Universidade da Beira Interior

Desenho de Linguagens de Programação e de Compiladores

Simão Melo de Sousa

Aula 5 - Analise Sintáctica Descendente

não é só o léxico que importa, a sintaxe e a morfologia da frase também

Man eating piranha mistakenly sold as pet fish

Babies are what the mother eats

o objectivos da análise sintáctica é reconhecer as frases pertencendo à sintaxe da linguagem

o que o fluxo de caracteres é para a análise léxica, o fluxo dos lexemas – resultado desta mesma análise – o são para a análise sintáctica

a saída da análise sintáctica é uma árvore de sintaxe abstracta sequência de caracteres

`"fun x -> ( x + 1 )"`

↓

análise léxica

↓

sequência de lexemas

$\boxed{\texttt{fun}}\ \boxed{\texttt{x}}\ \boxed{\texttt{->}}\ \boxed{\texttt{(}}\ \boxed{\texttt{x}}\ \boxed{\texttt{+}}\ \boxed{\texttt{1}}\ \boxed{\texttt{)}}$

sequência de lexemas

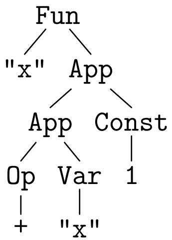
$\boxed{\texttt{fun}}\ \boxed{\texttt{x}}\ \boxed{\texttt{->}}\ \boxed{\texttt{(}}\ \boxed{\texttt{x}}\ \boxed{\texttt{+}}\ \boxed{\texttt{1}}\ \boxed{\texttt{)}}$

↓

análise sintáctica

↓

sintaxe abstracta



em particular, a análise sintáctica deve detectar os erros de sintaxe e

- localisá-los com precisão
- os identificar (na maioria dos casos assinalam-se por « Syntax Error » mas também « missing ending parenthesis », etc.)
- eventualmente, retomar a análise após detecção e identificação de erro para procurar outros potenciais erros

para a análise sintáctica vamos usar

- uma **gramática livre de contexto** (também conhecida como **gramática algébrica**, *context free grammar*, ou ainda CFG) para descrever a sintaxe
- um **autómato com pilha** para a reconhecer

é o análogo das expressões regulares e dos autómatos de estados finitos utilizados na análise léxica

Definição

Uma gramática algébrica é um 4-tuplo (N, T, S, R) *onde*

- N *é um conjunto finito de* ***símbolos não terminais***
- T *é um conjunto finito de* ***símbolos terminais***
- $S \in N$ *é um símbolo de partida/inicial (designado de* ***axioma****)*
- $R \subseteq N \times (N \cup T)^{\star}$ *é um conjunto finito de* ***regras de produção***

$N = \{E\}$, $T = \{+, *, (,), \mathtt{int}\}$, $S = E$,
e $R = \{\, (E, E{+}E), (E, E{*}E), (E, (E)), (E, \mathtt{int}) \,\}$

na prática, apresentamos as regras na forma seguinte:

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

os terminais da gramática são os lexemas produzidos pela análise léxica

`int` designa aqui o lexema correspondendo a uma constante inteira
(*i.e.* a sua natureza, e não o seu valor)

Definição

*Uma palavra $u \in (N \cup T)^\star$ **deriva-se** numa palavra $v \in (N \cup T)^\star$, com a notação $u \rightarrow v$, se existe uma descomposição*

$$u = u_1 X u_2$$

com $X \in N$, $X \rightarrow \beta \in R$ e

$$v = u_1 \beta u_2$$

exemplo :

$$\underbrace{E * (}_{u_1} \; \underbrace{E}_{X} \; \underbrace{)}_{u_2} \quad \rightarrow \quad E * (\; \underbrace{E + E}_{\beta} \;)$$

uma sequência $w_1 \to w_2 \to \cdots \to w_n$ é designada de **derivação**

falamos de **derivação esquerda** (resp. **direita**) se o não-terminal reduzido é sistematicamente o mais a esquerda *i.e.* $u_1 \in T^\star$ (resp. o mais a direita *i.e.* $u_2 \in T^\star$)

nota-se $\to^\star$ o fecho reflexivo transitivo de $\to$

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

em particular

$$E \rightarrow^{\star} \texttt{int} * (\ \texttt{int} + \texttt{int}\ )$$

Definição

*A **linguagem** definida por uma gramática algébrica $G = (N, T, S, R)$ é o conjunto das palavras $T^\star$ derivadas do axioma, i.e.*

$$L(G) = \{\, w \in T^\star \mid S \to^\star w \,\}$$

no exemplo anterior : $\texttt{int} * (\ \texttt{int} + \texttt{int}\ ) \in L(G)$

outro exemplo : para a gramática $S \to aSb \mid \epsilon$, temos

$$L = \{\, a^n b^n \mid n \in \mathbb{N} \,\}$$

(exercício : demonstrar tal resultado)

Definição

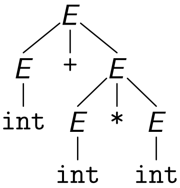
A qualquer derivação $S \rightarrow^{\star} w$, podemos associar uma ***árvore de derivação****, cujos nodos possuem etiquetas definidas da seguinte forma*

