Universidade da Beira Interior

Desenho de Linguagens de Programação e de Compiladores

Simão Melo de Sousa

Aula 6 - Análise Sintáctica Ascendente

lembrete da aula anterior

- noção de gramática
  - derivação
  - gramática ambígua
- análise descendente
  - suporta-se numa tabela que indica que expansão escolher
  - cálculo de null, first, follow
    - por cálculo de ponto fixo
  - gramática LL(1)

$$
\begin{array}{rcl}
E & \rightarrow & E + E \\
 & | & E * E \\
 & | & (E) \\
 & | & \texttt{int}
\end{array}
$$

análise ascendente

a ideia é sempre ler a entrada da esquerda para a direita, mas agora procuramos reconhecer os membros direitos das produções para construir a árvore de derivação de baixo para cima (*bottom-up parsing*, em inglês)

corresponde em descobrir a derivação direita, começando do fim e acabando no axioma

a análise trabalha com uma pilha que é uma palavra de $(T \cup N)^\star$

em cada instante, duas ações são possíveis

- operação de **leitura** (*shift* em inglês, nesta análise) : lê-se um terminal da entrada e colocamo-lo na pilha
- operação de **redução** (*reduce* em inglês) : reconhecemos no topo da pilha a parte direita β de uma produção $X \to \beta$, e substituimos β na pilha por X (no topo da pilha)

no estado inicial a pilha encontra-se vazia

quando não há mais acções possíveis, a entrada é reconhecida se esta foi totalmente lida e a pilha encontra-se reduzida a S

$$
\begin{array}{lcl}
E & \rightarrow & E + T \\
 & | & T \\
T & \rightarrow & T * F \\
 & | & F \\
F & \rightarrow & (E) \\
 & | & \texttt{int}
\end{array}
$$

pilha	entrada	ação
ϵ	`int+int*int`	leitura
`int`	`+int*int`	redução $F \rightarrow$ `int`
F	`+int*int`	redução $T \rightarrow F$
T	`+int*int`	redução $E \rightarrow T$
E	`+int*int`	leitura
E`+`	`int*int`	leitura
E`+int`	`*int`	redução $F \rightarrow$ `int`
$E+F$	`*int`	redução $T \rightarrow F$
$E+T$	`*int`	leitura
$E+T$`*`	`int`	leitura
$E+T*$`int`		redução $F \rightarrow$ `int`
$E+T*F$		redução $T \rightarrow T*F$
$E+T$		redução $E \rightarrow E+T$
E		sucesso

em cada momento do processo, como decidir entre leitura / redução ?

servindo-se de um autómato finito e examinar em antemão os k primeiros caracteres da entrada ; é a análise LR(k)

(LR significa « **L**eft to right scanning, **R**ightmost derivation »)

na prática $k = 1$
i.e. examinamos de antemão só o primeiro caractere da entrada

a pilha é da forma

$$s_0\, x_1\, s_1\, \ldots\, x_n\, s_n$$

onde s_i é um estado do autómato e $x_i \in T \cup N$ como anteriormente

seja a o primeiro caractere da entrada ; examinamos a transição do autómato para o estado s_n e a entrada a

- se está marcada como sucesso ou falha, paramos
- se é uma leitura, então coloca-se a e o estado resultado da transição na pilha
- se é uma redução $X \to \alpha$, com α de comprimento p, então devemos encontrar α no topo da pilha

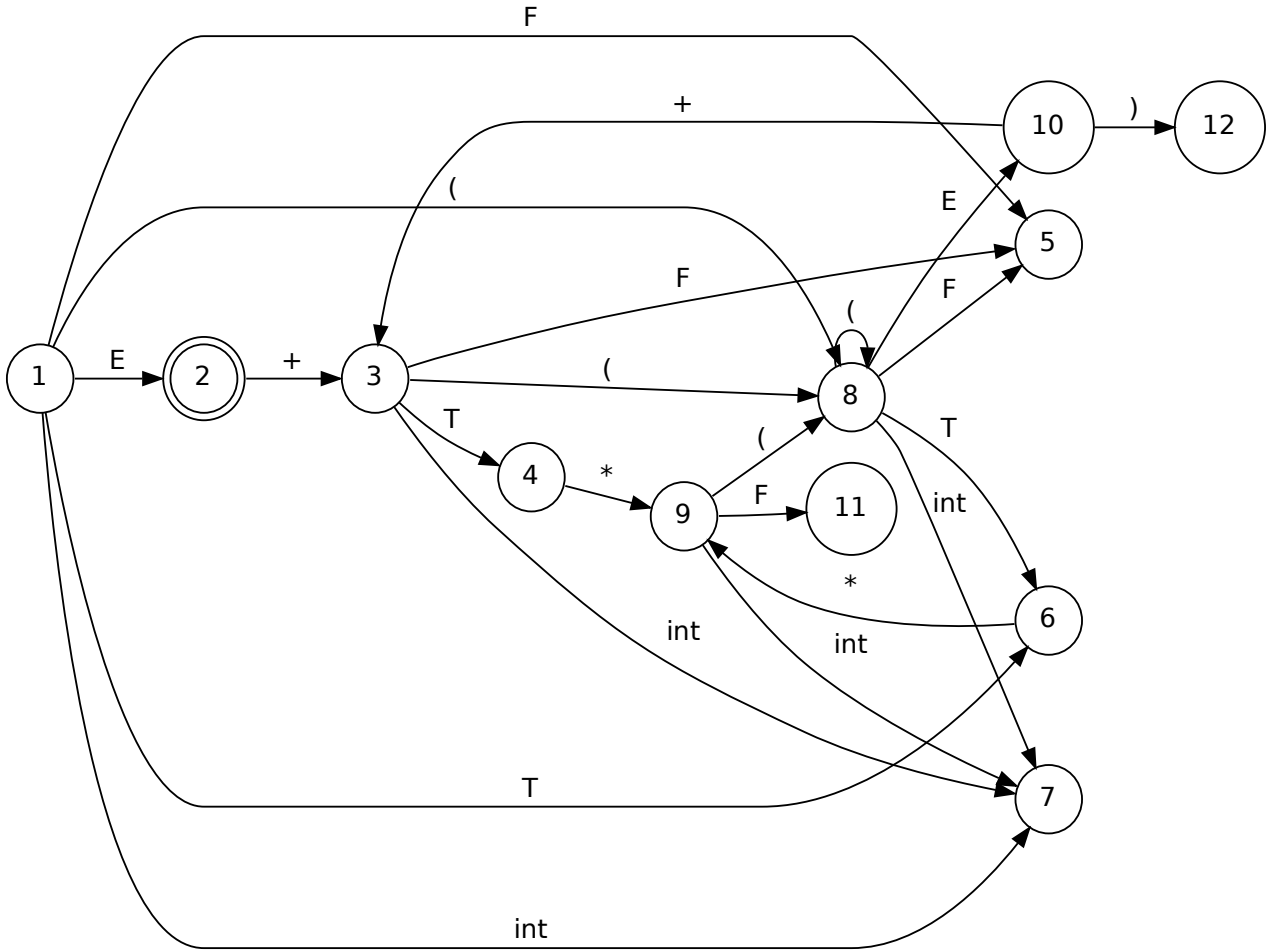
  $$s_0\, x_1\, s_1\, \ldots\, x_{n-p}\, s_{n-p}\, |\alpha_1\, s_{n-p+1} \ldots\, \alpha_p\, s_n$$

  tiramos então α da pilha e colocamos $X\, s$ no seu lugar, onde s é o estado resultado da transição $s_{n-p} \xrightarrow{X} s$, *i.e.*

  $$s_0\, x_1\, s_1\, \ldots\, x_{n-p}\, s_{n-p}\, X\, s$$

no exemplo anterior, servimo-nos do autómato seguinte

$$
\begin{array}{rcl}
E & \rightarrow & E + T \\
 & | & T \\
T & \rightarrow & T * F \\
 & | & F \\
F & \rightarrow & (E) \\
 & | & \texttt{int}
\end{array}
$$



construção do autómato e da tabela

fixemos por enquanto $k = 0$

começamos por construir um autómato **assíncrono**

isto é, contendo transições espontâneas
designadas de **transições** ϵ (notação $s_1 \xrightarrow{\epsilon} s_2$)

