Compiler

Parsing

Jyun-Ao Lin

iFIRST & CSIE, NTUT
jalin@ntut.edu.tw

Credits

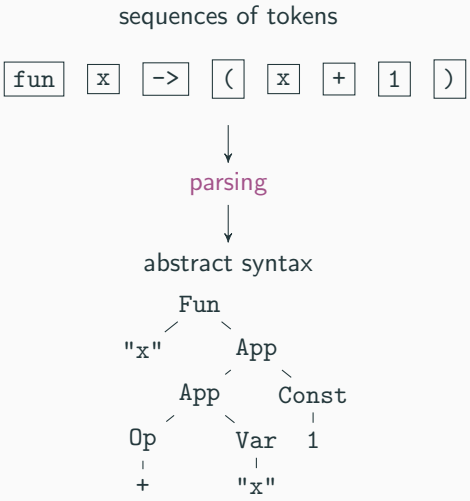
A large part of this course is based on the Compilation Course of J.-C. Filliâtre at ENS Ulm.

Parsing

The goal of parsing is to recognize sentences that belong to the syntax of the language

Its input is the flow of tokens constructed by lexical analysis, its output is an abstract syntax tree

Lexical analysis: example



Syntax errors

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 "non-closed parenthesis", etc.)
- possibly resume the analysis to discover further errors

Which tool?

To implement syntax analysis, we are using

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

Similar to regular expressions / finite automata used in lexical analysis

Overview of the course

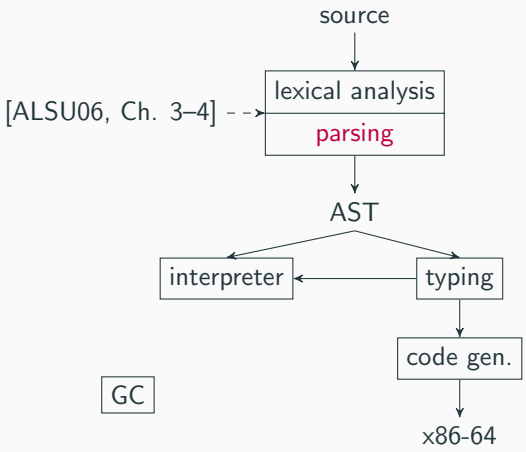


Table of contents

Grammars

Context-free grammar

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

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

Example: 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)

Derivation

Defn. (derivation)
A word $u \in (N \cup T)^*$ derives to a word $v \in (N \cup T)^*$, denoted by $u \to v$, if there exists 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

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

Derivation

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

It is called left derivation (resp. right) if the reduced non-terminal is systemically the leftmost, i.e. $u_1 \in T^*$ (resp. the rightmost $u_2 \in T^*$)

We denote by $\overset{*}{\to}$ the reflexive, transitive closure of $\to$.

Example

Example
Left derivation:

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

Then we have (in particular, but not uniquely)

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

Language

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

$$L(G) = \{w \in T^* \mid S \xrightarrow{*} w\}$$

□

Example
In the above example, we have

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

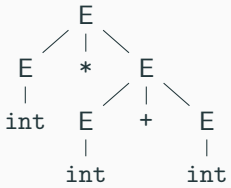
Derivation tree

Defn. (derivation tree)
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 non-terminal whose subnodes are labeled by $\beta \in (N \cup T)^*$ with $X \to \beta$ a production rule
- leaves are terminal symbols

□

Example



Careful: this is different from the abstract syntax tree

Derivation tree

For a derivation tree in whose leaves form the word w in infix order, it is clear that we have $S \xrightarrow{*} w$

Conversely, to any derivation $S \xrightarrow{*} w$, we can associate a derivation tree whose leaves form the word w in infix order

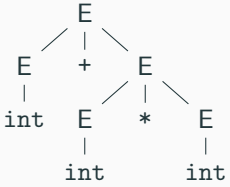
Idea: the derivation tree captures a whole set of derivations that we wish to identify

Example

The left derivation

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

gives the derivation tree



but the right derivation

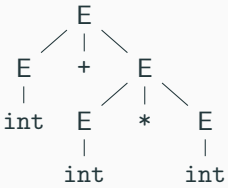
$$E \to E \texttt{ + } E \to E \texttt{ + } E \texttt{ * } E \to E \texttt{ + } E \texttt{ * int} \to E \texttt{ + int * int} \to \texttt{int + int * int}$$

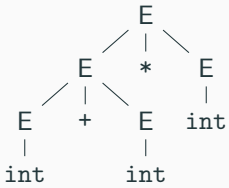
too

Ambiguous

Defn. (ambiguous)
A grammar is called ambiguous if at least a word admits several derivation trees. □

Example
The word `int + int * int` admits two derivation trees





and thus our grammar is ambiguous

Non-ambiguous grammar

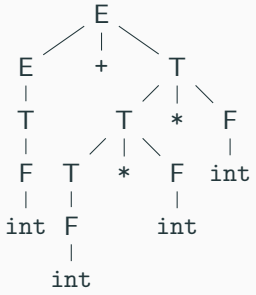
It is possible to propose another grammar, that is not ambiguous and that defines the same language

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

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 the left derivation

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

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 are called
LR(0), SLR(1), LALR(1), LR(1), LL(1), etc.

