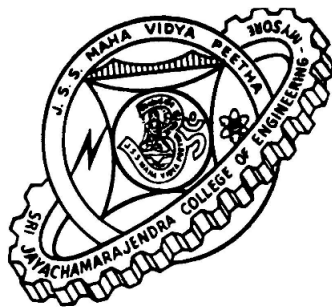


**VISVESVARAYA TECHNOLOGICAL UNIVERSITY
BELGAUM**



**SRI JAYACHAMARAJENDRA COLLEGE OF ENGINEERING
MYSORE-570006**
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING



Project on
DEFINATION OF OUR OWN LAGUAGE β PARSE

Guidance of
P.M.SHIVAMURTHY
Lecturer
Dept of CS&E,SJCE Mysore.

Team:

NAME	ROLL NO	USN
SHARAD D	03	4JC06CS089
PRABHAKAR GOUDA	35	4JC07CS070
VIKRAM TV	59	4JC07CS120

Contents

1	Overview	1
2	Glossary	1
3	References	1
4	Requirement	1
4.1	User requirements	1
4.2	System requirements	2
4.3	Functional and domain requirements	2
4.4	Non-functional requirements	2
5	Form Based Approach	3
6	Data Flow Diagram	3
7	Life Cycle	4
8	Constructs of the Language	5
8.1	File Inclusion	5
8.2	Macro Definition	5
8.3	Function	6
9	Syntax of Statements	7
9.1	Operators and Symbols	7
9.2	Variables	8
9.3	Commenting	8
9.4	Declaration	8
9.5	Assignment	9
9.6	Relational Condition Checking - The ‘test-otherwise-endl’ construct	9
9.7	Looping Construct	10
9.8	Nesting of Statements	10
9.9	Program	11
10	Design of Parser	12
10.1	Architectural Design	12
10.1.1	System Organisation	13
10.1.2	Modules	13
10.1.3	Cohesion	13

10.2 Detailed Design	13
10.3 Interface Design	16
11 Example Program	16
12 Final Deliverable	19
13 Applications	19
14 Future Work	19
15 Conclusion	20

1 Overview

β - parse language is the beta version of parsing of our own language. It involves the best possibilities that we have seen from the classic languages ‘C’, ‘Scripting Languages’ and others. Language contains its own specifications for each statement. It takes program written as per the rules and syntax of our language as input and validates the syntax of the program and displays the result as *valid* or *invalid* program. In case of any error in the program the system counts number of errors as well as display the line number in which the error is detected.

2 Glossary

β parse	It is our own language defined.
source program	User program written in β parse spec.
keywords	The keywords supported by our language.
HTML	Hyper Text Markup Language.
dotty	Product to support graph display.
kgraphviewer	Product to support graph display (kde version).
*.vps	Our own language extension although any extension is supported.

3 References

- Lex and Yacc by John R. Levine, Tony Mason, Doug Brown - O'Reilly Publications
- A Compact Guide to Lex and Yacc by Tom Niemann at epaperpress.com
- Wikipedia

4 Requirement

4.1 User requirements

- System allows users to enter data from source file containing program in β parse.
- User can install the product by *make install*.
- Source program can be parsed using $\$betaparse \textit{filename}$

- User can view the created parse tree by *make tree*.
- The errors are displayed by *make view*. Displayed in the browser (firefox).
- The colours are used to differentiate errors in program.
- The product can be uninstalled by *make uninstall*.

4.2 System requirements

- Machine must run in Linux platform.
- Machine must support Lex and YACC.
- Machine must have the system with 256MB of RAM or better.
- GNU C compiler (cc or gcc).
- Machine must contain kgraphviewr or dotty for displaying the tree.

4.3 Functional and domain requirements

- All source files are visible to user.
- It must display the line number of the invalid syntax in case of β parse programs.
- The parse tree display is done using *kgraphviewer* or *dotty*.
- The final validated program is displayed in *html* format.
- The different colours are used for user friendliness.

4.4 Non-functional requirements

- Easy to use User interface.
- Efficiency in analyzing.
- Versatility in handling all types of C assignment source code.
- The creation of parse tree for both error and error-free program is must.
- The parse tree shows red colour for invalid statement in program.
- The program may contain any extention(like *.txt, *.vps)

5 Form Based Approach

Function:

Defination of language β parse.Checks for correctness of the program written in β parse.

Description:

Program takes input from file and analyses the syntax of the statements and returns the validity of the program.

Input:

Program defined in language β parse.

