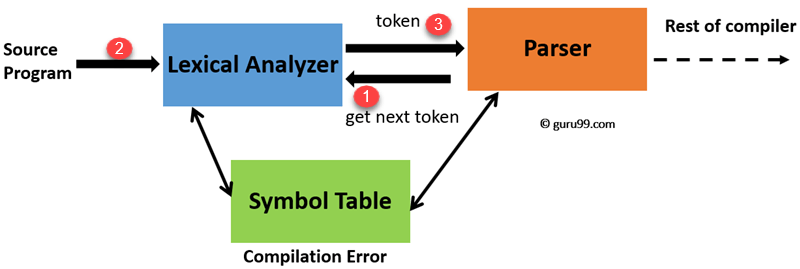
**Tiny Programming Language – Lex Analyzer and Parser**

The tiny program language is a newly developed language using Lex Analyzer and Parser. This language has its own unique tokens, identifiers, operators, keywords etc. This article gives insights into the language development and its coding implementation.

**Lexical Analysis**

The lexical analysis is the first part of the compiler. Whenever a language is developed, the compiler is developed to understand the language. Lexical analysis is used to design the grammar, tokens, identifiers, etc. of the language so that the compiler will be able to understand the language. The working of lex analyzer can be expressed as:



**Language Details**

The tiny language consists of complex grammar and tokens. Let’s consider a tiny language program:

{

This is a program to compute the factors of entered numbers.

It tests:

procedures

**repeat** loop

**if** statement

arithmetic

}

program factors:

var

i : integer;

**function** **Factor** ( i : integer ):integer;

var

j : integer;

begin

**if** i > 0 **then**

**for** (j := 1; j <= i; j:=j+1)

**if** i mod j = 0 **then** output ( j )

**end** Factor;

begin

**repeat**

read(i);

d:=Factor ( i )

**until** i <= 0

**end** factors.

The program is showing loop, function, statements, etc. and it’s clearly unique from other languages. The syntax of the language, it’s grammar and tokens are newly developed. The complete token list gives more understanding and is as under:

\_tokenRepresentationMap[Token::Program] = "program";

\_tokenRepresentationMap[Token::Variable] = "var";

\_tokenRepresentationMap[Token::Constant] = "const";

\_tokenRepresentationMap[Token::Type] = "type";

\_tokenRepresentationMap[Token::Function] = "function";

\_tokenRepresentationMap[Token::Return] = "return";

\_tokenRepresentationMap[Token::Begin] = "begin";

\_tokenRepresentationMap[Token::End] = "end";

\_tokenRepresentationMap[Token::Swap] = ":=:";

\_tokenRepresentationMap[Token::Assignment] = ":=";

\_tokenRepresentationMap[Token::Case] = "case";

\_tokenRepresentationMap[Token::Of] = "of";

\_tokenRepresentationMap[Token::Elipses] = "..";

\_tokenRepresentationMap[Token::Otherwise] = "otherwise";

\_tokenRepresentationMap[Token::Repeat] = "repeat";

\_tokenRepresentationMap[Token::Until] = "until";

\_tokenRepresentationMap[Token::Loop] = "loop";

\_tokenRepresentationMap[Token::Pool] = "pool";

\_tokenRepresentationMap[Token::Exit] = "exit";

\_tokenRepresentationMap[Token::LE] = "<=";

\_tokenRepresentationMap[Token::NE] = "<>";

\_tokenRepresentationMap[Token::LT] = "<";

\_tokenRepresentationMap[Token::GE] = ">=";

\_tokenRepresentationMap[Token::GT] = ">";

\_tokenRepresentationMap[Token::Eq] = "=";

\_tokenRepresentationMap[Token::Mod] = "mod";

\_tokenRepresentationMap[Token::And] = "and";

\_tokenRepresentationMap[Token::Or] = "or";

\_tokenRepresentationMap[Token::Not] = "not";

\_tokenRepresentationMap[Token::Read] = "read";

\_tokenRepresentationMap[Token::Successor] = "succ";

\_tokenRepresentationMap[Token::Predecessor] = "pred";

\_tokenRepresentationMap[Token::CharFun] = "chr";

\_tokenRepresentationMap[Token::OrdFun] = "ord";

\_tokenRepresentationMap[Token::Eof] = "eof";

**Code Implementation**

The source code is having Lexical Analyzer, Parser and Node Tree for parsing tree. Following is the output for the test program considered in previous section:

program(7)

. <identifier>(1)

. . factors(0)

. consts(0)

. types(0)

. dclns(1)

. . var(2)

. . . <identifier>(1)

. . . . i(0)

. . . <identifier>(1)

. . . . integer(0)

. subprogs(1)

. . fcn(8)

. . . <identifier>(1)

. . . . Factor(0)

. . . params(1)

. . . . var(2)

. . . . . <identifier>(1)

. . . . . . i(0)

. . . . . <identifier>(1)

. . . . . . integer(0)

. . . <identifier>(1)

. . . . integer(0)

. . . consts(0)

. . . types(0)

. . . dclns(1)

. . . . var(2)

. . . . . <identifier>(1)

. . . . . . j(0)

. . . . . <identifier>(1)

. . . . . . integer(0)

. . . block(1)

. . . . if(2)

. . . . . >(2)

. . . . . . <identifier>(1)

. . . . . . . i(0)

. . . . . . <integer>(1)

. . . . . . . 0(0)

. . . . . for(4)

. . . . . . assign(2)

. . . . . . . <identifier>(1)

. . . . . . . . j(0)

. . . . . . . <integer>(1)

. . . . . . . . 1(0)

. . . . . . <=(2)

. . . . . . . <identifier>(1)

. . . . . . . . j(0)

. . . . . . . <identifier>(1)

. . . . . . . . i(0)

. . . . . . assign(2)

. . . . . . . <identifier>(1)

. . . . . . . . j(0)

. . . . . . . +(2)

. . . . . . . . <identifier>(1)

. . . . . . . . . j(0)

. . . . . . . . <integer>(1)

. . . . . . . . . 1(0)

. . . . . . if(2)

. . . . . . . =(2)

. . . . . . . . mod(2)

. . . . . . . . . <identifier>(1)

. . . . . . . . . . i(0)

. . . . . . . . . <identifier>(1)

. . . . . . . . . . j(0)

. . . . . . . . <integer>(1)

. . . . . . . . . 0(0)

. . . . . . . output(1)

. . . . . . . . integer(1)

. . . . . . . . . <identifier>(1)

. . . . . . . . . . j(0)

. . . <identifier>(1)

. . . . Factor(0)

. block(1)

. . repeat(3)

. . . read(1)

. . . . <identifier>(1)

. . . . . i(0)

. . . assign(2)

. . . . <identifier>(1)

. . . . . d(0)

. . . . call(2)

. . . . . <identifier>(1)

. . . . . . Factor(0)

. . . . . <identifier>(1)

. . . . . . i(0)

. . . <=(2)

. . . . <identifier>(1)

. . . . . i(0)

. . . . <integer>(1)

. . . . . 0(0)

. <identifier>(1)

. . factors(0)

There are other examples present in test files along with their output trees and parsing steps.

**Conclusion**

The project has developed a parser for the new language with new syntax. The language is named as tiny because of its short tokens, identifiers, grammar etc. This language can be enhanced and developed to an advanced.