École Polytechnique

INF564 – Compilation

Jean-Christophe Filliâtre

parsing

the goal of parsing is to identify the programs that belong to the syntax of the language

its input is concrete syntax, that is a sequence of characters, and its output is abstract syntax

parsing is split into two phases

- **lexical analysis**, which splits the input in "words" called *tokens*
- **syntax analysis**, which recognizes legal sequences of tokens

source = sequence of characters

```
fun x -> (* my function *)
    x+1
```

↓

lexical analysis

↓

sequence of tokens

`fun` `x` `->` `x` `+` `1`

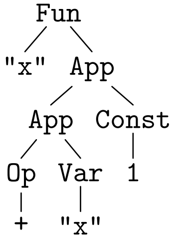
⋮

↓

syntax analysis

↓

abstract syntax



lexical analysis

blanks (spaces, newlines, tabs, etc.) play a role in lexical analysis; they can be used to separate two tokens

for instance, `funx` is understood as a single token (identifier `funx`) and `fun x` is understood as two tokens (keyword `fun` and identifier `x`)

yet several blanks are useless (as in `x + 1`)
and simply ignored

blanks do not appear in the returned sequence of tokens lexical conventions differ according to the languages, and some blanks may be significant

examples:

- tabs for `make`
- newlines and indentation in Python or Haskell
  (indentation defines the structure of blocks)

comments act as blanks

```
fun(* go! *)x -> x + (* adding one *) 1
```

here the comment `(* go! *)` is a significant blank (splits two tokens) and the comment `(* adding one *)` is a useless blank

note: comments are sometimes interpreted by other tools (`javadoc`, `ocamldoc`, etc.), which handle them differently in their own lexical analysis

```
val length: 'a list -> int
  (** Return the length (number of elements) of ...
```

to implement lexical analysis, we are going to use

- **regular expressions** to describe tokens
- **finite automata** to recognize them

we exploit the ability to automatically construct a deterministic finite automaton recognizing the language described by a regular expression

regular expressions

let A be some alphabet

$$
\begin{array}{rcll}
r & ::= & \emptyset & \text{empty language} \\
  & | & \epsilon & \text{empty word} \\
  & | & a & \text{character } a \in A \\
  & | & r\,r & \text{concatenation} \\
  & | & r \mid r & \text{alternation} \\
  & | & r\star & \text{Kleene star}
\end{array}
$$

conventions: in forthcoming examples, star has strongest priority, then concatenation, then alternation the **language** defined by the regular expression r is the set of words $L(r)$ defined as follows:

$$
\begin{aligned}
L(\emptyset) &= \emptyset \\
L(\epsilon) &= \{\epsilon\} \\
L(a) &= \{a\} \\
L(r_1\,r_2) &= \{w_1 w_2 \mid w_1 \in L(r_1) \wedge w_2 \in L(r_2)\} \\
L(r_1 \mid r_2) &= L(r_1) \cup L(r_2) \\
L(r\star) &= \textstyle\bigcup_{n\geq 0} L(r^n) \quad \text{where } r^0 = \epsilon,\ \ r^{n+1} = r\,r^n
\end{aligned}
$$

with alphabet $\{a, b\}$

- words with exactly three letters

$$(a|b)(a|b)(a|b)$$

- words ending with a

$$(a|b) \star a$$

- words alternating a and b

$$(b|\epsilon)(ab) \star (a|\epsilon)$$

decimal integer literals, possibly with leading zeros

$$(0|1|2|3|4|5|6|7|8|9)\,(0|1|2|3|4|5|6|7|8|9)\star$$

identifiers composed of letters, digits and underscore, starting with a letter

$$(a|b|\ldots|z|A|B|\ldots|Z)\,(a|b|\ldots|z|A|B|\ldots|Z|\_|0|1|\ldots|9)\star$$

floating point numbers (`3.14` `2.` `1e-12` `6.02e23` etc.)

