# 编译原理实验报告

## ——Parser of PCAT

# 1 实验目的

- 学习 LALR 文法与和 Bison 语法分析器生成器的用法；
- 熟悉 PCAT 语法；
- 使用 Bison 为 PCAT 编写语法分析器。

# 2 概述

语法分析器是 PCAT 的编译器中至关重要的一个组件。通常一个标准的编译器需要包含词法分析器、语法分析器、中间码生成这几个部分，其中语法分析器接受来自词法分析器的 Token 流，进行语法分析，构造语法分析树，并把抽象语法树传递给中间码生成步骤。

PCAT 的语法可以表示为用 LALR 文法描述，对于 LALR 文法，可以采用成熟的语法分析器生成器，输入 LALR 文法，并生成此文法对应的语法分析器代码。在本实验中采用 Bison 生成语分析器，输入 PCAT 语言的 BNF 文法描述文件，生成语法分析器，并把生成的语法分析器与上一个实验的词法分析器组合，完成了可以对源代码进行分析，输出抽象语法树的程序。

本报告将按照实验完成的顺序，介绍 Bison 的使用，PCAT 文法实现，词法分析器的整合，以及抽象语法树的生成等内容。

# 3 Bison 分析器生成器

Bison 是一个 GNU 系列项目中的语法分析器生成工具，它接受一个 LALR 文法（在最新版本中，已经提供了对限制更小的 GLR 文法的支持），并生成一份与之对应的语法分析器源码。Bison 接受的 LALR 文法以 BNF 范式描述，生成语法分析器代码。

## 3.1 文法描述文件

Bison 接受 BNF 范式描述的 LALR 文法，描述包含在文法描述文件之中。文法描述文件主要分为了三个部分，使用“%%”符号分割。三个部分如下所示。

```
/* 声明部分*/
%%
```

```
/* 规则部分 */
%%
/* 尾部 */
```

## 3.2 声明部分

声明部分主要负责引用头文件，设置 Bison 的参数，定义 Token 等工作。在文件的顶部可以引用头文件，目的是允许后续语义动作中调用其他源码文件中的函数，这一部分使用一对“%{”与“%}”符号来标识，其间写入 C 语言#include 语句。

此后设置 Bison 参数，定义 Token 和非终结符，这类设置通常以“%”开头。在 Bison 中，每一个 Token 或非终结符都对应一个数据类型，这就允许在进行语法分析的过程中，利用符号对应的信息来构造抽象语法树。在此实验中构造语法树的方法也是如此，每一个符号都是语法树的一个节点，在分析过程中利用语义动作来构造语法树。每一个终结符与非终结符的类型就是在声明部分定义的。

“%token”指令用于定义 Token 与它的种类，与词法分析器中输出的 Token 类型对应。“%type”指令用于定义非终结符的类型，通常可以定义为抽象语法树的节点。此外，“%left”、“%right”等指令定义运算符的结合顺序，用于消除冲突。关于声明部分具体参数的说明，可以从 Bison 参考文档中获取。

## 3.3 规则部分

描述文件最重要的部分是规则部分。规则部分是由 BNF 与语义动作构成的，每一条文法规则都可以用 BNF 描述，又可以为规则附加一条语义动作。Bison 接受无右递归的非二义性 LALR 文法。在表达式产生冲突时，可以使用算符优先级解决算符冲突。

语义动作是当采用一条规则归约后就会执行的一段代码，在这段代码中可以访问所归约语句的所有信息，包括获取终结符与非终结符的值，以及设置产生式左部非终结符的值。由于在本实验中，所有非终结符都保存抽象语法树节点类型的值，在语义动作中，根据不同的语句来构造不同的子树。

### 3.4 尾部

尾部在文件中是并不是必要的。有时会希望在语法分析器中写入函数等代码，则可以把它们写在这一部分。例如语义动作中调用的函数就可以在此处实现。实际上，也可以把用到的函数写在其他的 C 或 CPP 文件之中，在规则文件中仅仅引用其头文件，编译后链接到一个可执行文件。