Bottom-up parsing

Main idea

- scan the input from left to right, and
- look for right-hand sides of production rules to build the derivation tree from bottom to top (*bottom-up parsing*)

Principle

The parser uses a stack that is a word of $(T \cup N)^*$

At each step, two actions can be performed

- a shift operation: we read a terminal from the input and we push it into the stack
- a reduce operation: the top of the stack is the right-hand side β of a production $X \to \beta$, and we replace β with X on the stack

Initially, the stack is empty

When no more action can be performed, the input is recognized if it was read entirely and if the stack is limited to the axiom S

Example

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

stack	input	action
ϵ	`int+int*int`	shift
`int`	`+int*int`	reduce $F \rightarrow$ `int`
F	`+int*int`	reduce $T \rightarrow F$
T	`+int*int`	reduce $E \rightarrow T$
E	`+int*int`	shift
E`+`	`int*int`	shift
E`+int`	`*int`	reduce $F \rightarrow$ `int`
E`+`F	`*int`	reduce $T \rightarrow F$
E`+`T	`*int`	shift
E`+`T`*`	`int`	shift
E`+`T`*int`		reduce $F \rightarrow$ `int`
E`+`T`*`F		reduce $T \rightarrow T*F$
E`+`T		reduce $E \rightarrow E+T$
E		success

LR parser (Knuth, 1965)

How to choose between shift and reduce?

Using an automaton and considering the first k tokens of the input;
this is called LR(k) analysis (LR means "Left to right scanning, Rightmost derivation")

In practice $k = 1$
i.e. we only consider the first token to take the decision

LR parsing

The stack looks like

$$s_0 x_1 s_1 x_2 \dots x_n s_n$$

where s_i is the state of the automation and $x_i \in T \cup N$ as before.

let a be the first token from the input; we look in the action table for state s_n and character a

- if success or failure, we stop
- if shift, we push a and then the target state of the transition $s_n \xrightarrow{a} s$ into the stack
- if reduce rule $X \to \alpha$, with α of length p, then we have α on top of the stack

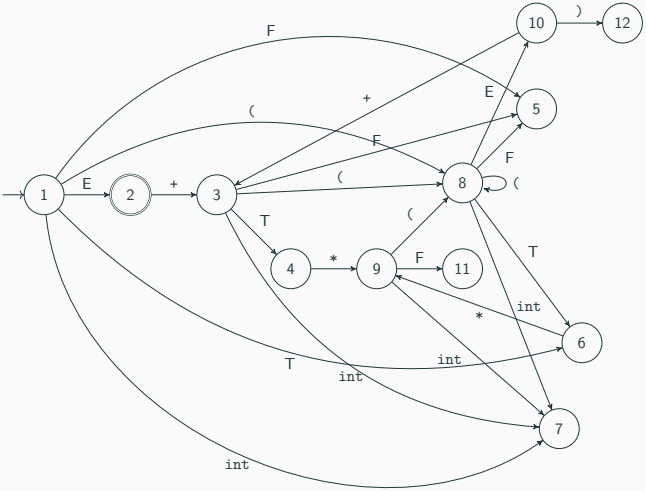
  $$s_0 x_1 s_1 \dots x_{n-p} s_{n-p} | \alpha_1 s_{n-p+1} \dots \alpha_p s_n$$

  we pop it and we push $X\ s$, where s is the target state of the transition $s_{n-p} \xrightarrow{X} s$, i.e.

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

Example

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



LR table

In practice, we do not work with automation but with two tables

- an action table with states as rows and terminals as columns; the box $\texttt{action}(s, a)$ indicates
  - $\texttt{shift}\ s'$ for a shift and a new state s'
  - $\texttt{reduce}\ X \to \alpha$ for a reduction
  - a success
  - a failure
- a transition table having states for rows and non-terminals for columns; the box $\texttt{goto}(s, X)$ indicates the state resulting from a reduction of X

End of input

We also add a special token, denoted by #, which designates the end of the input

One can view it as adding it as a new non-terminal S (which becomes an axiom) and a new rule

$$\begin{array}{lcl} S & \rightarrow & E\# \\ E & \rightarrow & \ldots \\ \vdots & & \end{array}$$

Example

$$
\begin{array}{llll}
1 & E & \to & E + T \\
2 & & | & T \\
3 & T & \to & T * F \\
4 & & | & F \\
5 & F & \to & (E) \\
6 & & | & \texttt{int}
\end{array}
$$

$\texttt{s}i = \texttt{shift}\ i$

$\texttt{r}j = \texttt{reduce}\ j$

	action						goto		
state	(	)	+	*	int	#	E	T	F
1	s8				s7		2	6	5
2			s3			success			
3	s8				s7			4	5
4		r1	r1	s9		r1			
5		r4	r4	r4		r4			
6		r2	r2	s9		r2			
7		r6	r6	r6		r6			
8	s8				s7		10	6	5
9	s8				s7				11
10		s12	s3						
11		r3	r3	r3		r3			
12		r5	r5	r5		r5			

Example of execution

stack	input	action
1	int+int*int#	s7
1 int 7	+int*int#	$F \rightarrow$ int, g5
1 F 5	+int*int#	$T \rightarrow F$, g6
1 T 6	+int*int#	$E \rightarrow T$, g2
1 E 2	+int*int#	s3
1 E 2 + 3	int*int#	s7
1 E 2 + 3 int 7	*int#	$F \rightarrow$ int, g5
1 E 2 + 3 F 5	*int#	$T \rightarrow F$, g4
1 E 2 + 3 T 4	*int#	s9
1 E 2 + 3 T 4 * 9	int#	s7
1 E 2 + 3 T 4 * 9 int 7	#	$F \rightarrow$ int, g11
1 E 2 + 3 T 4 * 9 F 11	#	$T \rightarrow T*F$, g4
1 E 2 + 3 T 4	#	$E \rightarrow E+F$, g2
1 E 2	#	success

