BITS Pilani, Hyderabad Campus
Department of Computer Science and Information Systems
Second Semester, 2024-25
CS F363 Compiler Construction
Lab-6: `LL(1)` parsing algorithm implementation

1 Objectives

The objectives of this lab sheet are given below.

1. **Implement FIRST and FOLLOW set computation**: Students should be able to write a program that calculates the FIRST and FOLLOW sets for a given context-free grammar (CFG) read from a file. This involves understanding the algorithms for computing these sets and translating them into code.
2. **Construct an LL(1) parsing table**: Given a CFG and its FIRST and FOLLOW sets, students should be able to construct the corresponding LL(1) parsing table. This requires understanding the relationship between FIRST, FOLLOW, and the predict sets, and how they are used to populate the table.
3. **Implement the LL(1) parsing algorithm**: Students should be able to implement the LL(1) parsing algorithm using the constructed parsing table and a stack. This includes handling terminal and non-terminal symbols, looking up productions in the table, and managing the stack. The program should take a string as input and determine whether the string can be derived from the given grammar.
4. **Analyze and modify grammars for LL(1) parsing**: (Optional, but highly recommended) Students should be able to analyze given grammars, identify if they are LL(1), and if not, apply transformations (like left factoring or eliminating left recursion) to make them LL(1) parsable. This objective fosters a deeper understanding of the limitations and requirements of LL(1) parsing.

2 LL(1) parser

`LL(1)` parsers are a type of top-down parser that can be used to parse a certain class of context-free grammars. They are called "`LL`" because they scan the input from left to right, and they use the leftmost derivation to construct a parse tree. The "`(1)`" indicates that they use one lookahead symbol to make parsing decisions.

LL(1) parsers are widely used in compiler design because they are efficient and relatively easy to implement. They can be used to parse many common programming languages, such as C and Java.

We implement the `LL(1)` parsing algorithm using C language in the following. Given a context-free grammar (`CFG`) `G`, we do the following things.

- Compute `FIRST` and `FOLLOW` sets for each grammar variable.

- Compute `PREDICT` set for each production in the grammar. This step requires the sets `FIRST` and `FOLLOW`.
- Compute the `LL(1)` parse table `M[A,a]` stores the production (or production number) that can be expanded for the given grammar variable `A` and terminal `a`. Some of the entries in the table can be empty.
- Finally, run the `LL(1)` parsing algorithm that uses the `LL(1)` table and a stack to check if the given input can be generated by the given `CFG G`.

2.1 Some `CFGs`

Here, we give three `CFGs` that can be used for the reference. We use slightly different notations here. Assume that the length of each variable and each terminal in the grammar is exactly 1. Further, the `empty` string ϵ is denoted with `#`. Further, each production $\texttt{A} \rightarrow \alpha$ is written as $\texttt{A} = \alpha$. In every grammar, the head of the first production is the start variable of the grammar.

```
//Grammar 1
E=TG
G=+TG
G=#
T=FH
H=*FH
H=#
F=(E)
F=i
```

```
//Grammar 2
S=aBDh
B=cC
C=bCc
C=#
D=EF
E=g
E=#
F=f
F=#
```

2.2 Computation of `FIRST` and `FOLLOW` sets

In compiler design, FIRST and FOLLOW sets are crucial for tasks like parsing and grammar analysis, particularly when constructing LL(1) parsing tables.

2.2.1 FIRST Sets

For a given grammar symbol (terminal or non-terminal) X, the FIRST set, denoted as $\text{FIRST}(X)$, contains all the terminal symbols that can appear at the beginning of any string derived from X.

1. **Terminals:** If X is a terminal symbol, then $\text{FIRST}(X) = \{X\}$.

2. **Epsilon:** If $X \to \epsilon$ (epsilon, representing an empty string) is a production, then add ϵ to FIRST(X).

3. **Non-terminals:** If X is a non-terminal and $X \to Y_1 Y_2 \dots Y_k$ is a production:

   (a) Add any non-ϵ symbols from FIRST(Y_1) to FIRST(X).

