Interpretation and Compilation of Languages

Master Programme in Computer Science

Mário Pereira mjp.pereira@fct.unl.pt

Nova School of Science and Technology, Portugal

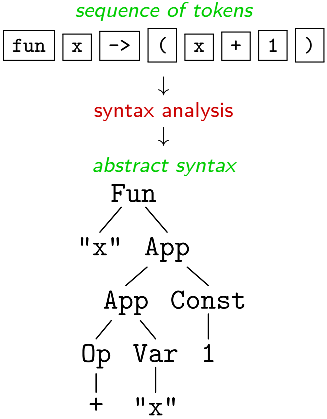
April 1, 2025

Lecture 4

based on lectures by Jean-Christophe Filliâtre and Léon Gondelman
previous editions by João Costa Seco, Luís Caires, and Bernardo Toninho

The Goal of Parsing

Is to recognize the phrases belonging to the syntax of language.
It's input is the flow of tokens constructed by lexical analysis.
It's output is an abstract syntax tree.



sequence of tokens

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

↓

syntax analysis

↓

abstract syntax

Syntax Errors

Additionally, syntax analysis must detect syntax errors and

- signal them with a position in the source
- explain them (most often limited to “syntax error” but also “unclosed parenthesis”, etc.)
- possibly resume the analysis to discover further errors

Which Tools?

To implement syntax analysis, we are using

- a context-free grammar to define the syntax
- a pushdown automaton to recognize it

This is similar to regular expressions / finite automata used for the lexical analysis.

Today: Syntactic Analysis (Parsing), part 1

1. context-free grammars (recall)
2. top-down parsing algorithm
   - how the algorithm works
   - constructing the expansion table (key ingredient)

Context-free Grammar

Definition

A context-free grammar is a tuple (N, T, S, R) *where*

- N *is a finite set of nonterminal symbols*
- T *is a finite set of terminal symbols*
- $S \in N$ *is the start symbol (the axiom)*
- $R \subseteq N \times (N \cup T)^\star$ *is a finite set of production rules*

Example 1: Arithmetic Expressions

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

In practice, we write production rules as follows:

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

The terminals are the tokens produced by the lexical analysis.

Here `int` stands for an integer literal token (*i.e.*, its nature, not its value).

Example 2: a^nb^n

Words are of the form a^nb^n with $n \in \mathbb{N}$

$N = \{S\},\ T = \{a, b\}$ and $R = \{(S, aSb), (S, \epsilon)\}$

$$\begin{array}{rcl} S & \rightarrow & aSb \\ & \mid & \epsilon \end{array}$$

Words recognized by the grammar: ϵ, ab, $aabb$, ...
Words not recognized by the grammar: a, b, aab, $aabbbb$, ...

Example 3: Words With $|a| = |b|$

Words over alphabet $\{a, b\}$ with the same number of a's and b's:

$N = \{S\}$, $T = \{a, b\}$ and
$R = \{\,(S, \epsilon), (S, aAS), (S, bBS), (A, aAA), (A, b), (B, bBB), (B, a)\}$

$$\begin{array}{rcl} S & \rightarrow & \epsilon \\ & \mid & a\ A\ S \\ & \mid & b\ B\ S \\ A & \rightarrow & a\ A\ A \\ & \mid & b \\ B & \rightarrow & b\ B\ B \\ & \mid & a \end{array}$$

Words recognized by the grammar: ϵ, *ab*, *abbaba*, ...
Words not recognized by the grammar: *a*, *b*, *aab*, *bbba*, ...

Derivation

Definition

A word $u \in (N \cup T)^\star$ is derived into a word $v \in (N \cup T)^\star$, which we write $u \to v$, if there is a decomposition

$$u = u_1 X u_2$$

with $X \in N$, $X \to \beta \in R$, and

$$v = u_1 \beta u_2$$

Example:

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

Derivation

A sequence $w_1 \to w_2 \to \cdots \to w_n$ is called a derivation.

It is a leftmost derivation (resp. rightmost) if the nonterminal X is the leftmost in each of w_i, *i.e.*, $u_1 \in T^\star$ (resp. the rightmost *i.e.* $u_2 \in T^\star$) where $w_i = u_1 X u_2$ for some u_1 and u_2.