os **estados** são *itens* (também designados de 0-itens) da forma

$$[X \to \alpha \bullet \beta]$$

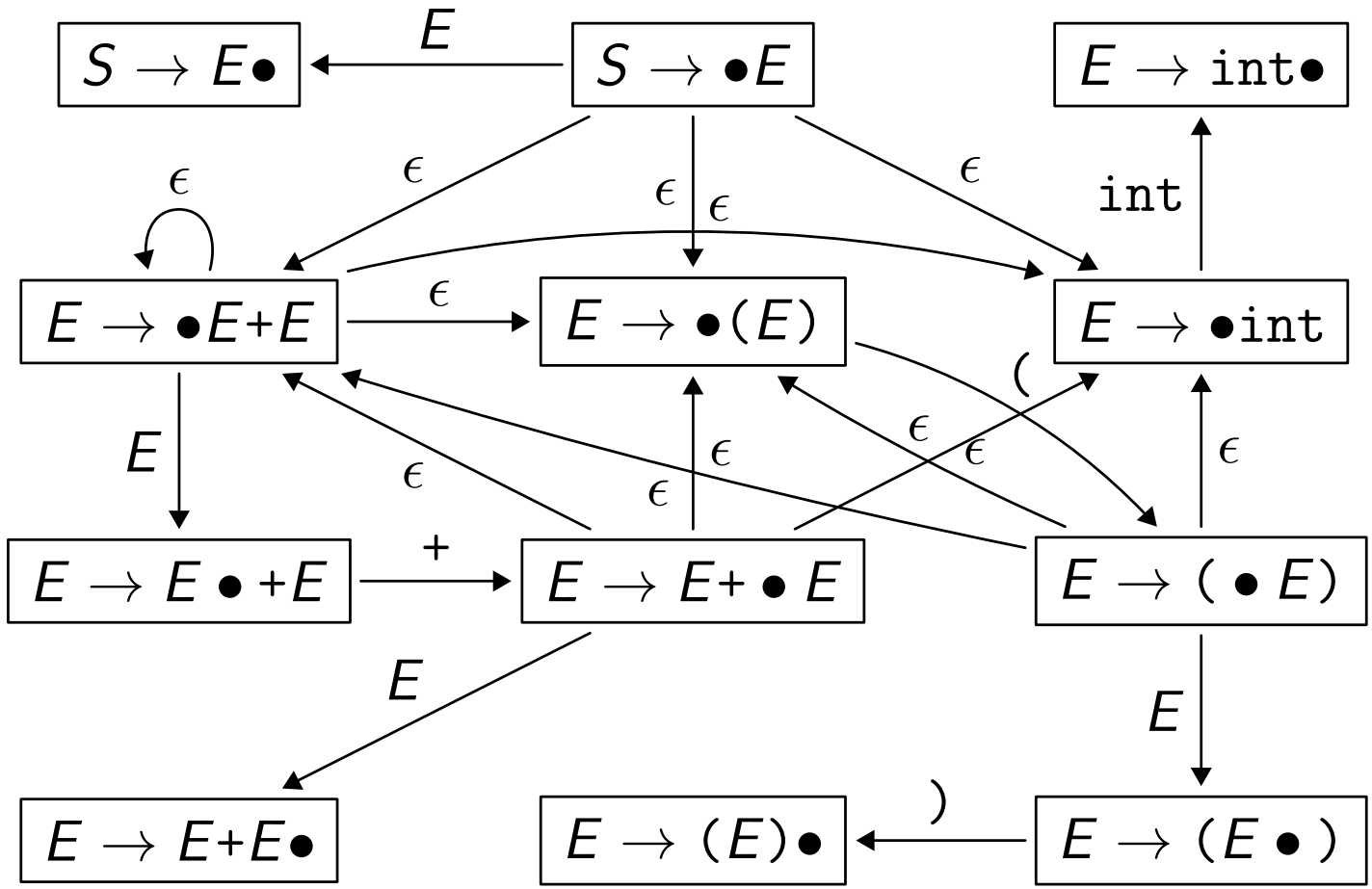
onde $X \to \alpha\beta$ é uma produção da gramática ; a intuição é
« procuro reconhecer X, já li α e resta me ler β »

as **transições** tem por label elementos de $T \cup N$ e são as seguintes

$$\begin{array}{rcll} [Y \to \alpha \bullet a\beta] & \xrightarrow{a} & [Y \to \alpha a \bullet \beta] & \\ [Y \to \alpha \bullet X\beta] & \xrightarrow{X} & [Y \to \alpha X \bullet \beta] & \\ [Y \to \alpha \bullet X\beta] & \xrightarrow{\epsilon} & [X \to \bullet\gamma] & \text{para toda a produção } X \to \gamma \end{array}$$

$S \rightarrow E$

$E \rightarrow E{+}E \mid (E) \mid \texttt{int}$



determinemos o autómato LR(0)

para tal, agrupamos os estados interligados por transições ϵ

os estados do autómato determinista são assim conjuntos de *itens*, tais como, por exemplo

$$
\begin{array}{l}
E \rightarrow E\texttt{+} \bullet E \\
E \rightarrow \bullet E\texttt{+}E \\
E \rightarrow \bullet\texttt{(}E\texttt{)} \\
E \rightarrow \bullet\texttt{int}
\end{array}
$$

cada estado s é **fechado** pela propriedade

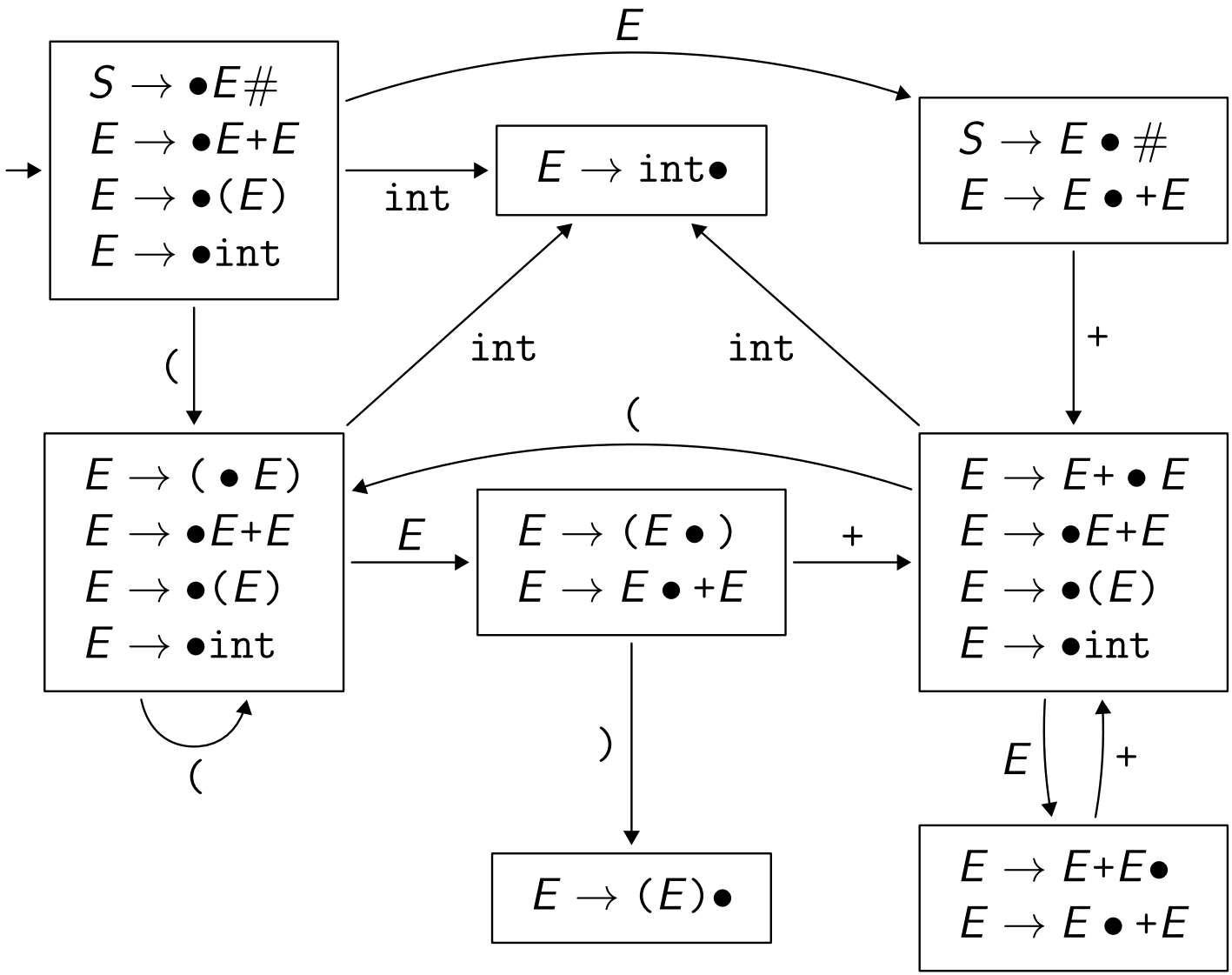
$$\begin{array}{rl} \text{se} & [Y \to \alpha \bullet X \beta] \in s \\ \text{e se} & X \to \gamma \text{ é uma produção} \\ \text{então} & [X \to \bullet \gamma] \in s \end{array}$$

o estado inicial é o estado (fechado) contendo $S \to \bullet E$

$S \rightarrow E$

$E \rightarrow E{+}E \mid (E) \mid \texttt{int}$

(nota: repare no ♯ significando *fim-de-entrada*)



na prática, não se trabalha directamente com o autómato mas sim com duas tabelas

- uma tabela de **ação** constituida nas linhas pelos estados do autómato, nas colunas pelos terminais ; a célula $\texttt{action}(s, a)$ indica
  - `shift` s', para uma leitura e um novo estado s'
  - `reduce` $X \to \alpha$ para uma redução
  - um sucesso
  - uma falha