$$d\,d\star\,(.d\star\mid(\epsilon\mid .d\star)(e|E)\,(\epsilon|+|-)d\,d\star)$$

with $d = 0|1|\ldots|9$

comments such as `(* ... *)`, **not nested**, can be described with the following regular expression

$$\boxed{(}\ \boxed{*}\ \left(\boxed{*}\star\, r_1 \mid r_2\right)\star\ \boxed{*}\ \boxed{*}\star\ \boxed{)}$$

where $r_1 =$ all characters but $*$ and $)$
and $r_2 =$ all characters but $*$

regular expressions are not expressive enough to describe **nested** comments (we say that the language of balanced parentheses is not regular)

we will explain later how to get around this problem

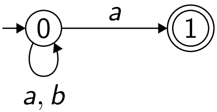
finite automata

Definition

A finite automaton over some A is a tuple (Q, T, I, F) where

- *Q is a finite set of states*
- *$T \subseteq Q \times A \times Q$ is a set of transitions*
- *$I \subseteq Q$ is a set of initial states*
- *$F \subseteq Q$ is a set of final states*

example: $Q = \{0,1\},\ T = \{(0,a,0),(0,b,0),(0,a,1)\},\ I = \{0\},\ F = \{1\}$



a word $a_1 a_2 \dots a_n \in A^\star$ is **recognized** by the automaton (Q, T, I, F) if and only if

$$s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \cdots s_{n-1} \xrightarrow{a_n} s_n$$

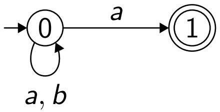
with $s_0 \in I$, $(s_{i-1}, a_i, s_i) \in T$ for all i, and $s_n \in F$

the **language** defined by an automaton is the set of words it recognizes

Theorem (Kleene, 1951)

Regular expressions and finite automata define the same languages.

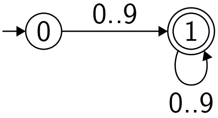
$$(a|b) \star a$$



regular expression

$$(0|1|2|3|4|5|6|7|8|9)\,(0|1|2|3|4|5|6|7|8|9)\star$$

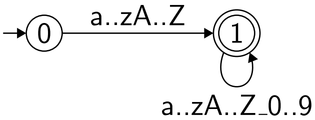
automaton



regular expression

$$(a|b|\dots|z|A|B|\dots|Z)\,(a|b|\dots|z|A|B|\dots|Z|\_|0|1|\dots|9)\star$$

automaton

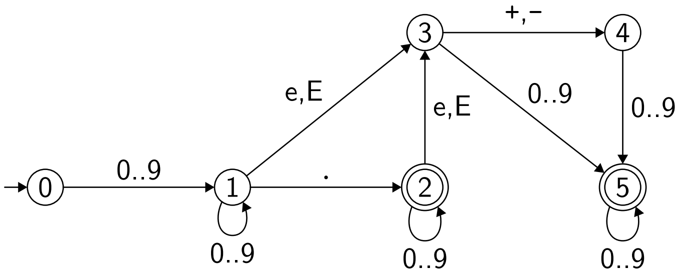


regular expression

$$d\, d\, \star\ (.d\, \star\ |\, (\epsilon\, |\, .d\star)(e|E)\, (\epsilon|+|-)d\, d\star)$$

where $d = 0|1|\ldots|9$

automaton



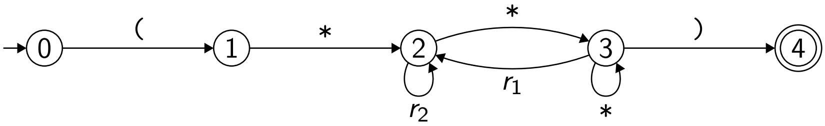
comments

regular expression

$$\boxed{(}\,\boxed{*}\,\left(\boxed{*}\star r_1 \mid r_2\right)\star\,\boxed{*}\,\boxed{*}\star\,\boxed{)}$$

where r_1 = all characters but $*$ and)
and r_2 = all characters but $*$

automaton



lexical analyzer

a **lexical analyzer** is a finite automaton for the "union" of all regular expressions describing the tokens

however, it differs from the mere analysis of a single word by an automaton, since

- we must split the input into a **sequence** of words
- there are possible **ambiguities**
- we have to build tokens (final states contain **actions**)

the word `funx` is recognized by the regular expression for identifiers, but contains a prefix recognized by another regular expression (keyword `fun`)

