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

Fun
   "x" App
   App Const
   Op Var 1
   + "x"
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, ocamlfixocamldoc, 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

$$r ::= \emptyset \quad \text{empty language}$$

$$\mid \epsilon \quad \text{empty word}$$

$$\mid a \quad \text{character } a \in A$$

$$\mid r \ r \quad \text{concatenation}$$

$$\mid r \mid r \quad \text{alternation}$$

$$\mid r^* \quad \text{Kleene star}$$

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:

\[
L(\emptyset) = \emptyset \\
L(\epsilon) = \{\epsilon\} \\
L(a) = \{a\}
\]

\[
L(r_1 r_2) = \{w_1 w_2 \mid w_1 \in L(r_1) \land w_2 \in L(r_2)\}
\]

\[
L(r_1 \mid r_2) = L(r_1) \cup L(r_2)
\]

\[
L(r^\ast) = \bigcup_{n \geq 0} L(r^n) \quad \text{where} \quad r^0 = \epsilon, \quad r^{n+1} = r \, r^n
\]
examples

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 \(b\)

\[(b|\epsilon)(ab) \star (a|\epsilon)\]
integer literals

decimal integer literals, possibly with leading zeros

$$(0|1|2|3|4|5|6|7|8|9) (0|1|2|3|4|5|6|7|8|9)^*$$
identifiers composed of letters, digits and underscore, starting with a letter

\[(a|b| \ldots |z|A|B| \ldots |Z)(a|b| \ldots |z|A|B| \ldots |Z|\_01\_| \ldots |9)\star\]
floating point literals

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

\[ d d \star (\cdot d \star | (\varepsilon \cdot d\star)(eE)(\varepsilon|+|-)d d\star) \]

with \( d = 0|1|\ldots|9 \)
comments such as (* . . .  *)

\( \text{not nested} \), can be described with the following regular expression

\[
( * ( * r_1 | r_2 ) * * * )
\]

where \( r_1 = \text{all characters but * and )} \)
and \( r_2 = \text{all characters but *} \)
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 $a_1a_2\ldots a_n \in A^*$ 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.

\[(a|b)^* a\]
integer literals

regular expression

$$(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$$

automaton

![Automaton diagram](attachment:image.png)
regular expression

$$(a|b|\ldots|z|A|B|\ldots|Z)(a|b|\ldots|z|A|B|\ldots|Z|_0|1|\ldots|9)^*$$

automaton

```
0 -> 1
a..zA..Z
```

```
1 -> 0
a..zA..Z_0..9
```
floating point literals

regular expression

\[
d d \ast (d \ast | (\epsilon | d \ast)(e|E)(\epsilon | +| -)d d \ast)
\]

where \(d = 0|1|\ldots|9\)

automaton
regular expression

\[
( \star ( \star \star r_1 \mid r_2 ) \star \star \star )
\]

where \( r_1 = \) all characters but \( \star \) and \( ) \)
and \( r_2 = \) all characters but \( \star \)

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 \texttt{funx} is recognized by the regular expression for identifiers, but contains a prefix recognized by another regular expression (keyword \texttt{fun})

\Rightarrow \text{we choose to match the } \textbf{longest} \text{ token}

the word \texttt{fun} is recognized by the regular expression for the keyword \texttt{fun} but also by that of identifiers

\Rightarrow \text{we order regular expressions using } \textbf{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

\[
\begin{array}{c}
\text{input} \\
\text{...already analyzed...}
\end{array}
\]

\[\text{current_pos}\]
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 \((a|b)^* a(a|b)\) we get

see the polycopié sec 3.2
tools
in practice, we have tools to build lexical analyzers from a description 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

```plaintext
... 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