We note $\to^\star$ the reflexive transitive closure of $\to$.

Example

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

In particular, we get

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

Language

Definition

The language defined by a context-free grammar $G = (N, T, S, R)$ is the set of words from $T^\star$ that are derived from the axiom, i.e.

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

In our example

$$\texttt{int * (int + int)} \in L(G)$$

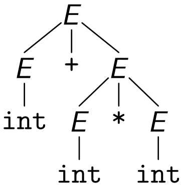
Derivation Tree

Definition

A derivation tree is a tree whose nodes are labeled with grammar symbols, such that

- *the root is the axiom S*
- *any internal node X is a nonterminal whose subnodes are labeled by $\beta \in (N \cup T)^\star$ with $X \to \beta$ a production rule*

Example:



Careful: this is different from the abstract syntax tree.

Observations

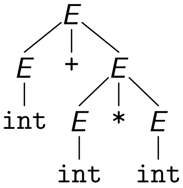
- Derivation trees describe acceptance of the input by a given grammar and can feature concrete syntax (e.g. parentheses).
- For a derivation tree whose leaves form the word w in infix order, it is clear that there exists $S \rightarrow^\star w$.
- Conversely, any derivation $S \rightarrow^\star w$ corresponds to a derivation tree whose leaves form the word w in infix order.
- The notion of derivation tree helps to characterize whether a grammar is ambiguous or not, which is important for building parsers.

Example

The leftmost derivation

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

Corresponds to the derivation tree



And so is the rightmost derivation

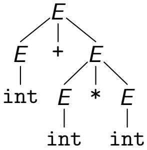
$$E \rightarrow E + E \rightarrow E + E * E \rightarrow E + E * \texttt{int} \rightarrow E + \texttt{int} * \texttt{int} \rightarrow \texttt{int} + \texttt{int} * \texttt{int}$$

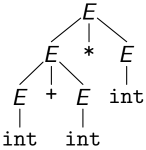
Ambiguity

Definition

A context-free grammar is ambiguous if at least one word accepts several derivation trees.

Example: the word `int + int * int` accepts two derivation trees





and thus our grammar is ambiguous:

- the parser can't know it should choose the left tree to build the AST
- if it chooses the right tree, it will build an AST with a wrong meaning

Non-ambiguous Grammar

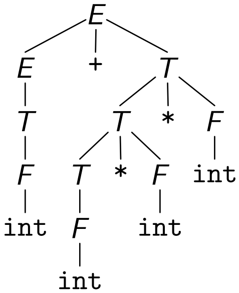
It is possible to propose another grammar, that is not ambiguous and that defines the same language.

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

This new grammar reflects the priority of multiplication over addition, and the choice of a left associativity for these two operations.

Non-ambiguous Grammar

Now, the word `int + int * int * int` has a single derivation tree,



corresponding to this leftmost derivation

$$
\begin{aligned}
E &\rightarrow E + T \rightarrow T + T \rightarrow F + T \rightarrow \texttt{int} + T \rightarrow \texttt{int} + T * F \\
&\rightarrow \texttt{int} + T * F * F \rightarrow \texttt{int} + F * F * F \rightarrow \texttt{int} + \texttt{int} * F * F \\
&\rightarrow \texttt{int} + \texttt{int} * \texttt{int} * F \rightarrow \texttt{int} + \texttt{int} * \texttt{int} * \texttt{int}
\end{aligned}
$$

Negative Result

Whether a context-free grammar is ambiguous is not decidable.

(*reminder*: decidable means that we can write a program that, for any input, terminates and outputs yes or no)

Approach

We are going to use decidable sufficient criteria to ensure that a grammar is not ambiguous, and for which we know how to decide membership efficiently (using a pushdown automaton).

The corresponding grammar classes defined by those criteria are called LL(1), LR(0), SLR(1), LALR(1), LR(1), etc.

LL	:	Left-to-right, leftmost derivation
LR	:	left-to-right, rightmost derivation
SLR	:	simple left-to-right, rightmost derivation
LALR	:	look-ahead, left-to-right, rightmost derivation

Top-down Parsing

General Idea

Proceed by successive expansions of the leftmost non-terminal (thus constructing a left derivation).

