**Emulator for Bottom Up**

**Parsing -CLR**

Project Phase-I report submitted to

M S RAMAIAH INSTITUTE OF TECHNOLOGY

Bengaluru – 560054

By

Anvitha G K Bhat (1MS13CS026)

Gayathri R (1MS13CS042)

Deepak Jayaprakash (1MS13CS037)

As part of the course Compiler Design (CS612)



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

M S RAMAIAH INSTITUTE OF TECHNOLOGY

Jan-May 2016

**Department of Computer Science and Engineering**

**M S Ramaiah Institute of Technology**

**Bengaluru – 54**



**CERTIFICATE**

This is to certify that **Anvitha G K Bhat (1MS13CS026), Gayathri R( 1MS13CS042) and Deepak Jayaprakash**

**( 1MS13CS037)**  have completed the **“Emulator for Bottom Up Parsing- CLR”** for project phase –I.

We declare that the entire content embodied in this B.E. 6th Semester Project report contents are not copied.

Submitted by Guided by

1. **Anvitha G K Bhat**

2. **Gayathri R**

3. **Deepak Jayaprakash**

Mrs. Sini Anna Alex

Dept of CSE, MSRIT)

(Assistant Professor, Dept. of CSE)

Department of Computer Science and Engineering

M S Ramaiah Institute of Technology

Bengaluru – 54



**Evaluation Sheet**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Sl.No** | **USN** | **Literature survey**  **(1m)** | **Abstract**  **(1m)** | **DFD/ Algorithm**  **(2m)** | **Report**  **(1m)** | **Total**  **(5m)** |
| 1. | **1MS13CS026** |  |  |  |  |  |
| 2. | **1MS13CS042** |  |  |  |  |  |
| 3. | **1MS13CS037** |  |  |  |  |  |

Evaluated By

Name: Sini Anna Alex

Designation: Assistant Professor

Department: Computer Science & Engineering, MSRIT

Signature:

ACKNOWLEDGEMENT

We express our sincere gratitude to Mrs. Sini Anna Alex, Dept. of Computer Science and Engineering, MSRIT, for her stimulating guidance, continuous encouragement and supervision throughout the course of present work.

Through the course and the time that we invested in completing the project, we understood the steps of parsing in detail and also the art of representing each individual solution in a systematic and graphical way using good front-end tools.

We thank Mrs. Sini Anna Alex for the opportunity to implement the algorithms that we had learnt in the theory in a real-time project which gave us an in-depth knowledge and expertise on the topic and the subject as a whole.

**Signature(s) of Students**

Anvitha G K Bhat (1MS13CS026)

Gayathri R (1MS13CS042)

Deepak Jayaprakash (1MS13CS037)

ABSTRACT

The objective of this project is to implement an efficient and robust CLR parser using a high level programming language, that is capable of correctly parsing any input fed to it. The LR(1) parsing is a technique of bottom-up parsing. ‘L’ says that the input string is scanned from left to right, ‘R’ says that the parsing technique uses rightmost derivations, and ‘1’ stands for the look-ahead. To avoid some of invalid reductions, the states need to carry more information. Extra information is put into state by adding a terminal symbol as the second component in the item.

Thus the canonical-LR parser makes full use of look-ahead symbols. This method uses a large set of items, called LR(1) items.

The LR(1) parsing method consists of a parser stack, that holds non-terminals, grammar symbols and tokens; a parsing table that specifies parser actions, and a driver function that interacts with the parser stack, table and scanner. The typical actions of a CLR parser include: shift, reduce, and accept or error.

The project work would include a set of predefined grammar and an interface which would convert each phase of the parsing process into a visual representation and would display onto webpage. The implementation is pretty straight forward and simple. Then it would take any input string belonging to the grammar language and would show the acceptance or rejection of that input string and also the steps one by one.