```flex
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

```sh
ejflex Lexer.flex
```

we get pure Java code in Lexer.java, with

- a constructor

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

- a method

  ```java
  Symbol next_token()
  ```
jflex regular expressions

. any character
a the character 'a'
"foobar" the string "foobar" (in particular ϵ = "")
[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

r₁ | r₂ alternation
r₁ r₂ concatenation
r * star
r + one or more repetitions of r (def r r*)
r ? zero or one occurrence of r (def ϵ | 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 = [' ' '	' '
']+
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

```haskell
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)
ocamllex regular expressions

- 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

$r_1 \mid r_2$ alternation

$r_1 \ r_2$ concatenation

$r \ *$ star

$r \ +$ one or more repetitions of $r$ ($\equiv r \ r^*$)

$r \ ?$ zero or one occurrence of $r$ ($\equiv \epsilon \mid r$)

( $r$ ) 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

  ⇒ allows us to recognize nested comments for instance

  (poly page 54)
syntax analysis
sequence of tokens

\[
\text{fun } x \rightarrow ( x + 1 )
\]

\[\downarrow\]

**syntax analysis**

\[\downarrow\]

**abstract syntax**

```
Fun
  "x"  App
    App  Const
      Op  Var  1
        +  "x"
```
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
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**)
- \(R \subseteq N \times (N \cup T)^*\) is a finite set of **production rules**
example: arithmetic expressions

\[ N = \{ E \}, \quad T = \{ +, *, (, ), \text{int} \}, \quad S = E, \]
and \[ R = \{ (E, E+E), (E, E*E), (E, (E)), (E, \text{int}) \} \]

in practice, we write production rules as follows:

\[
E \rightarrow E + E \\
| \quad E \ast E \\
| \quad ( E ) \\
| \quad \text{int}
\]

the terminals are the tokens produced by the lexical analysis

here \text{int} stands for an integer literal token (\textit{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** $\beta \in (N \cup T)^*$ **with** $X \rightarrow \beta$ **a production rule**
- **leaves are terminal symbols**

**example:**