Automation

Bottom-up parsing is powerful but computing the tables is complex

We have tools to automate the process

This is the big family of `yacc`, `bison`, `ocamlyacc`, `cup`, `menhir`, . . .
(YACC means *Yet Another Compiler Compiler*)

The tool Menhir

Menhir

Menhir is a tool that transforms a grammar into an OCaml parser
Menhir is based on an LR(1) parsing

Each production rule of the grammar is accompanied by a semantic action
i.e. an OCaml code building a semantic value
(typically an abstract syntax tree)

Menhir is used in conjunction with a lexical analyzer
(typically ocamllex)

Structure

A Menhir file has the suffix .mly and has the following structure

```
%{
    ... arbitrary OCaml code ...
%}
... declaration of tokens ...
... declaration of precedents and associativity ...
... declaration of entry points ...

%%
non-terminal-1:
| production { action }
| production { action }
;
non-terminal-2:
| production { action }
...
%%
    ... arbitrary OCaml code ...
```

Minimal example

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

phrase:
| e = expression; EOF { e }
;

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

Example

We compile the `arith.mly` file in the following way

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

We obtain pure OCaml code in arith.ml(i), which contains in particular

- the declaration of a token type

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

- for each non-terminal declared with `%start`, a function of the type

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

  as we can see, this function takes as an argument a lexical analyzer, of the type produced by `ocamllex`

ocamllex + menhir

When we combine ocamllex and menhir

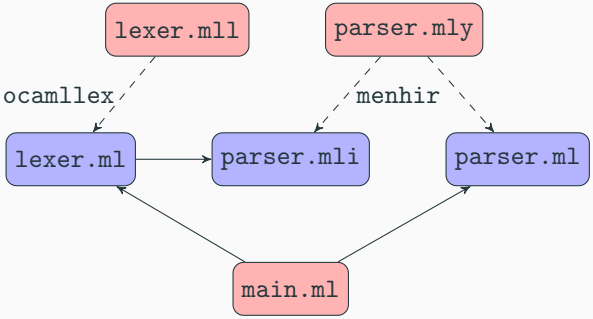
- lexer.mll refers to the tokens defined in parser.mly

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

- The lexical analyzer and the parser are combined as follows:

```
let c = open_in file in
let lb = Lexing.from_channel c in
let e = Parser.phrase Lexer.token lb in
...
```

ocamllex + menhir



= user source
= built automatically
= dependence

Conflicts

When the grammar is not LR(1), `Menhir` shows the conflicts to the user

- the file `.automaton` contains the LR(1) automaton (more later), with conflicts listed
- the file `.conflicts` contains an explanation for each conflict, as a sequence of tokens leading to two distinct derivation trees

Example

On the above grammar, Menhir reports a conflict

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

The file arith.automaton contains in 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 expressi
-- On PLUS reduce production expression -> expression PLUS expressi
-- On EOF reduce production expression -> expression PLUS expressio
** Conflict on PLUS
```

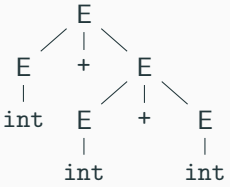
Example

The file arith.conflicts contains a clear explanation

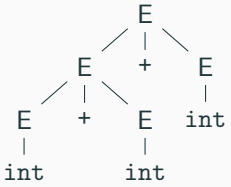
```
** Conflict (shift/reduce) in state 6.
** Token involved: PLUS
** This state is reached from phrase after reading:

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
```

Example

In other word, the question is to choose between



and



Conflict resolution

One way to resolve conflicts is to tell `Menhir` how to choose between shift and reduction

For this, we can give priorities to tokens and productions, and rules of associativity

By default, the priority of a production is that of its rightmost token (but it can be specified explicitly)

Conflict resolution

If the priority of the production is higher than that of the tokens to be read,
then reduction is favored

Conversely, if the priority of the tokens is higher,
then shifting is favored

In case of equality, associativity is consulted: a left-associative tokens favors reduction and a right-associative tokens favors shift

Example

In our example, it is enough to indicate for example that PLUS is left-associative

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

Priorities

To associate priorities with tokens, we use the following convention:

- the order of declaration of the associativity sets the priorities (the first tokens have the lowest priorities)
- Several tokens can appear on the same line, thus having the same priority

Example:

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

A great classic

The following grammar contains a conflict

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

(known in English as *dangling else*)

Explanation, resolution

It corresponds to the situation

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

To associate the ELSE with the closest THEN, it is necessary to favor shift

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

Menhir assets

Menhir offers many advantages over traditional tools such as ocamlyacc :

- non-terminals parameterized by (non-)terminals
  - in particular, easy to write regular expressions (`E?`, `E*` ,`E+`) and lists with separator
- explanation of conflicts
- interactive mode
- LR(1) parsing, where most tools only offer LALR(1)

read the Menhir manual! (accessible from the course page)

Localization

For the following phases of the analysis (typically typing) can localize error messages, it is advisable to keep a localization information in the abstract syntax tree

`Menhir` provides this information in `$startpos` and `$endpos`, two value of the type `Lexing.position`; this information was transmitted to it by the lexical analyzer

Be careful: `ocamllex` automatically maintains only the absolute position in the file;
to have the line and column numbers up to date, it is necessary to call `Lexing.new_line` for each carriage return

Localization

One way to keep location information in the abstract syntax tree is as follows

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

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

Each node is thus decorated by a localization

Localization

The grammar can therefore look like this

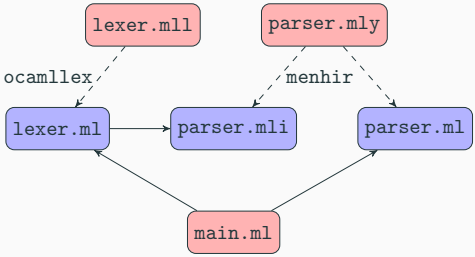
```
expression:
| d = desc { { desc = d; loc = $startpos, $endpos } }
;