- *a raíz é S*
- *as folhas formam a palavra w na ordem infixa*
- *todo o nodo interno X é um não-terminal cujos filhos tem por etiqueta $\beta \in (N \cup T)^{\star}$ com $X \rightarrow \beta$ regra de derivação*

de notar: não confundir com uma árvore de sintaxe abstracta, são coisas diferentes a derivação esquerda

$$E \to E + E \to \texttt{int} + E \to \texttt{int} + E * E \to \texttt{int} + \texttt{int} * E \to \texttt{int} + \texttt{int} * \texttt{int}$$

devolve a árvore de derivação



mas a derivação direita

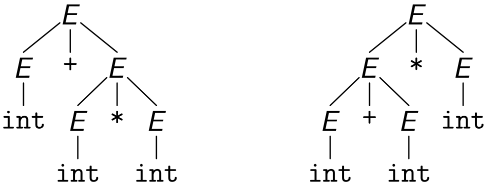
$$E \to E + E \to E + E * E \to E + E * \texttt{int} \to E + \texttt{int} * \texttt{int} \to \texttt{int} + \texttt{int} * \texttt{int}$$

também

Definição

Uma gramática é dita ***ambígua*** *se uma palavra da sua linguagem admite mais do que uma árvore de derivação*

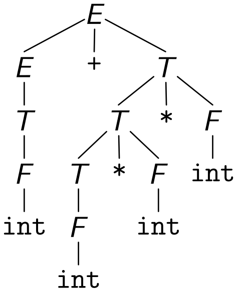
exemplo : a palavra `int + int * int` admite as duas árvores de derivação



é no entanto e neste caso possível propor uma outra gramática, não ambígua, que define a mesma linguagem

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

esta nova gramática traduz a prioridade do produto sobre a soma e a escolha da associatividade esquerda para estas duas operações assim, a palavra `int + int * int * int` só tem doravante uma árvore de derivação, ou seja



que corresponde a derivação esquerda

$$E \rightarrow E + T \rightarrow T + T \rightarrow F + T \rightarrow \texttt{int} + T \rightarrow \texttt{int} + T * F$$
$$\rightarrow \texttt{int} + T * F * F \rightarrow \texttt{int} + F * F * F \rightarrow \texttt{int} + \texttt{int} * F * F$$
$$\rightarrow \texttt{int} + \texttt{int} * \texttt{int} * F \rightarrow \texttt{int} + \texttt{int} * \texttt{int} * \texttt{int}$$

determinar se uma gramática é ou não ambígua **não é decidível**

(lembrete :*decidível* significa que não podemos definir um algoritmo/programa que, para toda a entrada possível, termina e responde correctamente pela positiva ou pela negativa) vamos utilizar **critérios de decisão suficientes** para garantir que uma gramática não é ambígua , e para as quais sabemos também decidir a pertença à linguagem de forma eficiente (com recurso a um autómato de pilha determinista)

as classes de gramáticas definidas por estes critérios chamam-se LL(1), LR(0), SLR(1), LALR(1), LR(1), etc.

execução por um autómato de pilha

reconhecer linguagens regulares = autómatos finitos

reconhecer linguagens livres de contexto = autómatos com pilha

num autómato finito, numa transição entre dois estados como por exemplo: $1 \xrightarrow{a} 2$, passar de 1 para 2 basta que a entrada forneça um a

num autómato com pilha, a diferença é que a transição é executada tendo em conta igualmente um **histórico** das sequências de caracteres já analisados: **pilha = histórico** conjunto dos estados: Q

estado inicial: $q_0 \in Q$

estados finais: $F \subseteq Q$

histórico/pilha: Q^+

função/relação de transição: $\Delta \in Q^+ \times (A \cup \{\epsilon\}) \times Q^*$

princípio:

- A cada estado está associada uma palavra de Q^+: a pilha.
- Uma transição = uma transformação no topo da pilha

consideremos a transição por x de $m_1 \in Q^+$ para $m_2 \in Q^*$ (notação $m_1 \xrightarrow{x} m_2$), então queremos que qualquer que seja a pilha com prefixo m_1, digamos mm_1, seja possível utilizar esta transição de cada vez que a entrada apresente x.

notação $mm_1 \overset{x}{\twoheadrightarrow} mm_2$

extendemos $\twoheadrightarrow$ às palavras de $(A \cup \{\epsilon\})^*$
se $m_1 \overset{\omega_1}{\twoheadrightarrow} m_2$ e $m_2 \overset{\omega_2}{\twoheadrightarrow} m_3$ então $m_1 \overset{\omega_1\omega_2}{\twoheadrightarrow} m_3$

uma palavra ω é **reconhecida** por um autómato com pilha se $q_0 \overset{\omega}{\twoheadrightarrow} mf$ com $f \in F$ no topo da pilha

construção dum autómato com pilha a partir duma CFG

seja $G = (N, T, S, P)$ uma gramática livre de contexto

vamos considerar todos os pares possíveis (Y, α, β), designados de **item**, tais que $Y \in N$, $\alpha, \beta \in (N \cup T)^*$ e $(Y ::= \alpha\beta) \in P$
notação $[Y \to \alpha.\beta]$ (poderemos omitir ocasionalmente os [])

alguns itens particulares:

se $\alpha = \epsilon$, $[Y \to .\beta]$

se $\beta = \epsilon$, $[Y \to \alpha.]$ este item é designado de **completo**

se $\alpha = \beta = \epsilon$, $[Y \to .]$

construção dum autómato com pilha a partir duma CFG

para construir o autómato com pilha correspondente introduzimos, se necessário, um novo símbolo inicial S' e a regra $S' \rightarrow S\sharp$. (Necessário se e a gramática não tiver esta configuração).

o símbolo $\sharp$ significa: marca de fim de buffer de entrada (ou ainda: fim de analise, fim de ficheiro etc...). Assim o buffer de entrada deverá considerar como último símbolo o símbolo $\sharp$

por exemplo:

dada a gramática

E ::= E + E
E ::= E * E
E ::= x

nova versão :

$\boxed{\text{S' ::= E } \sharp}$
E ::= E + E
E ::= E * E
E ::= x

conjunto dos itens:

$$
\begin{array}{llll}
S' ::= .\ E\sharp & E ::= .\ E + E & E ::= .\ E * E & E ::= .\ x \\
S' ::= E.\ \sharp & E ::= E.\ + E & E ::= E.\ * E & E ::= x. \\
S' ::= E\sharp. & E ::= E + .\ E & E ::= E * .\ E & \\
 & E ::= E + E. & E ::= E * E. &
\end{array}
$$

construção dum autómato com pilha a partir duma CFG

tendo uma gramática $G = (T, N, S, P)$, o autómato com pilha define-se então da seguinte forma:

Alfabeto: T.