$\Rightarrow$ we choose to match the **longest** token

the word `fun` is recognized by the regular expression for the keyword `fun` but also by that of identifiers

$\Rightarrow$ we order regular expressions using **priorities** with the three regular expressions

$$a, \quad ab, \quad bc$$

a lexical analyzer will **fail** on input

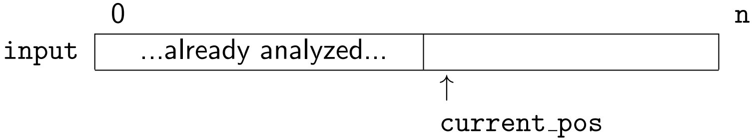
$$abc$$

(ab is recognized, as longest, then failure on c)

yet the word abc belongs to the language $a|ab|bc$

tokens are output one by one, on demand (from the syntax analyzer)

the lexical analyzer memorizes the position where the analysis will resume



when a new token is required, we start from the initial state of the automaton, from position `current_pos`

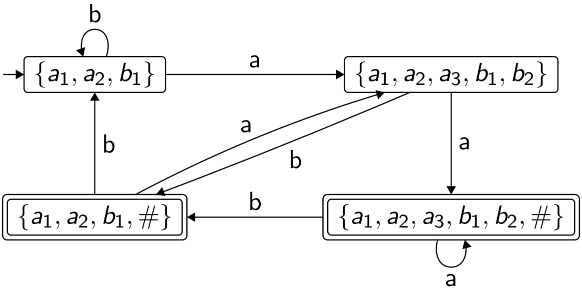
as long as a transition exists, we follow it, while memorizing any token that was recognized (any final state that was reached)

when there is no transition anymore, there are two cases:

- if a token was recognized, we return it and `current_pos` ← `last`
- otherwise, we signal a lexical error

one can build the finite automaton corresponding to a regular expression using an intermediate non deterministic finite automaton (Thompson, 1968)

but one can build a deterministic finite automaton in a single step (Berry, Sethi, 1986); for $(a|b) \star a(a|b)$ we get



see the polycopié sec 3.2

tools

in practice, we have tools to build lexical analyzers from a decription with regular expressions and actions

this is the `lex` family: `lex`, `flex`, `jflex`, `ocamllex`, etc.

we illustrate `jflex` (for Java) and `ocamllex` (for OCaml) to illustrate these tools, let us write a lexical analyzer for a language of arithmetic expressions with

- integer literals
- parentheses
- subtraction

jflex

a `jflex` file has suffix `.flex` and the following structure

```
... preamble ...
%{
... some Java code
%}
%%
<YYINITIAL> {
  regular expression { action }
  ...
  regular expression { action }
}
```

where each `action` is Java code
(returning a token most of the time)

we set up a file `Lexer.flex` for our language

```
import static sym.*;    /* imports the tokens */

%%

%class Lexer                /* our class will be Lexer   */
%unicode                    /* we use unicode characters */
%cup                        /* syntax analysis using cup */
%line                       /* activate line numbers     */
%column                     /* and column numbers        */
%yylexthrow Exception       /* we can raise Exception    */

%{
   /* no need for a Java preamble here */
%}
```

...

...

```
WhiteSpace          = [ \t\r\n]+       /* shortcuts */
Integer             = [:digit:]+
%%
<YYINITIAL> {
  "-"   { return new Symbol(MINUS, yyline, yycolumn); }
  "("   { return new Symbol(LPAR,  yyline, yycolumn); }
  ")"   { return new Symbol(RPAR,  yyline, yycolumn); }
  {Integer}
        { return new Symbol(INT, yyline, yycolumn,
                                Integer.parseInt(yytext())); }
  {WhiteSpace}
        { /* ignore */ }
  .     { throw new Exception (String.format (
           "Line %d, column %d: illegal character: '%s'\n",
           yyline, yycolumn, yytext())); }
}
```