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

Compilation and dependencies

As in the case of ocamllex, it is necessary to ensure the application of menhir before calculating the dependencies

If we use dune, we indicate the presence of a menhir file:

```
(ocamllex
  (modules lexer))
(menhir
  (flags --explain --dump)
  (modules parser))
(executable
  (name minilang)
...
```



Construction of the automation and the table

Definitions

Defn. (NULL)
Let $\alpha \in (N \cup T)^*$. $\mathrm{NULL}(\alpha)$ holds if and only if we can derive ϵ from α, ie, $\alpha \xrightarrow{*} \epsilon$ □

Defn. (FIRST)
Let $\alpha \in (N \cup T)^*$. $\mathrm{FIRST}(\alpha)$ is the set of all terminals starting words derived from α, ie,
$\{a \in T \mid \exists w.\alpha \xrightarrow{*} aw\}$ □

Defn. (FOLLOW)
Let $X \in N$. $\mathrm{FOLLOW}(X)$ is the set of all terminals that may appear after X in a derivation, ie,
$\{a \in T \mid \exists u, w.S \xrightarrow{*} uXaw\}$ □

Computing NULL, FIRST and FOLLOW

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

$\mathrm{NULL}(X)$ 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 $\mathrm{NULL}(Y_i)$ for all i

Issue: this is a set of mutually recursive equations

In other words, if $N = \{X_1, \dots, X_n\}$ and if $\vec{V} = (\mathrm{NULL}(X_1), \dots, \mathrm{NULL}(X_n))$, we look for the least fixpoint to an equation such as

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

Fixpoint computation

Thm. (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 $f : A \to A$, ie, such that $\forall x, y. x \leq y \implies f(x) \leq f(y)$, admits a *least fixpoint*. □

Proof.
As ϵ is the smallest element, we have $\epsilon \leq f(\epsilon)$.
The function f being increasing, therefore we have $f^k(\epsilon) \leq f^{k+1}(\epsilon)$ for every k.
Since A is a finite set, there exists a smallest k_0 such that $f^{k_0}(\epsilon) = f^{k_0+1}(\epsilon)$.
We have just found a fixed point $a_0 := f^{k_0}(\epsilon)$ of f.

Let b be another fixed point of f. We have $\epsilon \leq b$ and hence $f^k(\epsilon) \leq f^k(b)$ for every k. In particular, $a_0 = f^{k_0}(\epsilon) \leq f^{k_0}(b) = b$. Thus the fixed point a_0 is the smallest fixed point of f. ■

Computing NULL

To compute NULL, we have
$A = \mathrm{BOOL} \times \cdots \times \mathrm{BOOL}$ with $\mathrm{BOOL} = \{\texttt{true}, \texttt{false}\}$.

We can equip BOOL with the 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 then applies by taking

$$\epsilon = (\texttt{false}, \ldots, \texttt{false}).$$

since the function calculating $\mathrm{NULL}(X)$ from $\mathrm{NULL}(X_i)$ is increasing.

Computing NULL

To compute $\mathrm{NULL}(X_i)$, we thus start with

$$\mathrm{NULL}(X_1) = \texttt{false}, \ldots, \mathrm{NULL}(X_n) = \texttt{false}$$

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

Example

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

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

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 $\mathrm{NULL}(X) = \texttt{true}$ then $X \xrightarrow{*} \epsilon$

$\Leftarrow$ by induction on the number of steps of derivation $X \xrightarrow{*} \epsilon$,
we show that $\mathrm{NULL}(X) = \texttt{true}$ in the previous computation

Computing FIRST

Similarly, the equations defining FIRST are mutually recursive

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

and

$$\begin{aligned}
\mathrm{FIRST}(\epsilon) &= \emptyset \\
\mathrm{FIRST}(a\beta) &= \{a\} \quad \text{if } a \in T \\
\mathrm{FIRST}(X\beta) &= \begin{cases} \mathrm{FIRST}(X) & \text{if } \neg \mathrm{NULL}(X) \\ \mathrm{FIRST}(X) \cup \mathrm{FIRST}(\beta) & \text{if } \mathrm{NULL}(X) \end{cases}
\end{aligned}$$

Again, we compute a least fixpoint using Tarski's theorem, with

$$A = 2^T \times \cdots \times 2^T$$

equipped with point-wise ordered with $\subseteq$ and with $\epsilon = (\emptyset, \ldots, \emptyset)$

Example

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

NULL

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

FIRST

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

Computing FOLLOW

Again, the equations defining FOLLOW are mutually recursive

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

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

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

Example

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

NULL

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

FIRST

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

FOLLOW

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

Exercise

Compute NULL, FIRST, FOLLOW for the grammar LISP

$$
\begin{array}{lcl}
S & \rightarrow & E\ \# \\
E & \rightarrow & \texttt{sym} \\
 & | & (L) \\
L & \rightarrow & \epsilon \\
 & | & E\ L
\end{array}
$$

Exercise

Compute NULL, FIRST, FOLLOW for the grammar of the terms of λ-calculus

$$
\begin{array}{lcl}
E & \rightarrow & T\ \# \\
T & \rightarrow & \texttt{nat} \\
 & | & (T\ T) \\
 & | & \texttt{lam}\ T
\end{array}
$$

LR(0) automation

Let us use $k = 0$ for the moment

We start by constructing an asynchronous automata

that is, an automata containing ϵ-transitions $s_1 \xrightarrow{\epsilon} s_2$.