Output:

Validity of the program is displayed on screen and parse tree in created for the same program.

Action:

Program is parsed. The lexical analyser searches for tokens from input stream and returns the token. Yacc parses the input and sets the valid bit to '0' if statement is valid else sets to '1'.The statement containing error the line number of error is displayed.The parser make call to write the file for tree.Each statement in program indicates the node in parse tree.The final program execution can be simulated by reading the leaf nodes of the parse tree.

Requires:

Machine must contain Lex and Yacc packages.The kgraphviewer or dotty packages for tree display.

Pre-condition:

The input file must exist in the specified path.

6 Data Flow Diagram

The data flow diagram shows that the program writen in β parse language is taken as input, which is passed to the lexical analyser.The lexical analyser returns tokes to syntax analyser.The syntax scanned by the lexar is outputed in the form of parse tree.

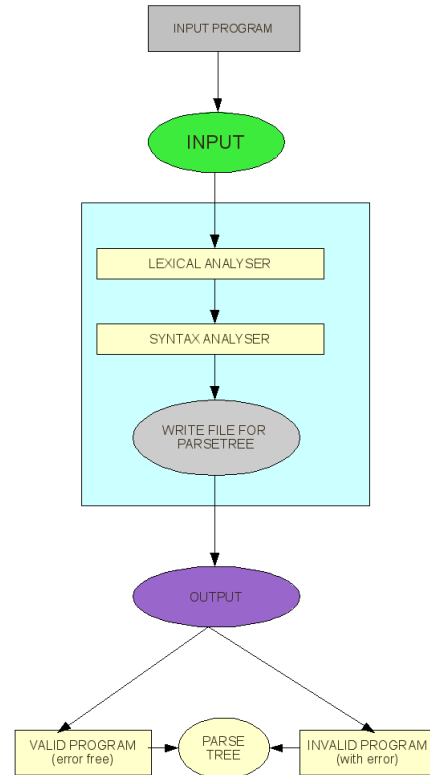


Figure 1: Data Flow Diagram

7 Life Cycle

The parser is evolved in various phases. It is to be evolved during

- Requirements
- Design
- Implementation
- Optimization
- Testing
- Installation

- Operation and Maintenance

The product is developed making use of the *spiral development model*. Each release is updated with new functionalities and features. Development activities starts from the design phase of previously released versions. The product is released based on user suggestions and optimizations.

8 Constructs of the Language

The program consists of 3 parts:

- The ‘file inclusion’ part
- The ‘macro definition’ part
- The ‘function definition’ part

8.1 File Inclusion

File inclusion is done using the keyword **include**.

Syntax:

include *filename*

Filenames should not contain any extensions. Hence unique files are included. The include directive can be placed anywhere in the program and every file that is to be included should in a separate line.

Example: include stdio

The directive includes the file ”stdio” to the current scope.

8.2 Macro Definition

Macro Definition has the keywords **macro** to begin the macro definition and **endm** to end the macro definition.

Syntax:

macro *macroName code* **endm**

The above syntax defines a macro with name ‘macroName’ having the body ‘code’ and ended with keyword endm. The macroName should not

contain any special characters and its declaration is similar to the declaration of the 'C' variables. The 'code' part can contain any assignments, function calls, relational condition structures, looping structures *except* the function definition itself. This is to distinguish between function characters from macro characters. The code can be spread over multiple lines.

The macro definitions can be included anywhere in the program but not inside any functions.

Example: macro aIncrement a++; endm

```
or
macro aIncrement
    a++;
    .
    .
endm
```

8.3 Function

Function can be started by specifying the function name and parameter list for the function enclosed within parenthesis. Every function is terminated by the keyword **endf**.

Syntax:

functionName (parameterList) *body* **endf**

The parameter list can be zero or more. The elements of the list should be separated by comma (,). The function body can contain any number of basic statements like assignment, increment, decrement that resemble the 'C' statements. The body can also include the relational operations, looping operations and function calls also.

Example: main(argc, argv) s = p + v; a++; endf

```
or
main(argc, argv)
    s = p + v;
    a++;
    .
    .
endf
```

Every statement within the function body should end with a semicolon (;) similar to 'C' language syntax. The *return value* of the function lies in

its name only.

The details of the various types of statements in our language construction is dealt in the next section.

9 Syntax of Statements

The following are the various types of syntax that are used:

9.1 Operators and Symbols

The following **operators** are supported in the language. It includes the basic arithmetic operators.