Estados: Conjunto dos itens associado à gramática G

Transições: $T \cup \{\epsilon\}$.

Estado inicial: $\boxed{S' \to .S\sharp}$
significa: preparada para iniciar o reconhecimento duma frase gerada por S

Estado final (único): $\boxed{S' \to S\sharp.}$
significa: uma frase gerada por S foi reconhecido e a marca de fim de analise foi atingida

3 tipos de transições:

- **expansão**: para cada $Y ::= \alpha$ de P (lembrete: $\alpha \in (N \cup T)^*$).
  $[X \to \beta.\ Y\gamma] \xrightarrow{\epsilon} [X \to \beta.\ Y\gamma][Y \to .\ \alpha]$
- **leitura** de $a \in T$.
  $[X \to \beta.\ a\gamma] \xrightarrow{a} [X \to \beta a.\ \gamma]$
- **redução** da regra $Y ::= \alpha$.
  $[X \to \beta.\ Y\gamma][Y \to \alpha.\ ] \xrightarrow{\epsilon} [X \to \beta Y.\ \gamma]$

reconhecimento de $aabb\sharp$ pela gramática

$$\begin{array}{lll} S & ::= & aSb \\ S & ::= & \epsilon \end{array}$$

$$\begin{array}{rl}
\textit{start} & [S' \to .\ S\sharp] \\
\textit{expansão } S ::= aSb & [S' \to .\ S\sharp][S \to .\ aSb] \\
\textit{leitura } a & [S' \to .\ S\sharp][S \to a.\ Sb] \\
\textit{expansão } S ::= aSb & [S' \to .\ S\sharp][S \to a.\ Sb][S \to .\ aSb] \\
\textit{leitura } a & [S' \to .\ S\sharp][S \to a.\ Sb][S \to a.\ Sb] \\
\textit{expansão } S ::= \epsilon & [S' \to .\ S\sharp][S \to a.\ Sb][S \to a.\ Sb][S \to .] \\
\textit{redução} & [S' \to .\ S\sharp][S \to a.\ Sb][S \to aS.\ b] \\
\textit{leitura } b & [S' \to .\ S\sharp][S \to a.\ Sb][S \to aSb.] \\
\textit{redução} & [S' \to .\ S\sharp][S \to aS.\ b] \\
\textit{leitura } b & [S' \to .\ S\sharp][S \to aSb.] \\
\textit{redução} & [S' \to S.\ \sharp] \\
\textit{leitura } \sharp & [S' \to S\sharp.\ ]
\end{array}$$

apresentar um reconhecimento de $id + id * id\sharp$ com base no autómato com pilha subjacente à gramática seguinte:

$$
\begin{array}{l}
E ::= T \mid E + T \\
T ::= F \mid T * F \\
F ::= id \mid (E)
\end{array}
$$

este processo consegue lidar com todas as gramática livres de contexto, mas

os autómatos gerados por este métodos são enormes (na prática, como vimos no exemplo anterior, nem o desenhamos)

e são igualmente não-deterministas

em termos conceptuais, um reconhecimento sintáctico baseado neste processo é completo mas, em prática, é particularmente ineficiente

que soluções? mais uma vez...visar um subconjunto das gramáticas livres de contexto representativo sobre o qual outros tipos de algoritmos de reconhecimento mais eficientes possam ser definidos: as gramáticas LL e LR

análise descendente

problema: tendo em conta o não determinismo do autómato com pilha, como escolher uma regra de expansão?

uma solução possível: olhar para o buffer de entrada **um ou mais** caracteres **a frente**.

contexto: tomemos por exemplo as produções $Y ::= \alpha_1|\alpha_2|\ldots|\alpha_n$. Se cada α_i começar de forma distinta dos outros α_j $(i \neq j)$ então ao ler um caracter a sabemos imediatamente que regras escolher: a regra $Y ::= \alpha_k$ com $\alpha_k \rightarrow^\star a \cdots$ (a na primeira posição).

$$\begin{array}{llll} 1 & S ::= A\ d\ S & 3 & A ::= a\ A\ b \\ 2 & S ::= b & 4 & A ::= c \end{array}$$

vamos analisar a situação mais cuidadosamente

imaginemos que estamos numa situação em que pretendemos derivar algo de S

- se a entrada começar por a ou c então a única hipótese é escolher a regra 1
- se a entrada começar por b então a única possibilidade é escolher a regra 2
- qualquer outra entrada não pode reconhecida por S

relativamente a A

- se a entrada for a então podemos escolher a 3^{ra} regra
- se a entrada for c então podemos escolher a 4^{ta} regra
- qualquer outra entrada não pode reconhecida por A

$$
\begin{array}{ll}
1 & S ::= A\ d\ S \\
2 & S ::= b \\
3 & A ::= a\ A\ b \\
4 & A ::= c
\end{array}
$$

esta gramática tem então a particularidade de permitir escolher de forma certa a produção para a derivação, e isto com base na **antevisão** de um só caracter na entrada

ou seja, com tal gramática a analise sintáctica é facilitada: em cada situação sabemos exactamente o que fazer entre escolher leituras, expansões de determinadas regras ou reduções