Contents

|  |  |  |
| --- | --- | --- |
| **Sl.No** | **USN** | **Page no** |
| 1. | Introduction | 1 |
| 2. | Literature Review | 2 |
| 3. | Data flow diagram/ Algorithm for design | 3 |
| 4. | Relevance w.r.t compiler phases | 6 |
| 5. | Conclusion | 7 |
| 6. | References | 8 |

Introduction

The LR parser is a non-recursive, shift-reduce, bottom-up parser. It uses a wide class of context-free grammar which makes it the most efficient syntax analysis technique.

LR parsers are also known as LR(k) parsers, where L stands for left-to- right scanning of the input stream; R stands for the construction of right-most derivation in reverse, and k denotes the number of look ahead symbols to make decisions. LR parsing does a rightmost derivation in reverse.

LR(1) parser works on complete set of LR(1) grammar, which makes full use of the look ahead symbols. It generates a large table with a large number of states. An LR(1) item is a two-component element of the form where the first component is a marked production, called the core of the item and the second is a look ahead character that belongs to the super set.

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |

Literature Review

The idea is to generate a good front end for the parsing process and to represent each step of the algorithm in a illustrative graphical pattern which is easy to interpret because of the visual appeal. So the choice of tools and tech used revolve around finding a API which gives us rich UI features and easy to use database to store session id’s and to pass values from screen to another.

**Choice of programming interfaces:**

* **Front end:** The front end is split into 2 things. A general technology which takes care of the overall look and feel and a specific framework which has built in features to achieve graphic utilities.
* **HTML5 +CSS3:** General functionality, rendering of pages, display of each step, navigation from servlet to another.
* **JavaScript:** Front end validations of input grammar. After constructing the parsing table, we accept the input string to check whether it gets accepted by the parser or not. To do so, before submitting to the server, we do some front-end checking. If the input string contains an invalid terminal or non-terminal, we pop it out on form submission itself.
* **Bootstrap:** Twitter Bootstrap is a web-application framework which contains a vast set of library functions to achieve some graphic utilities like modal, tool-tips, carousels, animations, transitions, etc
* **jQuery library:** jQuery UI is a curated set of user interface interactions, effects, widgets, and themes built on top of the jQuery JavaScript Library. Whether you're building highly interactive web applications or you just need to add a date picker to a form control, jQuery UI is the perfect choice.
* **Back –end:** The choice was between PHP which would be the optimal choice for web-programming and Java used as a servlet. But since this involves a lot of string manipulations and set generations we need to have an ample set of built in functions to do so.

Therefore, we went forward with the decision of making use of Java as a servlet.

As of now there exists no need of a separate database for storing temporal data, but for handling sessions and maintain synchronization between multiple servlets we may need a user friendly database.

Mysql is the obvious choice, since it’s easy to manipulate and query. It has got good embedding functions between native user code and sql database core operations. The schema of the database, not decided yet but would be pretty simple and would contain not more than 4-5 fields.

Algorithm

**Algorithm for constructing LR(1) sets of items**

Input: An augmented grammar Gˡ.

Output: The sets of LR(1) items that are the set of items valid for one or more viable prefixes of Gˡ.

SetOfItems CLOSURE(I) {

repeat

for (each item [A→α.B𝛽,a] in I )

for (each production B→ 𝛾 in Gˡ )

for (each terminal b in FIRST(𝛽a) )

add [B→. 𝛾,b] to set I;

until no more items are added to I;

return I;

}

SetOfItems GOTO(I,X) {

initialize J to be the empty set;

for (each item [A→α.X𝛽,a] in I )

add item [A→αX.𝛽,a] to set J;

return CLOSURE(J);

}

SetOfItems items(Gˡ) {

initialize C to CLOSURE({[Sˡ→.S,$]});

repeat

for (each set of items I in C )

for (each grammar symbol X )

if (GOTO(I,X) is not empty and not in C )

add GOTO(I,X) to C;

until no new set of items are added to C;

}

**Algorithm for constructing Canonical LR(1) parsing table**

Input: An augmented grammar Gˡ.

Output: The canonical-LR parsing table functions ACTION and GOTO for Gˡ.