- + to perform binary addition of two variables.
- to perform binary subtraction of two variables.
- * to perform binary multiplication of two variables.
- / to perform binary division of two variables.
- , (comma) to separate the variable declarations and also to separate assignment statements.
- = to assign the rvalue obtained by an expression to the lvalue.
- ^ to perform the binary power operation.
- % to perform the modulo operation.

Relational Operators:

The following operators on two variables or expressions returns a boolean value. These operators are used to test the various conditions in the language.

- == checks for equivalence of two variables or expressions.
- != checks if left variable or expression is equal to or not to the right.
- > checks if left variable or expression is greater than to the right.
- < checks if left variable or expression is less than or equal to the right.
- >= checks if left variable or expression is greater than or equal to the right.
- <= checks if left variable or expression is less than or equal to the right.

Ternary Operators:

? : Checks for the condition and assigns the value between '?' and ':' or the value after ':'.

Structural Operators:

- . to access the members of a structure using the object (as implemented in C).
- > to access the members of a structure using pointer operation (as implemented in C).

The following **symbols** are used:

- ; to terminate a statement in the program.
- () to specify a function call and the precedence of evaluation.
- [] to specify the array related activities (similar to 'C')

9.2 Variables

The language supports all the characters (both uppercase and lowercase) of the English Language. It also supports the integer values. Any variable can be declared on the lines of variable declaration in 'C'.

- The variable can have both the characters and integers in it.
- The variable should start with an English character and not by an integer.
- The variable can be of any length.
- Keywords of the language cannot be used as variables.

9.3 Commenting

Commenting in the language is similar to the commenting styles in C++ language.

Syntax:

`// comment`

The entire line after '`//`' is considered as a comment and it is not parsed.

9.4 Declaration

The type declaration of variables used in the program are not needed and the variables can be used directly, although some of the basic datatypes (integer, float, character) are provided. The variable is casted according to the rvalue used in the program. Also multiple declaration of variables in a single line separated by commas are possible. This form of declaration is inspired from the 'shell scripting' language. The declarations are to be terminated by a semicolon to mark the end.

The declaration of a new variable is made at its first assignment.

Example: `a = 3;`

This declares the variable 'a' and assigns to it the value 3.

9.5 Assignment

The assignment statements are implemented based on the syntax of assignments done in 'C' Language. The 'rvalue' is assigned to the 'lvalue'. The left value can consist only one variable and not a number as a single value cannot be assigned to two or more variables on left. The right value can consist of a variable, number or a combination of both to form a compound expression.

All assignment statements should terminate with a semicolon.

The following assignment statements are supported:

- Direct Assignment as '`a = b;`'
- Compound Assignment as '`a += b;`'
- Multiple Assignment (partially implement as of now) as '`a = b = ... = z;`'
- Increment and Decrement as '`a++;`', '`++a;`', '`a--;`' or '`--a;`'.
- Structural operational assignments (as in C) as '`a->node = b.leaf;`'.
- Evaluation of expressions within '`()`', that has the highest order.
- Array assignments as '`a[i] = b[j];`'
- *Function Calls* can also be made and the return values can be assigned as '`a = hello();`'.

9.6 Relational Condition Checking - The 'test-otherwise-endl' construct

The language uses the following syntax to perform the relational condition checking.

Syntax:

`test condition statements1 endt`

or

test *condition* statements1 **otherwise** statements2 **endt**

The test keyword tests for the boolean value of the condition. The condition uses the relational operators between two variables or expressions. If the condition is true, then the *statements1* are executed. In the second case, if the condition results false then the keyword otherwise is parsed to execute the *statements2*. The keyword endt is compulsory to mark the end of the ‘test-otherwise-endt’ relational construct.

The statements1 and statements2 can inturn contain any number of nested ‘test-otherwise-endt’ conditions.

9.7 Looping Construct

The looping construct is implement in the language using the keywords **loop**, **from**, **step** and **endl**.

Syntax:

loop *condition* statements **endl**

or

loop *condition* **from** assignment **step** assignment statements **endl**

Looping of the *statements* is done only if the *condition* is satisfied. It can also be supplement with keywords - from and step that determine the start and end of the looping. The looping structure is finally terminated by endl keyword.

The statements can inturn contain any number of nested looping structures.

9.8 Nesting of Statements

The language supports nesting of statements.