   (b) If ϵ is in FIRST(Y_1), add any non-ϵ symbols from FIRST(Y_2) to FIRST(X), and so on, until you reach a Y_i where ϵ is not in FIRST(Y_i) or you have considered all Ys.

   (c) If ϵ is in FIRST(Y_i) for all i, then add ϵ to FIRST(X).

2.2.2 FOLLOW Sets

For a given non-terminal A, the FOLLOW set, denoted as FOLLOW(A), contains all the terminal symbols that can appear immediately to the right of A in some sentential form (a string of terminals and non-terminals derived from the start symbol).

1. **Start Symbol:** If S is the start symbol, then add '$' (end-of-input marker) to FOLLOW(S).

2. **Productions:** For a production $A \to \alpha B \beta$, where α and β are any strings of grammar symbols:

   (a) Add all non-ϵ symbols from FIRST(β) to FOLLOW(B).

   (b) If ϵ is in FIRST(β), or if β is empty, then add everything in FOLLOW(A) to FOLLOW(B).

2.2.3 Example

Consider the following grammar:

```
E -> T E'
E' -> + T E' | \epsilon
T -> F T'
T' -> * F T' | \epsilon
F -> ( E ) | id
```

FIRST Sets:

- FIRST(E) = FIRST(T) = FIRST(F) = $\{(, \text{id}\}$
- FIRST(E') = $\{+, \epsilon\}$
- FIRST(T') = $\{*, \epsilon\}$

FOLLOW Sets:

- FOLLOW(E) = FOLLOW(E') = $\{), \$\}$
- FOLLOW(T) = FOLLOW(T') = $\{+,), \$\}$
- FOLLOW(F) = $\{+, *,), \$\}$

3 C code to compute `FIRST and FOLLOW` sets

The complete code is uploaded on LMS/Google Classroom and can be found at [1].

Please see the appendix at the end of the document for the explanation.

4 Computation of Predict set

The **Predict Set** of production in a context-free grammar (`CFG`) is used in top-down parsing, particularly in LL(1) parsers. It helps determine which production to use based on the next input symbol.

For a production of the form:

$$A \to \alpha$$

where A is a non-terminal and α is a sequence of grammar symbols, the **Predict Set** is defined as:

$$\text{Predict}(A \to \alpha) = (\text{First}(\alpha) - \{\epsilon\}) \cup \{\text{Follow}(A) : \text{ if } \varepsilon \in \text{First}(\alpha)\}$$

That is:

- If α can derive a string starting with a terminal, those terminals belong to the Predict Set.
- If α can derive ε (empty string), then Follow(A) is also included in the Predict Set.

4.1 Computation of Predict Set

The Predict Set for production $A \to \alpha$ is computed as follows:

1. Compute First(α):
   - If α starts with a terminal, First(α) = {first terminal in α}.
   - If α starts with a non-terminal B, then include First(B) (except ε).
   - If α can derive ε, include ε in First(α).
2. If $\varepsilon \in$ First(α), compute Follow(A):
   - Include Follow(A) in the Predict Set.
3. The final Predict Set is:

$$\text{Predict}(A \to \alpha) = (\text{First}(\alpha) - \{\epsilon\}) \cup \text{Follow}(A) \text{ if } \varepsilon \in \text{First}(\alpha).$$

4.2 Example

Consider the following CFG:

$$S \to aA \mid \varepsilon$$

[1] https://www.geeksforgeeks.org/program-calculate-first-follow-sets-given-grammar/

$$A \to b$$

We compute the Predict Sets:

- First$(aA) = \{a\}$, so Predict$(S \to aA) = \{a\}$.
- First$(\varepsilon) = \{\varepsilon\}$, so Predict$(S \to \varepsilon)$ = Follow(S). If Follow$(S) = \{\$\}$ (end-of-input marker), then Predict$(S \to \varepsilon) = \{\$\}$.
- First$(b) = \{b\}$, so Predict$(A \to b) = \{b\}$.