- uma tabela de **deslocação** tendo por linhas os estados do autómato e por colunas os não.terminais ; a célula $\texttt{goto}(s, X)$ indica o estado que resulta de uma redução de X

construimos a tabela `action` desta forma

- $\texttt{action}(s, \#) =$ sucesso se $[S \to E \bullet \#] \in s$
- $\texttt{action}(s, a) = \texttt{shift}\ s'$ se temos uma transição $s \xrightarrow{a} s'$
- $\texttt{action}(s, a) = \texttt{reduce}\ X \to \beta$ se $[X \to \beta\bullet] \in s$, para todo o a
- falha, nos restantes casos

construimos a tabela `goto` desta forma

- $\texttt{goto}(s, X) = s'$ se e só se temos uma transição $s \xrightarrow{X} s'$

no exemplo, as tabelas são :

	action					*goto*
estado	(	)	+	int	#	*E*
1	shift 4			shift 2		3
2	reduce $E \rightarrow$ int					
3			shift 6		sucesso	
4	shift 4			shift 2		5
5		shift 7	shift 6			
6	shift 4			shift 2		8
7	reduce $E \rightarrow (E)$					
8			shift 6			
	reduce $E \rightarrow E$+E					

a tabela LR(0) pode conter dois tipos de conflitos

- uma conflito **leitura / redução** (*shift/reduce*), se num estado s podemos efectuar uma leitura mas também uma redução
- um conflito **redução / redução** (*reduce/reduce*), se num estado s são possíveis duas reduções diferentes

Definição (classe LR(0))

Uma gramática é designada de LR(0) se as tabelas desta forma construídas não contêm conflitos

temos um conflito leitura/redução no estado 8

$$\boxed{\begin{array}{l} E \to E{+}E\bullet \\ E \to E\bullet{+}E \end{array}}$$

este ilustra precisamente a ambiguidade da gramática sobre palavras tais como `int+int+int`

podemos resolver este conflito de duas formas

- se favorecemos a **leitura**, estabelecemos uma associatividade a direita
- se favorecemos a **redução**, estabelecemos uma associatividade a esquerda

vamos dar prioridade à redução

ilustremos com um exemplo

	(	)	+	int	#	E
1	s4			s2		3
2	reduce $E \rightarrow$ int					
3			s6		ok	
4	s4			s2		5
5		s7	s6			
6	s4			s2		8
7	reduce $E \rightarrow (E)$					
8	reduce $E \rightarrow E$+E					

pilha	entrada	ação
1	int+int+int	s2
1 int 2	+int+int	$E \rightarrow$ int, g3
1 E 3	+int+int	s6
1 E 3 + 6	int+int	s2
1 E 3 + 6 int 2	+int	$E \rightarrow$ int, g8
1 E 3 + 6 E 8	+int	$E \rightarrow E$+E, g3
1 E 3	+int	s6
1 E 3 + 6	int	s2
1 E 3 + 6 int 2	#	$E \rightarrow$ int, g8
1 E 3 + 6 E 8	#	$E \rightarrow E$+E, g3
1 E 3	#	sucesso

a construção LR(0) gera muito facilmente conflitos
vamos então procurar limitar as reduções

uma ideia muito simples consiste em considerar
$\texttt{action}(s, a) = \texttt{reduce}\ X \to \beta$ se e só se

$$[X \to \beta\bullet] \in s \quad \text{e} \quad a \in \text{follow}(X)$$

Definição (classe SLR(1))

uma gramática é dita SLR(1) se as tabelas construídas não contêm conflitos.

(SLR significa *Simple LR*)

a gramática

$$
\begin{array}{lcl}
S & \rightarrow & E\# \\
E & \rightarrow & E + T \\
 & | & T \\
T & \rightarrow & T * F \\
 & | & F \\
F & \rightarrow & (E) \\
 & | & \texttt{int}
\end{array}
$$

é SLR(1)

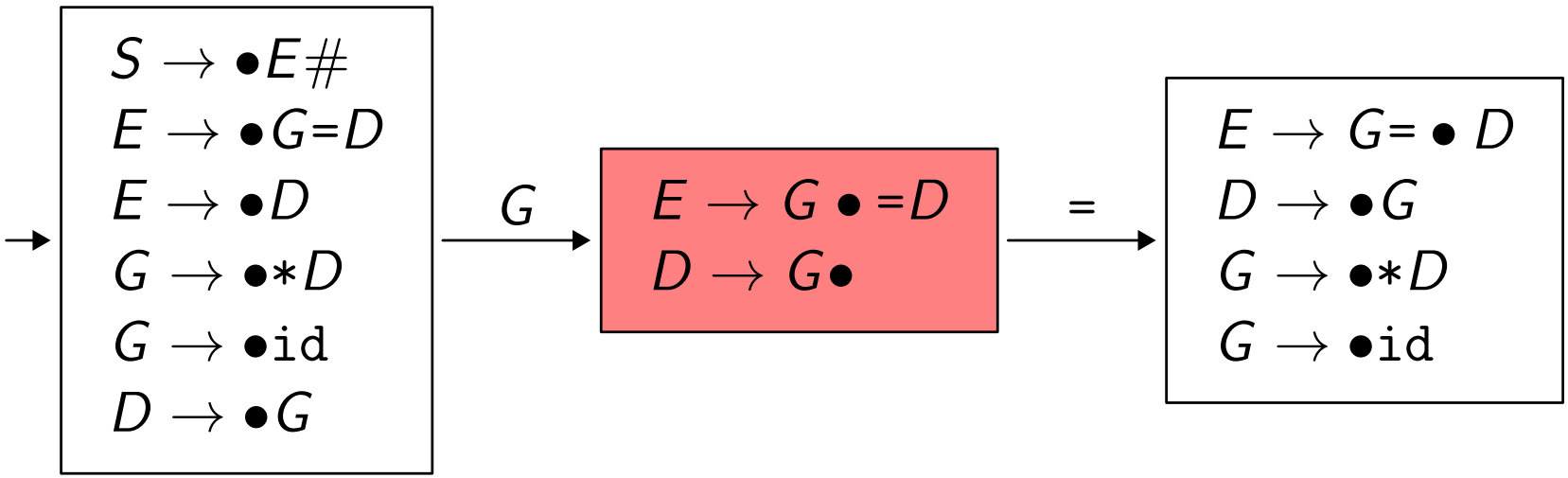
exercício : verificar tal afirmação, em particular constatar que o autómato tem 12 estados

na prática a classe SLR(1) continua demasiada restritiva

exemplo :

$$
\begin{array}{lcl}
S & \rightarrow & E\# \\
E & \rightarrow & G = D \\
 & | & D \\
G & \rightarrow & * D \\
 & | & \texttt{id} \\
D & \rightarrow & G
\end{array}
$$

	=	
1	...	...
2	shift 3 reduce $D \rightarrow G$	...
3	⋮	⋱



introduzimos uma classe de gramáticas ainda mais largada, **LR(1)**, mas com o custo de umas tabelas bem maiores

na análise LR(1), os *itens* têm agora a forma

$$[X \to \alpha \bullet \beta, a]$$

cujo significado é : « procuro reconhecer X, já li α, devo ainda ler β e em seguida verificar que o caractere que segue é a »

um item desta natureza é designado de **1-item**

as transições do autómato LR(1) não determinista (i.e. assíncrono) são

$$
\begin{aligned}
[Y \to \alpha \bullet a\beta, b] &\xrightarrow{a} [Y \to \alpha a \bullet \beta, b] \\
[Y \to \alpha \bullet X\beta, b] &\xrightarrow{X} [Y \to \alpha X \bullet \beta, b] \\
[Y \to \alpha \bullet X\beta, b] &\xrightarrow{\epsilon} [X \to \bullet\gamma, c] \quad \text{para todo o } c \in \text{first}(\beta b)
\end{aligned}
$$

o estado inicial é o estado $[S \to \bullet\alpha, \#]$

procedemos de forma semelhante ao caso LR(0)

agrupamos os estados interligados por transições ϵ

concretamente cada estado s é fechado pela propriedade

$$\begin{array}{rl} \text{se} & [Y \to \alpha \bullet X\beta, a] \in s \\ \text{e se} & X \to \gamma \text{ é uma produção} \\ \text{então} & [X \to \bullet\gamma, b] \in s, \quad \text{para todo o } b \in \text{first}(\beta a) \end{array}$$