LR(0) automation

The states are *items* of the form

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

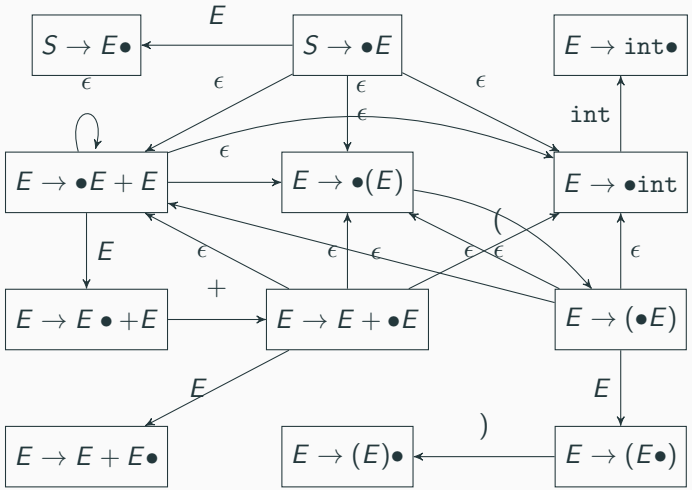
where $X \to \alpha\beta$ is a production of the grammar; the intuition is
"We want to recognize X, we have already seen α and we still have to see β"

The transitions are labeled by $N \cup T$ and are as follows

$$[Y \to \alpha \bullet a\beta] \xrightarrow{a} [Y \to \alpha a \bullet \beta]$$
$$[Y \to \alpha \bullet X\beta] \xrightarrow{X} [Y \to \alpha X \bullet \beta]$$
$$[Y \to \alpha \bullet X\beta] \xrightarrow{\epsilon} [Y \to \bullet\gamma] \quad \text{for all production } X \to \gamma$$

Example

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



Deterministic LR(0) automation

Let us determinize the LR(0) automation

For this, we group the states connected by ϵ-transitions

The states of the deterministic automaton are therefore sets of items, such that

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

Deterministic LR(0) automation

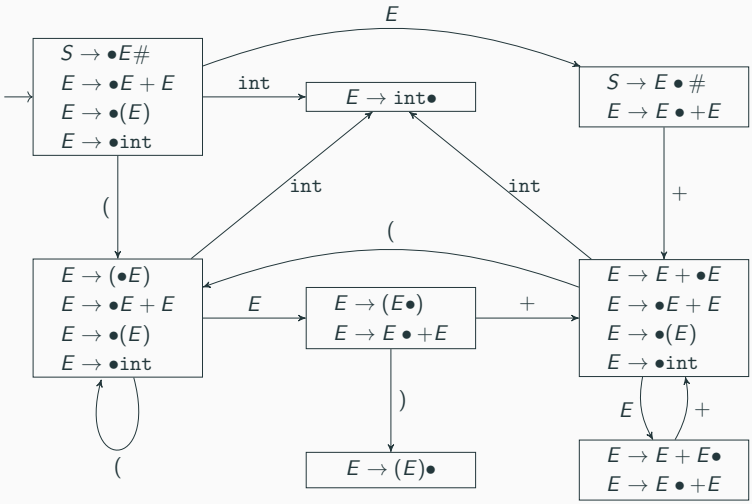
By construction, each state is saturated by the property

$$\begin{aligned} \text{if} \quad & Y \to \alpha \bullet X\beta \in s \\ \text{and if} \quad & X \to \gamma \text{ is a production} \\ \text{then} \quad & X \to \bullet\gamma \in s \end{aligned}$$

The initial state is the one containing $S \to \bullet E \ \#$

Example

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



Building the LR(0) table

For the table `action`, we set

- $\texttt{action}(s, \#) =$ success if $[S \to E \bullet \#] \in s$;
- $\texttt{action}(s, a) = \texttt{shift}\ s'$ if there is a transition $s \xrightarrow{a} s'$;
- $\texttt{action}(s, a) = \texttt{reduce}\ X \to \beta$ if $[S \to \beta\bullet] \in s$ for each a;
- fail in all the other cases.

For the table `goto`, we set

- $\texttt{goto}(s, X) = s'$ if and only if there is a transition $s \xrightarrow{X} s'$.

Example

In our example, the table is of follows:

	action					*goto*
state	(	)	+	int	#	E
1	shift 4			shift 2		3
2	reduce $E \to$ int					
3			shift 6		success	
4	shift 4			shift 2		5
5		shift 7	shift 6			
6	shift 4			shift 2		8
7	reduce $E \to (E)$					
8			shift 6			
	reduce $E \to E + E$					

Conflicts

The LR(0) table can contain several possible actions in the same box. We call this a conflict. There are two kinds of conflicts:

- a conflict shift/reduce if in a state s we can perform a shift but also a reduction;
- a conflict reduce/reduce, if in a state s two different reductions are possible.

