Universitat Pompeu Fabra Barcelona

Aranich, Marcel - 251453
Quera, Clàudia - 231197
Villarino, Jorge - 231351
Solans, Arnau - 216530
Garcia, David - 251587
Prat, Ariadna - 251281

DESIGN DOCUMENT

Project Overview

In this lab, we aim to develop the bottom-up parsing algorithm using a shift/reduce automaton (sra) for a given grammar. The grammar consists of production rules for arithmetic expressions involving addition, multiplication, parenthesis, and numeric literals.

The sra will also create an abstract syntax tree (ast) and we will translate it to the corresponding instructions in assembly.

Project Goals and Objectives

The main goal of this project is to implement a parser that can read and evaluate simple arithmetic operations involving addition and multiplication (alongside with parenthesis).

- Use the bottom-up parsing algorithm to detect the correct strings that are accepted by the automaton.
- Use the AST to determine how the operations should be performed.
- Designing a well-structured codebase that works correctly for all the specifications described in the lab statement.

Design of the program

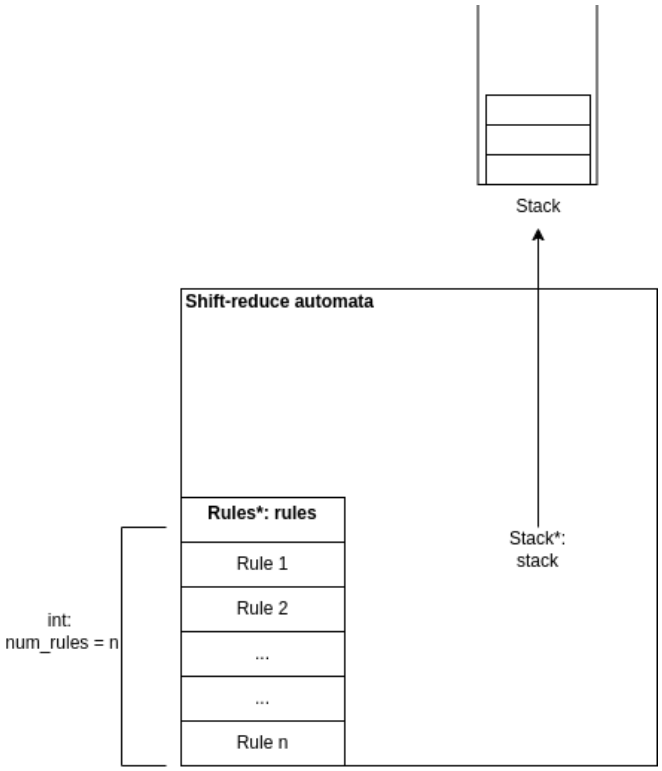
In order to explain the design of our Parser, we will explain the different types of data structures we have used and provide a list of functions that treat that type of struct.

Universitat Pompeu Fabra Barcelona

Aranich, Marcel - 251453
Quera, Clàudia - 231197
Villarino, Jorge - 231351
Solans, Arnau - 216530
Garcia, David - 251587
Prat, Ariadna - 251281

- **Automata**

  - **Shift/Reduce Automata (extension of automata)**



The reduce-shift automata (as an extension of the automata implemented in previous assignment) is the structure in charge of processing the input and applying the production rules usable with the input read. The shift-reduce automata, or RSA in our implementation, has a Stack (which will hold the tokens read and allow to apply production rules if right-hand side or RHS is matched), the set of production rules in our language, an integer to track the number of production rules, and an integer to specify maximum amount of rules that the automata holds pointers to (in order to manage memory when automata needs to track more rules and has reached max capacity).