```
    E
   / \  \\
  E + E
 /   |  \\
int E * E
     /   \\
    int int
```

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 $w \in T^*$ for which there is a derivation tree whose leaves form the word $w$.

In our example

$$\text{int} + \text{int} \times \text{int} \in L(G)$$
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

\[
E \rightarrow E + T \\
| \quad T \\
T \rightarrow T * F \\
| \quad F \\
F \rightarrow (E) \\
| \quad \text{int}
\]

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 \((T \cup N)^*\)

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 \(\beta\) of a production \(X \rightarrow \beta\), and we replace \(\beta\) 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

E → E + E  
| ( E )  
| int

<table>
<thead>
<tr>
<th>stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>ε</td>
<td>int+int+int</td>
<td>shift</td>
</tr>
<tr>
<td>int</td>
<td>+int+int</td>
<td>shift</td>
</tr>
<tr>
<td>E</td>
<td>+int*int</td>
<td>shift</td>
</tr>
<tr>
<td>E+</td>
<td>int+int</td>
<td>shift</td>
</tr>
<tr>
<td>E+int</td>
<td>+int</td>
<td>reduce E → int</td>
</tr>
<tr>
<td>E+E</td>
<td>+int</td>
<td>reduce E → E+E</td>
</tr>
<tr>
<td>E</td>
<td>+int</td>
<td>shift</td>
</tr>
<tr>
<td>E+</td>
<td>int</td>
<td>shift</td>
</tr>
<tr>
<td>E+int</td>
<td>int</td>
<td>reduce E → int</td>
</tr>
<tr>
<td>E+E</td>
<td></td>
<td>reduce E → E+E</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>success</td>
</tr>
</tbody>
</table>
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
the automaton is implemented as follows:

<table>
<thead>
<tr>
<th>state</th>
<th>action</th>
<th>goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>shift 4</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
<td>shift 2</td>
</tr>
<tr>
<td>int</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>#</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>reduce $E \rightarrow \text{int}$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>shift 6</td>
<td>success</td>
</tr>
<tr>
<td>3</td>
<td>shift 6</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>shift 4</td>
<td>shift 2</td>
</tr>
<tr>
<td>5</td>
<td>shift 7</td>
<td>shift 6</td>
</tr>
<tr>
<td>6</td>
<td>shift 4</td>
<td>shift 2</td>
</tr>
<tr>
<td>7</td>
<td>reduce $E \rightarrow (E)$</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>reduce $E \rightarrow E+E$</td>
<td></td>
</tr>
</tbody>
</table>

(we show later how to built it)
LR analysis

the stack looks like

\[ S_0 \ x_1 \ s_1 \ x_2 \ \ldots \ x_n \ s_n \]

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 \( X \rightarrow \alpha \), with \( \alpha \) of length \( p \), then we have \( \alpha \) on top of the stack
  \[ S_0 \ x_1 \ s_1 \ \ldots \ x_{n-p} \ s_{n-p} | \alpha_1 \ s_{n-p+1} \ \ldots \ \alpha_p \ s_n \]
  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.
  \[ S_0 \ x_1 \ s_1 \ \ldots \ x_{n-p} \ s_{n-p} \ X \ s \]
## Execution Example

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>int+int+int</td>
<td>s2</td>
</tr>
<tr>
<td>1 int 2</td>
<td>+int+int</td>
<td>E → int, g3</td>
</tr>
<tr>
<td>1 E 3</td>
<td>+int+int</td>
<td>s6</td>
</tr>
<tr>
<td>1 E 3 + 6</td>
<td>int+int</td>
<td>s2</td>
</tr>
<tr>
<td>1 E 3 + 6 int 2</td>
<td>+int</td>
<td>E → int, g8</td>
</tr>
<tr>
<td>1 E 3 + 6 E 8</td>
<td>+int</td>
<td>E → E+E, g3</td>
</tr>
<tr>
<td>1 E 3</td>
<td>+int</td>
<td>s6</td>
</tr>
<tr>
<td>1 E 3 + 6</td>
<td>int</td>
<td>s2</td>
</tr>
<tr>
<td>1 E 3 + 6 int 2</td>
<td>#</td>
<td>E → int, g8</td>
</tr>
<tr>
<td>1 E 3 + 6 E 8</td>
<td>#</td>
<td>E → E+E, g3</td>
</tr>
<tr>
<td>1 E 3</td>
<td>#</td>
<td>success</td>
</tr>
</tbody>
</table>

**Table Description:**
- **Stack:** The current state of the stack.
- **Input:** The next input to be processed.
- **Action:** What action is taken based on the current state and input.
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

```c
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;
predence 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

\[ \text{Parser(Scanner scanner)} \]

and a method

\[ \text{Symbol parse()} \]
we combine the code generated by jflex and CUP as follows:

```java
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

```ocaml
%{
    ... 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

```plaintext
%left MINUS
```

(which favors reduction)

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

```plaintext
%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`

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

- a function

  ```ocaml
val file : (Lexing.lexbuf -> token) -> Lexing.lexbuf -> int
  ```
we combine ocamllex and menhir as follows:

```ocaml
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**)  

\[ \text{Let } \alpha \in (T \cup N)^*. \text{ NULL}(\alpha) \text{ holds if and only if we can derive } \epsilon \text{ from } \alpha \text{ i.e. } \alpha \rightarrow^* \epsilon. \]

Definition (**FIRST**)  

\[ \text{Let } \alpha \in (T \cup N)^*. \text{ FIRST}(\alpha) \text{ is the set of all terminals starting words derived from } \alpha, \text{ i.e. } \{ a \in T \mid \exists w. \alpha \rightarrow^* aw \}. \]

Definition (**FOLLOW**)  

\[ \text{Let } X \in N. \text{ FOLLOW}(X) \text{ is the set of all terminals that may appear after } X \text{ in a derivation, i.e. } \{ a \in T \mid \exists u, w. S \rightarrow^* uXaw \}. \]
computing NULL, FIRST, and FOLLOW