e o estado inicial é o estado (fechado) que contém $[S \to \bullet\alpha, \#]$

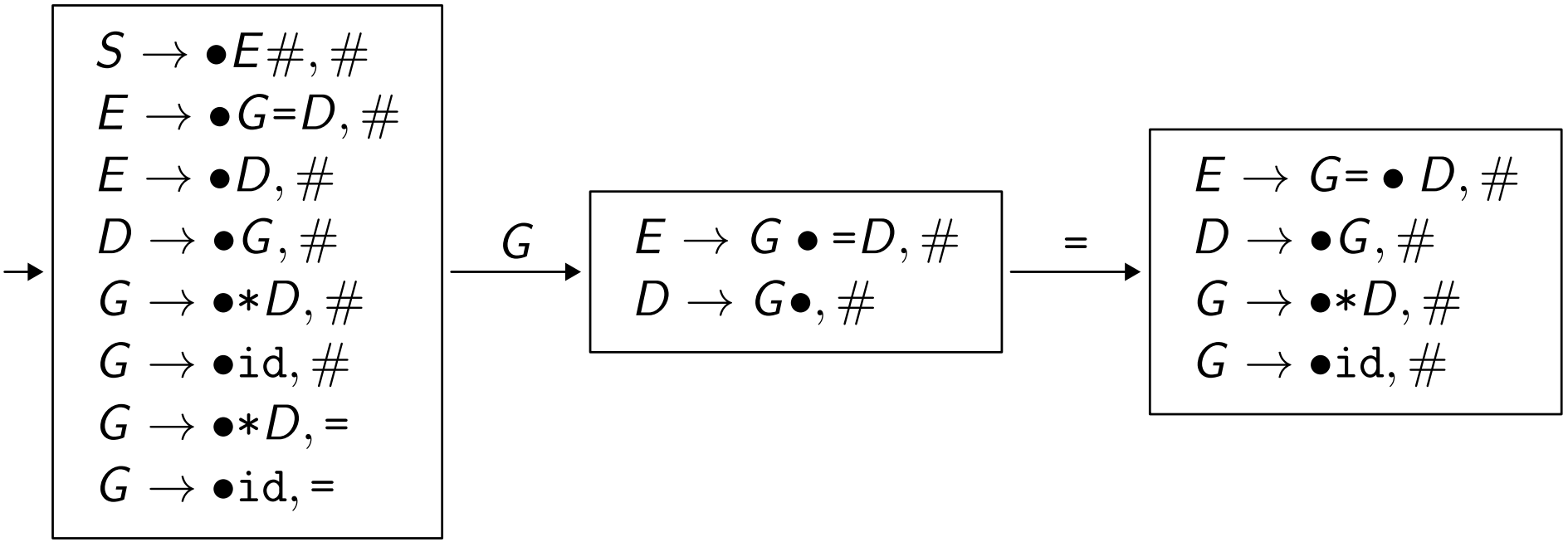
podemos então construir a tabela correspondente ;
introduzimos uma ação de redução para (s, a) somente quando s contiver um item da forma $[X \to \alpha\bullet, a]$

Definição (classe LR(1))

Uma gramática é designada por LR(1) se a tabela assim construída não contém conflitos.

$$
\begin{array}{lcl}
S & \rightarrow & E\# \\
E & \rightarrow & G = D \\
 & | & D \\
G & \rightarrow & * D \\
 & | & \texttt{id} \\
D & \rightarrow & G
\end{array}
$$

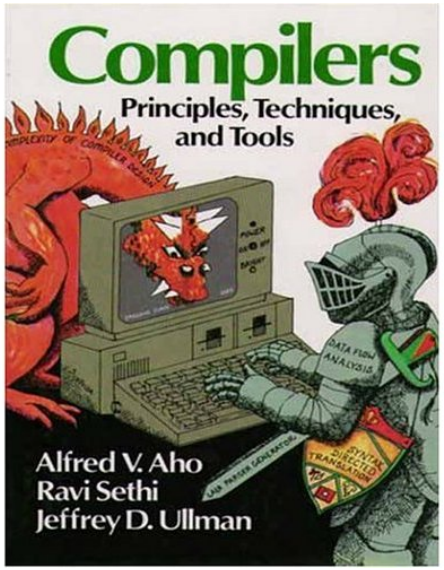
	#	=	
1	...	...	...
2	reduce $D \rightarrow G$	shift 3	...
3	⋮	⋮	⋱

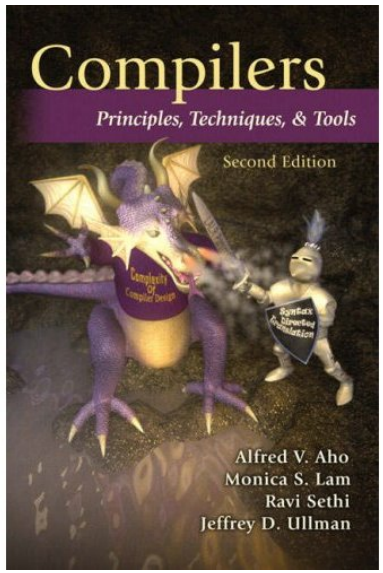


a construção LR(1) implica as vezes um custo que se pode querer minimizar (e.g. tamanho da tabela), existam aproximações

a classe LALR(1) (*lookahead LR*) é uma destas aproximações e esta está na base de ferramentas da família `yacc`

para mais informações sobre estas aproximações : consultar por exemplo *Compilers : principles techniques and tools* (conhecido como « o livro do dragão ») de A. Aho, R. Sethi, J. Ullman, secção 4.7





os estados formados de 1-item na analise $LR(1)$ correspondem aos estados da analise $LR(0)$ mas nos quais cada item ficou anotado com uma lista dos símbolos seguintes. Podemos ter um estado 0-item duplicado em vários estados da analise

em prática o número de estados canónicos de $LR(1)$ é bem maior do que o de analises como $LR(0)$ ou $SLR(1)$. A ideia aqui é assim actuar ao nível dos estados gerados numa analise $LR(0)$ tentando agrupar todos os estados que partilham a mesma estrutura $LR(0)$ subjacente. Aqui consideraremos a união dos caracteres seguintes

consideremos que os estados do autómato $LR(1)$ são $I_1 \ldots I_n$

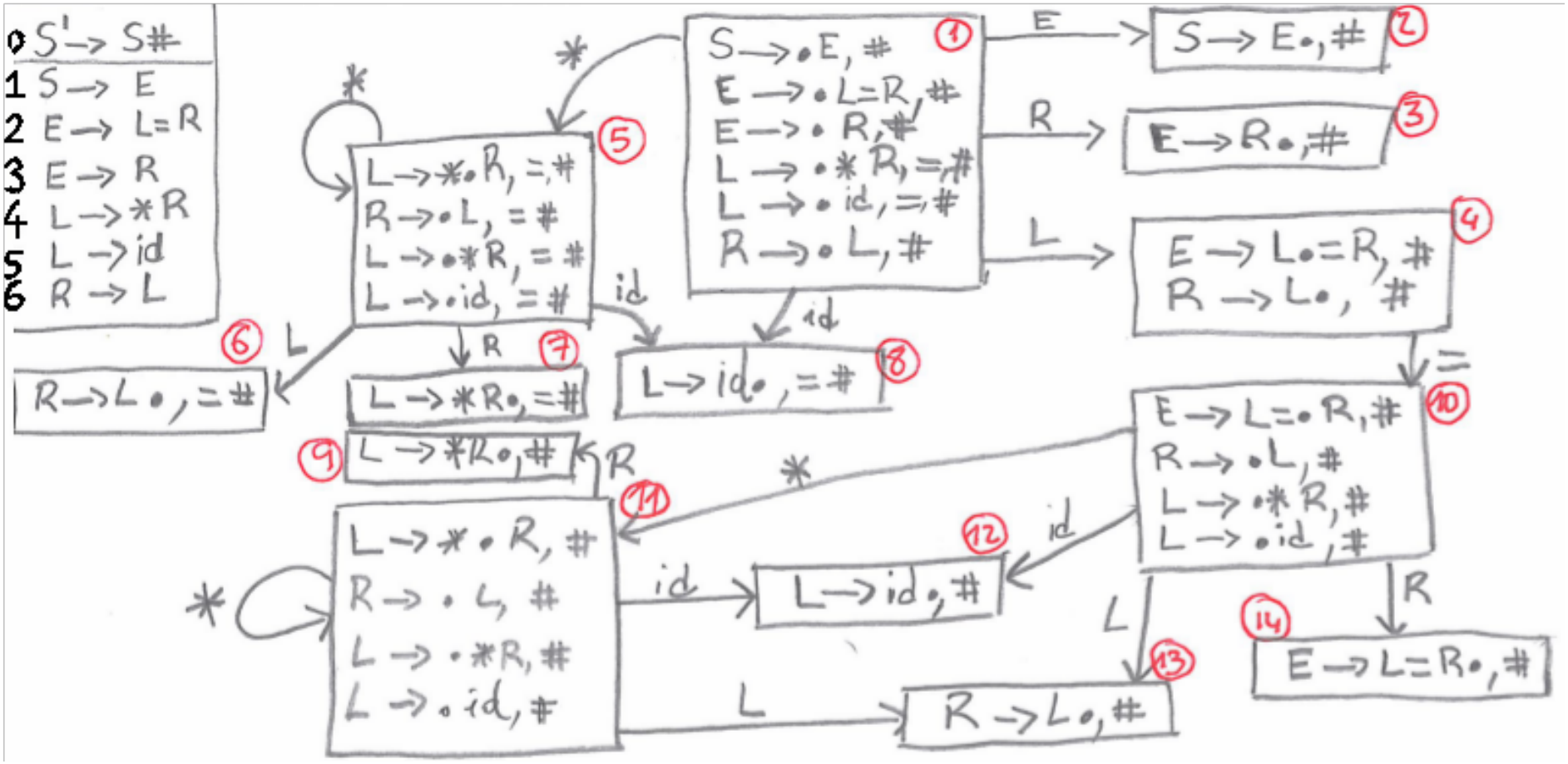
agrupamos todos os estados que tem a mesma estrutura $LR(0)$, obtemos os estados $J_1 \ldots J_p$

determina-se a tabela de deslocação da mesma forma que no caso $LR(0)$.