Thus, the Predict Sets are:

$$\text{Predict}(S \to aA) = \{a\}, \quad \text{Predict}(S \to \varepsilon) = \{\$\}, \quad \text{Predict}(A \to b) = \{b\}.$$

Refer to lecture slides 7-9 and tutorial 4 for more examples of the prediction set.

Task 1 `Compute the predict set for each production in the given CFG.`

5 `LL(1)` table construction

An LL(1) parser is a top-down parser that uses a single lookahead symbol to make parsing decisions. The LL(1) parsing table is a two-dimensional table `M[ , ]` where:

- Rows correspond to non-terminals.
- Columns correspond to terminals (including the end-of-input marker \$).
- Entries contain the production rules to be applied when a specific non-terminal and terminal pair is encountered.

5.1 Fill the LL(1) Parsing Table

For each production $A \to \alpha$, place it in the parsing table `M[ , ]` as follows:

- For each terminal $a \in$ Predict$(A \to \alpha)$, add the production to the parsing table at row A and column a, i.e.,
$$\texttt{M[A, a]} = \texttt{A} \to \alpha$$

A grammar is LL(1) if:

- Each table entry contains at most **one** production.
- If multiple productions appear in any cell, the grammar is not LL(1).

5.2 Example

Given the CFG:

$$
\begin{aligned}
&(1) \quad E \to TG \\
&(2) \quad G \to +TG \\
&(3) \quad G \to \varepsilon \\
&(4) \quad T \to FH \\
&(5) \quad H \to *FH \\
&(6) \quad H \to \varepsilon \\
&(7) \quad F \to (E) \\
&(8) \quad F \to id
\end{aligned}
$$

The First and Follow sets are computed as follows:

Non-terminal	FIRST Set	FOLLOW Set
E	$\{(, id\}$	$\{), \$\}$
G	$\{+, \varepsilon\}$	$\{), \$\}$
T	$\{(, id\}$	$\{+,), \$\}$
H	$\{*, \varepsilon\}$	$\{+,), \$\}$
F	$\{(, id\}$	$\{*, +,), \$\}$

Table 1: FIRST and FOLLOW Sets

The predict set for each production is computed as follows:

- Predict(1) = First(TG) = $\{(, id\}$
- Predict(2) = First($+TG$) = $\{+\}$
- Predict(3) = Follow(G) = $\{), \$\}$ (since $\varepsilon \in$ First(ε))
- Predict(4) = First(FH) = $\{(, id\}$
- Predict(5) = First($*FH$) = $\{*\}$
- Predict(6) = Follow(H) = $\{+,), \$\}$ (since $\varepsilon \in$ First(ε))
- Predict(7) = First((E)) = $\{(\}$
- Predict(8) = First(id) = $\{id\}$

`LL(1) Parse Table is given below`

	id	(	)	+	*	$
E	$E \to TG$	$E \to TG$				
G			$G \to \varepsilon$	$G \to +TG$		$G \to \varepsilon$
T	$T \to FH$	$T \to FH$				
H			$H \to \varepsilon$	$H \to \varepsilon$	$H \to *FH$	$H \to \varepsilon$
F	$F \to id$	$F \to (E)$				

Task 2 Compute the LL(1) table for a given CFG.

For simplicity, take `M[ , ]` as a two-dimensional array of integers, and the integer we store here is the production number instead of the production.

The resultant table for the above grammar is given below:

$M[A, a] =$

	id	(	)	+	$*$	$
E	1	1				
G			3	2		3
T	4	4				
H			6	6	5	6
F	8	7				

6 LL(1) parsing algorithm

An LL(1) parser uses a stack and the parsing table to determine the sequence of productions to apply. The algorithm follows these steps:

1. Initialize a stack with the start symbol (at the top) and push the end-of-input marker $ (below the top). At this stage, the stack contains only two symbols: $ and the grammar's start symbol.
2. Read the first input symbol.
3. Repeat until the top of stack is not equal to `$`:
   - If the top of the stack is a terminal and matches the input, pop it and advance the input.
   - If the top of the stack is a terminal but does not match the input, print an error message and stop.
   - If the top of the stack is a non-terminal, consult the parsing table using the current input symbol.
   - If a production is found, replace the non-terminal on the stack with the production's right-hand side.
   - If no entry exists, report a parsing error.
4. If the stack is empty (the top is `$`) and the input is completely read, accept the input; otherwise, reject it.