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 $X \rightarrow Y_1 \ldots Y_m$ where $\text{NULL}(Y_i)$ for all $i$

issue: this is a set of mutually recursive equations

said otherwise, if $N = \{X_1, \ldots, X_n\}$ and if

$\vec{V} = (\text{NULL}(X_1), \ldots, \text{NULL}(X_n))$, 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 $\varepsilon$. 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 $\varepsilon$ 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 $f$
to compute **NULL**, we have
\[ A = \text{BOOL} \times \cdots \times \text{BOOL} \text{ avec } \text{BOOL} = \{\text{false, true}\} \]

we can equip **BOOL** with order \( \text{false} \leq \text{true} \) and \( A \) with point-wise order
\[
(x_1, \ldots, x_n) \leq (y_1, \ldots, y_n) \text{ if and only if } \forall i. x_i \leq y_i
\]

the theorem applies with
\[
\varepsilon = (\text{false}, \ldots, \text{false})
\]

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

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

and we use the equations until we get a fixpoint \(\text{i.e. until the values NULL}(X_i)\) do not change anymore
\[
\begin{align*}
E & \to TE' \\
E' & \to + TE' \\
& \mid \epsilon \\
T & \to FT' \\
T' & \to \ast FT' \\
& \mid \epsilon \\
F & \to (E) \\
& \mid \text{int}
\end{align*}
\]

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(E)</td>
<td>(E')</td>
<td>(T)</td>
<td>(T')</td>
<td>(F)</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>
why do we seek for a least fixpoint?

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

⇐ by induction on the number of steps of derivation \( X \rightarrow^* \epsilon \), we show that \( \text{NULL}(X) = \text{true} \) in the previous computation
similarly, the equations defining \textsc{first} are mutually recursive

\[
\text{first}(X) = \bigcup_{X \rightarrow \beta} \text{first}(\beta)
\]

and

\[
\begin{align*}
\text{first}(\varepsilon) &= \emptyset \\
\text{first}(a\beta) &= \{a\} \\
\text{first}(X\beta) &= \text{first}(X), \quad \text{if } \neg \text{null}(X) \\
\text{first}(X\beta) &= \text{first}(X) \cup \text{first}(\beta), \quad \text{if } \text{null}(X)
\end{align*}
\]

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
\[
\varepsilon = (\emptyset, \ldots, \emptyset)
\]
null

<table>
<thead>
<tr>
<th>E</th>
<th>E'</th>
<th>T</th>
<th>T'</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

first

<table>
<thead>
<tr>
<th>E</th>
<th>E'</th>
<th>T</th>
<th>T'</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>∅</td>
<td>∅</td>
<td>{+}</td>
<td>∅</td>
<td>{∗}</td>
</tr>
<tr>
<td>∅</td>
<td>∅</td>
<td>{+}</td>
<td>{(, int)}</td>
<td>{∗}</td>
</tr>
<tr>
<td>{(, int)}</td>
<td>{+}</td>
<td>{(, int)}</td>
<td>{∗}</td>
<td>{(, int)}</td>
</tr>
<tr>
<td>{(, int)}</td>
<td>{+}</td>
<td>{(, int)}</td>
<td>{∗}</td>
<td>{(, int)}</td>
</tr>
</tbody>
</table>
again, the equations defining \texttt{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 \texttt{FIRST}

we add a special symbol \# in \texttt{FOLLOW}(S)

(which we can do directly, or by adding a rule \( S' \to S\# \))
### NULL

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>$E'$</td>
<td>$T$</td>
<td>$T'$</td>
<td>$F$</td>
<td></td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td></td>
</tr>
</tbody>
</table>

### FIRST

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>$E'$</td>
<td>$T$</td>
<td>$T'$</td>
<td>$F$</td>
<td></td>
</tr>
<tr>
<td>${ (, \text{int} }</td>
<td>{ + }</td>
<td>{ (, \text{int} }</td>
<td>{ * }</td>
<td>{ (, \text{int} }</td>
<td></td>
</tr>
</tbody>
</table>