as acções de redução são definidas como no caso $LR(1)$

uma gramática que não gera conflitos numa tabela $LALR(1)$ é designada de gramática $LALR(1)$

exemplo - Autómato LR(1)



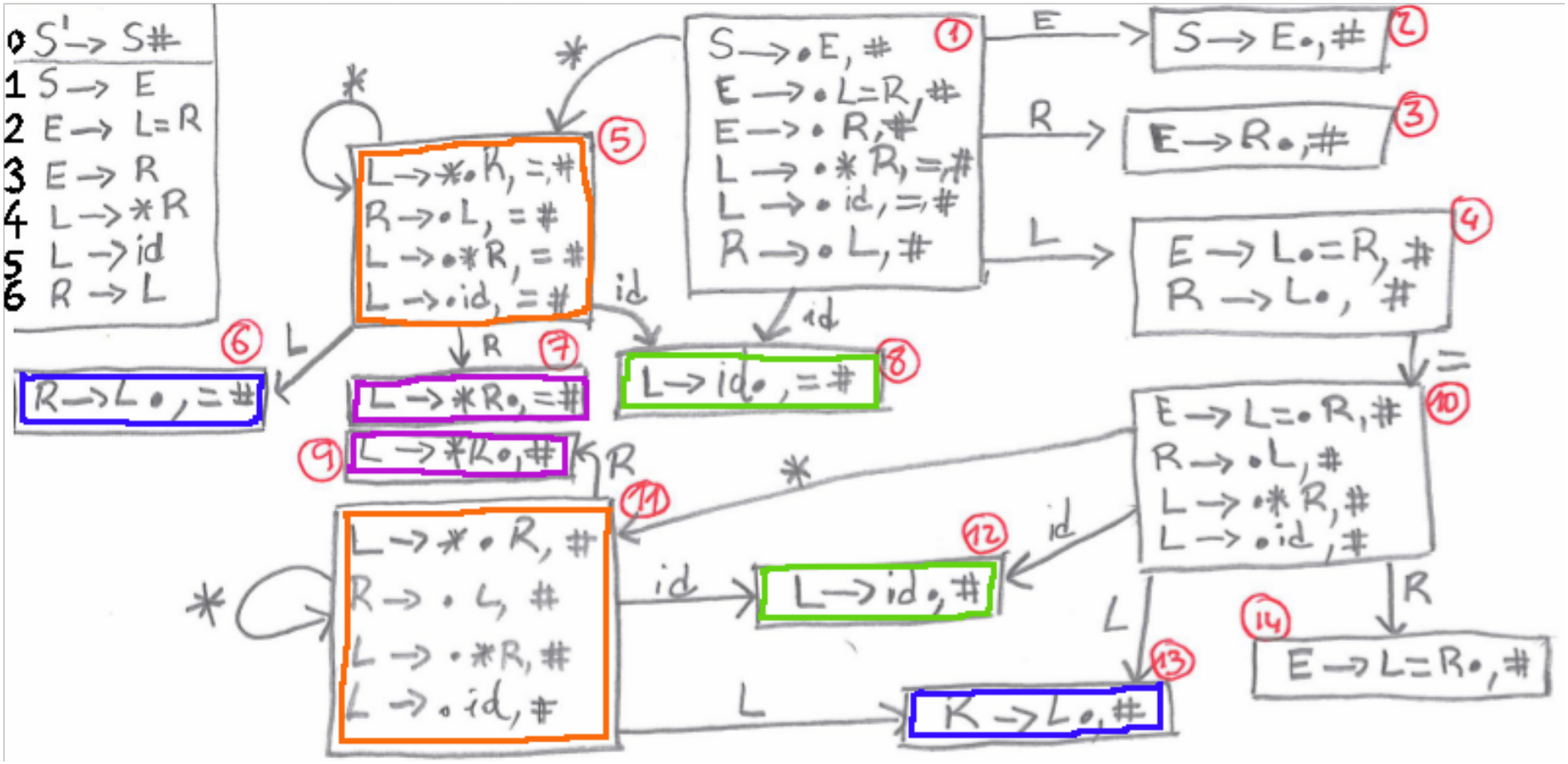
exemplo - Tabelas Acção/Deslocação LR(1)

0	S' → S #
1	S → E
2	E → L = R
3	E → R
4	L → * R
5	L → id
6	R → L

↑
ao que se referem os "reduce"

	id	*	=	#	E	L	R
1	S8	S5			G2	G4	G3
2				r1 ≡ SUCESSO			
3				r3			
4			S10	r6			
5	S8	S5				G6	G7
6			r6	r6			
7			r4	r4			
8			r5	r5			
9				r4			
10	S12	S11				G13	G14
11	S12	S11				G13	G9
12				r5			
13				r6			
14				r2			

exemplo - Autómato LALR(1) via LR(1)



exemplo - Tabelas LALR(1) via LR(1)

0	S' → S #
1	S → E
2	E → R
3	E → R
4	L → *R
5	L → id
6	R → L

LR(1) → LALR(1)

5,11 → 5

6,13 → 6

7,9 → 7

8,12 → 8

	id	*	=	#	E	L	R
1	S8	S5			G2	G4	G3
2				✓			
3				r3			
4			S10	r6			
5	S8	S5				G6	G7
6			r6	r6			
7			r4	r4			
8			r5	r5			
10	S8	S5				G6	G14
14				r2			

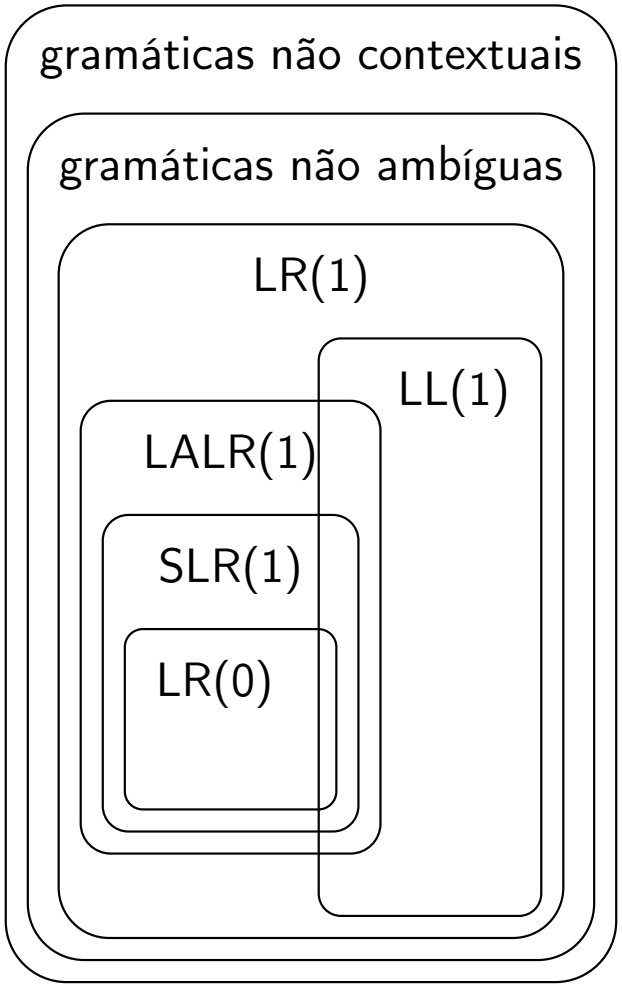
conflitos: os únicos conflitos que surgem relativamente a analise $LR(1)$ são conflitos entre duas reduções

de facto se existe um conflito leitura/redução então significa que existem dois estados $X \to \gamma., a$ e $Y \to \alpha.a\beta, c$, mas neste caso existe o item $Y \to \alpha.a\beta, d$ já no estado do autómato $LR(1)$ na origem. **Contradição**

