# L2c语法说明

### 1、程序定义

program ::= {{ decls }}

decls ::= type\_block //类型块

| const\_block //常量块

| user\_op\_decl //节点定义

程序的定义是一个声明定义列表，每个定义可以是类型定义、常量定义或用户节点定义。

### 2、类型定义

type\_block ::= **type** {{ type\_decl；}}

type\_decl ::= ID [[ = type\_def ]]

type\_def ::= type\_expr

| **enum** { ID {{ , ID }}}

type\_expr ::= **bool** //布尔类型

| **char** //字符类型

| **byte** //8位带符号整型

| **ubyte** //8位无符号整型

| **short** //16位带符号整型

| **ushort** //16位无符号整型

| **int** //32位带符号整型

| **uint** //32位无符号整型

| **long** //64位带符号整型

| **ulong** //64位无符号整型

| **float** //32位浮点

| **real** //64位浮点

| typevar //类型名

| { field\_decl {{, field\_decl}} } //结构体类型

| type\_expr ^ const\_expr //数组类型

field\_decl ::= ID : type\_expr

typevar ::= ID

类型定义需要注意类型名定义和数组定义。类型名支持引用其他已经定义好的类型。数组类型的数组长度支持常量表达式，常量表达式必须是可计算可确定的，并且计算结果为整型。下面是类型定义的示例。

type

struct1 = {label1 : int, label2 :bool};

struct\_type = {label1 : int, label2 : bool^3, label3 : struct1};

enum\_type = enum {a1, a2, a3};

### 3、常量定义

const\_block ::= **const** {{ const\_decl; }} //常量块定义

const\_decl ::= ID : type\_expr [[= const\_expr]] //常量声明

const\_expr ::= ID //常量ID

| atom //基本常量

| unary\_arith\_op const\_expr //常量的一元运算

| const\_expr bin\_arith\_op const\_expr //常量的二元算术运算

| const\_expr bin\_bool\_op const\_expr //常量的二元布尔运算

| const\_expr bin\_realtion\_op const\_expr //常量的二元比较运算

| [ const\_list ] //数组常量

| { const\_label\_expr {{ , const\_label\_expr }} } //结构体常量

const\_list ::= [[ const\_expr {{, const\_expr}} ]] //数组常量内部定义

const\_label\_expr ::= ID : const\_expr //结构体常量内部定义

常量表达式主要用在全局常量定义、类型定义中的数组长度定义、高阶运算中的迭代次数、fby中的常数定义和数组运算中。除全局常量定义以外，其他定义中都要求常量表达式必须是可计算可确定的，并且计算结果为整型。

下面是常量定义的示例：

const

arr1 : int ^4 = [1, 2, 3, 4]; //数组常量

stru:struct1 = {label1:len, label2:false}; //结构体常量

len:int = 5; //整数常量

enu:enum\_type = a1; //枚举常量

c1: int = 2+3; //int常量表达式

c2: uint = 2u+3u; //uint常量表达式

c3: short = 2s+3s; //short常量表达式

c4: ushort = 2us+3us; //ushort常量表达式

c5: real = 3.4+2.5; //real常量表达式

c6: float = 7.5f+6.4f; //float常量表达式

### 4、变量定义

var\_decls ::= var\_id {{, var\_id}} : type\_expr [[ when\_decl ]] //变量声明

[[ last\_decl ]]

var\_id ::= ID //变量ID定义

when\_decl ::= **when** clock\_expr //变量时钟声明

clock\_expr ::= ID //时钟表达式定义

| **not** ID

last\_decl ::= last = const\_expr

变量定义既支持对每个变量单独定义，也支持对类型和时钟相同的变量集中定义。

### 5、节点定义

user\_op\_decl ::= op\_kind ID //节点定义

params **returns** params //输入参数和输出参数

op\_body //节点体

op\_kind ::= **function** //节点类型分为函数和变量

| **node**

params ::= ( [[ var\_decls {{ ; var\_decls}} ]] ) //参数列表定义

opt\_body ::= ； //空节点

| [[ local\_block ]] //局部变量定义可以为空

**let** {{ equation ; }} **tel** [[; ]]

local\_block ::= **var** {{ var\_decls ;}} //局部变量定义

需要注意的是如果节点类型为函数，则节点内部不能包含时态运算。

### 6、外部函数定义

user\_op\_decl ::= **function imported** ID //外部函数定义

params **returns** params //输入参数和输出参数

### 7、等式定义

equation ::= lhs = expr //等式定义

｜state\_machine return

lhs ::= ( ) //左值定义

| lhs\_id {{, lhs\_id}}

lhs\_id ::= ID

return ::= returns returns\_var

returns\_var ::= {{ ID , }} (( ID | .. ))