Method:

1. Construct Cˡ = { *I0, I1,....,In*}, the collection of sets of LR(1) items for Gˡ.

2. State *i* of the parser is constructed from *Ii*. The parsing action for

state i is determined as follows.

1. If [A→α.a𝛽,b] is in *Ii* and GOTO(*Ii* ,a) =*Ij* , then set ACTION[*i*,a] to “shift *j*”. Hence a must be a terminal.

(b) If [A→α.,α] is in *Ii*, A ≠ Sˡ, then set ACTION[*i*,a] to

“reduce A→α.”

(c) If [Sˡ→S.,$] is in *Ii*, then set ACTION[*i*,$] to “accept”.

If any conflicting actions result from the above rules, we say the grammar is not LR(1). The algorithm fails to produce a parser in this case.

3. The goto transitions for state *i* are constructed for all nonterminals A using the rule: If GOTO(*Ii* ,a) = *Ij* ,then GOTO[*i*, A] = *j*.

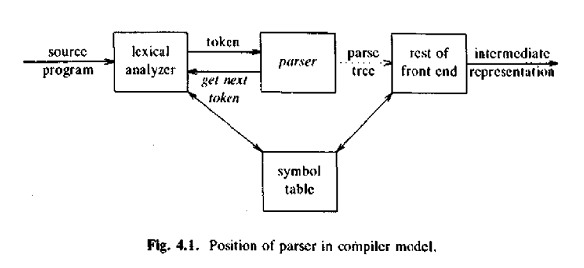
4. All entries not defined by rules (2) and (3) are made “error”.

5. The initial state of the parser is the one constructed from the set of items containing [Sˡ→.S,$].

Data Flow Diagram

Relevance with respect to other compiler phases

In a typical compiler model, the parser obtains a string of tokens from the lexical analyzer, as shown in Fig. 4.1, and verifies that the string can be generated by the grammar for the source language. We expect the parser to report any syntax errors in an intelligible fashion. It should also recover from commonly occurring errors w that it can continue promising the remainder of its input.



The parsing takes place in the syntax analysis phase. The job of this phase is to build a relationship between the lexime values and generate a syntax tree.

One more job of the phase to handle errors. Application programmers frequently write incorrect programs, and a good compiler should assist the programmer in identifying and locating errors.

The generated tree is then passed onto the intermediate code representation. Output of the parser is some representation of the parse tree for the stream of tokens produced by the lexical analyzer. In practice, there are a number of tasks that might be conducted during parsing, such as collecting information about various tokens into the symbol table, per- forming type checking and other kinds of semantic analysis, and generating intermediate code.

Conclusion

The final application interface would represent a pictorial and a visual figure of each step of the process of CLR parsing. The grammar for the parser is assumed i.e. fixed in the initial phase itself and the user is given the freedom of entering any input string possible. The parser would generate LR(1) set of items and display that. Then it would generate a parsing table given when the generated item set is fed on to it.

Then the parsing of any user input is shown step by step. But the only static thing in the project is the set of pre-defined grammar. The add-on to the project would be to generalize any grammar that is accepted by the user and then generate LR(1) set of tokens on it and then parse the input string.

We could add the other parsing techniques to the implementation too and generate an illustrative case study on which parser would be the best fit for a given grammar and for a given set of input strings.

References

* **Compilers: Principles, techniques and tools** by Alfred V Rao, Ravi Sethi, Jeffrey D Ullman
* **LR(1**) **Parsing**: Handout written by Maggie Johnson and revised by Julie Zelenski, Stanford University
* **Clemson University:** [www.cs.clemson.edu/parsing.pdf](http://www.cs.clemson.edu/parsing.pdf)
* **Parsing tables examples and solved grammar: ORCCA**
* Compiler Design Lectures by **Ravi Chandra Babu**
* Bottom Up Parser **Tutorials Point:** www.tutorialspoint.com/.../compiler\_design\_bottom\_up\_**parser**.htm