在本实验中，语法规则的尾部就没有内容。这是因为在语义动作中调用的函数实现在“ast.c”中。

# 4 PCAT 文法实现

PCAT 是一个类似 Pascal 的面向过程的程序语言。附录 1 中包含了 PCAT 的原始文法描述。在这一部分，将根据文法描述写出 BNF 供 Bison 使用。

首先需要进行的修改是文法中的重复元素。重复元素是指在 PCAT 原始文法描述中出现的一种，允许某一个符号重复任意次数的形式。在 Bison 的 BNF 中并没有直接解析这类文法的规则，为了实现重复元素，需要改写文法。对于文法中的重复部分，如：

```
{ B }
```

可改写为：

```
B_array: %empty | B_array B
```

由此引入了一类新的非终结符，它们表示语法树中的一组节点，除此之外，其他的非终结符则表示一个普通的语法树节点。因此对于任何改写文法引入的这类非终结符，在本实验中的节点类型设为 ast_list 指针，而其他的正常非终结符，类型设为 ast 指针。

另一种常见的规则是用括号标识的可选元素。这类规则与重复元素是类似的，它允许文法中的某一部分元素出现或是不出现。处理方法与重复元素也类似，对于可选的部分，如：

```
( B )
```

可改写为：

```
B_optional: %empty | B
```

遵循以上规则，把全部的文法翻译为 BNF 后，文法的翻译工作就完成了绝大部分。此后还有两个问题需要处理，一是整合词法分析器，完成一个完整的语法分析器结构，这将在下一节中详细描述；另一个问题是，由于以上的转换，产生了某些语义冲突，需要解决这些冲突。

直接使用 Bison 编译规则文件后，会产生一个编译输出。在输出中经过查找可以定位到冲突主要集中在二元与一元表达式中。这是因为在文法中并没有定义表达式的优先级与结合方向。因此，参考 PCAT 语言手册，在语法规则文件的声明部分添加如下：

```
%left EQ NEQ GT LT GE LE
%left PLUS MINUS OR
%left STAR SLASH MOD DIV AND
%right NOT UOPT
```

其中，除了常规的二元运算符与“非”一元运算符以外，还添加了一条 UOPT 规则。UOPT 的优先级被列为最高，表示数字前的正号与符号。之所以不能直接定义加号和减号为一元运算符，是因为这两个符号已经被定义为低优先级的二元运算符了。为了避免冲突，定义一个高优先级的 UOPT 规则，并在正号、负号表达式中使用%prec 指令来提高表达式的优先级：

```
expression: PLUS expression %prec UOPT
          | MINUS expression %prec UOPT
```

完成对冲突规则的处理后，就得到了一个合法的 LALR 文法。

# 5 词法分析器整合

语法分析器需要接受来自词法分析器的 Token 流，在编译完整的语法分析器前需要整合词法分析器。词法分析器除了为语法分析器提供 Token 流，在每一个 Token 之中还应当附加一些必要的信息，例如对于字符串 Token 应附加字符串内容，对于实数 Token 应附加实数的值，而所有的 Token 都应该附加它们在源码中的起始与结束位置。为此需要对词法分析器规则文件进行修改。

词法分析器使用一个全局变量 yylval 来传递 Token 的值内容，此值是一个联合体，在 Bison 语法规则文件的声明部分定义。本实验中定义如下：

```
%union {
    char*      Tstring;
    int        Tint;
    double     Treal;
    ast*       Tast;
    ast_list*   Tast_list;
}
```

在以上的定义之下，可以使用 yylval.Tstring 以字符串指针的类型来设置 Token 值，同理也可以使用 yylval.Tint 以整数类型来设置 Token 值。在词法分析器的分析动作中，对字符串、整数、实数，以及标识符几种类型的 Token 添加代码，逐一对 yylval 赋值。

与 Token 关联的内容，除了值以外还有源码位置信息。在 Flex 规则文件的声明部分加入%location 参数，就可以开启词法分析器的位置跟踪。词法分析器中，位置跟踪也是通过一个全局变量 yylloc 传递给语法分析器的。