Universitat Pompeu Fabra Barcelona

Aranich, Marcel - 251453
Quera, Clàudia - 231197
Villarino, Jorge - 231351
Solans, Arnau - 216530
Garcia, David - 251587
Prat, Ariadna - 251281

Function Name	Parameters	Type	Functionality
initialize_rsa	RSA* rsa	void	Initializes the automata (all its components) given a memory space to store it.
add_rule	RSA* rsa, Rule rule	void	Checks if input RSA has capacity for more rules (if not, expands) and adds input rule at current index (counter of rules: num_rules) before incrementing the counter of rules.
advance_rsa	RSA* rsa, Token* token	void	Receives input token and shifts automata (cloning the token in stack) then tries to reduce automata (apply production rules with tokens in stack) making use of next two functions.
shift_rsa	RSA* rsa, Token* token	void	Clones input token and pushes it to the stack of input RSA.
reduce_rsa	RSA* rsa	bool	For each rule stored in RSA retrieves number of operands in RHS of the rule and tries to match it with that same number of tokens from top of the RSA stack. If there is a match, it pops that number of tokens from the top of the

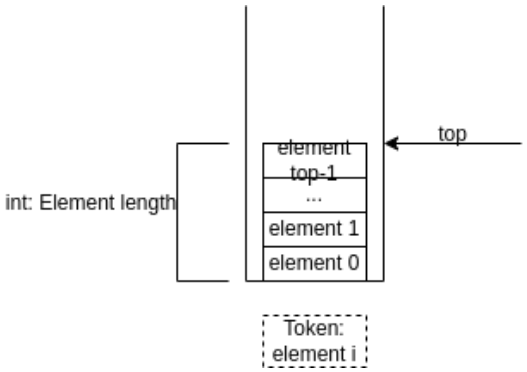
Aranich, Marcel - 251453
Quera, Clàudia - 231197
Villarino, Jorge - 231351
Solans, Arnau - 216530
Garcia, David - 251587
Prat, Ariadna - 251281

			RSA stack and pushes cloned left-hand side or LHS from the rule on top of the RSA stack. If a production rule has been applied, it returns true, if unable to apply, it returns false after trying each production rule.
is_starting_token	RSA* rsa	bool	Checks if the stack has a single token and then checks if that token is the starting element (or goal) of our language. If it is, it returns true, if not, returns false.
print_rsa	RSA* rsa	void	Prints each rule and the stack making use of each structure respective prints.
free_rsa	RSA* rsa	void	Frees each rule the RSA points to, then frees the set of rules, and finally frees the stack.

Universitat Pompeu Fabra Barcelona

Aranich, Marcel - 251453
Quera, Clàudia - 231197
Villarino, Jorge - 231351
Solans, Arnau - 216530
Garcia, David - 251587
Prat, Ariadna - 251281

- **Stack**



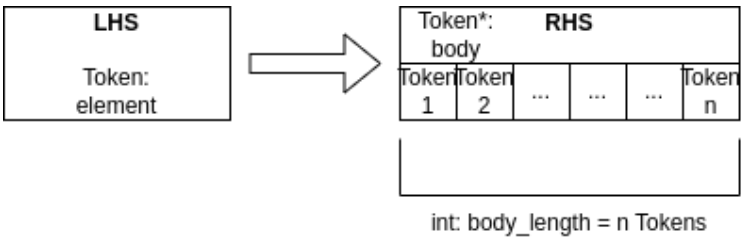
Function name	Parameters	Type	Functionality
initialize_stack	Stack* stack	void	Initialize the elements of the stack
peek_stack	Stack* stack	Option*	Get the top of the stack.
pop_stack	Stack* stack	Option*	Take out the token from the top of the stack
push_stack	Stack* stack, Token* new_token	void	Put the token into the stack
print_stack	Stack* stack	void	Print the tokens from the stack
free_stack	Stack* stack	void	Free the memory from the stack

Universitat Pompeu Fabra Barcelona

Aranich, Marcel - 251453
Quera, Clàudia - 231197
Villarino, Jorge - 231351
Solans, Arnau - 216530
Garcia, David - 251587
Prat, Ariadna - 251281

- **Language**

  - **Production rules**



The language, or set of production rules, that the shift-reduce automata recognizes and processes tokens according to. The language is a collection of production rules that we implemented as a Rule, which has a body (that is, a set of tokens representing RHS), a body length (the number of operands in the RHS), and an element (or the LHS of the production rule).