The statements containing the ‘test-otherwise-endt’ and ‘loop-from-step-endl’ can be nested. Any number of statements can be used in them. Also the nested can be to any level in depth. The ‘test’ can nest ‘test’ or ‘loop’ or both.

Similarly, ‘loop’ can nest ‘loop’ or ‘test’ or both. The macro definitions can also be nested and they may be defined anywhere in the program but between the functions.

Thus any of the *nesting* combinations works well with the language.

9.9 Program

The language supports to have *any number of* ‘functions’, ‘macros’, ‘file inclusions’, ‘conditional executions’, ‘looping constructs’ and ‘statements’ in the program. Each construct is supposed to be terminated by appropriate terminator. Finally the program itself needs to be terminated by the keyword **endp**.

10 Design of Parser

The source program contains our language. The parser checks the source program for errors, that is identifies which part in the source program has deviated from our language specification. It finally output the errors.

10.1 Architectural Design

The design involves identifying the different tokens and keywords of the language and integrate them with rules that are specified in the language specification.

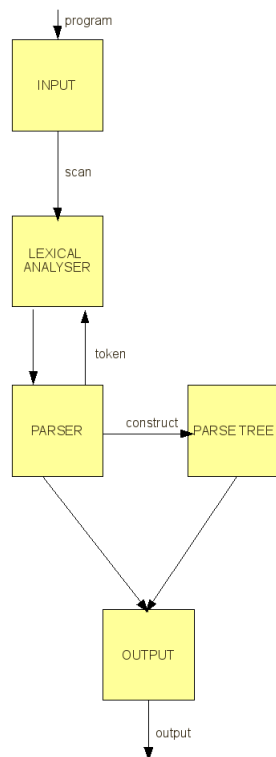


Figure 2: Design Process

10.1.1 System Organisation

The source file can be accessed by the tokens and keywords identifier *and* also by the grammar checker. Hence a shared data repository model is feasible. The shared data here is the source program.

10.1.2 Modules

One module needs to fetch the keywords and tokens that are defined in the language. The other module receives the tokens and keywords and checks for the grammar required by the language. The ‘lexer’ does the job of fetching the keywords and token from the source program and the ‘parser’ does the job of handling them to match the grammar as specified by the language. If the grammar is violated, then the source program is not in our language specification and an error message is outputted. If an error is found in the source file, then the line number of error occurrence from source file is outputted and the grammar checking continues with the entire source file.

10.1.3 Cohesion

The modules are interconnected by a common buffer storage area. The lexer identifies and loads the keywords and tokens from source file to a buffer which is also referred to by the parser.

10.2 Detailed Design

The current design consists of few tokens and keywords. The lexer identifies the following tokens and keywords from the source file.

- It can identify the keywords - test, otherwise, endt, loop, from, step, endl, include, macro, endm, endf, endp, //
- It can also identify the variables, numbers, operators, symbols and newline character.

The parser checks for the following grammars.

- Check for expression. An expression can consist of left and right parts. Left part can have only a single variable. Right part can have variables, numbers or both forming compound statements.
The grammar could be something as

- Implementing ‘statements’:

statement: assign

| relCond | loopCond
 | funcCall ;
 | statement assign
 | statement funcCall ;
 | statement relCond
 | relCond statement
 | statement loopCond
 | loopCond statement

- The file inclusion, macro and function can be implemented as:

include : include variable

macro : macro variable code endm

func : funcCall code endf

- The recursive call in the program is implemented using the ‘code’, ‘parts’ and ‘program’ as:

code : statement | relCond | loopCond

| code statement | code relCond | code loopCond

parts : include | macro

| include macro | parts include | parts macro

program : parts | program parts

| func | parts func | program func

- Finally the entire program is ended with the keyword ‘endp’ and the implementation is

full : program endp

If the above rule is satisfied then the source program contains no errors. Otherwise the errors that occur in between are to be outputted with linenumber.

10.3 Interface Design

- The tokens and keywords are scanned by the lexer and put into a buffer. The parser requests for a particular token at which the lexer starts scanning the tokens. The tokens put into the buffer by the lexer are picked up the parser for grammar checking.
- During the course of grammar checking, if an error is encountered the particular linenumber from the source program is outputted. The line numbers with errors are further stored into an array 'errorlines'.
- In order to display the error lines from the source file, the 'errorlines' are sent to the display function. The error lines are then recognized and displayed suitably.
- Parse tree is constructed by referring to the line numbers containing the errors and formatted to the .dot format.