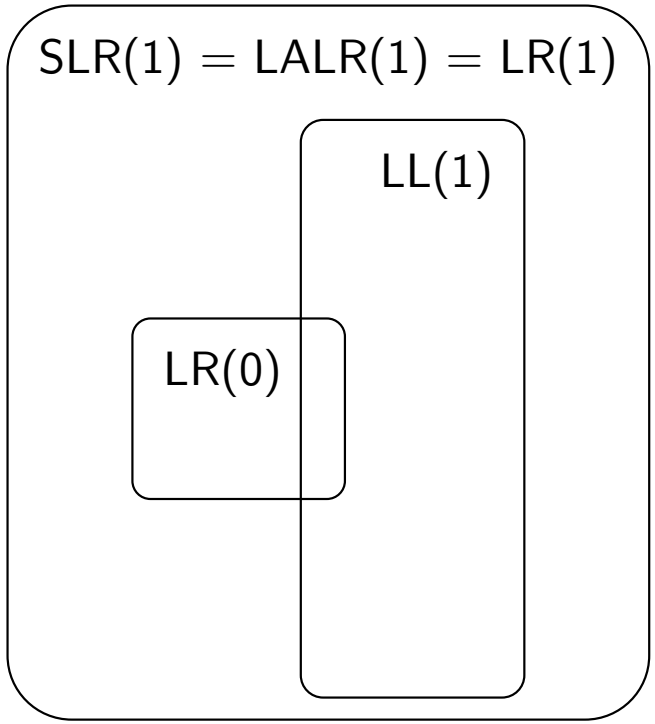
as gramáticas $LALR(1)$ são menos poderosas que as gramáticas $LR(1)$ mas são em contrapartida mais eficientes, devido ao facto de construírem autómatos mais compactos

ferramentas como o `yacc` e os seus derivados geram analizadores sintácticos ascendentes que aceitam gramáticas $LALR(1)$

gramáticas



linguagens



a análise ascendente é potente mas os cálculos das tabelas é complexa

este trabalho pode ser automatizado por várias ferramentas existentes

são constituintes da grande família de ferramentas `yacc`, `bison`, `ocamlyacc`, `cups`, `menhir`, ...
(YACC significa *Yet Another Compiler Compiler*)

OCamlyacc

nesta aula vamos dar mais ênfase à ferramenta menhir

a introdução ao ocamlyacc far-se-á em consequência por uma explicação prática dos problemas encontrados na **engenharia das gramáticas** LALR(1) (que se estendem naturalmente às gramáticas LR(1), mais permissivas no que diz respeito ao risco de conflito)

todos os exemplos apresentados são compatíveis com o menhir (podemos usar o menhir no lugar do ocamlyacc sobre os mesmos ficheiros apresentados)

```
%{
(*  Cabeçalho.  Opcional, pode ser omitido. *)
(*  Contém eventual código OCaml *)
%}
%token INT PLUS MINUS TIMES DIV EOF
%start expr
%type <unit> expr
%%

expr: INT              { (* acção - código OCaml *) }
    | expr PLUS expr   {}
    | expr MINUS expr  {}
    | expr TIMES expr  {}
    | expr DIV expr    {}
    | MINUS expr       {};

%%
  (* Trailer. Opcional : pode ser omitido.*)
  (* Contém eventual código OCaml *)
```

precedência e associatividade - (ocamlyacc -v calc.mly)

```
   0  $accept : %entry% $end
   1  expr : INT
   2       | expr PLUS expr
   3       | expr MINUS expr
   4       | expr TIMES expr
   5       | expr DIV expr
   6       | MINUS expr
   7  %entry% : '\001' expr
(.....)
14: shift/reduce conflict (shift 7, reduce 5) on PLUS
14: shift/reduce conflict (shift 8, reduce 5) on MINUS
14: shift/reduce conflict (shift 9, reduce 5) on TIMES
14: shift/reduce conflict (shift 10, reduce 5) on DIV
state 14
expr : expr . PLUS expr  (2)
expr : expr . MINUS expr  (3)
expr : expr . TIMES expr  (4)
expr : expr . DIV expr  (5)
expr : expr DIV expr .  (5)

PLUS  shift 7
MINUS  shift 8
TIMES  shift 9
DIV  shift 10
$end  reduce 5
(.....)
State 14 contains 4 shift/reduce conflicts.
9 terminals, 3 nonterminals
8 grammar rules, 15 states
```

precedência e associatividade: a solução

```
%{
%}
%token INT PLUS MINUS TIMES DIV EOF
%left PLUS MINUS
%left TIMES DIV
%left uminus
%start expr
%type <unit> expr
%%

expr: INT                              {}
    | expr PLUS expr                   {}
    | expr MINUS expr                  {}
    | expr TIMES expr                  {}
    | expr DIV expr                    {}
    | MINUS expr %prec uminus          {};
```

dangling else

Consideremos as três regras: (1) $S \rightarrow \mathit{if}\ E\ \mathit{then}\ S\ \mathit{else}\ S$,
(2) $S \rightarrow \mathit{if}\ E\ \mathit{then}\ S$ e (3) $S \rightarrow \mathit{resto}$

- Em $\mathit{if}\ a\ \mathit{then}\ \mathit{if}\ b\ \mathit{then}\ x\ \mathit{else}\ y \implies$ a que *if* pertence o *else*?
- Conflito *shift*/*reduce* com *else* entre
  (*reduce*) $S \rightarrow \mathit{if}\ E\ \mathit{then}\ S\ .$ e (*shift*) $S \rightarrow \mathit{if}\ E\ \mathit{then}\ S\ .\ \mathit{else}\ S$
- uma solução: reescrever a gramática desta forma
  1 $S \rightarrow M$
  2 $S \rightarrow U$
  3 $M \rightarrow \mathit{if}\ E\ \mathit{then}\ M\ \mathit{else}\ M$
  4 $M \rightarrow \mathit{resto}$
  5 $U \rightarrow \mathit{if}\ E\ \mathit{then}\ S$
  6 $U \rightarrow \mathit{if}\ E\ \mathit{then}\ M\ \mathit{else}\ U$
- outra solução: deixar como está! De facto em caso de conflito *shift*/*reduce*, as ferramentas como yacc assumem a interpretação *shift*.
- Este truque deve ser usado com cuidado! Em caso de conflito verificar com cuidado se podem deixar o yacc decidir sempre pelo *shift*.

em concreto...(ocamlyacc -v exemplo.mly)

```
%{
%}

%token ID WHILE BEGIN END DO IF THEN ELSE SEMI ASSIGN EOF
%start prog
%type <unit>prog
%%

prog: stmlist EOF              {};

stmlist: stm                   {}
       | stmlist SEMI stm      {};

stm: ID ASSIGN ID              {}
   | WHILE ID DO stm           {}
   | BEGIN stmlist END         {}
   | IF ID THEN stm            {}
   | IF ID THEN stm ELSE stm   {};
```

em concreto...(ocamlyacc -v exemplo.mly)

```
   0  $accept : %entry% $end
   1  prog : stmlist EOF
   2  stmlist : stm
   3          | stmlist SEMI stm
   4  stm : ID ASSIGN ID
   5      | WHILE ID DO stm
   6      | BEGIN stmlist END
   7      | IF ID THEN stm
   8      | IF ID THEN stm ELSE stm
   9  %entry% : '\001' prog
(......)
22: shift/reduce conflict (shift 23, reduce 7) on ELSE
state 22
stm : IF ID THEN stm .  (7)
stm : IF ID THEN stm . ELSE stm  (8)

ELSE  shift 23
END  reduce 7
SEMI  reduce 7
EOF  reduce 7
(......)
State 22 contains 1 shift/reduce conflict.
14 terminals, 5 nonterminals
10 grammar rules, 25 states
```

conflitos *reduce*/*reduce*

```
%token ID PLUS MINUS AND OR EQUAL ASSIGN EOF
%left OR
%left AND
%left PLUS
%start stm
%type <unit> stm
%%

stm: ID ASSIGN ae     {}
   | ID ASSIGN be     {};

be: be OR be          {}
  | be AND be         {}
  | ae EQUAL ae       {}
  | ID                {};

ae: ae PLUS ae        {}
  | ID                {};
```

conflitos *reduce*/*reduce*

```
   0  $accept : %entry% $end
   1  stm : ID ASSIGN ae
   2      | ID ASSIGN be
   3  be : be OR be
   4     | be AND be
   5     | ae EQUAL ae
   6     | ID
   7  ae : ae PLUS ae
   8     | ID
(.....)
6: reduce/reduce conflict (reduce 6, reduce 8) on $end
state 6
be : ID .  (6)
ae : ID .  (8)

$end  reduce 6
PLUS  reduce 8
AND  reduce 6
OR  reduce 6
EQUAL  reduce 8
(.....)
State 6 contains 1 reduce/reduce conflict.
11 terminals, 5 nonterminals
10 grammar rules, 19 states
```

aqui a dificuldade é que um identificador pode ocorrer tanto numa expressão booleana (*be*) como numa expressão aritmética (*ae*)

na analise dum identificador, que produção escolher para a redução? a regra 6 (identificador booleano) ou a regra 8 (identificador aritmético)?

nesta fase da compilação não temos forma fácil de inferir em que situação nos encontramos. Por isso o mais sensato é reescrever a gramática de tal forma que a escolha seja feita posteriormente: na **análise semântica** (tipagem, análise de porte etc.)