1	$S ::= A\ d\ S$	3	$A ::= a\ A\ b$
2	$S ::= b$	4	$A ::= c$

um exemplo: derivação da palavra $acbdb\sharp$

	Estado	antevisão da entrada	decisão e acção
0	$[.S\sharp]$		**Início**
1	$[.S\sharp]$	a	expansão regra 1
2	$[.S\sharp][.A\ d\ S]$	a	expansão regra 3
3	$[.S\sharp][.A\ d\ S][.a\ A\ b]$	a	leitura a
4	$[.S\sharp][.A\ d\ S][a.\ A\ b]$	c	expansão regra 4
5	$[.S\sharp][.A\ d\ S][a.\ A\ b][.c]$	c	leitura c
6	$[.S\sharp][.A\ d\ S][a.\ A\ b][c.]$	b	redução regra 4
7	$[.S\sharp][.A\ d\ S][aA.\ b]$	b	leitura b
8	$[.S\sharp][.A\ d\ S][a\ A\ b.]$	d	redução da regra 3
9	$[.S\sharp][A.\ d\ S]$	d	leitura d
10	$[.S\sharp][A\ d.\ S]$	b	expansão regra 2
11	$[.S\sharp][A\ d.\ S][.b]$	b	leitura b
12	$[.S\sharp][A\ d.\ S][b.]$	$\sharp$	redução da regra 2
13	$[.S\sharp][A\ d\ S.]$	$\sharp$	redução da regra 1
14	$[S.\sharp]$	$\sharp$	leitura $\sharp$
15	$[S\sharp.]$	ϵ	**sucesso**

esta analise corresponde a derivação esquerda seguinte:

$$S' \Longrightarrow S\sharp \Longrightarrow AdS\sharp \Longrightarrow aAbdS\sharp \Longrightarrow acbdS\sharp \Longrightarrow acbdb\sharp$$

E se juntarmos mais uma regra?

1	$S ::= A\ d\ S$
2	$S ::= b$
3	$A ::= a\ A\ b$
4	$A ::= c$
5	$A ::= \epsilon$

o facto de A poder derivar o ϵ obriga aqui a um cuidado especial: temos de olhar para além de A para saber que regra escolher

relativamente a S

- Se a entrada começar por a, c ou d então a única hipótese é escolher a regra 1.
- Se a entrada começar por b então a única possibilidade é escolher a regra 2
- Qualquer outra entrada não pode reconhecida por S.

relativamente a A

- Se a entrada for a então podemos escolher a 3^{ra} regra.
- Se a entrada for c então podemos escolher a 4^{ta} regra.
- Se a entrada for b ou d então podemos escolher a 5^{ta} regra.

Para poder ter uma analise sintáctica nestes moldes é preciso conseguir determinar para toda a palavra m de $(N \cup T)^*$ (em particular não-terminais)

se esta pode **derivar a palavra vazia**,

que conjunto de caracteres (letras de T) **apareçam em primeira posição** das palavras geradas por m e finalmente

que conjunto de caracteres terminais **apareçam em primeira posição a seguir** a qualquer palavra gerada por m.

estes dados permitirão construir um analisador (descendente) eficiente e determinístico.

ideia : proceder por expansões sucessivas do não terminal o mais a esquerda (construimos assim uma derivação esquerda) partindo de S e servindo-se de uma **tabela** que indica, para um não terminal X por expandir e os k primeiros caracteres da entrada, qual a expansão $X \to \beta$ por realizar
(em inglês falamos de *top-down parsing*)

supomos $k = 1$ no resto da apresentação e notemos $T(X, c)$ esta tabela

na prática, tendo em conta o símbolo terminal $\#$, a tabela indica também as expansões X quando o fim da entrada foi atingido utilizamos uma pilha que é uma palavra de $(N \cup T)^\star$; inicialmente a pilha está reduzida ao símbolo inicial

em cada instante, examinamos o topo da pilha e o primeiro caracter c da entrada

- se a pilha encontra-se vazia, paramos ; há sucesso se e só se $c = \sharp$
- se o topo da pilha é um terminal a, então a deve ser igual a c, tiramos a da pilha (*pop*) e consumimos c da entrada ; senão a analise falha
- se o topo da pilha é um não-terminal X, então substituímos X pela palavra $\beta = T(X, c)$ no topo da pilha: i.e. *pop* seguido de, quando apropriado, *push* para cada letra de β, começando pela última ; senão, a análise falha

consideremos mais uma gramática para as expressões aritméticas
(assumimos implicitamente $S \to E\sharp$)
e a tabela de expansão seguinte

$$
\begin{array}{rcl}
E & \to & T\ E' \\
E' & \to & +\ T\ E' \\
 & | & \epsilon \\
T & \to & F\ T' \\
T' & \to & *\ F\ T' \\
 & | & \epsilon \\
F & \to & (\ E\ ) \\
 & | & \mathtt{int}
\end{array}
$$

	$+$	$*$	(	)	int	#
E			TE'		TE'	
E'	$+TE'$			ϵ		ϵ
T			FT'		FT'	
T'	ϵ	$*FT'$		ϵ		ϵ
F			(E)		int	

(veremos mais adiante como construir tal tabela) ilustremos a análise descendente da palavra

```
int + int * int
```