### FOLLOW

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>$E'$</td>
<td>$T$</td>
<td>$T'$</td>
<td>$F$</td>
<td></td>
</tr>
<tr>
<td>${ # }</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td></td>
</tr>
<tr>
<td>${ #, ) }</td>
<td>{ # }</td>
<td>{ +, # }</td>
<td>\emptyset</td>
<td>{ * }</td>
<td></td>
</tr>
<tr>
<td>${ #, ) }</td>
<td>{ #, ) }</td>
<td>{ +, #, ) }</td>
<td>{ +, # }</td>
<td>{ *, +, # }</td>
<td></td>
</tr>
<tr>
<td>${ #, ) }</td>
<td>{ #, ) }</td>
<td>{ +, #, ) }</td>
<td>{ +, #, ) }</td>
<td>{ *, +, #, ) }</td>
<td></td>
</tr>
<tr>
<td>${ #, ) }</td>
<td>{ #, ) }</td>
<td>{ +, #, ) }</td>
<td>{ +, #, ) }</td>
<td>{ *, +, #, ) }</td>
<td></td>
</tr>
</tbody>
</table>
let us use $k = 0$ for the moment

we consider the following grammar:

\[
S \rightarrow E \\
E \rightarrow E + E \\
\quad \mid (E) \\
\quad \mid \text{int}
\]
states are sets of items of the shape

\[ [X \rightarrow \alpha \bullet \beta] \]

where \( X \rightarrow \alpha \beta \) is a grammar production rule; interpretation is “we want to recognize \( X \), we have already seen \( \alpha \) and we still need to see \( \beta \)”

the initial state is that containing \( S \rightarrow \bullet E \# \)
LR(0) automaton

each state $s$ is **closed** under

\[
\text{if } Y \to \alpha \bullet X \beta \in s \\
\text{and if } X \to \gamma \text{ is a production} \\
\text{then } X \to \bullet \gamma \in s
\]

example:

\[
\begin{align*}
E & \to E + E \\
E & \to E + E \\
E & \to (E) \\
E & \to \text{int}
\end{align*}
\]
transitions are labeled with $T \cup N$ and are as follows:

\[
\begin{align*}
[Y \rightarrow \alpha \cdot a\beta] & \xrightarrow{a} [Y \rightarrow \alpha a \cdot \beta] \\
[Y \rightarrow \alpha \cdot X\beta] & \xrightarrow{X} [Y \rightarrow \alpha X \cdot \beta]
\end{align*}
\]
\[ S \rightarrow \bullet E \# \]
\[ E \rightarrow \bullet E + E \]
\[ E \rightarrow \bullet (E) \]
\[ E \rightarrow \bullet \text{int} \]

\[ E \rightarrow \text{int} \bullet \]

\[ S \rightarrow E \bullet \# \]
\[ E \rightarrow E \bullet + E \]

\[ E \rightarrow (E) \bullet \]
\[ E \rightarrow E + + E \]
\[ E \rightarrow \bullet E + E \]
\[ E \rightarrow \bullet (E) \]
\[ E \rightarrow \bullet \text{int} \]

\[ E \rightarrow \text{int} \bullet \]
\[ E \rightarrow E + + E \bullet \]
\[ E \rightarrow E \bullet + E \]
the **action** table is built as follows:

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

the **goto** table is built as follows:

- \( \text{goto}(s, X) = s' \) if and only if we have \( s \xrightarrow{X} s' \)
on our example, we get