Defn. (LR(0) grammar)
A grammar is said to be LR(0) if the table thus constructed does not contain any conflicts. □

Conflicts

In our example, there is a conflict in state 8

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

This conflict precisely illustrates the ambiguity of grammar on a word such as $\texttt{int} + \texttt{int} + \texttt{int}$, after reading $\texttt{int} + \texttt{int}$.

We can resolve the conflict in two ways:

- either we favor the shift, then translating an associativity on the right;
- either we favor the reduction, then translating a left associativity.

Exercise

Show that the grammar for the terms of the λ-calculus is LR(0)

SLR(1) parsing

The construction of LR(0) generates conflicts too easily
We will therefore seek to limit the reductions

A very simple idea is to set $\texttt{action}(s, a) = \texttt{reduce}\ X \to \beta$ if and only if

$$[X \to \beta\bullet] \in s \quad \text{and} \quad a \in \text{FOLLOW}(X).$$

Defn. (the class SLR(1))
A grammar is called SLR(1) if the constructed table does not contain any conflict. (SLR stands for *simple LR*.) □

Example

The grammar

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

is SLR(1)

Exercise: Verify it (the automata contains 12 states).

Exercise: Show that the grammar for LISP is SLR(1).

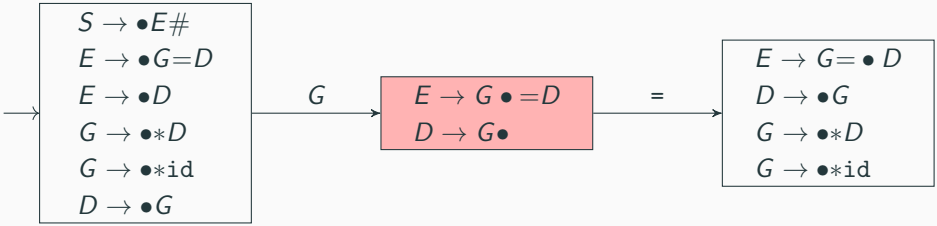
Limitation of SLR(1) parsing

In practice, SLR(1) grammars are not powerful enough

Example

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

	=	
1	...	...
2	shift 3 reduce $D \to G$	...
3	$\vdots$	$\ddots$



LR(1) parsing

We introduce a larger class of grammars, LR(1), with larger tables

items now look like

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

and the meaning is "we want to recognize X , we have already seen α, we still need to see β and then to check that the next token is a"

LR(1) parsing

The non-deterministic LR(1) automaton has transitions

$$[Y \to \alpha \bullet a\beta, b] \xrightarrow{a} [Y \to \alpha a \bullet \beta, b]$$
$$[Y \to \alpha \bullet X\beta, b] \xrightarrow{X} [Y \to \alpha X \bullet \beta, b]$$
$$[Y \to \alpha \bullet X\beta, b] \xrightarrow{\epsilon} [Y \to \bullet\gamma, c] \quad \text{for each } c \in \mathrm{FIRST}(\beta b)$$

the initial state is that containing $[S \to \bullet\alpha, \#]$

As before, we can determinize the automaton and construct the corresponding table; we introduce a reduction action for (s, a) only when s contains an item of the form $[X \to \alpha\bullet, a]$

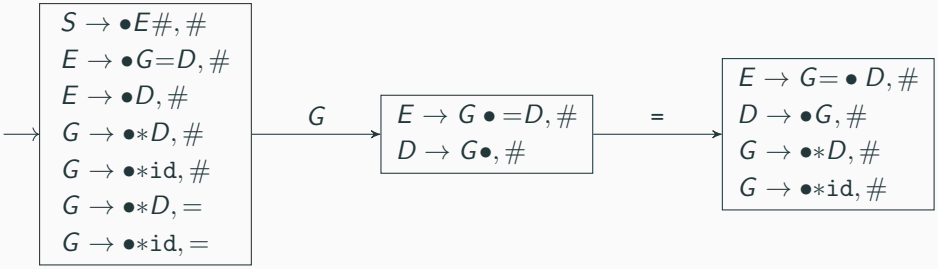
Defn. (the class LR(1))
A grammar is said to be LR(1) if the resulting table contains no conflict. □

Example

Example

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

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



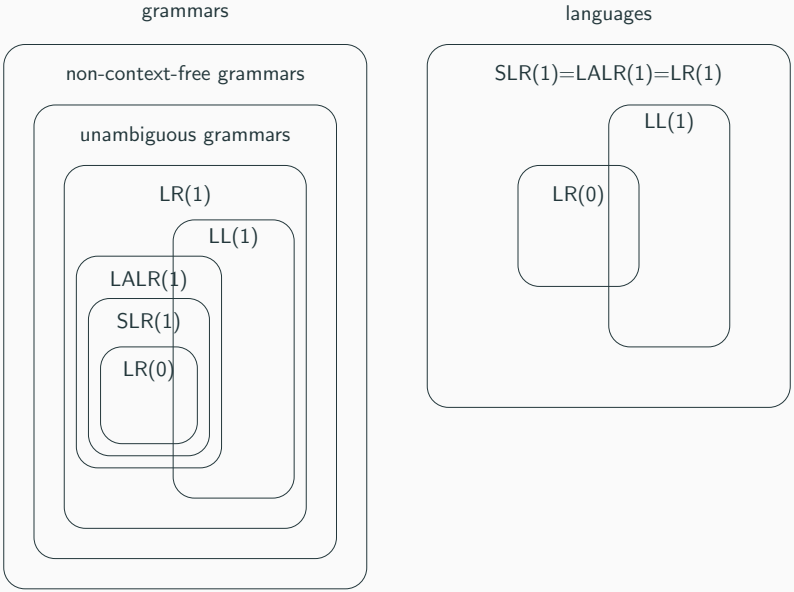
LALR(1) parsing

The LR(1) tables can be large, so we introduced approximations

The class LALR(1) (*lookahead LR*) is such an approximation, used in tools of the yacc family

