada item LR(1) corresponde a um item LR(0), que é o seu núcleo, e um *token* de verificação à frente (*lookahead* = 1).
  - É construído um AFD cujos estados são os itens LR(1) acima.
  - Este AFD é simplificado: os estados com os mesmos núcleos LR(0) são agrupados, e as suas verificações são combinadas adiante.
- Esta técnica é utilizada amplamente porque atende a praticamente todas as linguagens de programação e pode ser implementada de forma eficiente.

A ferramenta *Yacc*

- A ferramenta *Yacc* (*Yet another compiler-compiler*) é um conhecido gerador de *parsers* baseado na análise LALR(1).
- Entre as diversas implementações abertas e gratuitas, a versão *Bison* é disponibilizada pelo *GNU Project* (www.gnu.org), da *Free Software Foundation*.
- https://www.gnu.org/software/bison/manual/bison.html
- *Bison* gera uma rotina em código C chamada **`yyparse()`**, que faz a análise sintática de um arquivo de entrada, retornando 0 quando não são encontrados erros, ou 1 em caso contrário.
- *Bison* e *Flex* foram desenvolvidos para serem integrados em um mesmo compilador.

Formato do arquivo de entrada

- Um arquivo de entrada do *Bison* tem o seguinte formato:

```
{ definições }
%%
{ regras }
%%
{ rotinas auxiliares }
```

- A seção de definições, que é opcional, contém informações sobre os *tokens*, os tipos de dados e o código C, delimitado entre `%{` e `%}`, que deverá ser inserido no início do arquivo de saída. Quando essa seção não existir, o primeiro `%%` pode ser omitido.
- Na seção de regras, há as produções gramaticas em formato BNF, seguidas pelos códigos em C a serem executados em caso de reconhecimento.
- Na terceira seção, que também é opcional, podem estar os códigos em C de rotinas auxiliares e do programa principal. Quando essa seção não existir, o segundo `%%` pode ser omitido.

Algumas notações utilizadas

- Produções gramaticais:
  - Formato BNF: **`nãoterminal: expansão1 | expansão2 ;`**
  - Símbolo inicial: através da declaração **`%start`**
    - Na sua ausência, o lado esquerdo da primeira produção será o símbolo inicial
  - Produções recursivas: é preferível recursão à esquerda
- *Tokens* :
  - Nomes simbólicos declarados com **`%token`**
    - *Yacc* atribuirá automaticamente um valor numérico com **`#define`**
  - Representação direta com um único caractere entre aspas simples
  - Operadores:
    - **`%left`**: indica associatividade à esquerda
    - **`%right`**: indica associatividade à direita
    - **`%nonassoc`**: indica que operador é não-associativo
    - Operadores declarados na mesma linha têm mesma precedência
    - Operadores declarados na última linhas têm maior precedência

Exemplo

- Consideremos a gramática abaixo:
  - `E → E + E`
    `E → E * E`
    `E → ( E )`
    `E → VALOR`
- Os *tokens* podem ser declarados da seguinte maneira:
  - `%start` E
  - `%token` VALOR ABRPAR FECPAR
  - `%left` SOMA
  - `%left` MULT
- Significado das precedências:
  - `%left` SOMA — A + B + C é interpretado como (A + B) + C
  - `%right` SOMA — A + B + C é interpretado como A + (B + C)
  - `%nonassoc` SOMA — A + B + C é considerado incorreto

Variáveis e pseudovariáveis

- Quando uma produção gramatical é reconhecida, cada símbolo desta produção possui um valor, que são preservados pelo *Yacc* numa pilha mantida em paralelo com a pilha de análise sintática.
- Cada um desses valores pode ser referenciado com pseudovariáveis, cujos nomes começam com **`$`**:
  - **`$$`**: valor do não-terminal que acabou de ser reconhecido;
  - **`$1`**, **`$2`**, **`$3`**, ...: valores de cada símbolo do lado direita da produção reconhecida (da esquerda para a direita).
- *Yacc* assume que, cada vez que um *token* é reconhecido, o *scanner* atribui seu valor à variável **`yylval`**.
- O tipo padrão desses valores é **`int`**, mas pode ser alterado com a redefinição da *string* **`YYSTYPE`**.
- Nomes dos procedimentos: **`yyparse`** (*parser*), **`yylex`** (*scanner*), **`yyerror`** (tratamento de erros).

Exemplo: uma calculadora

```
%{
#include <stdio.h>
#include <ctype.h>
%}

%token NUMBER

%%

command : exp  {printf("%d\n", $1);}
         /* imprime o resultado */
        ;

exp     : exp '+' term      {$$ = $1 + $3;}
        | exp '-' term      {$$ = $1 - $3;}
        | term              {$$ = $1;}
        ;

term    : term '*' factor   {$$ = $1 * $3;}
        | factor            {$$ = $1;}
        ;

factor  : NUMBER            {$$ = $1;}
        | '(' exp ')'       {$$ = $2;}
        ;
```

```
%%

main ()
{ return yyparse(); }

int yylex (void)
{ int c;
  while ((c = getchar()) == ' ');
     /* elimina espaços em branco */
  if (isdigit (c)) {
     ungetc (c, stdin);
     scanf("%d", &yylval);
     return (NUMBER);
  }
  if (c == '\n') return 0;
     /* interrompe a análise sintática */
  return (c);
}

int yyerror (char *s)
{ fprintf(stderr, "%s\n", s);
  /* imprime mensagem de erro */
  return 0;
}
```

Exemplo: uma calculadora

- Considerando o exemplo anterior, as regras abaixo utilizam **`%left`**, produzindo o mesmo resultado:

```
%{
#include <stdio.h>
#include <ctype.h>
%}

%token NUMBER
%left '+' '-' '*'

%%

command : exp                {printf("%d\n", $1);}
        ;

exp     : NUMBER             {$$ = $1;}
        | exp '+' exp        {$$ = $1 + $3;}
        | exp '-' exp        {$$ = $1 - $3;}
        | exp '*' exp        {$$ = $1 * $3;}
        | '(' exp ')'        {$$ = $2;}
        ;

%%
```

Recuperação de erros no *Yacc*

- Modo pânico do *Yacc* : ao detectar um erro, permanece neste estado até encontrar uma sequência de três *tokens* legais consecutivos.
- Uma alternativa é utilizar produções gramaticais com um *pseudotoken* **`erro`** no seu lado direito.
- Uma produção deste tipo define um contexto para que os *tokens* errôneos sejam consumidos, até se encontrar *tokens* apropriados para sincronização.

Eliminação de ambiguidades no *Yacc*

- *Yacc* segue algumas regras para eliminar ambiguidades:
  - entre reduzir e empilhar, prefere empilhar;
  - entre duas possíveis reduções, escolhe a produção especificada antes no arquivo de entrada.
- Também podem ser utilizados os mecanismos para especificação de precedência e associatividade de operadores, já mencionados:
  - **`%left`**, **`%right`**, **`%nonassoc`**

Integração *Yacc/Lex*

- Opções para a linha de comando do *Bison* :
  - **-d**: produz arquivo com as definições de valores para os nomes simbólicos associados aos *tokens*
  - **-v**: produz arquivo com informações sobre a análise sintática (útil para resolução de ambiguidades)
  - **-g**: produz arquivo descrevendo os estados do AFD
- Considere os seguintes arquivos de entrada:
  - **file.y** para o *Bison*
  - **file.l** para o *Flex*
- Exemplos de comandos:
  - **flex file.l**
  - **gcc -c lex.yy.c**
  - **bison -d file.y**
  - **gcc -o output lex.yy.o file.tab.c -ly -lfl**