An outline of a C implementation is given below.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define STACK_SIZE 100
#define INPUT_SIZE 100

char stack[STACK_SIZE];
int top = -1;
```

```c
void push(char c) {
    if (top < STACK_SIZE - 1)
        stack[++top] = c;
}

char pop() {
    return (top >= 0) ? stack[top--] : '\0';
}

char peek() {
    return (top >= 0) ? stack[top] : '\0';
}

void parse(char *input) {
    push('$');
    push('E'); // Start symbol

    int i = 0;
    while (top >= 0) {
        char X = peek();
        char a = input[i];

        if (X == a) {
            pop();
            i++;
        } else if (//X is a grammar symbol ) {

            // Lookup table (simplified example for demonstration)


            pop();


        // Push corresponding production onto stack (reverse order)
        } else {
            printf("Error: Unexpected symbol %c\n", a);
            return;
        }
    }
    printf("Parsing successful!\n");
}

int main() {
    char input[INPUT_SIZE];
    printf("Enter input string: ");
    scanf("%s", input);
    strcat(input, "$"); // Append end marker
    parse(input);
    return 0;
}
```

Task 3 Give a complete C implementation of the LL(1) parsing algorithm that takes a CFG (from a .txt file) and a string as input and verifies where the string can be generated by the grammar.

Sample input and output for the grammar 1 (given above) are as follows:

```
Enter input string: i + ( i * i )
Output: Accepted
```

Stack	Input	Rule
$\$E$	$i+(i*i\$$	
$\$GT$	$i+(i*i)\$$	$E \to TG$
$\$GHF$	$i+(i*i)\$$	$T \to FH$
$\$GHi$	$i+(i*i)\$$	$F \to i$
$\$GH$	$+(i*i)\$$	
$\$G$	$+(i*i)\$$	$H \to \#$
$\$GT+$	$+(i*i)\$$	$G \to +TG$
$\$GT$	$(i*i)\$$	
$\$GHF$	$(i*i)\$$	$T \to FH$
$\$GH)E($	$(i*i)\$$	$F \to (E)$
$\$GH)E$	$i*i)\$$	
$\$GH)GT$	$i*i)\$$	$E \to TG$
$\$GH)GHF$	$i*i)\$$	$T \to FH$
$\$GH)GHi$	$i*i)$	$F \to i$
$\$GH)GH$	$*i)\$$	
$\$GH)GHF*$	$*i)\$$	$H \to *FH$
$\$GH)GHF$	$i)\$$	
$\$GH)GHi$	$i)\$$	$F \to i$
$\$GH)GH$	$)\$$	
$\$GH)G$	$)\$$	$H \to \#$
$\$GH)$	$)\$$	$G \to \#$
$\$GH$	$\$$	
$\$G$	$\$$	$H \to \#$
$\$$	$\$$	$G \to \#$

