Language Processors - 2024-2025

Degree in Applied Mathematics and Computation

FIRST Assignment - Recursive Descent Parser

Introduction

The task involves designing a translator that converts expressions from a simple prefix notation language into their equivalent infix notation.

An expression in infix notation like `1+2*3` is translated to prefix notation as `(+ 1 (* 2 3))`. In the conversion process[1], parentheses are used to establish the relationship between operators and operands, ensuring that precedence and associativity follow conventional rules.

Additionally, parentheses allow for overriding the default evaluation order. For example, the prefix expression `(* (+ 1 2) 3)` translates to the infix notation `(1+2)*3`, explicitly altering the standard precedence of operations..

The translator's task is to convert prefix expressions into their equivalent infix form:

```
(+ 1 (* 2 3))     ⇒     (1 + (2 * 3))
(* (+ 1 2) 3)     ⇒     ((1 + 2) * 3)
```

Parentheses are included in the output, which in some cases are not necessary, because they ensure that the expression's intended evaluation order is always preserved.

Tasks to perform

1. **Read the Full Statement**.
2. **Review the code provided in `drLL.C`**. What lexical items does `rd_lex()` provide?
3. **Design a Grammar** that represents the syntax of prefix arithmetic expressions as specified.
4. **Redesign the Grammar until LL(1) Compliance is met**.
   a. **Use the JFLAP tool** to determine if the grammar satisfies LL(1) conditions, and to calculate First and Follow Sets.
   b. **Start with a Simplified Grammar** containing at most one or two operators and single-digit numbers.
   c. **Expand the Grammar** gradually including variables and additional specifications.
   d. **Follow Notation Rules** using single lowercase letters for terminals and uppercase letters for non-terminals. Avoid whitespaces in JFLAP.
5. **Develop a Recursive Descent Parser**
   a. **Based on the Grammar** to recognize prefix notation expressions.
   b. **Incorporate Translation Semantics** to output the equivalent infix expressions.
6. **Test Extensively** with a comprehensive set of test expressions.

[1] Conversions between prefix, postfix and infix notations are performed by applying traversals in pre-, post- and in-order of the syntactic tree that represents the expression.

Initial Specifications

The input language follows these rules:

- A single **`InputLine`**[2] contains an **`Expression`** finished by **`\n`**. The output consists of the **`Expression`** translated into infix notation, also ending with **`\n`**
- An **`Expression`** can be:
  - A sequence enclosed in parentheses: **`(Operator Parameter Parameter)`**. For simplicity, the first element will be an **`Operator`**, followed by exactly two elements of type **`Parameter`**.
  - a single element of type **`Number`** (or later a **`Variable`**).
- An **`Operator`** will be one of **`+`**, **`-`**, **`*`**, **`/`**, representing basic arithmetic operations. It is treated as a *Token.*
- A **`Parameter`** can be either an **`Expression`**, or a `Number`.
- A **`Number`** will be an integer of one or more digits, also treated as a *Token*.

Constraints:

- We will not consider the unary signs **`-`** and + as part of our grammar (neither at the syntactic nor lexical level). Cases such as **`-(1+2)`**, **`--1`** and **`-+2`** will not be handled.
- The parser should reject expressions like **`(((* 3 2)))`** or **`((* 3 2))`** as they are not syntactically valid.

Once the grammar is designed and validated as LL(1), additional specifications can be incorporated.

Extended Specifications:

- **`Variables:`** We will include single-letter variables (uppercase or lowercase), optionally followed by a digit. A **`Variable:`**
  - will be represented by a specific *Token*.
  - can appear in an expression as a **`Parameter`**, just like a **`Number`**. A translation example would be:

    `(+ A2 (* B 3))` ⇒ **`(A2 + (B * 3))`**
  - can be assigned the value of an **`Expression`** using the formulation (= **`Variable Parameter`**). Translation example:

    `(= A3 (* 2 B))` ⇒ **`(A3 = (2 * B))`**

- **Chained Assignments** are allowed in some programming languages:
  - **`a=b=c+3 ;`** is valid in C.
  - **`(setq a (setq b (+ c 3)))`** is valid in LISP, a language that we will see later (**`setq`** is equivalent to = in our case).

  Translation example:

  `(= a (= b (+ c (+ 2 3))))` ⇒ **`(a = (b = (c = (2 + 3)))`**

- **Mixed expressions**. Although it may seem unusual, the following expression:
  - **`a = (b=2) + (c=3) ;`** is valid in C Language
  - **`(setq a (+ (setq b 2) (setq c 3)))`** is valid in LISP

  The parser should translate:

  `(= a1 (= b (+ c3 (+ 2 3))))` ⇒**`(a1 = (b = (c3 = (2 + 3)))`**

[2] The parser will evaluate one or more expressions using the **`while`** loop in the **`main()`** function.