- tokens are freely implemented;
  here, we use the class `Symbol` that comes with cup (see later)
- `MINUS`, `LPAR`, `RPAR` and `INT` are integers (token kinds) built by the tool cup and imported from `sym.java`
- variables `yyline` and `yycolumn` are updated automatically
- `yytext()` returns the string that was recognized by the regular expression

we compile file `Lexer.flex` with `jflex`

```
jflex Lexer.flex
```

we get pure Java code in `Lexer.java`, with

- a constructor

```
    Lexer(java.io.Reader)
```

- a method

```
    Symbol next_token()
```

`.`	any character
`a`	the character `'a'`
`"foobar"`	the string `"foobar"` (in particular $\epsilon =$ `""`)
`[`*characters*`]`	set of characters (e.g. `[a-zA-Z]`)
`[^`*characters*`]`	set complement (e.g. `[^"]`)
`[:`*ident*`:]`	predefined set of characters (e.g. `[:digit:]`)
`{`*ident*`}`	named regular expression
r_1 `|` r_2	alternation
$r_1\ r_2$	concatenation
r `*`	star
r `+`	one or more repetitions of r ($\stackrel{\text{def}}{=}\ r\,r\star$)
r `?`	zero or one occurrence of r ($\stackrel{\text{def}}{=}\ \epsilon \mid r$)
`(` r `)`	grouping

ocamllex

an `ocamllex` file has suffix `.mll` and the following structure

```
{
  ... some OCaml code ...
}
rule ident = parse
| regular expression { action }
| regular expression { action }
| ...
{
  ... some OCaml code ...
}
```

where each `action` is some OCaml code

```
let white_space = [' ' '\t' '\n']+
let integer     = ['0'-'9']+
rule next_token = parse
  | white_space
      { next_token lexbuf }
  | integer as s
      { INT (int_of_string s) }
  | '-'
      { MINUS }
  | '('
      { LPAR }
  | ')'
      { RPAR }
  | eof
      { EOF }
  | _ as c
      { failwith ("illegal character" ^ String.make 1 c) }
```

- we assume the following type for the tokens

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

(will be built by the syntax analyzer)

- contrary to `jflex`
  - we explicitly call `next_token` to ignore blanks
  - we do not handle lines and columns explicitly

we compile the file `lexer.mll` with `ocamllex`

```
ocamllex lexer.mll
```

it outputs some pure OCaml code in `lexer.ml`, which provides

```
val next_token: Lexing.lexbuf -> token
```

(such an argument can be built with `Lexing.from_channel`)

`_`	any character
`'a'`	the character `'a'`
`"foobar"`	the string `"foobar"` (in particular $\epsilon =$ `""`)
`[`*characters*`]`	set of characters (e.g. `['a'-'z' 'A'-'Z']`)
`[^`*characters*`]`	set complement (e.g. `[^ '"']`)
ident	named regular expression
r_1 `|` r_2	alternation
$r_1\ r_2$	concatenation
r `*`	star
r `+`	one or more repetitions of r ($\stackrel{\text{def}}{=}\ r\ r\star$)
r `?`	zero or one occurrence of r ($\stackrel{\text{def}}{=}\ \epsilon \mid r$)
`(` r `)`	grouping
`eof`	end of input

for more details, see the documentations of `jflex` and `ocamllex`, available from

- the course website
- lab 3

- **regular expressions** are the basis of lexical analysis
- the job is **automatized** with tools such as `jflex` and `ocamllex`
- `jflex`/`ocamllex` are **more expressive** than regular expressions

  indeed, actions can call the lexical analyzer recursively
  $\Rightarrow$ allows us to recognize nested comments for instance
  (poly page 54)

syntax analysis

sequence of tokens

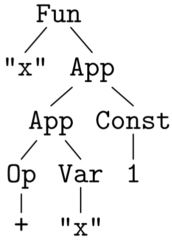
`fun` `x` `->` `(` `x` `+` `1` `)`

↓

syntax analysis

↓

abstract syntax



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

to implement syntax analysis, we are using

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

similar to regular expressions / finite automata used in lexical analysis

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*

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

in practice, we write production rules as follows:

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

the terminals are the tokens produced by the lexical analysis

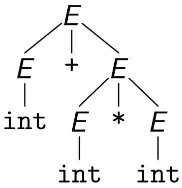
here `int` stands for an integer literal token (*i.e.* its nature, not its value)

