Théorie des langages
TP 2 - An Introduction to LL(1) Parsing

Jonathan Fabrizio and Adrien Pommellet, EPITA

October 9, 2023

Please download on Moodle files `parsing.c`, `parsing.h` and `grammar0.c` beforehand.

1 A C Parser

Our goal is to design a straightforward C implementation of the LL(1) parsing algorithm. To this end, you need nothing but your trusty C compiler.

1.1 Manipulating the Input Stream

We will not be interfacing this parser with an actual lexer yet; we instead assume that the input (terminal) symbols are of type `char` and model the input stream as a pointer of type `char **` to a string. We first include the header file `parsing.h` in order to import the following functions on the input stream:

```
void eat(char ** stream_pointer, char token)
void wrong_look_ahead(char ** stream_pointer, char symbol)
```

Function `eat` checks if the first character of the input stream pointed by `stream_pointer` is `token`: it is indeed true, then it increments the input stream so that it points to the next character. However, if it is not the case, it triggers instead an error and prints the following message on the error stream before halting:

```
Read token '%**stream_pointer', expected token '%token' instead.
```

Function `wrong_look_ahead` triggers an error and prints the following message on the error stream:

```
No rule matched to non-terminal '%symbol' can generate look-ahead '%**stream_pointer'.
```

1.2 Designing a LL Parser

LL(1) is a simple enough algorithm; we can design a C implementation of a LL(1) parser for a given grammar by carrying out the following steps:

1. Compute the LL(1) table and check if the grammar is LL(1) in the first place; obvious enough, but it's nonetheless worth mentioning.
2. For each non-terminal `X`, create a function `void X(char ** stream_pointer)`.
3. In the body of function `X`, use a `switch` or `if` statement that filters the possible values of the incoming look-ahead `**stream_pointer`:

   ```
   switch (**stream_pointer) {
     case 'a':
       // Apply rule (X, a) of the LL(1) parsing table, assuming it exists.
       break;
     case 'b':
       // Apply rule (X, b) of the LL(1) parsing table, assuming it exists.
       break;
     default:
       // Catch the terminal symbols that are not matched to any rule.
       wrong_look_ahead(stream_pointer, 'X');
   }
   ```

Note that the non-terminal symbol `$` corresponds to the end of string character `'\0'`.

4. In order to apply a grammar rule, read its right pattern from left to right, eating the terminal symbols and calling the functions associated with the non-terminal symbols as you do so. As an example, in order to apply rule $X \to aYb$, we write the following instructions in the body of function `X`:

   ```
   eat(stream_pointer, 'a');
   Y(stream_pointer);
   eat(stream_pointer, 'b');
   ```

5. Once the only rule $Z \to S\$$ associated with the axiom `Z` has been applied, the parsing process is over.
6. In order to start the parsing process, call the axiom `Z`.

File `grammar0.c` features a simple `LL(1)` parser that recognizes the following grammar $\mathcal{G}_0$:

$$\begin{aligned} Z &\to S\$ && (1)\\ S &\to (S) && (2)\\ &\mid n && (3) \end{aligned}$$

Question 1. Compile `grammar0.c` and test the resulting parser on a few examples (remember that the input string should be written between quotation marks if it contains parentheses).

Question 2. We create a new grammar $\mathcal{G}_1$ by adding a rule $S \to \{S\}$ to $\mathcal{G}_0$. Create a new `LL(1)` parser `grammar1.c` based on `grammar0.c` that recognizes $\mathcal{G}_1$, then test it on a few examples.

2 Parsing Arithmetic Expressions

Our goal is to write a LL(1) parser than can recognize and compute simple arithmetic expressions.

2.1 Parsing Simple Sums

Consider the following LL(1) grammar $\mathcal{G}_2$:

$$\begin{aligned} Z &\to S\$ && (1)\\ S &\to IU && (2)\\ U &\to +IU && (3)\\ &\mid \varepsilon && (4)\\ I &\to n && (5) \end{aligned}$$

It generates the language `n`, `n+n`, `n+n+n`, etc. Prefix factorization thanks to the non-terminal U ensures that this grammar is indeed LL(1).

Question 3. Compute rigorously $\mathcal{G}_2$'s LL(1) table using Null, First, and Follow.

Question 4. Write a new LL(1) parser `grammar2.c` that recognizes $\mathcal{G}_2$, then test it on a few examples.

