**Compiler Principle Experiment 2:**

**syntax analysis program design**

**The purpose:**

1. Reasoning the requirement of the syntax analysis.

2. Write parsing grammar analysis program .

3. Analyze the shortage of LL(1)/ recursive descent algorithm.

**The experiment content:**

1, Read the example given by teacher, which show the process of constructing a basic recursive grammar analysis program.

2, Choose a simple grammar that is suitable to construct the LL(1) parsing. The following three are recommended, P190 4.8, P190 4.9 , P190 4.10. but it is not compulsory. You can try any others.

3, Rewrite the LL (1) grammar if needed.

4, Write a recursive grammar analysis program (reference P148-149 Top-down parsing by recursive descent), implement the basic recursive drop analyzer, which can analyze as to whether the strings of symbols defined legal sentence for this method.

5, Design output form of syntax analysis program (output should be a syntax tree or deduction), a reference example, can be seen in figure 1.

6, debug the program and give the result analysis.

**Lab reports:**

1. List the grammar you will use.
2. Rewrite the grammar if necessary, List the result LL(1) grammar

3. Describe the algorithm you use and draw flow chart.

4. According to the grammar you choose, analysis if left recursion or left factor will influence the outcome of the algorithm.

5. List the problems and questions in the duration of experimental design (list the most disturbing problems in the experiment, at least 3 problems).

6. Compare the difference between homework assignments and the experimental results, analysis of recursive-descent and LL (1) the differences and similarities of the algorithm.

7. Listing the code of your program.

8. Give the experiment result (one legal input sentence and one illegal).

9. Self evaluation of this experiment, we give the experimental results of self-assessment system (100 points) Evaluation standard experimental principle about 30%, 40% experimental process, the analysis of the experiment report part (30%)

**The Grammars of Cool**

Before we dive into the implementation of a Cool Parser, we need first specify the grammars of Cool exactly( as the requirement of this experiment, we use **Backus–Naur form**).

According to the Cool Reference Manual, we have these terminals and non-terminals below:

**terminal CLASS, ELSE, FI, IF, IN, INHERITS, LET, LET\_STMT, LOOP, POOL, THEN, WHILE;**

**terminal CASE, ESAC, OF, DARROW, NEW, ISVOID;**

**terminal ASSIGN, NOT, LE, ERROR;**

**terminal PLUS, DIV, MINUS, MULT, EQ, LT, DOT, NEG, COMMA, SEMI, COLON;**

**terminal LPAREN, RPAREN, AT, LBRACE, RBRACE;**

**terminal AbstractSymbol STR\_CONST, INT\_CONST;**

**terminal Boolean BOOL\_CONST;**

**terminal AbstractSymbol TYPEID, OBJECTID;**

**nonterminal programc program;**

**nonterminal Classes class\_list;**

**nonterminal class\_c class;**

Notice that we specify the type of some terminals and non-terminals. And also, we can add any non-terminals if we wish.

**The BNF:**

**( In our text book, the syntax of a BNF is non-terminal → terminal, however, in the real word the syntax of a BNF is slightly different.**

**See** [**https://en.wikipedia.org/wiki/Backus%E2%80%93Naur\_form**](https://en.wikipedia.org/wiki/Backus–Naur_form) **in detail.)**

At the beginning, we have grammars like these:

**<program> ::= <class\_list>**

**<class\_list> ::= <class\_list> <class> ;**

**| <class> ;**

According the cool reference manual, the class definitions have the form:

class <type> [ inherits <type> ] {

<feature\_list>

};

The notation [ ...] denotes an optional construct.

Hence the grammar of class is:

**<class> ::= CLASS TYPEID <inheritance> LBRACE <feature\_list\_opt> RBRACE SEMI**

And we add a new non-terminal – inheritance:

**<inheritance> ::= INHERITS TYPE**

**| <empty>**

Then the feature\_list\_opt grammar:

**<feature\_list\_opt> ::= <feature\_list>**

**| <empty>**

**<feature\_list> ::= <feature\_list> <feature>**

**| <feature>**

In addition, we need add two non-terminals:

**nonterminal Features feature\_list;**

**nonterminal Features feature\_list\_opt;**

Now we already get in the most complex part. The body of a class definition consists of a list of feature definitions. A feature is **either an attribute or a method**.

An attribute of class A specifies a variable that is part of the state of objects of class A.

A method of class A is a procedure that may manipulate the variables and objects of class A.

Thus the feature grammar can be:

**<feature> ::= <attribute>**

**| <method>**

**<attribute> ::= OBJECTID COLON TYPEID assign\_opt SEMI**

**<assign\_opt> ::= ASSIGN <expression>**

**| <empty>**

**<method> ::= OBJECTID LPAREN <formal\_list\_opt> RPAREN COLON TYPEID LBRACE <expression> RBRACE**

Again we need add some new non-terminals:

nonterminal Feature feature;

nonterminal Attribute attribute;

nonterminal Assign assign\_opt;

nonterminal Expression expression;

nonterminal Method method;

nonterminal Formals formal\_list\_opt;

Then we dive in the method grammars:

**<formal\_list\_opt> ::= <formal\_list>**

**| <empty>**

**<formal\_list> ::= <formal\_list> ,** **<****formal>**

**| <formal>**

**<formal> ::= OBJECTID COLON TYPEID**

Then the expression grammars:

Problem:

Forget that in a class definition, its features can be empty( no variables and methods) !!

**Old one:**

dummy\_feature\_list → dummy\_feature\_list feature

| feature

**New one:**

dummy\_feature\_list\_opt→ dummy\_feature\_list

| Э

dummy\_feature\_list → dummy\_feature\_list feature

| feature

Also, the hardest part of the grammars of Cool is its class features, which can have lots of possibility.