conflitos *reduce*/*reduce*

```
%token ID PLUS MINUS AND OR EQUAL ASSIGN EOF
%left OR
%left AND
%left PLUS
%start stm
%type <unit> stm
%%

stm: ID ASSIGN expr     {};

expr: expr OR expr      {}
    | expr AND expr     {}
    | expr EQUAL expr   {}
    | expr PLUS expr    {}
    | ID                {};
```

gestão implícita dos conflitos pelo yacc

(Documentação do OCamlYacc)

The precedence declarations are used in the following way to resolve reduce/reduce and shift/reduce conflicts:

- Tokens and rules have precedences. By default, the precedence of a rule is the precedence of its rightmost terminal. You can override this default by using the %*prec* directive in the rule.
- A reduce/reduce conflict is resolved in favor of the first rule (in the order given by the source file), and ocamlyacc outputs a warning.
- A shift/reduce conflict is resolved by comparing the precedence of the rule to be reduced with the precedence of the token to be shifted. If the precedence of the rule is higher, then the rule will be reduced; if the precedence of the token is higher, then the token will be shifted.
- A shift/reduce conflict between a rule and a token with the same precedence will be resolved using the associativity: if the token is left-associative, then the parser will reduce; if the token is right-associative, then the parser will shift. If the token is non-associative, then the parser will declare a syntax error.
- When a shift/reduce conflict cannot be resolved using the above method, then ocamlyacc will output a warning and the parser will always shift.

yacc: o token `error`

o que acontece se o yacc detecta um erro (sintáctico) durante a análise sintáctica?

isto acontece quando a palavra actualmente analisada deixa de poder ter continuação que permita ser reconhecida

neste caso ferramentas como *yacc* utilizam a técnica da **correcção local do erro**

esta técnica baseia-se na introdução dum token especial, o token *error*, no local onde o erro foi detectado e na presença de tokens que desempenham um papel de **sincronização**

esses tokens são em geral tokens de pontuação que permitam delimitar componentes importantes da linguagem em causa

o token *error* pode assim ser utilizado como um token normal na gramática. A sua utilização num local particular da gramática representa o facto que a produção em questão pretende tratar um eventual erro sintáctico onde o token *error* aparece

que faz o yacc com o símbolo *error* introduzido aquando da detecção efectiva dum erro?

1. esvaziar a pilha de estados (se necessário) até chegar a um estado no qual uma acção para o token *error* seja *shift*
2. executar *shift error*
3. descartar símbolos de entrada (se necessário) até atingir um estado que tenha uma acção normal (que não seja de "erro") sobre o token de antevisão (**lookahead token**). Este token é o **token de sincronização**
4. retomar o *parsing* normalmente.

os tokens de sincronização podem ser pontuação que separam instruções

neste caso se um erro ocorrer, vamos procurar o próximo ponto de sincronização e ignorar todo o input que ocorre entre o local de erro e este ponto

a análise pode recomeçar normalmente a seguir. I.e. deita fora toda a instrução onde o erro ocorreu e retoma a analise normalmente com a instrução seguinte

(Documentação do OCamlYacc)

- Error recovery is supported as follows: when the parser reaches an error state (no grammar rules can apply), it calls a function named `parse_error` with the string "syntax error"as argument. The default `parse_error` function does nothing and returns, thus initiating error recovery (see below). The user can define a customized `parse_error` function in the header section of the grammar file.
- The parser also enters error recovery mode if one of the grammar actions raises the `Parsing.Parse_error` exception.
- In error recovery mode, the parser discards states from the stack until it reaches a place where the error token can be shifted. It then discards tokens from the input until it finds three successive tokens that can be accepted, and starts processing with the first of these. If no state can be uncovered where the error token can be shifted, then the parser aborts by raising the `Parsing.Parse_error` exception.

a ferramenta `menhir`

Menhir é um ferramenta que transforma uma gramática num analisador OCaml ; este assenta numa análise LR(1)

cada produção da gramática está acompanhada de uma **ação semântica** *i.e.* de código OCaml que constrói um valor semântico (tipicamente uma árvore de sintaxe abstracta)

Menhir utiliza-se em conjunto com um analisador léxico (tipicamente `ocamllex`)

um ficheiro Menhir tem o sufixo `.mly` e tem por estrutura

```
%{
  ...(opcional)  código OCaml arbitrário ...
%}
...declaração dos tokens...
...declaração de precedências e associatividade...
...declaração dos pontos de entrada...
%%
não-terminal-1:
| produção { ação }
| produção { ação }
;

não-terminal-2:
| produção { ação } ;
...
%%
  ... (opcional)  código OCaml arbitrário ...
```

```
%token PLUS LPAR RPAR EOF
%token <int> INT
%start <int> frase
%%

frase:
  e = expression; EOF  { e }
;

expression:
| e1 = expression; PLUS; e2 = expression { e1 + e2 }
| LPAR; e = expression; RPAR              { e }
| i = INT                                 { i }
;
```

compilamos o ficheiro `arith.mly` da forma seguinte

```
% menhir -v arith.mly
```

obtemos código OCaml puro no ficheiro `arith.ml(i)`, que contem em particular

- a declaração de um tipo `token`

```
type token = RPAR | PLUS | LPAR | INT of int | EOF
```

- para cada não-terminal declarado com `%start`, uma função de tipo

```
val frase: (Lexing.lexbuf -> token) -> Lexing.lexbuf -> int
```

como vemos, esta função toma como parâmetro um analisador léxico, do tipo dos que `ocamllex` produz (cf aula sobre análise léxica)

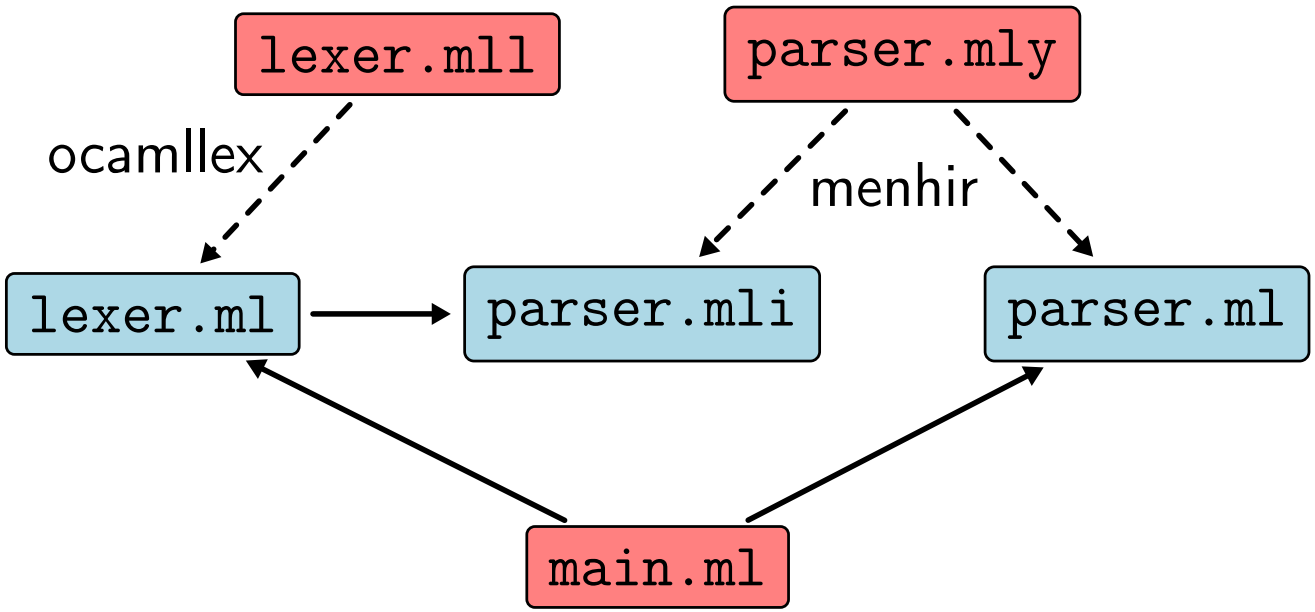
quando combinamos `ocamllex` e `menhir`

- `lexer.mll` faz referência aos lexemas definidos em `parser.mly`

```
{
  open Parser
}
...
```

- o analisador léxico e o analisador sintáctico são combinados da forma seguinte:

```
let c  = open_in file in
let lb = Lexing.from_file c in
let e  = Parser.frase Lexer.token lb in
...
```