Translation examples:

-	`A`	⇒ `A`
-	`123`	⇒ `123`
-	`z7`	⇒ `z7`
-	`(+ A 122)`	⇒ `(A + 122)`
-	`(= A (+ A 1))`	⇒ `(A = (A + 1))`
-	`(+ A (* B 4))`	⇒ `(A + (B * 4))`
-	`(= A (* 1237 B))`	⇒ `(A = (1237 * B))`
-	`(= a (= b (+ c (+ 2 3))))`	⇒ `(a = (b = (c = (2 + 3)))`
-	`(= a (+ (= b 2) (= c 3)))`	⇒ `(a = ((b = 2) + (c = 3)))`
-	`(= a1 (= b (+ c3 (+ 2 3))))`	⇒ `(a1 = (b = (c3 = (2 + 3)))`

Submission Guidelines:

- **First Submission:** Submit your work completed so far (on the first day)-.
- **Final Submission:** Due on Monday, March 10th.

Check the submission entry in Aula Global for specific instructions.

For the Final Submission, upload the Following Files:

1. Code file **drLL.c** , identifying the authors in the header.
    a. **First line**: Full names and team number.
    b. **Second line**: Emails, separated by spaces.
2. Report **drLL.pdf** (about 10p), describing (use the guided practice as an outline):
    a. What lexical items does the **`rd_lex()`** function provide?
    b. Grammar
        i. Initial one.
        ii. Final complete LL(1) which must be clearly Identified.
        iii. The transformations applied.
        iv. Clear distinction between lexical and syntactic levels.
        v. Other relevant information.
    c. Summary of the parser design.
    d. Syntactic diagram representing the designed grammar.
    e. Translation and evaluation tests carried out.
3. Test suite **drLL.txt** with an extensive and well-thought-out set of test cases.

Important:

- **Submit as a draft first**—do not consolidate immediately. Draft submissions are always easier to correct and adjust.
- Before finalizing, **download and verify** the submitted files to ensure they work correctly and comply with the instructions.
- **Respect the regulations and the code of ethics**. Including the name of a teammate who did not participate in the assignment will be considered fraud, which could result in a failing grade in the continuous assessment for both and even other sanctions.

General Recommendations:

- Consult your instructors about your designed grammar before proceeding with the parser development.
- Carefully study the provided code (**drLL.c**).
- To simplify implementation, you can first develop the parser without including parentheses in the infix notation output. Later, analyze where to insert parentheses to ensure the infix expression maintains the same operation as the input.
- Use the following convention: The function **`MatchSymbol()`** will ensure that the next *Token* to be processed is always available by calling the **`rd_lex()`** function (check the code). If you always use **`MatchSymbol()`** to check *Tokens*:
    - There will be no need to include direct calls to **`rd_lex()`**.
    - The token value being checked should be preserved beforehand or accessed through **`tokens.old_token`**.
- To validate expressions in prefix notation you can use the online Lisp interpreter https://rextester.com/l/clisp:
    - Assignments in Lisp are made by changing **`=`** by **`setq`**
    - To print a Lisp expression use **`(print <expression>)`**, For example: **`(print (= a (+ 1 2)))`**

Keep the following indications in mind, as they will influence the evaluation:

- The grammar should be as concise and simple as possible while maintaining a clear and well-structured design.
- Each **`ParseX()`** function must:
    - Include comments specifying the productions it represents.
    - Match the grammar described in the report.
    - Have a meaningful name (the same as the non-terminals).
- The program should be simple, well-structured, properly indented, easy to understand, and without unnecessary complexity. **Avoid**:
    - Adding global variables beyond those already provided.
    - Printing to strings or temporary files.
    - Using additional data structures such as stacks.
    - Using **`while`** or **`for`** loops outside of functions **`main()`** and **`rd_lex()`**, Instead, recursion should be used, following the philosophy of recursive descent parsers.
    - Modifying the **`main()`** function.
    - Modifying the **`rd_lex()`** function, as it is functionally valid as provided. Changes should only be considered if altering the *Token* structure.
    - Writing any output other than the translated expression.
- The translator should be able
    - To output infix expressions that evaluate identically as the corresponding prefix input.
    - To reject input expressions that are not syntactically valid.

Recommendations for the Report:

You are required to write a report explaining the grammar and the translator you have developed. It is recommended to follow the same format used in the guided practices.

Report Specifications

- **Length**: Approximately 10 pages.
- **Page numbering**: Check that pages are numbered.
- **Identification of authors:** should be included in the header of the report.
- **Grammar**: Provide a brief description of the development of grammar, including information relevant to the Recursive Downward Parser. Remember to differentiate the lexical level from the syntactic level.
- **Parser design**: Describe concisely how the grammar productions have been translated into code, highlighting relevant issues.
- **Formatting style**: Follow the format of the guided practice, interspersing explanations with code and productions, for example.
- **Syntactic Diagrams** representing the designed grammar are requested.
- **Testing Information**: Discuss the tests you designed in an annex.

Presentation Recommendations

- **Avoid Screenshots for code and tests:**
    - Screenshots are difficult to read and can appear visually harsh, especially with black backgrounds.
    - Explore alternative methods such as using Markdown (easily generated in Visual Studio), exporting code to RTF, or using built-in formatting options in Google Docs. These options provide a more professional and readable presentation.
    - If a screenshot is necessary (e.g., for JFLAP), ensure that its content is readable without requiring zoom.
- **Use of Colors**: This is optional, but if preferred, some terminals:
    - Allow copying and pasting code while preserving colors.
    - You can also configure color schemes for better readability.
- **Font Choice**: Use fixed-width fonts such as Consolas or Courier for grammars and code to enhance readability.