词法分析器中 yylloc 的值需要在分析的过程中动态修改，一种方法是在词法分析器的所有动作中加入一条代码来更新 yylloc 的值，但这样会导致词法分析器的文件冗余严重。一个更好的方法是利用 Flex 的“YY_USER_ACTION”宏，这个宏定义了一条在每一个 Token 产生后需要执行的动作，此时可以更新 yylloc。在 Flex 的规则文件中定义“YY_USER_ACTION”宏：

```
    #define YY_USER_ACTION yylex_updatelocation();
```

然后实现 yylex_updatelocation()函数。在此函数中调用 calc_column 函数，根据当前 Token 的起始位置与文本内容来计算 Token 的结束位置，全部代码如下：

```
void calc_column(int column, int* delta_line, int* end_column)
{
    int dline = 0;
    for(int i=0; i<yyleng; i++)
    {
        if(yytext[i] == '\n')
            column = 1, dline++;
        else
            column++;
    }

    if(end_column) *end_column = column;
    if(delta_line) *delta_line = dline;
}

void yylex_updatelocation()
{
    int delta_line;
    int end_column;

    calc_column(columnno, &delta_line, &end_column);

    yylloc.first_line = lineno;
    yylloc.last_line = lineno + delta_line;
    yylloc.first_column = columnno;
    yylloc.last_column = end_column;

    lineno = lineno + delta_line;
    columnno = end_column;
}
```

完成后，在 Bison 的语法规则文件中就可以引用 Token 的位置了。Bison 文件的语义动作中，每一个终结符或非终结符都可以通过“@”符号来引用起位置信息。对于终结符，位置

信息就是 Token 的位置，对于非终结符，默认会根据它所包含的第一个 Token 与最后一个 Token 来确定它的位置信息。

完成了词法分析器的整合后，就可以使用 Makefile 来编译出完整的词法分析器了。虽然至此还没有生成抽象语法树，但目前的分析器已经可以判断一份代码是否可以接受了。

# 6 抽象语法树的生成

最后的步骤是生成抽象语法树。抽象语法树的生成是通过语义动作来完成的。对于终结符，其语法动作中利用 yylval，调用“ast.h”中定义的语法树相关函数来创建节点，并赋予左端的非终结符。以整数为例，语义动作定义如下：

```
number: INTEGERT { $$ = mk_int(&(@$), yylval.Tint); }
```

其中 mk_int 函数接受两个参数，第一个参数是指向位置信息的指针，第二个参数是 Token 的整数值，返回一个 ast 指针类型的树节点，表示一个整数。语义动作中，“@$”表示归约结果的位置信息，传递给 mk_int 函数用于构造语法树；而“$$”表示归约结果的值，把构造出的语法树赋予归约结果，以便后续分析进一步使用。

除了 mk_int 函数，“ast.h”中还定义了一系列构造语法树的函数。这一类函数几乎都接受一个位置信息的指针，用于在构造语法树时保存位置数据。此外，不同的函数接受不同的其他参数。对于终结符，通常需要接受一个具体的 Token 值，例如前文提到的整数节点需要接受一个整数值，而对于字符串节点与标识符节点，则需要接受一个字符串值；对于非终结符，则需要接受一组 ast 指针类型的语法树节点，例如 mk_binexp 用于构造二元运算表达式，其定义如下：

```
ast* mk_binexp(void* loc, ast_kind tag, ast* val1, ast* val2)
```

此函数除了接受一个位置指针，还接受一个运算符类型 tag，以及两个子表达式 val1、val2。

所有的常规非终结符都是 ast 指针类型的抽象语法树节点。这类节点可以包含具体的值（如整数、实数、字符串等），或是一颗子树。结构体 ast 中定义了一个 ast_tag 枚举字段，用于唯一确定此节点的类型，通过这个类型就可以确定节点是否为子树，以及子树的具体结构。对于每一种语法树节点都创建一个枚举值。mk_int、mk_real 等函数创建一个单独的节点，而 mk_node 函数可以创建一个通用的子树。然而每一棵子树都通过 mk_node 来创建是十分烦琐的，因此在本实验中还定义了如下函数，用于简化构造过程：