<table>
<thead>
<tr>
<th>state</th>
<th>(</th>
<th>)</th>
<th>+</th>
<th>int</th>
<th>#</th>
<th>goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>shift 4</td>
<td></td>
<td></td>
<td>shift 2</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>reduce $E \rightarrow \text{int}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>shift 6</td>
<td></td>
<td></td>
<td>success</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>shift 4</td>
<td></td>
<td></td>
<td>shift 2</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>shift 7</td>
<td>shift 6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>shift 4</td>
<td></td>
<td></td>
<td>shift 2</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>reduce $E \rightarrow (E)$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>shift 6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>reduce $E \rightarrow E+E$</td>
</tr>
</tbody>
</table>
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

\[
E \rightarrow E + E \\
E \rightarrow E \cdot + E
\]

it illustrates the ambiguity of the grammar on input \text{int+int+int}

we can remove the conflict in two different ways:

- if we favor \text{shift}, we make + right associative
- if we favor \text{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 $\text{action}(s, a) = \text{reduce } X \rightarrow \beta$ if and only if $[X \rightarrow \beta\bullet] \in s$ and $a \in \text{FOLLOW}(X)$.

**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

\[
\begin{align*}
S & \rightarrow E \# \\
E & \rightarrow E + T \\
& \quad | \quad T \\
T & \rightarrow T \ast F \\
& \quad | \quad F \\
F & \rightarrow (E) \\
& \quad | \quad \text{int}
\end{align*}
\]

is SLR(1)

exercise: check it (the automaton has 12 states)
limits of SLR(1) analysis

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

example:

\[
\begin{align*}
S & \rightarrow E\# \\
E & \rightarrow G = D \\
& \quad | \quad D \\
G & \rightarrow * D \\
& \quad | \quad \text{id} \\
D & \rightarrow G
\end{align*}
\]

\[
\begin{array}{|c|c|}
\hline
1 & \ldots \quad \ldots \\
2 & \text{shift 3} \quad \ldots \\
3 & \vdots \quad \ldots \\
\hline
\end{array}
\]

\[
\begin{align*}
S & \rightarrow \bullet E\# \\
E & \rightarrow \bullet G=D \\
E & \rightarrow \bullet D \\
G & \rightarrow \bullet * D \\
G & \rightarrow \bullet \text{id} \\
D & \rightarrow \bullet G
\end{align*}
\]

\[
\begin{align*}
E & \rightarrow G \bullet = D \\
D & \rightarrow \bullet G \\
G & \rightarrow \bullet * D \\
G & \rightarrow \bullet \text{id}
\end{align*}
\]
we introduce a larger class of grammars, \textbf{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 $\alpha$, we still need to see $\beta$ and then to check that the next token is $a$”
the LR(1) automaton has transitions

\[
\begin{align*}
[Y \to \alpha \bullet a\beta, b] & \xrightarrow{\text{a}} [Y \to \alpha a \bullet \beta, b] \\
[Y \to \alpha \bullet X\beta, b] & \xrightarrow{\text{X}} [Y \to \alpha X \bullet \beta, b]
\end{align*}
\]

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.
\[
S \rightarrow E\# \\
E \rightarrow G = D \\
| \quad D \\
G \rightarrow *D \\
| \quad \text{id} \\
D \rightarrow G
\]

<table>
<thead>
<tr>
<th>#</th>
<th>=</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td>\text{reduce } D \rightarrow G</td>
</tr>
<tr>
<td>3</td>
<td>:</td>
</tr>
</tbody>
</table>

\[
S \rightarrow \bullet E\#,\# \\
E \rightarrow \bullet G=D,\# \\
E \rightarrow \bullet D,\# \\
D \rightarrow \bullet G,\# \\
G \rightarrow \bullet*D,\# \\
G \rightarrow \bullet\text{id},\# \\
G \rightarrow \bullet*D,= \\
G \rightarrow \bullet\text{id},=
\]

\[
E \rightarrow G = \bullet D,\# \\
D \rightarrow \bullet G,\# \\
G \rightarrow \bullet* D,\# \\
G \rightarrow \bullet\text{id},\#
\]
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
other approach

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
grammar hierarchies

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