	$+$	$*$	(	)	int	$\#$
E			TE'		TE'	
E'	$+TE'$			ϵ		ϵ
T			FT'		FT'	
T'	ϵ	$*FT'$		ϵ		ϵ
F			(E)		int	

pilha	entrada
E	$\texttt{int} + \texttt{int} * \texttt{int}\#$
$E'T$	$\texttt{int} + \texttt{int} * \texttt{int}\#$
$E'T'F$	$\texttt{int} + \texttt{int} * \texttt{int}\#$
$E'T'\texttt{int}$	$\texttt{int} + \texttt{int} * \texttt{int}\#$
$E'T'$	$+\texttt{int} * \texttt{int}\#$
E'	$+\texttt{int} * \texttt{int}\#$
$E'T+$	$+\texttt{int} * \texttt{int}\#$
$E'T$	$\texttt{int} * \texttt{int}\#$
$E'T'F$	$\texttt{int} * \texttt{int}\#$
$E'T'\texttt{int}$	$\texttt{int} * \texttt{int}\#$
$E'T'$	$*\texttt{int}\#$
$E'T'F*$	$*\texttt{int}\#$
$E'T'F$	$\texttt{int}\#$
$E'T'\texttt{int}$	$\texttt{int}\#$
$E'T'$	$\#$
E'	$\#$
ϵ	$\#$

programa-se um analisador descendente com alguma facilidade, introduzindo uma função para cada símbolo não-terminal da gramática

cada função examina a entrada e, consoante o caso, consome esta ou invoca recursivamente as funções correspondentes a outros não-terminais, conforme a tabela de expansão

façamos a escolha de uma programação puramente aplicativa,
onde a entrada é uma lista de lexemas do tipo

```
type token = Tplus | Tmult | Tleft | Tright | Tint | Teof
```

vamos assim escrever cinco funções que « consumam » a lista de entradas

```
val e : token list -> token list
val e': token list -> token list
val t : token list -> token list
val t': token list -> token list
val f : token list -> token list
```

e o reconhecimento de uma entrada pode assim fazer-se por

```
let recognize l =
  e l = [Teof]
```

as funções operam por *pattern matching* sobre a entrada em conformidade com a tabela

	+	*	(	)	int	#
E			TE'		TE'	

```
let rec e = function
  | (Tleft | Tint) :: _ as m -> e' (t m)
  | _ -> error ()
```

	+	*	(	)	int	#
E'	$+TE'$			ϵ		ϵ

```
and e' = function
  | Tplus :: m -> e' (t m)
  | (Tright | Teof) :: _ as m -> m
  | _ -> error ()
```

	+	*	(	)	int	#
T			FT'		FT'	

```
and t = function
  | (Tleft | Tint) :: _ as m -> t' (f m)
  | _ -> error ()
```

	+	*	(	)	int	#
T'	ϵ	$*FT'$		ϵ		ϵ

```
and t' = function
  | (Tplus | Tright | Teof) :: _ as m -> m
  | Tmult :: m -> t' (f m)
  | _ -> error ()
```

	+	*	(	)	int	#
F			(E)		int	

```
and f = function
  | Tint :: m -> m
  | Tleft :: m -> begin match e m with
      | Tright :: m -> m
      | _ -> error ()
    end
  | _ -> error ()
```

algumas notas

- a tabela de expensão não está explícita : está embutida no código de cada funções
- a pilha também não : ela é materializada pela pilha de chamadas
- em alternativa à implementação cá exposta, poderíamos tê-las tornadas explícitas no programa

- poderiamos ter preferido uma programação mais imperativa

```
val next_token : unit -> token
```

falta-nos abordar uma questão importante : como construir a tabela de expansão ?

a ideia é simples : para decidir se realizamos uma expansão $X \to \beta$ quando o primeiro caracter da entrada é c, vamos procurar determinar se c faz parte dos **primeiros** caracteres das palavras reconhecidos por β

como já vimos, surge uma dificuldade neste processo quando existam produções de tipo $X \to \epsilon$,
o primeiro caracter que pode ser gerado neste ponto pertence ao conjunto dos caracteres que podem **seguir** X

para determinar os primeiros caracteres e os caracteres seguintes é preciso também determinar se uma palavra se pode derivar em ϵ

Definição (null)

Seja $\alpha \in (T \cup N)^\star$*.* $\mathrm{null}(\alpha)$ *é verdade se e só se podemos derivar* ϵ *partindo de* α *i.e.* $\alpha \rightarrow^\star \epsilon$*.*

Definição (first)

Seja $\alpha \in (T \cup N)^\star$*.* $\mathrm{first}(\alpha)$ *é o conjunto de todos os primeiros terminais das palavras derivadas de* α*, i.e.* $\{a \in T \mid \exists w.\, \alpha \rightarrow^\star aw\}$*.*

Definição (follow)

Seja $X \in N$*.* $\mathrm{follow}(X)$ *é o conjunto de todos os terminais que podem aparecer a seguir a* X *numa derivação, i.e.* $\{a \in T \mid \exists u, w.\, S \rightarrow^\star uXaw\}$*.*

para calcular null(α), basta determinar null(X) para todo o $X \in N$

null(X) é verdade se e só se

- existe uma produção $X \to \epsilon$,
- ou existe uma produção $X \to Y_1 \dots Y_m$ onde null(Y_i) para todo i

(note que é falso em qualquer outra situação, em particular quando há sempre terminais na parte direita das produções)

problema : trata-se de um conjunto de equação mutuamente recursivas

dito de outra forma,
se $N = \{X_1, \dots, X_n\}$ e se $\vec{V} = (\text{null}(X_1), \dots, \text{null}(X_n))$,
vamos procurar a **menor solução** de uma equação da forma

$$\vec{V} = F(\vec{V})$$

Theorema (existência de uma menor ponto fixo (Tarski))

Seja A um conjunto finito equipado de uma relação de ordem $\leq$ e de um menor elemento ε. Toda a função $f : A \rightarrow A$ crescente, i.e. tal que $\forall x, y. x \leq y \Rightarrow f(x) \leq f(y)$, admite um menor ponto fixo.