- mk_noder：同 mk_node，但允许指定两个位置信息，用于构造覆盖一段范围的节点；
- mk_unaryexp：创建一元运算表达式节点；

- mk_binexp：创建二元运算表达式节点；
- mk_statblock：创建语句块节点；
- mk_idlist：创建标识符序列节点。

附录 2 中的 Bison 规则文件中已经包含了语义动作，这组语义动作可以在进行语法分析的过程中构造出语法树，并打印出来。

在拥有抽象语法树后，把树的内容打印出来，就可以十分方便的检验语法分析结果是否正确。在”ast.c“中实现了语法树的打印函数 print_ast，在文法的开始符号中，创建一个打印整棵语法树的语义动作，就可以在整个代码接受时打印出来语法树了。

打印语法树时为了美观，采用了部分缩进的策略。在打印的过程中，根据根据当前节点的位置信息来决定是否换行缩进：如果当前节点与上一节点不在同一行，则换行并添加缩进，反之则不换行缩进。这样可以取得很好的效果，既不会导致过度的换行，也不会线性的书写。输出的语法树与源码结构接近，具有很强的可读性。

# 7 总结

通过编写 Bison 的语法规则文件，并修改 Flex 规则文件和语法树相关的代码，最终在本实验中实现了一个完整的 PCAT 语法分析器。输入源码后，可以分析并生成抽象语法树。

# 附录 1：PCAT 原始文法

```
program         -> PROGRAM IS body ';'
body            -> {declaration} BEGIN {statement} END
declaration     -> VAR {var-decl}
                -> TYPE {type-decl}
                -> PROCEDURE {procedure-decl}
var-decl        -> ID { ',' ID } [ ':' typename ] ':=' expression ';'
type-decl       -> ID IS type ';'
procedure-decl  -> ID formal-params [':' typename] IS body ';'
typename        -> ID
type            -> ARRAY OF typename
                -> RECORD component {component} END
component       -> ID ':' typename ';'
formal-params   -> '(' fp-section {';' fp-section } ')'
                -> '(' ')'
fp-section      -> ID {',' ID} ':' typename
statement       -> lvalue ':=' expression ';'
                -> ID actual-params ';'
                -> READ '(' lvalue {',' lvalue} ')' ';'
                -> WRITE write-params ';'
                -> IF expression THEN {statement}
                   {ELSIF expression THEN {statement}}
                   [ELSE {statement}] END ';'
                -> WHILE expression DO {statement} END ';'
                -> LOOP {statement} END ';'
                -> FOR ID ':=' expression TO expression [ BY expression ]
                   DO {statement} END ';'
                -> EXIT ';'
                -> RETURN [expression] ';'
write-params    -> '(' write-expr {',' write-expr } ')'
                -> '(' ')'
write-expr      -> STRING
                -> expression
expression      -> number
                -> lvalue
                -> '(' expression ')'
                -> unary-op expression
                -> expression binary-op expression
                -> ID actual-params
                -> ID record-inits
                -> ID array-inits
lvalue          -> ID
                -> lvalue '[' expression ']'
                -> lvalue '.' ID
actual-params   -> '(' expression {',' expression} ')'
                -> '(' ')'
record-inits    -> '{' ID ':=' expression { ';' ID ':=' expression} '}'
array-inits     -> '[<' array-init { ',' array-init } '>]'
array-init      -> [ expression OF ] expression
number          -> INTEGER | REAL
unary-op        -> '+' | '-' | NOT
binary-op       -> '+' | '-' | '*' | '/' | DIV | MOD | OR | AND
                -> '>' | '<' | '=' | '>=' | '<=' | '<>'
```