- starting from S and
- using a table $T(X, \overrightarrow{c})$ that, for a non-terminal X to be expanded and the first k characters of the input $\overrightarrow{c}$, indicates the expansion $X \rightarrow \beta$ to be carried out.

(This is referred to as *top-down parsing*).

Suppose $k = 1$ subsequently, and let $T(X, c)$ denote this table.
That is why we will refer to this technique as LL(1) grammars.

In practice, we assume that a terminal symbol # denotes the end of the input, and thus the table also indicates the expansion of X when the end of the input is reached.

How it Works

We use a stack, which is a word from $(N \cup T)^\star$

Initially, the stack is reduced to the start symbol and at each step we scan the top of the stack and the first character c of the input:

- if the stack is empty, we stop and accept the input if and only if c is $\#$
- if the top of the stack is a terminal a, then a must be equal to c. We pop a from the stack and consume c; otherwise, we fail
- if the top of the stack is a non-terminal X, then we replace X by the word $\beta = T(X, c)$ on the top of the stack, by pushing the characters of β starting from the last one; otherwise, we fail.

Example

Let's rewrite the grammar of the arithmetic expressions and take a look on the following expansion table:

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

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

(we'll see in a moment how to construct such tables)

Example

But first let's illustrate the top-down analysis of the input

```
int + int * int
```

Note: the elements in the stack are presented in reverse order.

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

stack $\mapsto$	input
E	`int+int*int#`
$E'T$	`int+int*int#`
$E'T'F$	`int+int*int#`
$E'T'$`int`	`int+int*int#`
$E'T'$	`+int*int#`
E'	`+int*int#`
$E'T+$	`+int*int#`
$E'T$	`int*int#`
$E'T'F$	`int*int#`
$E'T'$`int`	`int*int#`
$E'T'$	`*int#`
$E'T'F*$	`*int#`
$E'T'F$	`int#`
$E'T'$`int`	`int#`
$E'T'$	`#`
E'	`#`
ϵ	`#`

Example

But first let's illustrate the top-down analysis of the input

`int + int * int`

Note: the elements in the stack are presented in reverse order.

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

stack $\mapsto$	input
E	`int+int*int`#
$E'T$	`int+int*int`#
$E'T'F$	`int+int*int`#
$E'T'$`int`	`int+int*int`#
$E'T'$	`+int*int`#
E'	`+int*int`#
$E'T+$	`+int*int`#
$E'T$	`int*int`#
$E'T'F$	`int*int`#
$E'T'$`int`	`int*int`#
$E'T'$	`*int`#
$E'T'F*$	`*int`#
$E'T'F$	`int`#
$E'T'$`int`	`int`#
$E'T'$	#
E'	#
ϵ	#

Example

But first let's illustrate the top-down analysis of the input

```
int + int * int
```

Note: the elements in the stack are presented in reverse order.

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

stack ↦	input
E	int+int*int#
$E'T$	int+int*int#
$E'T'F$	int+int*int#
$E'T'$int	int+int*int#
$E'T'$	+int*int#
E'	+int*int#
$E'T+$	+int*int#
$E'T$	int*int#
$E'T'F$	int*int#
$E'T'$int	int*int#
$E'T'$	*int#
$E'T'F*$	*int#
$E'T'F$	int#
$E'T'$int	int#
$E'T'$	#
E'	#
ϵ	#

Example

But first let's illustrate the top-down analysis of the input

```
int + int * int
```

Note: the elements in the stack are presented in reverse order.

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

stack ↦	input
E	`int+int*int#`
$E'T$	`int+int*int#`
$E'T'F$	`int+int*int#`
$E'T'$`int`	`int+int*int#`
$E'T'$	`+int*int#`
E'	`+int*int#`
$E'T+$	`+int*int#`
$E'T$	`int*int#`
$E'T'F$	`int*int#`
$E'T'$`int`	`int*int#`
$E'T'$	`*int#`
$E'T'F*$	`*int#`
$E'T'F$	`int#`
$E'T'$`int`	`int#`
$E'T'$	`#`
E'	`#`
ϵ	`#`

Example

But first let's illustrate the top-down analysis of the input

```
int + int * int
```

Note: the elements in the stack are presented in reverse order.