demonstração :
como ε é o menor elemento, temos $\varepsilon \leq f(\varepsilon)$
sendo f crescente, temos $f^k(\varepsilon) \leq f^{k+1}(\varepsilon)$ para todo k
sendo A finito, existe assim um menor k_0 tal que $f^{k_0}(\varepsilon) = f^{k_0+1}(\varepsilon)$
$a_0 = f^{k_0}(\varepsilon)$ é assim um menor ponto fixo de f

seja b um outro ponto fixo de f
temos $\varepsilon \leq b$ e então $f^k(\varepsilon) \leq f^k(b)$ para todo k
em particular $a_0 = f^{k_0}(\varepsilon) \leq f^{k_0}(b) = b$
a_0 é assim o menor ponto fixo de f $\square$

o teorema de Tarski apresenta condições **suficientes** para a existência do menor ponto fixo, mas não apresenta condições **necessárias** no caso do cálculo de null, temos $A = \text{Bool} \times \cdots \times \text{Bool}$ com $\text{Bool} = \{\texttt{false}, \texttt{true}\}$

podemos equipar Bool da ordem $\texttt{false} \leq \texttt{true}$ e A da ordem *produto*

$$(x_1, \dots, x_n) \leq (y_1, \dots, y_n) \quad \text{se e só se} \quad \forall i.\, x_i \leq y_i$$

o teorema aplica-se então tomando

$$\varepsilon = (\texttt{false}, \dots, \texttt{false})$$

porque a função calculando $\text{null}(X)$ a partir dos $\text{null}(X_i)$ é crescente para calcular os $\text{null}(X_i)$, podemos começar por

$$\text{null}(X_1) = \texttt{false}, \dots, \text{null}(X_n) = \texttt{false}$$

e aplicar as equações até obter o ponto fixo *i.e.* até os valores de $\text{null}(X_i)$ não se alterarem mais

$$
\begin{array}{rcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \\
 & | & \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \\
 & | & \epsilon \\
F & \rightarrow & (\ E\ ) \\
 & | & \texttt{int}
\end{array}
$$

$$
\begin{aligned}
\text{null}(E) &= \text{null}(T) \wedge \text{null}(E') \\
\text{null}(E') &= \text{null}(+\ T\ E') \vee \text{null}(\epsilon) \\
\text{null}(T) &= \text{null}(F) \wedge \text{null}(T') \\
\text{null}(T') &= \text{null}(*\ F\ T') \vee \text{null}(\epsilon) \\
\text{null}(F) &= \text{null}((\ E\ )) \vee \text{null}(int)
\end{aligned}
$$

E	E'	T	T'	F
false	false	false	false	false
false	true	false	true	false
false	true	false	true	false

porque procuramos o **menor** ponto fixo ?

$\Rightarrow$ por recorrência sobre o número de passos do cálculo anterior, mostra-se que se $\mathsf{null}(X) = \mathtt{true}$ então $X \rightarrow^\star \epsilon$

$\Leftarrow$ por recorrência sobre o número de passos da derivação $X \rightarrow^\star \epsilon$, mostra-se que $\mathsf{null}(X) = \mathtt{true}$ pelo cálculo anterior de forma semelhante, as equações que definam first são mutuamente recursivas

$$\mathsf{first}(X) = \bigcup_{X \to \beta} \mathsf{first}(\beta)$$

e

$$\begin{array}{rcl}
\mathsf{first}(\epsilon) & = & \emptyset \\
\mathsf{first}(a\beta) & = & \{a\} \\
\mathsf{first}(X\beta) & = & \mathsf{first}(X), \quad \text{se } \neg\mathsf{null}(X) \\
\mathsf{first}(X\beta) & = & \mathsf{first}(X) \cup \mathsf{first}(\beta), \quad \text{se } \mathsf{null}(X)
\end{array}$$

mais uma vez, procedemos por cálculo de ponto fixo sobre o produto cartesiano $A = \mathcal{P}(T) \times \cdots \times \mathcal{P}(T)$ equipado da ordem produto sobre $\subseteq$ e com $\varepsilon = (\emptyset, \ldots, \emptyset)$

null

E	E'	T	T'	F
`false`	`true`	`false`	`true`	`false`

$$
\begin{array}{rcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \\
 & | & \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \\
 & | & \epsilon \\
F & \rightarrow & (\ E\ ) \\
 & | & \mathtt{int}
\end{array}
$$

$$
\begin{array}{l}
\mathit{first}(E) = \mathrm{first}(T\ E') = \mathrm{first}(T) \\
\mathit{first}(E') = \mathrm{first}(+\ T\ E') \cup \mathrm{first}(\epsilon) = \mathrm{first}(+) \cup \emptyset = \{+\} \\
\mathit{first}(T) = \mathrm{first}(F\ T') = \mathrm{first}(F) \\
\mathit{first}(T') = \mathrm{first}(*\ F\ T') \cup \mathrm{first}(\epsilon) = \mathrm{first}(*) \cup \emptyset = \{*\} \\
\mathit{first}(F) = \mathrm{first}((\ E\ )) \cup \mathrm{first}(\mathtt{int}) = \{(, \mathtt{int}\}
\end{array}
$$

first

E	E'	T	T'	F
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\emptyset$	$\{+\}$	$\emptyset$	$\{*\}$	$\{(, \mathtt{int}\}$
$\emptyset$	$\{+\}$	$\{(, \mathtt{int}\}$	$\{*\}$	$\{(, \mathtt{int}\}$
$\{(, \mathtt{int}\}$	$\{+\}$	$\{(, \mathtt{int}\}$	$\{*\}$	$\{(, \mathtt{int}\}$
$\{(, \mathtt{int}\}$	$\{+\}$	$\{(, \mathtt{int}\}$	$\{*\}$	$\{(, \mathtt{int}\}$