# 附录 2：Bison 规则文件

```
%{
#include <stdio.h>
extern int linenno;
extern int columnno;

#include "ast.h"

#include "pcat.tab.h"
#include "pcat.yy.c"
void yyerror(const char* s);  // error handler defined in pcat.lex

%}

%locations

%union {
    char*       Tstring;
    int         Tint;
    double      Treal;
    ast*        Tast;
    ast_list*   Tast_list;
}

%token <Tast>   IDENTIFIER INTEGERT REALT STRINGT
                PROGRAM IS BEGINT END VAR TYPE PROCEDURE ARRAY RECORD
                IN OUT READ WRITE IF THEN ELSE ELSIF WHILE DO LOOP
                FOR EXIT RETURN TO BY AND OR NOT OF DIV MOD
                LPAREN  RPAREN LBRACKET RBRACKET LBRACE RBRACE COLON DOT
                SEMICOLON COMMA ASSIGN PLUS MINUS STAR SLASH BACKSLASH EQ
                NEQ LT LE GT GE LABRACKET RABRACKET ERROR

%token <int>    EOFF 0;


/* nonterminals */

%type <Tast>    program
                body
                declaration
                var_decl
                type_decl
                type_name opt_typename
                compound_type
                field_decl_array
                field_decl
                formal_params
                formal_section
                statement
                statement_if_opt_else
                statement_for_opt_by
                statement_return_opt_expr
                read_args
                write_args
                write_expr
                expression
                l_value
                arguments
                field_inits
                array_values
                array_value
                number
                id
                string

%type <Tast_list>   declaration_array
                    var_decl_array
                    type_decl_array
                    procedure_decl_array
                    field_decl_array
                    formal_params_optext
                    statement_array
                    statement_if_opt_elseif
                    read_args_optext
                    write_args_optext
                    arguments_optext
                    field_inits_optext
                    array_values_optext
                    id_array_optext


/* operation precedence */

%left EQ NEQ GT LT GE LE
%left PLUS MINUS OR
%left STAR SLASH MOD DIV AND
%right NOT UOPT

%%

start: program { printf("Accepted!\n"); print_ast($1); printf("\n"); };

program: PROGRAM IS body SEMICOLON { $$ = $3 };

body: declaration_array BEGINT statement_array END    { $$ = mk_node(&(#5), body_node, clst(2, mk_node(&(#1), genr_node, reverse($1)), mk_statblock(&(#3), $3))); };

declaration: VAR var_decl var_decl_array                { $$ = mk_node(&(#2), var_decls, cons($2, reverse($3))); }
           | TYPE type_decl type_decl_array             { $$ = mk_node(&(#2),type_decls, cons($2, reverse($3))); }
           | PROCEDURE procedure_decl procedure_decl_array  { $$ = mk_node(&(#5),proc_decls, cons($2, reverse($3))); };
declaration_array: /* empty */                          { $$ = NULL; }
                 | declaration_array declaration        { $$ = cons($2, $1); };

var_decl: id id_array_optext opt_typename ASSIGN expression SEMICOLON     { $$ = mk_node(&(#5), var_decl, clst(3, mk_idlist(&(#1), &(#2), $1, $2), $3, $5)); };
var_decl_array: /* empty */                             { $$ = NULL; }
              | var_decl_array var_decl                 { $$ = cons($2, $1); };

type_decl: type_name IS compound_type SEMICOLON     { $$ = mk_node(&(#5), type_decl, clst(1, $3)); }
type_decl_array: /* empty */                        { $$ = NULL; }
               | type_decl_array type_decl          { $$ = cons($2, $1); };

type_name: id;
opt_typename: /* empty */     { $$ = NULL; }
            | COLON type_name { $$ = $2};

compound_type: ARRAY OF type_name                   { $$ = mk_node(&(#5), arr_type, clst(1, $3)); }
             | RECORD field_decl field_decl_array END  { $$ = mk_node(&(#5), rec_type, cons($2, reverse($3))); };

procedure_decl: id formal_params opt_typename IS body SEMICOLON     { $$ = mk_node(&(#5), proc_decl, clst(4, $1, $2, $3, $5)); };
procedure_decl_array: /* empty */                                   { $$ = NULL; }
                    | procedure_decl_array procedure_decl           { $$ = cons($2, $1); };

field_decl: id COLON type_name SEMICOLON    { $$ = mk_node(&(#5), field_decl, clst(2, $1, $3)); };
field_decl_array: /* empty */               { $$ = NULL; }
                | field_decl_array field_decl   { $$ = cons($2, $1); };