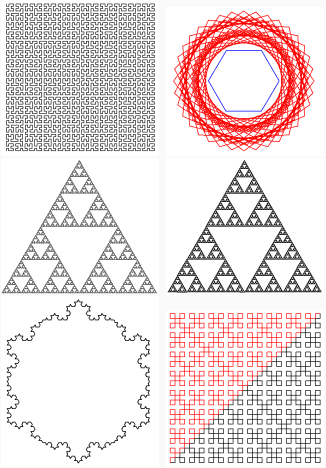
For more details, see Compilers ("the dragon book") by A. Aho, R. Sethi, J. Ullman, [ALSU06, section 4.7]

Grammar hierarchies



Next

- HW4
  - syntax analysis of mini-Turtle
  - OCaml
- more parsing



Localization

Localization

The ocamllex tool maintains, in the Lexing.lexbuf type structure, the current position in the source text that is analyzed (file name, line, column)

```
type position = ...
```

We can obtain the location of the last string recognized by ocamllex

```
val lexeme_start_p: lexbuf -> position
val lexeme_end_p : lexbuf -> position
```

Localization

This information is used to locate a syntax error

```
let lb = Lexing.from_channel c in
try
  let f = Parser.file Lexer.next_token lb in
  ...
with
| Parser.Error ->
    let pos = Lexing.lexeme_start_p lb in
    eprintf "%d: syntax error" pos.pos_lnum;
    exit 1
```

but also possibly lexical errors like a string or unclosed comments

Localization in Menhir

The menhir tool retrieves this information and provides it in two values $startpos and $endpos of the Lexing.position type

In a semantic action, they correspond to the beginning and the end of the text that has been recognized by the grammar rule

This information can be stored in the abstract syntax tree

```
expression:
| e1 = expression; PLUS; e2 = expression
    { Add ($startpos, $endpos, e1, e2) }
```

(anyway we have seen an even more elegant solution)

Elementary parsing

A good exercise

Let us write, in the most elementary way possible, a parser for arithmetic expressions including

- constants
- addition
- multiplication
- parentheses

More precisely

Starting point: a lexical analyzer (e.g. written with `ocamllex`)

```
type token =
  | CONST of int
  | PLUS
  | TIMES
  | LEFTPAR
  | RIGHTPAR
  | EOF
```

Goal: an abstract syntax tree

```
type expr =
  | Const of int
  | Add   of expr * expr
  | Mul   of expr * expr
```

A good advice

Start by writing a print function (pretty-printer)

```
let rec print fmt = function
  | Add (e1, e2) -> fprintf fmt "%a +@ %a" print e1 print e2
  | e            -> print2 fmt e
and print2 fmt = function
  | Mul (e1, e2) -> fprintf fmt "%a *@ %a" print2 e1 print2 e2
  | e            -> print3 fmt e
and print3 fmt = function
  | Const n      -> fprintf fmt "%d" n
  | e            -> fprintf fmt "(@[%a@])" print e
```

(here we use the library `Format`)

Recursive top-down parsing

the parser follows the same structure as the print function

```
let rec parse_expr () =
  let e = parse_term () in
  if !t = PLUS then (next (); Add (e, parse_expr ())) else e
and parse_term () =
  let e = parse_factor () in
  if !t = TIMES then (next (); Mul (e, parse_term ())) else e
and parse_factor () = match !t with
  | CONST n -> next (); Const n
  | LEFTPAR -> next ();
      let e = parse_expr () in
      if !t <> RIGHTPAR then error ();
      next (); e
  | _ -> error ()
```

Remark

- one could include lexical analysis in such a code, with other recursive functions to read integer constants, ignore blanks, etc.
- for left-associative operators (e.g. subtraction), the code will be slightly different but the principle remains the same

Top-down parsing

General idea

Idea: proceed by successive expansions of the leftmost non-terminal
(we therefore construct a left derivation) starting from S and using a table indicating,
for a non-terminal X to be expanded and the k first characters of the input, the production $X \to \beta$ to be performed
(this is the so-called top-down parsing)

Let us suppose $k = 1$ thereafter and let us denote by $T(X, c)$ this table

In practice we suppose that a terminal symbol $\#$ denotes the end of the input, and the table therefore also indicates the expansion of X when we reach the end of the input

How the analysis work

We use a stack that is a word in $(N \cup T)^*$: initially the stack is reduced to the start symbol

At each step, we examine the top of the stack and the first character c of the input

- if the stack is empty, we stop; there is success 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 and consume c; otherwise we fail
- if the top of the stack is a non-terminal X , then we replace X by the word $\beta = T(X, c)$ at the top of the stack, if any, by pushing the characters of β starting from the last one;
  otherwise, we fail

Example

Take the grammar of arithmetic expressions and consider the table of expansion as follows:

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

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

(we will explain how to construct the table later)