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*
- *leaves are terminal symbols*

example:



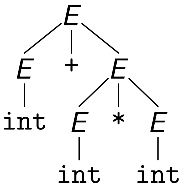
careful: this is **different** from the abstract syntax tree

Definition

The **language** $L(G)$ *defined by a context-free grammar* $G = (N, T, S, R)$ *is the set of words* $w \in T^\star$ *for which there is a derivation tree whose leaves form the word* w.

in our example

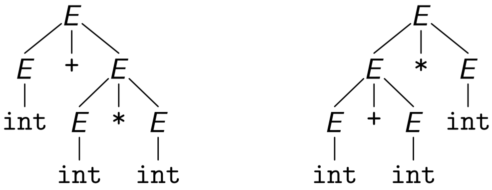
`int + int * int` $\in L(G)$



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

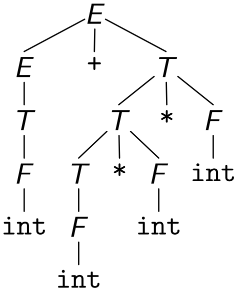


and thus our grammar is ambiguous

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

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

this new grammar reflects the priority of multiplication over addition, and the choice of a left associativity for these two operations now, the word `int + int * int * int` has a single derivation tree,



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) we are going to us **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

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

at each step, two actions can be performed

- a **shift** operation: we read a terminal from the input and we push it on 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

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

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

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 "**L**eft to right scanning, **R**ightmost derivation")

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

the automaton is implemented as 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	reduce $E \to E{+}E$					

(we show later how to built it)

the stack looks like

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

where s_i is a state of the automaton 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 on 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 **goto** table for s_{n-p} and X, *i.e.*

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

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

stack	input	action
1	int+int+int	s2
1 int 2	+int+int	$E \to$ int, g3
1 E 3	+int+int	s6
1 E 3 + 6	int+int	s2
1 E 3 + 6 int 2	+int	$E \to$ int, g8
1 E 3 + 6 E 8	+int	$E \to E$+E, g3
1 E 3	+int	s6
1 E 3 + 6	int	s2
1 E 3 + 6 int 2	#	$E \to$ int, g8
1 E 3 + 6 E 8	#	$E \to E$+E, g3
1 E 3	#	success

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*)

here we illustrate `cup` (for Java) and `menhir` (for OCaml) we keep using the language of arithmetic expressions with

- integer literals
- parentheses
- subtraction

we assume that abstract syntax and lexical analysis are already implemented

CUP (Java)

in a file `Parser.cup`, we start with a prelude where we declare terminals and nonterminals

```
terminal Integer INT;
terminal LPAR, RPAR, MINUS;

non terminal Expr file;
non terminal Expr expr;
```

...

the we declare the grammar production rules and the corresponding actions

```
start with file;

file ::=
  expr:e
    {: RESULT = e; :}
;

expr ::=
  INT:n
    {: RESULT = new Ecst(n); :}
| expr:e1 MINUS expr:e2
    {: RESULT = new Esub(e1, e2); :}
| LPAR expr:e RPAR
    {: RESULT = e; :}
;
```

we compile file `Parser.cup` with

```
java -jar java-cup-11a.jar -parser Parser Parser.cup
```

which signals an error:

```
Warning : *** Shift/Reduce conflict found in state #6
  between expr ::= expr MINUS expr (*)
  and     expr ::= expr (*) MINUS expr
  under symbol MINUS
  Resolved in favor of shifting.

Error : *** More conflicts encountered than expected
        -- parser generation aborted
```

we can declare `MINUS` to be left associative

```
precedence left MINUS;
```

(which favors reduction)

if there are more operators, we list them in increasing priorities

```
precedence left PLUS, MINUS;
precedence left TIMES, DIV, MOD;
```

now CUP successfully terminates and produces two Java files:

- `sym.java` contains the declarations of tokens (`INT`, `LPAR`, `RPAR`, etc.)
- `Parser.java` contains the syntax analyzer, with a constructor

  ```
  Parser(Scanner scanner)
  ```

  and a method

  ```
  Symbol parse()
  ```

we combine the code generated by `jflex` and `CUP` as follows:

```
Reader reader = new FileReader(file);
Lexer lexer = new Lexer(reader);
Parser parser = new Parser(lexer);
Expr e = (Expr)parser.parse().value;
try {
  System.out.println(e.eval());
} catch (Error err) {
  System.out.println("error: " + err.toString());
  System.exit(1);
}
```

the program must include the library `java-cup-11a-runtime.jar`

Menhir (OCaml)

in a file parser.mly, we first declare terminals and nonterminals

```
%{
  ... arbitrary OCaml code ...
%}

%token MINUS LPAR RPAR EOF
%token <int> INT

%start <expr> file
```

...

note: contrary to CUP, one has to declare EOF then we give the grammar production rules and the corresponding actions

```
%%

file:
  e = expr; EOF  { e }
;

expr:
| i = INT                          { Cte i }
| e1 = expr; MINUS; e2 = expr { Sub (e1, e2) }
| LPAR; e = expr; RPAR            { e }
;

%%
```

note: contrary to CUP, one has to add `EOF` we compile file `arith.mly` as follows:

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

it emits a warning

```
Warning: one state has shift/reduce conflicts.
Warning: one shift/reduce conflict was arbitrarily resolved.
```

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

we can declare `MINUS` to be left associative

```
%left MINUS
```

(which favors reduction)

if there are more operators, we list them in increasing priorities

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

now `menhir` successfully terminates and outputs two OCaml files `arith.ml(i)` that contain

- a data type `token`

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

- a function

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

we combine `ocamllex` and `menhir` as follows:

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

building the automaton

Definition (NULL)

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

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.* $\{a \in T \mid \exists w.\, \alpha \rightarrow^\star aw\}$.

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.* $\{a \in T \mid \exists u, w.\, S \rightarrow^\star uXaw\}$.

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

$\text{NULL}(X)$ holds if and only if

- there exists a production $X \rightarrow \epsilon$,
- or there exists a production $X \rightarrow Y_1 \dots Y_m$ where $\text{NULL}(Y_i)$ for all i

issue: this is a set of mutually recursive equations

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

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

two examples

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 $f : A \to A$, i.e. such that $\forall x, y.\, x \leq y \Rightarrow f(x) \leq f(y)$, has a least fixpoint.

proof: since ε is a least element, we have $\varepsilon \leq f(\varepsilon)$
f being increasing, we have $f^k(\varepsilon) \leq f^{k+1}(\varepsilon)$ for any k
A being finite, there exists a least k_0 such that $f^{k_0}(\varepsilon) = f^{k_0+1}(\varepsilon)$
$a_0 = f^{k_0}(\varepsilon)$ is thus a fixpoint of f

let b another fixpoint of f
we have $\varepsilon \leq b$ and thus $f^k(\varepsilon) \leq f^k(b)$ for any k
in particular $a_0 = f^{k_0}(\varepsilon) \leq f^{k_0}(b) = b$
a_0 is thus a least fixpoint of f $\square$

to compute NULL, we have
$A = \text{Bool} \times \cdots \times \text{Bool}$ avec $\text{Bool} = \{\texttt{false}, \texttt{true}\}$

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

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

the theorem applies with

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

since computing $\text{NULL}(X)$ from $\text{NULL}(X_i)$ is monotonic to compute $\text{NULL}(X_i)$, we thus start with

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

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

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

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

why do we seek for a **least** fixpoint?

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

$\Leftarrow$ by induction on the number of steps of derivation $X \rightarrow^\star \epsilon$, we show that $\text{NULL}(X) = \texttt{true}$ in the previous computation similarly, the equations defining FIRST are mutually recursive

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

and

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

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 $\varepsilon = (\emptyset, \ldots, \emptyset)$

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

NULL

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

FIRST

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

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

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

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

NULL

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

FIRST

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

FOLLOW

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

let us use $k = 0$ for the moment

we consider the following grammar:

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

states are sets of *items* of the shape

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

where $X \to \alpha\beta$ is a grammar production rule; interpretation is "we want to recognize X, we have already seen α and we still need to see β"