mais uma vez, as equações definindo follow são mutuamente recursivas

$$\text{follow}(X) = \bigcup_{Y \to \alpha X \beta} \text{first}(\beta) \;\cup \bigcup_{Y \to \alpha X \beta,\, \text{null}(\beta)} \text{follow}(Y)$$

procedemos por cálculo de ponto fixo, sobre o mesmo domínio usado para o cálculo de first

nota : é necessário introduzir $\#$ no calculo dos caracteres seguintes do símbolo inicial
(podemos fazê-lo directamente ou então juntando a regra $S' \to S\#$)

$$
\begin{array}{rcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \\
 & | & \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \\
 & | & \epsilon \\
F & \rightarrow & (\ E\ ) \\
 & | & \texttt{int}
\end{array}
$$

null

E	E'	T	T'	F
false	true	false	true	false

first

E	E'	T	T'	F
$\{(, \texttt{int}\}$	$\{+\}$	$\{(, \texttt{int}\}$	$\{*\}$	$\{(, \texttt{int}\}$

$$
\begin{aligned}
\text{follow}(E) &= \{\#\} \cup \text{first}(\ )\ ) = \{\#,)\} \\
\text{follow}(E') &= \text{first}(\epsilon) \cup \text{first}(\epsilon) \cup \text{follow}(E) \cup \text{follow}(E') \\
&= \text{follow}(E) \cup \text{follow}(E') \\
\text{follow}(T) &= \text{first}(E') \cup \text{follow}(E) \cup \text{follow}(E') \\
&= \{+\} \cup \text{follow}(E) \cup \text{follow}(E') \\
\text{follow}(T') &= \text{first}(\epsilon) \cup \text{follow}(T) \cup \text{follow}(T') \\
&= \text{follow}(T) \cup \text{follow}(T') \\
\text{follow}(F) &= \text{first}(T') \cup \text{follow}(T) \cup \text{follow}(T') \\
&= \{*\} \cup \text{follow}(T) \cup \text{follow}(T')
\end{aligned}
$$

follow

E	E'	T	T'	F
$\{\#\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\{\#,)\}$	$\{\#\}$	$\{+, \#\}$	$\emptyset$	$\{*\}$
$\{\#,)\}$	$\{\#,)\}$	$\{+, \#,)\}$	$\{+, \#\}$	$\{*, +, \#\}$
$\{\#,)\}$	$\{\#,)\}$	$\{+, \#,)\}$	$\{+, \#,)\}$	$\{*, +, \#,)\}$
$\{\#,)\}$	$\{\#,)\}$	$\{+, \#,)\}$	$\{+, \#,)\}$	$\{*, +, \#,)\}$

com base nestes três conceitos, construímos a tabela de expansão $T(X, a)$ da forma seguinte

para cada produção $X \to \beta$,

- consideramos $T(X, a) = \beta$ para todo o $a \in \text{first}(\beta)$
- se $\text{null}(\beta)$, consideramos também $T(X, a) = \beta$ para todo o $a \in \text{follow}(X)$

$$
\begin{array}{rcl}
E & \to & T\ E' \\
E' & \to & +\ T\ E' \\
 & | & \epsilon \\
T & \to & F\ T' \\
T' & \to & *\ F\ T' \\
 & | & \epsilon \\
F & \to & (\ E\ ) \\
 & | & \mathtt{int}
\end{array}
$$

first

E	E'	T	T'	F
$\{(,\mathtt{int}\}$	$\{+\}$	$\{(,\mathtt{int}\}$	$\{*\}$	$\{(,\mathtt{int}\}$

follow

E	E'	T	T'	F
$\{\#,)\}$	$\{\#,)\}$	$\{+,\#,)\}$	$\{+,\#,)\}$	$\{*,+,\#,)\}$

	$+$	$*$	$($	$)$	$\mathtt{int}$	$\#$
E			TE'		TE'	
E'	$+TE'$			ϵ		ϵ
T			FT'		FT'	
T'	ϵ	$*FT'$		ϵ		ϵ
F			(E)		$\mathtt{int}$	

Definição (gramática LL(1))

Uma gramática é dita LL(1) se, na sua tabela de expansão, há no máximo uma produção em cada célula.

LL significa « **L**eft to right scanning, **L**eftmost derivation » nem todas as gramáticas livres de contexto são LL(1), como tal é frequente ser necessário transformar uma gramática para obter uma gramática equivalente que seja LL(1)

vejamos alguns **critérios suficientes** para saltar fora do âmbito das gramáticas LL(1) já sabemos que são gramáticas que permitam uma analise descendente eficiente e simples

sabemos igualmente $LL(1) \iff$ uma entrada no máximo em cada célula da tabela de expansão

mas não haverá uma forma informal ou intuitiva de caracterizar tais gramáticas? Sim:

- é possível prever, sempre, que regra escolher só com base nos k primeiros caracteres.
- ou seja, para $k = 1$ e um não-terminal A tal que $A \to \alpha_1 | \alpha_2 | \ldots | \alpha_n$ e para i e j distintos :
    1. no máximo um dos α_i é tal que $\text{null}(\alpha_i) = \mathit{true}$
    2. $\text{first}(\alpha_i) \cap \text{first}(\alpha_j) = \emptyset$
    3. $\exists i.\text{null}(\alpha_i) = \mathit{true} \implies \forall j.\text{first}(\alpha_j) \cap \text{follow}(A) = \emptyset$

uma gramática **recursiva esquerda (direta)**, *i.e.* contendo uma produção da forma

$$X \to X\alpha$$

nunca será LL(1)

torná-la LL(1) passará sempre por remover a recursividade esquerda (direta ou indireta)

de igual forma é necessário factorizar as produções que começam pelo mesmo prefixo (**fatorização esquerda**)

suficiente?
em geral, não

como proceder então?
completar a transformação da gramática consoante diagnósticos feitos às anomalias presentes na tabela de transição (soluções *Ad-Hoc*)

por exemplo a redução da gramatica por remoção de não-terminais.

o que é a recursividade esquerda?

uma gramática é recursiva esquerda se existir uma derivação tal $X \rightarrow^{+} X\alpha$

uma gramática recursiva esquerda não pode ser $LL(1)$

se pretendemos obter uma gramática $LL(1)$ então é preciso eliminar a recursividade esquerda.

situação:

$$\begin{array}{lcl} A & \rightarrow & A\,\alpha_1 \mid A\,\alpha_2 \mid \cdots \mid A\,\alpha_n \\ A & \rightarrow & \beta_1 \mid \beta_2 \mid \cdots \mid \beta_m \qquad\qquad \text{os } \beta_i \text{ não começam por } A \end{array}$$

transformar em

$$\begin{array}{lcl} A & \rightarrow & \beta_1\,A' \mid \beta_2\,A' \mid \cdots \mid \beta_m\,A' \\ A' & \rightarrow & \alpha_1\,A' \mid \alpha_2\,A' \mid \cdots \mid \alpha_n\,A' \\ A' & \rightarrow & \epsilon \end{array}$$

caso particular: caso surgem regras do tipo $A \rightarrow A$, estas regras podem ser removidas consideremos esta gramática:

$$\begin{array}{lcl} S & \rightarrow & A\ a \\ S & \rightarrow & b \\ A & \rightarrow & A\ c \\ A & \rightarrow & S\ d \end{array}$$

$S \rightarrow^{+} S\ d\ a$

como proceder para remover esta recursividade esquerda?

organizar os não-terminais de 1 a n (no caso de $|N| = n$): $A_1, A_2, \ldots, A_n$

mas: **a ordem escolhida tem a sua importância!** Preferir atribuir os índices menores aos não-terminais do fundo da gramática e o índice final ao símbolo inicial

processo em n etapas:

etapa 1: Eliminar a recursividade directa de A_1

etapa i:

- enquanto existir uma produção $A_i \to A_j\alpha$ (com $i > j$) olhar para todas as produções tais que $A_j \to \beta$ e substituir $A_i \to A_j\alpha$ por $A_i \to \beta\alpha$
- retirar então a recursividade esquerda directa (quando $i = j$)
- no final, se β começa por um A_k então necessariamente $k > i$.

o processo iterativo anterior só acaba quando temos produções $A_i \to A_j\alpha$ tal que $i < j$

$$\begin{array}{lcl} S & \rightarrow & A\ a \\ S & \rightarrow & b \\ A & \rightarrow & A\ c \\ A & \rightarrow & S\ d \end{array}$$

ordem: S, A (contrária à sugestão feita...)

comecemos por S: $\boxed{S \rightarrow A\ a \quad S \rightarrow b}$

para A, em $A \rightarrow S\ d$ substituímos S por $A\ a$ <u>e</u> b logo:
$A \rightarrow A\ c \quad A \rightarrow A\ a\ d \quad A \rightarrow b\ d$

podemos então eliminar a recursividade esquerda directa:

$$\boxed{\begin{array}{ll} & A' \rightarrow c\ A' \\ A \rightarrow b\ d\ A' & A' \rightarrow a\ d\ A' \\ & A' \rightarrow \epsilon \end{array}}$$

conflito por prefixo comum

$$
\begin{array}{rcl}
A & \rightarrow & \underline{\alpha}\beta_1 \\
A & \rightarrow & \underline{\alpha}\beta_2 \\
& \vdots & \\
A & \rightarrow & \underline{\alpha}\beta_n
\end{array}
$$

neste caso transformar tais regras em:

$$
\begin{array}{rcl}
A & \rightarrow & \underline{\alpha}A' \\
A' & \rightarrow & \beta_1 \\
A' & \rightarrow & \beta_2 \\
& \vdots & \\
A' & \rightarrow & \beta_n
\end{array}
$$

remoção prévia de Não-Terminais

pode acontecer que duas produções de um não terminal Y tenham conjuntos first não disjuntos sem no entanto apresentarem explicitamente prefixo comum

pode assim ser interessante tentar explicitar estes prefixos e proceder à transformação anterior

$$Y \quad \rightarrow \quad \beta\underline{X}\gamma \qquad (X \neq Y)$$

neste caso substituir tal regra pelas regras:

$$Y \quad \rightarrow \quad \beta\underline{\alpha}\gamma$$

para todo α tal que $(X \rightarrow \alpha) \in P$.

proceder em seguida à remoção de factorização esquerda caso necessário

conclusão

os analisadores LL(1) são relativamente simples de implementar
(ver aulas práticas)

mas obrigam à escrita de gramáticas que podem não serem *naturais*

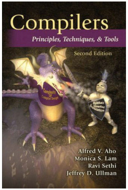
iremos ver na próxima aula um outro tipo de solução

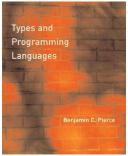
- aulas práticas
  - analisadores LL(1)

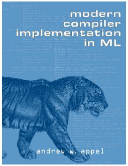
- aulas teóricas
  - análise sintáctica (segunda parte)

leituras de referência

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)