Function Name	Parameters	Type	Functionality
initialize_rule	Rule* rule, Token* body, int body_length, Token element	void	Clones each operand in the input body into the input rule body (copies RHS), then assigns input body_length to input rule body_length, and assigns input element (LHS) to input rule LHS. Then frees the input collection of tokens.
follows_rule	Rule* rule, Token* rule_candidates, int candidate_len	bool	It first checks if the candidate_len (or input RHS number of operators) matches the input rule RHS number of operators. If not, return false. Else, it checks for each

Aranich, Marcel - 251453
Quera, Clàudia - 231197
Villarino, Jorge - 231351
Solans, Arnau - 216530
Garcia, David - 251587
Prat, Ariadna - 251281

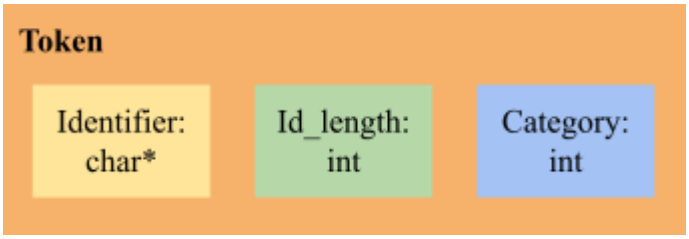
			element of the input rule body and the input rule_candidates if the category of each corresponding token matches. If there is a mismatch, returns false. If all elements match category, returns true.
print_rule	Rule* rule	void	Prints the rule LHS and RHS (token by token), making use of the print_token function of token datastructure.
free_rule	Rule* rule	void	Frees token by token the RHS of the rule, then frees the collection of tokens representing the RHS itself, and finally frees the LHS token.

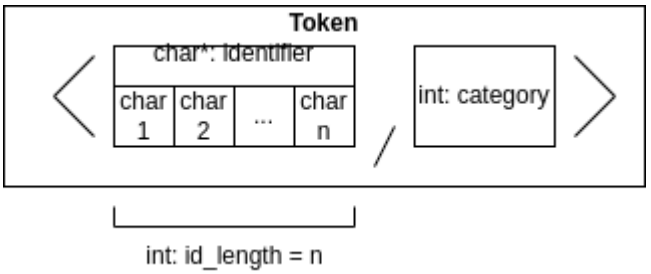
Universitat Pompeu Fabra Barcelona

Aranich, Marcel - 251453
Quera, Clàudia - 231197
Villarino, Jorge - 231351
Solans, Arnau - 216530
Garcia, David - 251587
Prat, Ariadna - 251281

- **Token**

The basic structure that will store the information are the tokens. The tokens store the string of literals, its length and the category they belong to. The file input will contain the tokens that will need to be parsed.





Function name	Parameters	Type	Functionality
getTokens	char* filename	Result<Token* , int>	Parses the content of a file into a list of tokens.
initialize_token	Token* t, char* _identifier, int _categoty	void	Initialize the token with the exact parameters.
clone_token	Token* original_token	clone <Token*>	This function clones the token and returns the same token.
print_token	Token* token	void	This function prints the content inside the token
filter_token	Token* token_list, int* len	void	If the category of the token is good, the token is copied in the writing index position.
free_token	Token* token	void	Free the memory of the token

Universitat Pompeu Fabra Barcelona

Aranich, Marcel - 251453
Quera, Clàudia - 231197
Villarino, Jorge - 231351
Solans, Arnau - 216530
Garcia, David - 251587
Prat, Ariadna - 251281

** Result is an Algebraic data type of sum types. It is generic and has a varian for the result (Ok) and another for the error (Err). In this case it holds Token* if it is the Ok variant or int if it is the error variant.

How it works:

The following block diagram shows the major functions that determine the workflow of our implementation given a preprocessed input and making use of a previously implemented scanner to tokenize the preprocessed input.

When the process starts, we will read the text and parse all the tokens (transform the string into a list of tokens). Then we will create the RSA with its rules. Then we will call advance() on the rsa. This will make the rsa shift (add it to the stack) and start reducing until it can no longer apply any rule. This will repeat until all tokens in the token list are parsed. During that time print statements will be executed to display how the rsa parses the tokens. After parsing everything, if there is only 1 token in the stack of the RSA and it is the starting symbol, then the parsing is successful, otherwise it fails.

Here the code could be expanded to generate the corresponding AST and then the assembly code.

Universitat Pompeu Fabra Barcelona

Aranich, Marcel - 251453
Quera, Clàudia - 231197
Villarino, Jorge - 231351
Solans, Arnau - 216530
Garcia, David - 251587
Prat, Ariadna - 251281