2.2 Computing Simple Sums

We introduce a new grammar $\mathcal{G}_3$ by replacing $\mathcal{G}_2$'s fifth rule with a new rule $I \to \mathit{integer}$, where *integer* stands for any non-empty sequence of digits. Our next objective is not only to parse but also compute the actual value of the arithmetic expressions generated by $\mathcal{G}_3$. To do so, we must ensure that the non-terminal functions have return type `int` instead of `void` and perform additions as the parser applies grammar rules; we must also entirely rewrite function `I`.

Question 5. Rewrite function `I` of `grammar2.c` so that it eats and returns instead the first integer read on the input stream. As an example, given an input stream `0123+45`, `I` should eat `0123` and return the `int` value `123`. Use the function `isdigit` of library `ctype` to check if the look-ahead is indeed a digit; if it is not, trigger an error instead.

Question 6. Write a new LL(1) parser `grammar3.c` that recognizes $\mathcal{G}_3$ and computes then prints the value of the input arithmetic expression. Test it on a few examples. Note that the function `isdigit` can be used to detect integer looks-ahead.

2.3 Products and Subtractions

Now consider the following LL(1) grammar $\mathcal{G}_4$:

$$\begin{aligned} Z &\to S\$ && (1)\\ S &\to PU && (2)\\ U &\to +PU && (3)\\ &\mid \varepsilon && (4)\\ P &\to IV && (5)\\ V &\to *IV && (6)\\ &\mid \varepsilon && (7)\\ I &\to \mathit{integer} && (8) \end{aligned}$$

It generates the language of arithmetic expressions with addition and product on integers. Intuitively, an arithmetic expression is written as a sum S of products P of integers I. Thanks to the non-terminals U and V, prefix factorization ensures again that this grammar is indeed LL(1).

Question 7. Write a new LL(1) parser `grammar4.c` based on `grammar3.c` that recognizes $\mathcal{G}_4$ and computes then prints the value of the input arithmetic expression. Test it on a few examples.

Question 8. We create a new grammar $\mathcal{G}_5$ by adding a rule $U \to -PU$ to $\mathcal{G}_4$. Create a new LL(1) parser `grammar5.c` based on `grammar4.c` that recognizes $\mathcal{G}_5$ and computes then prints the value of the input arithmetic expression. Test it on a few examples.

2.4 Parentheses and Divisions

Finally, we extend our grammar and parser in order to handle parentheses and divisions. The former extension is fairly straightforward; the latter, however, may witness unexpected associativity issues caused by rounding quotients down to integers.

Question 9. We create a new grammar $\mathcal{G}_6$ by adding a rule $I \to (S)$ to $\mathcal{G}_5$. Create a new LL(1) parser `grammar6.c` based on `grammar5.c` that recognizes $\mathcal{G}_6$ and computes then prints the value of the input arithmetic expression. Test it on a few examples.

Question 10. We create a new grammar $\mathcal{G}_7$ by adding a rule $V \to /IV$ to $\mathcal{G}_6$. Create a new LL(1) parser `grammar7.c` based on `grammar6.c` that recognizes $\mathcal{G}_7$ and tries to compute then print the value of the input arithmetic expression. Test it on the examples `2/4*8`, `2*8/4` and `8/4*2`. Is the program working as intended? Is its implementation of division left-associative or right-associative? Note that there are various ways to implement division: you may, as an example, add one or more new argument to function `V` that carry over the dividend and divisor evaluated so far.

3 Starred Grammars

We introduce a new class of grammars known as *starred grammars* that allows the use of the Kleene star in the body of grammar rules. As an example, the rule $X \to A(B)^*$ stands for the countable set of production rules $X \to A$, $X \to AB$, $X \to ABB$, etc. We consider the starred grammar $\mathcal{G}_8$:

$$Z \to S\$ \tag{1}$$
$$S \to I(/I)^* \tag{2}$$
$$I \to \mathit{integer} \tag{3}$$

$\mathcal{G}_8$ generates arithmetic expressions with division on integers.

Question 11. Write a new `LL(1)` parser `grammar8.c` that recognizes $\mathcal{G}_8$ and computes then prints the value of the input arithmetic expression. To do so, you may use a `while` instruction in the body of the function `S`. Then test this parser on a few examples. Is this grammar left-associative or right-associative?