**Compiler Principle Experiment 2:**

**syntax analysis program design**

**Introduction**

**The Grammars Definition 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.)**

Class

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 TYPEID**

**| <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;**

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 SEMI**

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**

Now we are getting into **the most complex part**. Expressions are the largest syntactic category in Cool.

**1 Constants**

The simplest expressions are constants. The **boolean** **constants** are true and false. **Integer** **constants** are unsigned strings of digits such as 0, 123, and 007. **String constants** are sequences of characters enclosed in double quotes, such as "This is a string." String constants may be at most 1024 characters long.

The constants belong to the basic classes Bool, Int, and String. The value of a constant is an object of the appropriate basic class.

The constants grammars:

**<constant> ::= STR\_CONST**

**| INT\_CONST**

**| BOOL\_CONST**

**<expression> ::= <constant>**

**| ...**

**2 Identifiers**

**<expression> ::= <constant>**

**| OBJECTID**

**| ...**

**3 Assignment**

An assignment has the form

<id> <- <expr>

The static type of the expression must conform to the declared type of the identifier. The value is the value of the expression. The static type of an assignment is the static type of <expr>.

**<expression> ::= <constant>**

**| OBJECTID**

**| OBJECTID ASSIGN <expression>**

**| ...**

**4 Dispatch**

There are **three forms** of dispatch (i.e. method call) in Cool. The three forms differ only in how the called method is selected. The most commonly used form of dispatch is

**<expr>.<id>(<expr>,...,<expr>)**

Consider the dispatch e0.f(e1 , . . . , en). To evaluate this expression, the arguments are evaluated in left-to-right order, from e1 to en. Next, e0 is evaluated and its class C noted (if e0 is void a runtime error is generated). Finally, the method f in class C is invoked, with the value of e0 bound to self in the body of f and the actual arguments bound to the formals as usual. The value of the expression is the value returned by the method invocation.

**The other forms of dispatch are:**

**<id>(<expr>,...,<expr>)**

**<expr>@<type>.id(<expr>,...,<expr>)**

The first form is shorthand for self.<id>(<expr>,...,<expr>).

The second form provides a way of accessing methods of parent classes that have been hidden by redefinitions in child classes. Instead of using the class of the leftmost expression to determine the method, the method of the class explicitly specified is used. For example, e@B.f() invokes the method f in class B on the object that is the value of e. For this form of dispatch, the static type to the left of “@”must conform to the type specified to the right of “@”.

Then, we get the grammars:

**<dispatch> ::= <expression> DOT OBJECTID LPAREN <expression\_list> RPAREN**

**| OBJECTID LPAREN <expression\_list> RPAREN**

**| <expression> AT TYPEID DOT OBJECTID LPAREN <expression\_list> RPAREN**

**<expression\_list> ::= <expression> COMMA <expression\_list>**

**| <expression>**

**<expression> ::= <constant>  
 | OBJECTID  
 | OBJECTID ASSIGN <expression>  
 | <dispatch>**

**| …**

**5 Conditionals**

A conditional has the form

**if <expr> then <expr> else <expr> fi**

The grammar:

**<expression> ::= <constant>  
 | OBJECTID  
 | OBJECTID ASSIGN <expression>  
 | <dispatch>**

**| IF <expression> THEN <expression> ELSE <expression> FI**

**| ...**

**6 Loops**

A loop has the form

**while <expr> loop <expr> pool**

The grammar:

**<expression> ::= <constant>  
 | OBJECTID  
 | OBJECTID ASSIGN <expression>  
 | <dispatch>**

**| IF <expression> THEN <expression> ELSE <expression> FI**

**| WHILE <expression> LOOP <expression> POOL**

**| …**

**7 Blocks**

A block has the form

**{ <expr>; ... <expr>; }**

The grammar:

**<expression> ::= <constant>  
 | OBJECTID  
 | OBJECTID ASSIGN <expression>  
 | <dispatch>**

**| IF <expression> THEN <expression> ELSE <expression> FI**

**| WHILE <expression> LOOP <expression> POOL**

**| LBRACE <expressions> RBRACE**

**| …**

**<expressions> ::= <expressions> <expression> SEMI**

**| <expression> SEMI**

**8 Let**

A let expression has the form

**let <id1> : <type1> [ <- <expr1> ], ..., <idn> : <typen> [ <- <exprn> ] in <expr>**

The grammar:

**<expression> ::= <constant>  
 | OBJECTID  
 | OBJECTID ASSIGN <expression>  
 | <dispatch>**

**| IF <expression> THEN <expression> ELSE <expression> FI**

**| WHILE <expression> LOOP <expression> POOL**

**| LBRACE <expressions> RBRACE**

**| <let\_expression>**

**| …**

**<let\_expression> ::= LET <nest\_lets> IN <expression>**

**<nest\_lets> ::= <formal> <assign\_opt> COMMA <nest\_lets>**

**| <formal> <assign\_opt>**

**9 Case**

A case expression has the form

**case <expr0> of**

**<id1> : <type1> => <expr1>;**

**. . .**

**<idn> : <typen> => <exprn>;**

**esac**

The grammar is:

**<expression> ::= <constant>  
 | OBJECTID  
 | OBJECTID ASSIGN <expression>  
 | <dispatch>**

**| IF <expression> THEN <expression> ELSE <expression> FI**

**| WHILE <expression> LOOP <expression> POOL**

**| LBRACE <expressions> RBRACE**

**| <let\_expression>**

**| <case\_expression>**

**<case\_expression> ::= CASE <expression> OF <cases> ESAC**

**<cases> ::= <case\_branch>**

**| <case\_branch> cases**

**<case\_branch> ::= <formal> DARROW <expression> SEMI**

**10 New, Isvoid**

A new expression has the form

**new <type>**

**11 Isvoid**

The expression

**isvoid expr**

evaluates to true if expr is void and evaluates to false if expr is not void.

**<expression> ::= <constant>  
 | OBJECTID  
 | OBJECTID ASSIGN <expression>  
 | <dispatch>**

**| IF <expression> THEN <expression> ELSE <expression> FI**

**| WHILE <expression> LOOP <expression> POOL**

**| LBRACE <expressions> RBRACE**

**| <let\_expression>**

**| <case\_expression>**

**| NEW TYPEID**

**| ISVOID <expression>**

**12 Arithmetic and Comparison Operations**

Cool has four binary arithmetic operations: +, -, \*, /. The syntax is

expr1 <op> expr2

Cool has three comparison operations: <, <=, =.

Finally, there is one arithmetic and one logical unary operator.

The expression ~<expr> is the integer complement of <expr>.

The expression not <expr> is the boolean complement of <expr>.

The grammars:

**<arith> ::= <expression> PLUS <expression>**

**| <expression> MINUS <expression>**

**| <expression> MULT <expression>**

**| <expression> DIV <expression>**

**| NEG <expression>**

**<comp>** ::= **<expression> LT <expression>**

**| <expression> LE <expression>**

**| <expression> EQ <expression>**

**| NOT <expression>**

Also, we can have a pair of parentheses outside a expression.

| …

| LPAREN <expression> RPAREN

Hence, we get the final grammars of Cool:

<program> ::= <class\_list>  
<class\_list> ::= <class\_list> <class>  
| <class>  
<class> ::= CLASS TYPEID <inheritance> LBRACE <feature\_list\_opt> RBRACE SEMI  
<inheritance> ::= INHERITS TYPEID  
| <empty>  
  
<feature\_list\_opt> ::= <feature\_list>  
| <empty>  
<feature\_list> ::= <feature\_list> <feature>  
| <feature>  
  
<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 SEMI  
<formal\_list\_opt> ::= <formal\_list>  
| <empty>  
<formal\_list> ::= <formal\_list> , <formal>  
| <formal>  
<formal> ::= OBJECTID COLON TYPEID  
  
  
<expression> ::= <constant>  
| OBJECTID <assign\_opt>  
| <dispatch>  
| IF <expression> THEN <expression> ELSE <expression> FI  
| WHILE <expression> LOOP <expression> POOL  
| LBRACE <expressions> RBRACE  
| <let\_expression>  
| <case\_expression>  
| NEW TYPEID  
| ISVOID <expression>  
| <arith>  
| <comp>  
| LPAREN <expression> RPAREN  
  
  
<constant> ::= STR\_CONST  
| INT\_CONST  
| BOOL\_CONST  
  
<dispatch> ::= <expression> DOT OBJECTID LPAREN <expression\_list> RPAREN  
| OBJECTID LPAREN <expression\_list> RPAREN  
| <expression> AT TYPEID DOT OBJECTID LPAREN <expression\_list> RPAREN  
  
<expression\_list> ::= <expression> COMMA <expression\_list>  
| <expression>  
  
<expressions> ::= <expressions> <expression> SEMI  
| <expression> SEMI  
  
<let\_expression> ::= LET <nest\_lets> IN <expression>  
<nest\_lets> ::= <formal> <assign\_opt> COMMA <nest\_lets>   
| <formal> <assign\_opt>  
  
<case\_expression> ::= CASE <expression> OF <cases> ESAC  
<cases> ::= <case\_branch>  
| <case\_branch> cases  
<case\_branch> ::= <formal> DARROW <expression> SEMI  
  
<arith> ::= <expression> PLUS <expression>  
| <expression> MINUS <expression>  
| <expression> MULT <expression>  
| <expression> DIV <expression>  
| NEG <expression>  
  
<comp> ::= <expression> LT <expression>  
| <expression> LE <expression>  
| <expression> EQ <expression>  
| NOT <expression>

**The grammars Implementation in CUP**

Problem:

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

**Old one:**

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

| feature

**New one:**

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

| <empty>

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

| feature