= fonte utilizador

= construido automaticamente

= dependência

revisitemos a noção de conflito, mas desta vez com o Menhir

quando a gramática não é LR(1), Menhir apresenta os **conflitos** ao utilizador

- o ficheiro `.automaton` contém uma descrição do autómato LR(1) ; os conflito estão ali referenciados
- o ficheiro `.conflicts` contém, se assim for o caso, uma explicação de cada conflito, na forma de uma sequência de lexemas que conduz a duas árvores de derivação (i.e. a uma manifestação do conflito)

na gramática anterior, Menhir assinala o conflito

```
% menhir -v arith.mly
Warning: one state has shift/reduce conflicts.
Warning: one shift/reduce conflict was arbitrarily resolved.
```

o ficheiro `arith.automaton` contém em particular

```
State 6:
expression -> expression . PLUS expression [ RPAR PLUS EOF ]
expression -> expression PLUS expression . [ RPAR PLUS EOF ]
- On PLUS shift to state 5
- On RPAR reduce production expression -> expression PLUS expression
- On PLUS reduce production expression -> expression PLUS expression
- On EOF reduce production expression -> expression PLUS expression
** Conflict on PLUS
```

o ficheiro `arith.conflicts` apresenta a seguinte explicação do conflito

```
** Conflict (shift/reduce) in state 6.
** Token involved: PLUS
** This state is reached from phrase after reading:

expression PLUS expression

** In state 6, looking ahead at PLUS, shifting is permitted
** because of the following sub-derivation:

expression PLUS expression
                  expression . PLUS expression

** In state 6, looking ahead at PLUS, reducing production
** expression -> expression PLUS expression
** is permitted because of the following sub-derivation:

expression PLUS expression // lookahead token appears
```

uma forma de resolução dos conflitos passa por indicar ao Menhir como escolher entre leituras e reduções

para esse efeito, podemos dar **prioridades** aos lexemas e às produções e regras de **associatividades**

como no caso do yacc

por omissão, a prioridades de uma produção é a prioridade do seu lexema mais a direita (mas esta pode ser especificada de forma explícita)

se a prioridade da produção é maior do que o lexema por ler,
então favorece-se a redução

reciprocamente, sa prioridade do lexema for maior,
então a leitura é privilegiada

em caso de igualdade, consulta-se as regras de associatividade : um lexema associativo à esquerda favorece a redução e um lexema associativo à direita favorece a leitura

no nosso exemplo, basta indicar por exemplo que PLUS é associativo à esquerda

```
%token PLUS LPAR RPAR EOF
%token <int> INT
%left PLUS
%start <int> frase
%%
frase:
  e = expression; EOF  { e }
;
expression:
| e1 = expression; PLUS; e2 = expression { e1 + e2 }
| LPAR; e = expression; RPAR              { e }
| i = INT                                 { i }
;
```

para associar prioridades aos lexemas, utilizamos a convenção seguinte :

- a ordem de declaração da associatividade fixa as prioridades
  (os primeiros lexemas têm as prioridades mais fracas)
- vários lexemas podem aparecer numa mesma declaração (na mesma linha),
  tendo assim a mesma prioridade

exemplo :

```
%left PLUS MINUS
%left TIMES DIV
```

a gramática contém um conflito

```
expression:
| IF e1 = expression; THEN; e2 = expression
    { ... }
| IF e1 = expression; THEN; e2 = expression;
  ELSE; e3 = expression
    { ... }
| i = INT
    { ... }
| ...
```

corresponde à situação

```
IF a THEN IF b THEN c ELSE d
```

para associar o ELSE ao THEN mais próximo, devemos privilegiar a leitura

```
%nonassoc THEN
%nonassoc ELSE
```

(esta situação é conhecida em inglês como *dangling else*)

Menhir oferece numerosas vantagens relativamente a ferramentas como, por exemplo, `ocamlyacc` :

- não-terminais parametrizados por (não-)terminais
  - em particular, mecanismos para escrever expressões regulares na gramática ($E?$, $E*$, $E+$), listas com separadores

```
expr: (...);
lista_exprs:  ve = separated_list(COMMA,expr)  { ve } ;
```

- explicações dos conflitos
- modo interactivo, modo depuração
- gramáticas parametrizadas
- análise LR(1) no lugar de LALR(1)
- etc.

ler o manual de Menhir !

para que as fases seguintes da análise sintáctica (e.g. a tipagem) possam dar indicações de **posição** nas mensagens de erro, convém guardar informação de localização directamente na árvore de sintaxe abstracta

Menhir fornece esta informação via `$startpos` e `$endpos`, dois valores de tipo `Lexing.position` ; esta informação é-lhe transmitida pelo analisador léxico

cuidado : relembramos que `ocamllex` só mantém por omissão a posição absoluta dentro do ficheiro ; ter os números de linha e coluna actualizados necessita de um tratamento explícito realizado pelo programador (quem define os analisadores), (ver por exemplo, os vários ficheiros `lexer.mll` fornecidos nas aulas práticas)

uma forma de conservar a informação de localização na AST é a seguinte (ver aula sobre tipagem)

```
type expression =
  { desc: desc;
    loc : Lexing.position * Lexing.position }

and desc =
  | Econst of int
  | Eplus  of expression * expression
  | Eneg   of expression
  | ...
```

a gramática pode então parecer-se com

```
expression:
| d = desc { { desc = d; loc = $startpos, $endpos } }
;

desc:
| i = INT                                    { Econst i }
| e1 = expression; PLUS; e2 = expression { Eplus (e1, e2) }
| ...
```

como no caso de d'`ocamllex`, é necessário assegurar a aplicação de `menhir` antes do cálculo das dependências

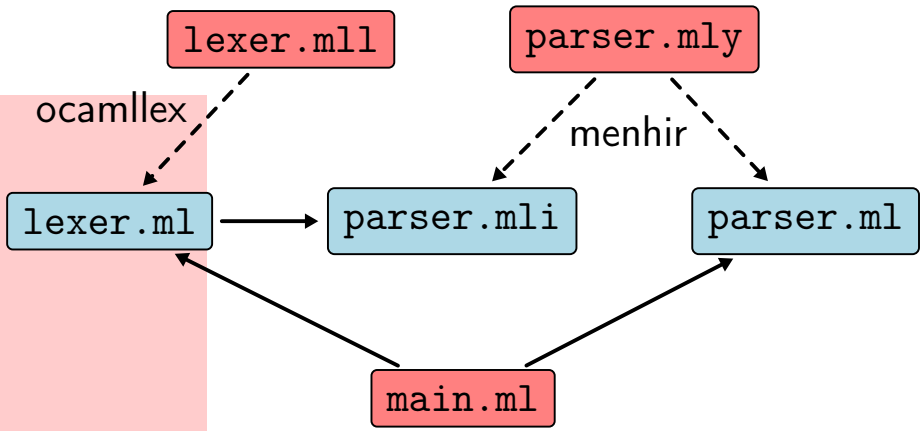
o `Makefile` pode ser algo como :

```
lexer.ml: lexer.mll
        ocamllex lexer.mll

parser.mli parser.ml: parser.mly
        menhir -v parser.mly

.depend: lexer.ml parser.mli parser.ml
        ocamldep *.ml *.mli > .depend

include .depend
```

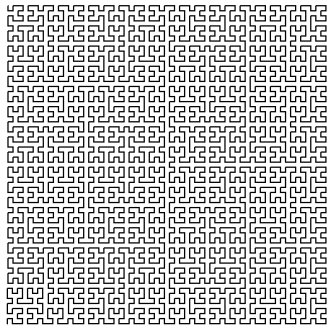
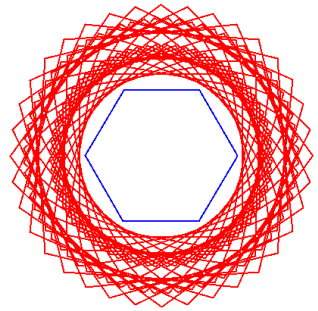
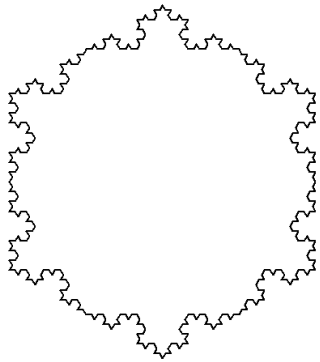
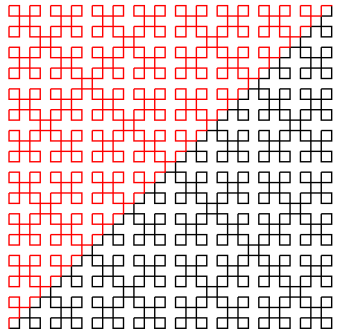


(ver `Makefile` fornecido nas aulas práticas ou então, utilizar `ocamlbuild`)

conclusão

- aulas práticas

utilização prática de ocamllex + menhir
numa pequena linguagem Logo
(tartaruga gráfica)

- aulas teóricas
  - production de codigo

estes acetatos resultam essencialmente de uma adaptação do material pedagógico gentilmente cedido pelo Jean-Christophe Filliâtre (link1, link2)

adicionalmente poderá consultar as obras seguintes

- **Modern Compilers: Principles, Techniques, and Tools**, Alfred V. Aho, Monica S. Lam, Ravi Sethi et Jeffrey D. Ullman
- **Types and Programming Languages**, Benjamin C. Pierce
- **Modern Compiler Implementation**, Andrew W. Appel (3 versões: ML, C, Java)