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

each state s is **closed** under

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

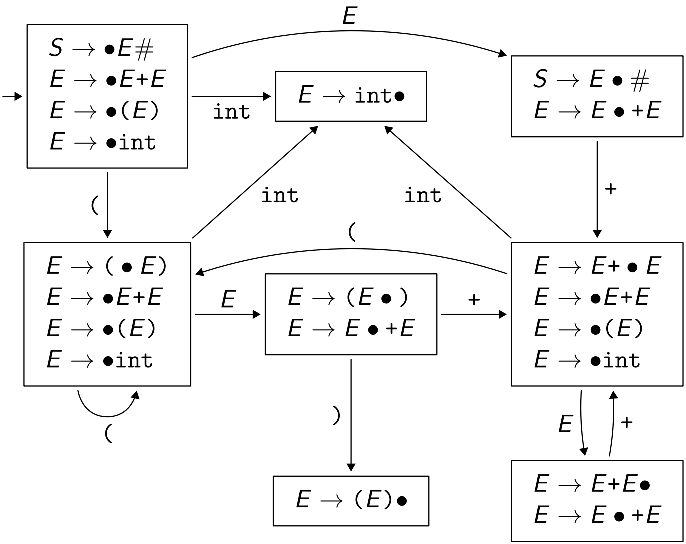
example:

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

transitions are labeled with $T \cup N$ and are as follows:

$$
\begin{array}{rcl}
[Y \rightarrow \alpha \bullet a\beta] & \xrightarrow{a} & [Y \rightarrow \alpha a \bullet \beta] \\
[Y \rightarrow \alpha \bullet X\beta] & \xrightarrow{X} & [Y \rightarrow \alpha X \bullet \beta]
\end{array}
$$

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



the `action` table is built as follows:

- $\texttt{action}(s, \#) = \text{success if } [S \to E \bullet \#] \in s$
- $\texttt{action}(s, a) = \texttt{shift}\ s'$ if we have $s \xrightarrow{a} s'$
- $\texttt{action}(s, a) = \texttt{reduce}\ X \to \beta$ if $[X \to \beta \bullet] \in s$, for all a
- failure otherwise

the `goto` table is built as follows:

- $\texttt{goto}(s, X) = s'$ if and only if we have $s \xrightarrow{X} s'$

on our example, we get

	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					

the LR(0) table may contain two kinds of conflicts

- a **shift/reduce** conflict, if we can do both a shift and a reduce action
- a **reduce/reduce** conflict, if we can do two different reduce actions

Definition (LR(0) grammar)

A grammar is said to be LR(0) if the table contains no conflict.

we have a shift/reduce conflict in state 8

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

it illustrates the ambiguity of the grammar on input `int+int+int`

we can remove the conflict in two different ways:

- if we favor **shift**, we make + right associative
- if we favor **reduce**, we make + left associative
  (and we get the table we used earlier)

LR(0) tables quickly contain conflicts,
so let us try to remove some reduce actions

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

$$[X \rightarrow \beta \bullet] \in s \quad \text{and} \quad a \in \textsc{follow}(X)$$

Definition (SLR(1) grammar)

A grammar is said to be SLR(1) if the resulting table contains no conflict.

(SLR means *Simple LR*)

the grammar

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

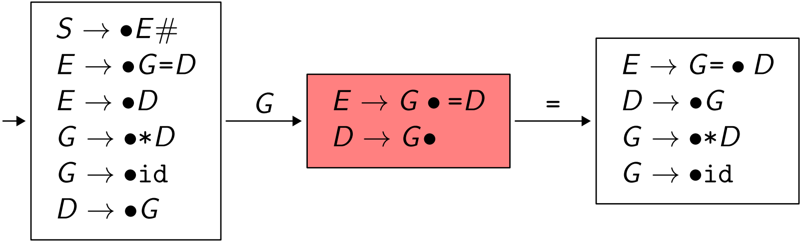
is SLR(1)

exercise: check it (the automaton has 12 states) in practice, SLR(1) grammars are not powerful enough

example:

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

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



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

items now look like

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

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"

the LR(1) automaton has transitions

$$\begin{array}{rcl} [Y \to \alpha \bullet a\beta, b] & \xrightarrow{a} & [Y \to \alpha a \bullet \beta, b] \\ [Y \to \alpha \bullet X\beta, b] & \xrightarrow{X} & [Y \to \alpha X \bullet \beta, b] \end{array}$$

and in a state containing $[Y \to \alpha \bullet X\beta, b]$ we only include

$$[X \to \bullet\gamma, c] \quad \text{for all } c \in \text{FIRST}(\beta b)$$

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

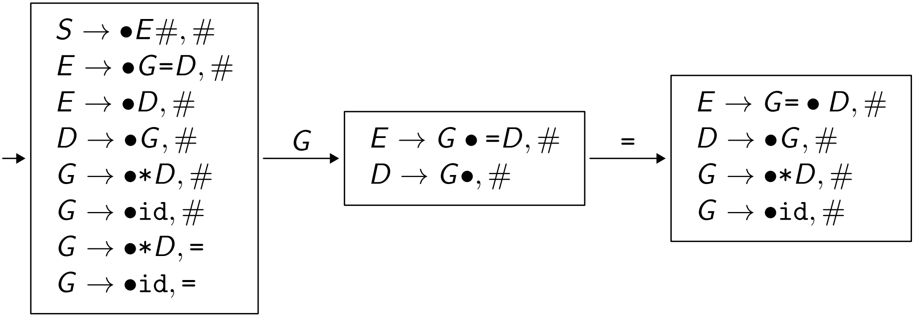
there is a reduce action for (s, a) only when s contains an item $[X \to \alpha\bullet, a]$

Definition (LR(1) grammar)

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

$$
\begin{array}{rcl}
S & \rightarrow & E\# \\
E & \rightarrow & G = D \\
  & | & D \\
G & \rightarrow & * \, D \\
  & | & \mathtt{id} \\
D & \rightarrow & G
\end{array}
$$

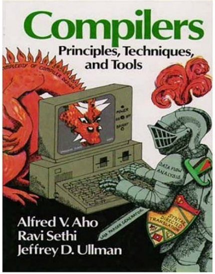
	#	=	
1	...	...	...
2	reduce $D \rightarrow G$	shift 3	...
3	⋮	⋮	⋱

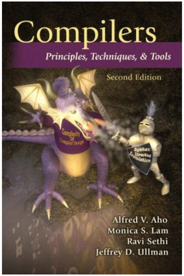


the LR(1) tables can be large, so we introduced approximations

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, section 4.7

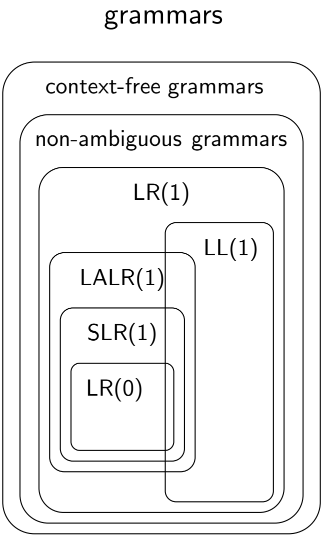


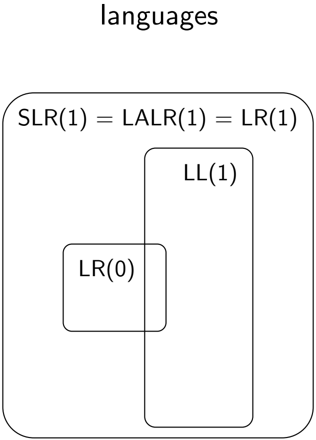


one can also use a **recursive descent parser** = successive expansions of the leftmost nonterminals, starting from S, using an expansion table

defines the LL(k) classes of grammars; cf poly chapter 4

LL(1) analyzers are rather simple to implement
but they require grammars that are less natural


grammars
context-free grammars
non-ambiguous grammars
LR(1)
LL(1)
LALR(1)
SLR(1)
LR(0)


languages
SLR(1) = LALR(1) = LR(1)
LL(1)
LR(0)

- lab 3
  - syntax analysis of mini-Turtle
  - (interpreter is given)
  - Java or OCaml

- poly chapters 3 and 4

- next lecture
  - typing
  - lab: project start