11 Example Program

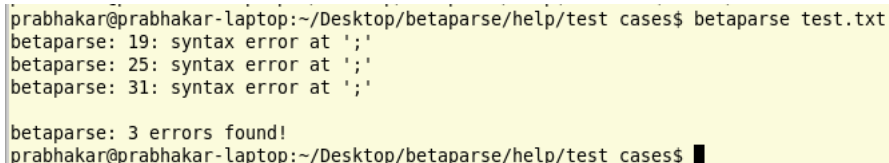
The following screenshot example shows the usage of the *betaparse*.

The test code is given in the figure 'Code'.

The executing of the code using betaparse is shown in 'Terminal'.

The parse tree that is generated is shown in 'Parse Tree'.

The error lines are differentiated from source program in 'HTML Error View'.



```
prabhakar@prabhakar-laptop:~/Desktop/betaparse/help/test cases$ betaparse test.txt
betaparse: 19: syntax error at ';'
betaparse: 25: syntax error at ';'
betaparse: 31: syntax error at ';'

betaparse: 3 errors found!
prabhakar@prabhakar-laptop:~/Desktop/betaparse/help/test cases$
```

Figure 3: Terminal

```

include stdio
macro hi    a = b; endm
main(a, b)
    a = b;
    test a > b
        s++;
        test a > b
            r+=r;
        endt
        otherwise
            loop a > b // xyz
                r = r;
                loop r < b
                    s--;
                endl
            endl
        endt
        test a == b
            r-;
        endt
    endf
a()
    loop a<b
        a+;
        b--;
    endl
endf
b()
    c++;
    x=;
endf
endp

```

Figure 4: Code

```
test a == b
    r-;
endt
endf

a()
loop a
    a+;
    b--;
endl
endf
b()
    c++;
    x=;
endf
endp
```

Figure 5: HTML Error View

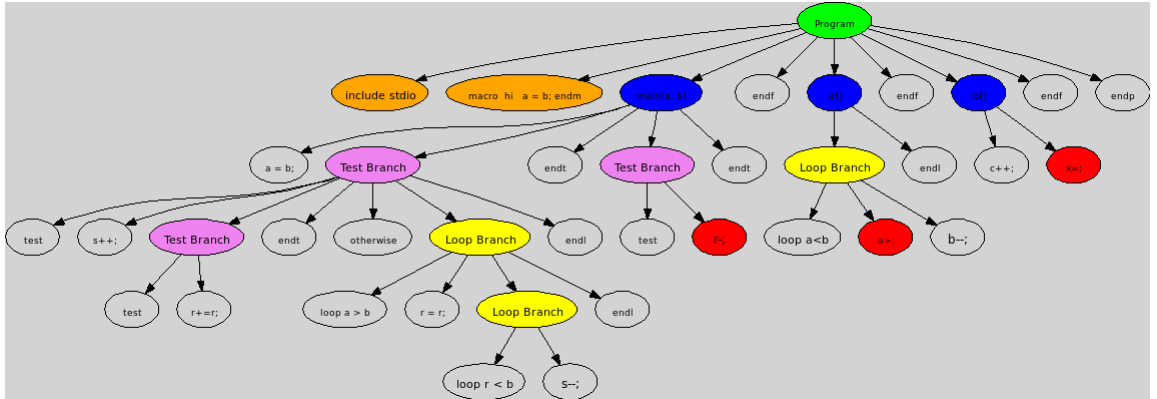


Figure 6: Parse Tree

12 Final Deliverable

- Archive of Product with Source code.
- Documentation.
- File containing suitable test cases.

13 Applications

- The designed β parse can be effectively used to verify the syntax of most of the possible programs written in β parse syntax.
- Can be used to simulate the *parse tree* creation.

14 Future Work

- The number of tokens and keywords currently supported are to be increased in future releases.
- Further evolve the current version of 'parse tree' implementing terminals for each tokens in the language.
- Implementing *auto make* and *auto configure* before installation.
- Develop a compiler that supports our language β parse and the needed optimizations.

15 Conclusion

Syntax analyzer is the major module used in any compiler. By repeated testing, we have found out that syntax analyser for β parse program statement successfully analyses and provides an easily understandable interface for the user to interact with system. The syntax that are to be followed in the language are simple with most them beign English statements and less number of brackets and datatype declarations. The language currently supports *any number* of file inclusions, macros, functions, loop and conditional structures and statements of various types. The syntax errors are successfully found. The expected results have been obtained and hence the implementation of the Syntax analyzer for β parse has been found successful.