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

stack ↦	input
E	`int+int*int#`
$E'T$	`int+int*int#`
$E'T'F$	`int+int*int#`
$E'T'$`int`	`int+int*int#`
$E'T'$	`+int*int#`
E'	`+int*int#`
$E'T+$	`+int*int#`
$E'T$	`int*int#`
$E'T'F$	`int*int#`
$E'T'$`int`	`int*int#`
$E'T'$	`*int#`
$E'T'F*$	`*int#`
$E'T'F$	`int#`
$E'T'$`int`	`int#`
$E'T'$	`#`
E'	`#`
ϵ	`#`

Example

But first let's illustrate the top-down analysis of the input

```
int + int * int
```

Note: the elements in the stack are presented in reverse order.

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

stack $\mapsto$	input
E	`int+int*int#`
$E'T$	`int+int*int#`
$E'T'F$	`int+int*int#`
$E'T'$`int`	`int+int*int#`
$E'T'$	`+int*int#`
E' _	`+int*int#`
$E'T+$	`+int*int#`
$E'T$	`int*int#`
$E'T'F$	`int*int#`
$E'T'$`int`	`int*int#`
$E'T'$	`*int#`
$E'T'F*$	`*int#`
$E'T'F$	`int#`
$E'T'$`int`	`int#`
$E'T'$	`#`
E'	`#`
ϵ	`#`

Implementing a Top-down Parser

A top-down parser can be implemented quite easily by introducing a function for each non-terminal of the grammar.

Each function scans the input and, depending on the case as described above, either consumes the input or recursively calls the functions corresponding to other non-terminals, according to the expansion table.

Implementing a Top-down Parser

Let’s opt for a purely functional programming approach, where the input is a list of tokens of the type:

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

we will therefore construct five functions that “consume” the input list:

```
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
```

and the recognition of an input can then be done like this

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

Implementing a Top-down Parser

The functions proceed by pattern matching on the input and follow the table:

	+	*	(	)	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 ()
```

Implementing a Top-down Parser

	+	*	(	)	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 ()
```

Implementing a Top-down Parser

	+	*	(	)	int	#
F			(E)		int	

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

Implementing a Top-down Parser

Note that in our implementation

- the expansion table is *implicit*: it is within the code of each function
- the stack is also *not explicit*: it is implemented by the *call stack*

- they could have been made explicit
- alternatively, a more imperative programming approach could have been chosen

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

One important question remains: how to construct the expansion table systematically for a given grammar?

Expansion Table

Constructing the Expansion Table

The idea itself is simple: in order to decide whether to perform the expansion $X \to \beta$ when the first character of the input is c, we will try to determine if c is among the first characters of the words recognized by β.

But there is a small catch: a difficulty arises for a production such as $Y \to \epsilon$, and in that case, it's necessary to consider also the set of characters that can follow Y (e.g, it could be that $\beta = Y\beta'$ for some β' so the first of β' are also the first of β).

... but for a rule $Z \to \gamma$, it could be that $Z \to^{\star} \epsilon$ (i.e. indirectly via other rules), so determining the first and follow sets also requires determining if an expansion γ can derive ϵ, i.e. have to compute the sets of null.

Definitions

Definition (NULL)

Let $\alpha \in (T \cup N)^\star$. $\textsc{null}(\alpha)$ *holds if and only if we can derive* ϵ *from* α *i.e.* $\alpha \to^\star \epsilon$.

Definition (FIRST)

Let $\alpha \in (T \cup N)^\star$. $\textsc{first}(\alpha)$ *is the set of all terminals starting words derived from* α, *i.e.* $\{a \in T \mid \exists w. \alpha \to^\star aw\}$.

Definition (FOLLOW)

Let $X \in N$. $\textsc{follow}(X)$ *is the set of all terminals that may appear after* X *in a derivation, i.e.* $\{a \in T \mid \exists u, w. S \to^\star uXaw\}$.