formal_params: LPAREN formal_section formal_params_optext RPAREN    { $$ = mk_node(&(#5), param_node, cons($2, reverse($3))); }
             | LPAREN RPAREN                                        { $$ = mk_node(&(#5), param_node, NULL); }
formal_params_optext: /* empty */                                   { $$ = NULL; }
                    | formal_params_optext SEMICOLON formal_section     { $$ = cons($3, $1); };

formal_section: id id_array_optext COLON type_name                  { $$ = mk_node(&(#5), param_sec, clst(2, mk_idlist(&(#1), &(#2), $1, $2), $4)); };

statement: l_value ASSIGN expression SEMICOLON                      { $$ = mk_node(&(#5), assgn_stat, clst(2, $1, $3)); }
         | id arguments SEMICOLON                                   { $$ = mk_node(&(#5), fcall_exp, clst(2, $1, $2)); }
         | READ read_args SEMICOLON                                 { $$ = mk_node(&(#5), read_stat, clst(1, $2)); }
         | WRITE write_args SEMICOLON                               { $$ = mk_node(&(#5), write_stat, clst(1, $2)); }
         | IF expression THEN
             statement_array
           statement_if_opt_elseif
           statement_if_opt_else
           END SEMICOLON
                                                                    {
                                                                        $$ = mk_node(&(#5), if_stat,
                                                                            clst
                                                                            (
                                                                                3, mk_node(&(#1), &(#4), if_node, clst(2, $2, $4)),
                                                                                mk_statblock(&(#5), $5),
                                                                                $6
                                                                            )
                                                                        );
                                                                    }
         | WHILE expression DO statement_array END SEMICOLON        { $$ = mk_node(&(#5), while_stat, clst(2, $2, mk_statblock(&(#4), $4))); }
         | LOOP statement_array END SEMICOLON                       { $$ = mk_node(&(#5),  loop_stat, clst(1, mk_statblock(&(#2), $2))); }
         | FOR l_value ASSIGN expression TO expression statement_for_opt_by DO
             statement_array
           END SEMICOLON
                                                                    {
                                                                        $$ = mk_node(&(#5), for_stat,
                                                                            clst
                                                                            (
                                                                                5, $2, $4, $6, $7,
                                                                                mk_statblock(&(#9), $9)
                                                                            )
                                                                        );
                                                                    }
         | EXIT SEMICOLON                                           { $$ = mk_node(&(#5), exit_stat, NULL); }
         | RETURN statement_return_opt_expr SEMICOLON               { $$ = mk_node(&(#5), rets_stat, clst(1, $2)); };

statement_array: /* empty */                    { $$ = NULL; }
               | statement_array statement      { $$ = cons($2, $1); };

statement_if_opt_elseif: /* empty */                                    { $$ = NULL; }
                       | statement_if_opt_elseif ELSIF expression THEN statement_array  { $$ = cons(mk_node(&(#5), &(#5), if_node, clst(2, $3, $5)), $1); };

statement_if_opt_else: /* empty */          { $$ = NULL; }
                     | ELSE statement_array { $$ = mk_node(&(#5), else_node, clst(1, $2)); };

statement_for_opt_by: /* empty*/     { $$ = NULL; }
                    | BY expression  { $$ = $2; };

statement_return_opt_expr: /* empty */ { $$ = NULL; }
                         | expression  { $$ = $1; };

read_args: LPAREN l_value read_args_optext RPAREN     { $$ = mk_node(&(#5), io_node, cons($2, reverse($3))); }
         | LPAREN RPAREN                              { $$ = mk_node(&(#5), io_node, NULL); };
read_args_optext: /* empty */                         { $$ = NULL; }