Example

Top-down parsing of the word

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

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

stack	input
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	$\#$

Programming a top-down parser

A top-down parser can be very easily programmed by introducing a function for each non-terminal of the grammar

Each function examines the input and, depending on the case, consumes it or recursively calls the functions corresponding to other non-terminals, according to the expansion table

(this is what we did at the beginning of the course)

Programming a top-down parser

Let's choose a purely application programming language, for a change, 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 which "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 as follows:

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

Programming a top-down parser

The functions proceed by filtering 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 ()
```

Programming 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 ()
```

Programming 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 ()
```

Remark

- the expansion table is not explicit: it is in the code of each function
- the stack is not explicit either: it is realized by the call stack
- we could have made them explicit

- in practice, we must also build the abstract syntax tree

Build the expansion table

It remains an important question: how to build the table?

The idea is simple: to decide whether we carry out the expansion $X \to \beta$ when the first character of the input is c, we will try to determine whether c is part of the first characters of the words recognized by β

A difficulty arises for a production such as $X \to \epsilon$, and we must then also consider the set of characters that can follow X

we saw last week how to calculate first and follow

Build the expansion table

using the FIRST and the FOLLOW, we construct the expansion table $T(X, a)$ in the following manner

for each production $X \to \beta$

- we set $T(X, a) = \beta$ for all $a \in \mathrm{FIRST}(\beta)$
- if $\mathrm{NULL}(\beta)$, we also set $T(X, a) = \beta$ for all $a \in \mathrm{FOLLOW}(X)$

Example

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

FIRST

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

FOLLOW

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

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

LL(1) grammar

Defn. (LL(1) grammar)
A grammar is said to be LL(1) if, in the preceding table, there is at most one production in each box. □

LL stands for "Left to right scanning, Leftmost derivation", which means that the input is scanned from left to right and a left derivation is constructed.

Counter-example

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

FIRST

E	T	F
{(, int}	{(, int}	{(, int}

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

In general

A left recursive grammar, i.e. containing productions of the form

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

is never LL(1)

Indeed, the FIRST will be the same for these two productions (whatever the word β is)

Solution

You have to remove the left recursion, for example, like this

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

Same

If a grammar contains

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

it is never LL(1)

Solution

It is necessary to factorize the productions that start with the same terminal (left factorization)

$$\begin{array}{lcl} X & \rightarrow & aX' \\ X' & \rightarrow & \alpha \\ & | & \beta \end{array}$$

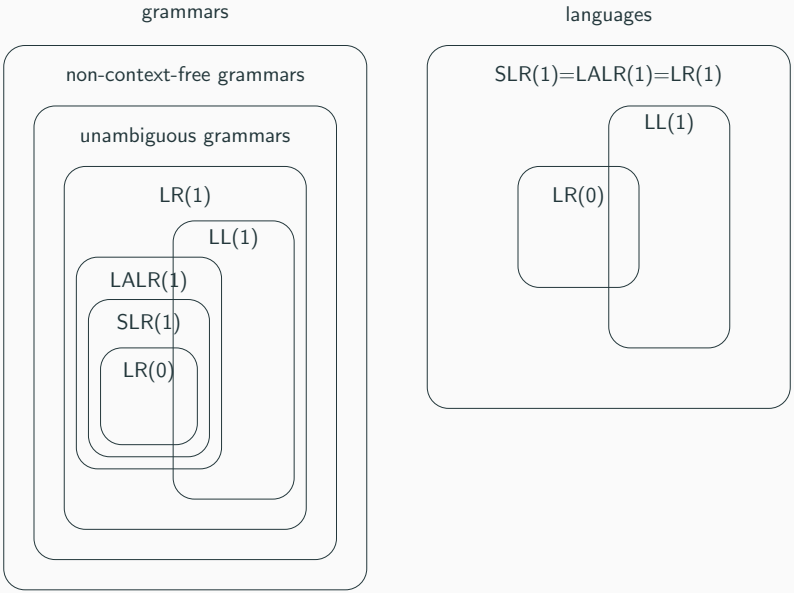
Exercise

Is the grammar LISP

$$
\begin{array}{lcl}
S & \rightarrow & E\ \# \\
E & \rightarrow & \texttt{sym} \\
 & | & (L) \\
L & \rightarrow & \epsilon \\
 & | & E\ L
\end{array}
$$

LL(1)?

Grammar hierarchies



Conclusion

LL(1) parsers are relatively simple to write

but they require writing unnatural grammars

However....

Many compilers use hand-written top-down analysis

Examples:

- `javac` ($\approx$ 3 KLOC of Java code)
- `rustc` ($\approx$ 16 KLOC of rust code)
- `gcc` ($\approx$ 25 KLOC of C++ code)

Indentation as syntax

Concept

In some languages, indentation (leading spaces, vertical alignment) is used to define syntax.

Examples

- Python
- Haskell
- Purescript

The idea is not new (Landin, 1966)

Example

In Python, indentation defines the block structure.

```
while q > 0:
    if q % 2 == 1:
        r += p
    p *= 2
    q //= 2
```

Principle

The lexical analyzer introduces tokens `NEWLINE` (end of line), `INDENT` (when the indentation increases) and `DEDENT` (when it decreases)

and the grammar of the language uses them

```
statement:
| IF expression COLON suite
| WHILE expression COLON suite
...
suite:
| statement NEWLINE
| NEWLINE INDENT statement+ DEDENT
...
```

In particular, it can be written with a yacc-like tool

References i


Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman.
Compilers: Principles, Techniques, and Tools (2nd Edition).
Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

Questions?