Computing $\textsc{null}$, $\textsc{first}$, and $\textsc{follow}$

To compute $\textsc{null}(\alpha)$, we simply need to compute $\textsc{null}(X)$ for $X \in N$

$\textsc{null}(X)$ holds if and only if

- there exists a production $X \to \epsilon$,
- or there exists a production $X \to Y_1 \dots Y_m$ where $\textsc{null}(Y_i)$ for all i

issue: this is a set of mutually recursive equations

said otherwise, if $N = \{X_1, \dots, X_n\}$ and if $\vec{V} = (\textsc{null}(X_1), \dots, \textsc{null}(X_n))$, we look for the least fixpoint to an equation such as

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

Fixpoint Computation

Theorem (existence of a least fixpoint (Tarski))

Let A be a finite set with an order relation $\leq$ and a least element ε. Any monotonically increasing function $f : A \rightarrow A$, i.e., such that $\forall x, y. x \leq y \Rightarrow f(x) \leq f(y)$, has a least fixpoint.

(worry not dear students, this is just for you to know that mathematical magic will take good care of problems)

Computing NULL

To compute NULL, we have
$A = \textsc{Bool} \times \cdots \times \textsc{Bool}$ with $\textsc{Bool} = \{\texttt{false}, \texttt{true}\}$

We can equip $\textsc{Bool}$ with order $\texttt{false} \leq \texttt{true}$ and A with point-wise order

$$(x_1, \ldots, x_n) \leq (y_1, \ldots, y_n) \quad \text{if and only if} \quad \forall i. x_i \leq y_i$$

the theorem applies with

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

since computing $\textsc{null}(X)$ from $\textsc{null}(X_i)$ is monotonic

Computing NULL

To compute $\textsc{null}(X_i)$, we thus start with

$$\textsc{null}(X_1) = \texttt{false}, \ldots, \textsc{null}(X_n) = \texttt{false}$$

and we use the equations until we get a fixpoint *i.e.* until the values $\textsc{null}(X_i)$ do not change anymore

Example

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

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

Example

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

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

Example

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

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

We have attained a fixpoint,
this is the result for $\text{NULL}(X_i)$.

Explanation

Why do we seek for a least fixpoint?

$\Rightarrow$ by induction on the number of steps of the fixpoint computation, we show that if $\textsc{null}(X) = \texttt{true}$ then $X \rightarrow^{\star} \epsilon$

(soundness)

$\Leftarrow$ by induction on the number of steps of derivation $X \rightarrow^{\star} \epsilon$, we show that $\textsc{null}(X) = \texttt{true}$ in the previous computation

(completeness)

Computing FIRST

Similarly, the equations defining FIRST are mutually recursive

$$\text{FIRST}(X) = \bigcup_{X \to \beta} \text{FIRST}(\beta)$$

and

$$\begin{array}{rcl}
\text{FIRST}(\epsilon) & = & \emptyset \\
\text{FIRST}(a\beta) & = & \{a\} \\
\text{FIRST}(X\beta) & = & \text{FIRST}(X), \quad \text{if } \neg\text{NULL}(X) \\
\text{FIRST}(X\beta) & = & \text{FIRST}(X) \cup \text{FIRST}(\beta), \quad \text{if } \text{NULL}(X)
\end{array}$$

Again, we compute a least fixpoint using Tarski's theorem, with $A = \mathcal{P}(T) \times \cdots \times \mathcal{P}(T)$, point-wise ordered with $\subseteq$, and with $\varepsilon = (\emptyset, \ldots, \emptyset)$

Example

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

NULL

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

FIRST

E	E'	T	T'	F
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Example

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

NULL

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

FIRST

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

Example

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

NULL

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

FIRST

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

Example

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

NULL

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

FIRST

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

Example

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

NULL

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

FIRST

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

We have attained a fixpoint,
this is the result for FIRST(X_i).