                | read_args_optext COMMA l_value      { $$ = cons($3, $1); };

write_args: LPAREN write_expr write_args_optext RPAREN  { $$ = mk_node(&(#5), io_node, cons($2, reverse($3))); }
          | LPAREN RPAREN                               { $$ = mk_node(&(#5), io_node, NULL); };
write_args_optext: /* empty */                          { $$ = NULL; }
                 | write_args_optext COMMA write_expr   { $$ = cons($3, $1); };

write_expr: string          { $$ = $1; }
          | expression      { $$ = $1; };

expression: number                          { $$ = $1; }
          | l_value                         { $$ = $1; }
          | LPAREN expression RPAREN        { $$ = $2; }
          // unary op expression
          | PLUS expression %prec UOPT      { $$ = mk_unaryexp(&(#5), pos_exp, $2); }
          | MINUS expression %prec UOPT     { $$ = mk_unaryexp(&(#5), neg_exp, $2); }
          | NOT expression %prec UOPT       { $$ = mk_unaryexp(&(#5), not_exp, $2); }
          // expression binary op expression
          | expression PLUS expression      { $$ = mk_binexp(&(#3),  plus_exp, $1, $3); }
          | expression MINUS expression     { $$ = mk_binexp(&(#3), minus_exp, $1, $3); }
          | expression STAR expression      { $$ = mk_binexp(&(#3), times_exp, $1, $3); }
          | expression SLASH expression     { $$ = mk_binexp(&(#3), fdiv_exp, $1, $3); }
          | expression DIV expression       { $$ = mk_binexp(&(#3),  div_exp, $1, $3); }
          | expression MOD expression       { $$ = mk_binexp(&(#3),  mod_exp, $1, $3); }
          | expression OR expression        { $$ = mk_binexp(&(#3),   or_exp, $1, $3); }
          | expression AND expression       { $$ = mk_binexp(&(#3),  and_exp, $1, $3); }
          | expression GT expression        { $$ = mk_binexp(&(#3),   gt_exp, $1, $3); }
          | expression LT expression        { $$ = mk_binexp(&(#3),   lt_exp, $1, $3); }
          | expression EQ expression        { $$ = mk_binexp(&(#3),   eq_exp, $1, $3); }
          | expression GE expression        { $$ = mk_binexp(&(#3),   ge_exp, $1, $3); }
          | expression LE expression        { $$ = mk_binexp(&(#3),   le_exp, $1, $3); }
          | expression NEQ expression       { $$ = mk_binexp(&(#3),  neq_exp, $1, $3); }
          | id arguments                    { $$ = mk_node(&(#5), fcall_exp, clst(2, $1, $2)); }
          | id field_inits                  { $$ = mk_node(&(#5), finit_exp, clst(2, $1, $2)); }  // T { .key1 := val1, ... }
          | id id_array_values              { $$ = mk_node(&(#5), ainit_exp, clst(2, $1, $2)); };  // T [K [$1 id] val1, ... >]

l_value: id
       | l_value LBRACKET expression RBRACKET   { $$ = mk_node(&(#5), idx_exp, clst(2, $1, $3)); }
       | l_value DOT id                         { $$ = mk_node(&(#5), mem_exp, clst(2, $1, $3)); };

arguments: LPAREN expression arguments_optext RPAREN  { $$ = mk_node(&(#5), args_node, cons($2, reverse($3))); }
         | LPAREN RPAREN                              { $$ = mk_node(&(#5), args_node, NULL); };
arguments_optext: /* empty */                         { $$ = NULL; }
                | arguments_optext COMMA expression   { $$ = cons($3, $1); };

field_inits: LBRACE id ASSIGN expression field_inits_optext RBRACE  { $$ = mk_node(&(#5), fasg_node, cons(mk_node(&(#2), &(#4), fasg_exp, clst(2, $2, $4)), reverse($5))); }
           | LBRACE RBRACE                                          { $$ = NULL; }
field_inits_optext: /* empty */                                     { $$ = NULL; }
                  | field_inits_optext COMMA id ASSIGN expression   { $$ = cons(mk_node(&(#3), &(#5), fasg_exp, clst(2, $3, $5)), $1); };

array_values: LABRACKET array_value array_values_optext RABRACKET { $$ = mk_node(&(#5), aval_node, cons($2, reverse($3))); };
array_values_optext: /* empty */                                    { $$ = NULL; }
                   | array_values_optext COMMA array_value          { $$ = cons($3, $1); };

array_value: expression                 { $$ = mk_node(&(#5), aval_exp, clst(1, $1)); }
           | expression OF expression   { $$ = mk_node(&(#5), aval_exp, clst(2, $1, $3)); };

number: INTEGERT    { $$ = mk_int(&(#5), yylval.Tint); }
      | REALT       { $$ = mk_real(&(#5), yylval.Treal); }
string: STRINGT     { $$ = mk_str(&(#5), yylval.Tstring); };

id: IDENTIFIER                  { $$ = mk_var(&(#5), yylval.Tstring); };
id_array_optext: /* empty */    { $$ = NULL; }
               | id_array_optext COMMA id   { $$ = cons($3, $1); };


%%
```