```
Enter input string: ( i + i ) * i
Output: Accepted
```

Stack	Input	Rule
$\$E$	$(i+i)*i\$$	
$\$GT$	$(i+i)*i\$$	$E \to TG$
$\$GHF$	$(i+i)*i\$$	$T \to FH$
$\$GH)E($	$(i+i)*i\$$	$F \to (E)$
$\$GH)E$	$i+i)*i\$$	
$\$GH)GT$	$i+i)*i\$$	$E \to TG$
$\$GH)GHF$	$i+i)*i\$$	$T \to FH$
$\$GH)GHi$	$i+i)*i\$$	$F \to i$
$\$GH)GH+$	$i)*i\$$	
$\$GH)G+$	$i)*i\$$	$H \to \#$
$\$GH)GT+$	$i)*i\$$	$G \to +TG$
$\$GH)GT$	$)*i\$$	
$\$GH)GHF$	$)*i\$$	$T \to FH$
$\$GH)GHi$	$)*i\$$	$F \to i$
$\$GH)GH)$	$*i\$$	
$\$GH)G)$	$*i\$$	$H \to \#$
$\$GH)$	$*i\$$	$G \to \#$
$\$GH*$	$i\$$	
$\$GHF*$	$i\$$	$H \to *FH$
$\$GHF$	$i\$$	
$\$GHi$	$i\$$	$F \to i$
$\$GH$	$\$$	
$\$G$	$\$$	$H \to \#$
$\$$	$\$$	$G \to \#$

You can use `https://jsmachines.sourceforge.net/machines/ll1.html` to check your answer.

Optional: print the parse tree if the input can be generated by the grammar.

Appendix

The C code (uploaded on LMS) implements the algorithms for calculating FIRST and FOLLOW sets. It reads the grammar from a file named `grammar.txt`, where each line represents a production rule in the format `NonTerminal=Production`. The code then computes and prints each non-terminal's FIRST and FOLLOW sets.

6.1 Reading Input (`main` function)

- The code opens the file `grammar.txt` for reading.
- The `fgets` function reads each line (production rule) from the file and stores it in the `production` array.
- `strcspn` is used to remove the newline character at the end of each line.

- The `count` variable stores the total number of production rules.

6.2 `findfirst(char c, int q1, int q2)`: Compute FIRST set

Purpose: Recursively computes the FIRST set for a given non-terminal `c`.

Base Cases:

- If `c` is a terminal (not uppercase), it is added to the `first` set.
- If a production rule for `c` starts with '#' (epsilon), '#' is added to the `first` set.

Recursive Cases:

- Iterates through all production rules where `c` is the left-hand side (LHS).
- If the right-hand side (RHS) starts with a terminal, that terminal is added to the `first` set.
- If the RHS starts with a non-terminal, `findfirst` is called recursively for that non-terminal.
- Handles epsilon productions. If the RHS of a production for `c` is epsilon, and `c` is encountered in another production, the first of the subsequent symbols is added to the first set of `c`.

Data Structures:

- `first[10]`: Stores the FIRST set of a symbol during computation.
- `calc_first[10][100]`: Stores the final calculated FIRST sets for all non-terminals.

6.3 `follow(char c)`: Compute FOLLOW set

Purpose: Computes the FOLLOW set for a given non-terminal `c`.

Base Case: If `c` is the start symbol (LHS of the first production), '$' (end-of-input) is added to its `follow` set.

Logic:

- Iterates through all production rules.
- If `c` appears on the RHS of a production:
  - If the symbol immediately following `c` is a terminal, that terminal is added to `follow(c)`.
  - If the symbol immediately following `c` is a non-terminal, the FIRST set of that non-terminal (excluding epsilon) is added to `follow(c)`.
  - If `c` is the last symbol on the RHS, the FOLLOW set of the non-terminal on the LHS is added to `follow(c)`.

Recursive Calls: `follow()` can call itself if the last symbol on the RHS is a non-terminal.

Data Structures:

- `f[10]`: Stores the FOLLOW set of a symbol during computation.
- `calc_follow[10][100]`: Stores the final calculated FOLLOW sets for all non-terminals.

6.4 `followfirst(char c, int c1, int c2)`

Purpose: Helper function used by `follow()` to handle cases where a non-terminal's FIRST set needs to be included in another non-terminal's FOLLOW set. Handles cases where the FIRST set might contain epsilon.

Logic:

- If `c` is a terminal, it's added to the `f` set.
- If `c` is a non-terminal, it retrieves the pre-calculated FIRST set of `c` from `calc_first`.
- Adds the non-epsilon symbols from `first(c)` to `f`.
- If epsilon is in `first(c)`, recursively calls `followfirst` for the symbol following `c` in the original production rule (handled by `c1` and `c2`).

6.5 `main()` Function (Orchestration)

- Reads the grammar from the file.
- Calls `findfirst()` for each non-terminal.
- Calls `follow()` for each non-terminal.
- Prints the calculated FIRST and FOLLOW sets.