Computing FOLLOW

Again, the equations defining FOLLOW are mutually recursive

$$\text{FOLLOW}(X) = \bigcup_{Y \to \alpha X \beta} \text{FIRST}(\beta) \; \cup \bigcup_{Y \to \alpha X \beta,\, \text{NULL}(\beta)} \text{FOLLOW}(Y)$$

We compute a least fixpoint, using the same domain as for FIRST

Remark: we add a special symbol $\#$ in $\text{FOLLOW}(S)$, which we can do directly, or by adding a rule $S' \to S\#$ to the grammar

Example

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

NULL

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

FIRST

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

FOLLOW

E	E'	T	T'	F
$\{\#\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Example

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

NULL

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

FIRST

E	E'	T	T'	F
{(, int}	{+}	{(, int}	{*}	{(, int}

FOLLOW

E	E'	T	T'	F
{#}	∅	∅	∅	∅
{#,)}	{#}	{+, #}	∅	{*}

Example

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

NULL

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

FIRST

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

FOLLOW

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

Example

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

NULL

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

FIRST

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

FOLLOW

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

Example

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

NULL

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

FIRST

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

FOLLOW

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

Building the Expansion Table

With $\textsc{first}$ and $\textsc{follow}$, we build the expansion table $T(X, a)$ as follows.

For each rule $X \to \beta$ of the grammar,

- define $T(X, a) = \beta$ for every $a \in \textsc{first}(\beta)$
- if $\textsc{null}(\beta)$, then also define $T(X, a) = \beta$ for every $a \in \textsc{follow}(X)$

Example

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

FIRST

E	E'	T	T'	F
{(, int}	{+}	{(, int}	{*}	{(, int}

FOLLOW

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

	+	*	(	)	int	#
E			TE'		TE'	

Example

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

FIRST

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

FOLLOW

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

	+	*	(	)	int	#
E			TE'		TE'	
E'	$+TE'$			ϵ		ϵ

Example

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

FIRST

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

FOLLOW

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

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

Example

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

FIRST

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

FOLLOW

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

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

Example

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

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)		$\mathtt{int}$	

LL(1) Grammar

Definition (LL(1) Grammar)

A grammar is said to be LL(1) if, in the expansion table, there is at most one production in each cell.

The grammar from the previous slide is LL(1).

LL stands for "Left to right scanning, Leftmost derivation".

Counter-example (1/2)

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

FIRST

E	T	F
$\{(, \texttt{int}\}$	$\{(, \texttt{int}\}$	$\{(, \texttt{int}\}$

	+	*	(	)	int	#
E			$E+T / T$		$E+T / T$	
T			$T*F / F$		$T*F / F$	
F			(E)		int	

Left-recursion Grammar

A left-recursion grammar, *i.e.*, containing a production of the form

$$\begin{array}{rcl} X & \rightarrow & X\alpha \\ & \mid & \beta \end{array}$$

will never be LL(1).

Indeed, the FIRST would be the same for these two productions (no matter the word β).

Solution

One needs to suppress the left recursion, for instance

$$\begin{array}{rcl} X & \rightarrow & \beta\, X' \\ X' & \rightarrow & \alpha\, X' \\ & | & \epsilon \end{array}$$

Counter-example (2/2)

Also, if a grammar contains

$$\begin{array}{rcl} X & \to & \mathtt{a}\,\alpha \\ & | & \mathtt{a}\,\beta \end{array}$$

it will never be LL(1).

The problem is, again, the FIRST would be the same for both productions.

Solution

One needs to suppress productions that start with same terminal (left factorization)

$$\begin{array}{rcl} X & \rightarrow & \mathtt{a}\, X' \\ X' & \rightarrow & \alpha \\ & | & \beta \end{array}$$

Conclusion

LL(1) parsers are relatively simple to write.

However, they require writing somewhat unnatural grammars.

We will turn to another solution next week.

However...

Many compilers use top-down hand-written analyzers.

Examples :

- `javac` (≈ 3 kloc of Java code)
- `rustc` (≈ 16 kloc of Rust code)
- `gcc` (≈ 25 kloc of C++ code)

Bibliography

- Compilers, Principles, Techniques, and Tools, Alfred Aho and Monica S. Lam and Ravi Sethi and Jeffrey D. Ullman, Second Edition[Chapter 4.1-4.4], Addison-Wesley (2006) (*"the dragon book"*).