这里的等式都是简单等式，即不允许时态运算、call运算、高阶运算等互相嵌套使用。

### 状态机

state\_machine ::= **automaton** [[ ID ]] {{ state\_decl }}+ //状态机声明

state\_decl ::= [[ **initial** ]] [[ **final** ]] **state** ID //状态声明

[[ **unless** {{ transition ; }}+ ]] //强迁移列表

data\_def //状态内语句

[[ **until** {{ transition ; }} //弱迁移列表

data\_def ::= equation ; //等式

| [[ local\_block ]] [[ let {{ equation ; }} tel ]] //子变量和等式列表

transition ::= **if** expr **resume** ID //resume切换的状态迁移语句

｜ **if** expr **restart** ID //restart切换的状态迁移语句

### 条件块

clocked\_block ::= activate [[ ID ]] (( if\_block | match\_block )) //条件块声明

if\_block ::= if expr then (( data\_def | if\_block )) //布尔类型条件块if分支

else (( data\_def | if\_block )) //布尔类型条件块else分支

match\_block ::= when expr match {{ | pattern : data\_def }}+ //枚举类型条件块定义

### 10、表达式定义

expr ::= simple\_expr //简单表达式

｜last 'ID //last运算符

| tempo\_expr //时态运算表达式

| bool\_expr //布尔运算表达式

| array\_expr //数组运算表达式

| struct\_expr //结构体运算表达式

| mixed\_constructor //mix运算表达式

| switch\_expr //条件分支表达式

| apply\_expr //prefix和高阶运算表达式。

注意上面的各种表达式中的子表达式只能是简单表达式。

list ::= [[ simple\_expr {{, simple\_expr}} ]] //表达式列表

**时态运算表达式**

tempo\_expr ::= **pre** simple\_expr //pre表达式

| simple\_expr -> simple\_expr //arrow表达式

| **fby** (simple\_expr; const\_expr; simple\_expr) //fby表达式

| simple\_expr **fby** simple\_expr //不带常数的fby表达式

| simple\_expr **when** clock\_expr //when表达式

| **merge** ID (simple\_expr) (simple\_expr) //merge表达式

**布尔运算**

bool\_expr ::= #（list） //bool运算表达式

**数组运算**

array\_expr ::= simple\_expr [ INTEGER .. INTEGER ] //数组分片运算

| ( simple\_expr . {{ index}}+ **default** simple\_expr ) //数组动态投影运算

| simple\_expr ^ const\_expr //数组初始化运算1

| [ list ] //数组初始化运算2

**结构体运算**

struct\_expr ::= {label\_expr {{, label\_expr}} } //结构体初始化运算

**mix运算**

mixed\_constructor ::= (ID **with** {{ label\_or\_index }}+ = simple\_expr) //mix运算

label\_expr ::= ID : simple\_expr

index ::= [ simple\_expr ]

label\_or\_index ::= . ID

| index

**条件分支运算**

swith\_expr ::= **if** simple\_expr **then** simple\_expr **else** simple\_expr //ifelse运算

| ( **case** simple\_expr **of** {{ case\_expr }}+ ) //case运算

case\_expr ::= | pattern : simple\_expr

pattern ::= ID

| CHAR

| [[-]] INTEGER

| true

| false

| \_

**高阶运算表达式**

apply\_expr ::= prefix\_operator (list)

| (iterator << prefix\_operator; const\_expr >>)(list)

| (**mapw**  << prefix\_operator; const\_expr >> **if** simple\_expr **default** (list))(list)

| (**mapwi** <<prefix\_operator; const\_expr>> **if** simple\_expr **default** (list))(list)

| (**foldw**  << prefix\_operator; const\_expr >> **if** simple\_expr)(list)

| (**foldwi** <<prefix\_operator; const\_expr>> **if** simple\_expr)(list)

prefix\_operator ::= ID

| prefix\_unary\_operator //一元prefix运算符

| prefix\_binary\_operator //二元prefix运算符

| (**make** ID) //make运算

| (**flatten** ID) //flatten运算

prefix\_unary\_operator ::= +$ | -$ | not$ | short$ | int$ | float$ | real$

prefix\_binary\_operator ::= $+$ | $-$ | $\*$ | $/$ | $mod$ | $div$

| $=$ | $<>$ | $<$ | $>$ | $<=$ | $>=$

| $and$ | $or$ | $xor$

interator ::= **map** | **fold** | **mapi** | **foldi**

| **mapfold**

### 11、简单表达式定义

simple\_expr ::= ID //变量或常量id

| atom //基本常量表达式

| simple\_expr [ const\_expr ] //数组取子元素运算

| simple\_expr . ID //结构体取子元素运算

| unary\_arith\_op simple\_expr //一元运算

| simple\_expr bin\_arith\_op simple\_expr //二元算术运算

| simple\_expr bin\_bool\_op simple\_expr //二元布尔运算

| simple\_expr bin\_realtion\_op simple\_expr //二元比较运算

| (type\_expr simple\_expr) //强制类型转换

unary\_arith\_op ::= - | + | **not** //一元运算符

bin\_arith\_op ::= + | - | \* | / | **mod** | **div** //二元算术运算符

bin\_relation\_op ::= =| <> | < | > | <= | >= //二元比较运算符

bin\_bool\_op ::= **and** | **or** | **xor** //二元布尔运算符

atom ::= **true** //bool常量true

| **false** //bool常量false

| CHAR //字符常量

| INTEGER //32位整型int常量

| UINT //无符号32位整型常量

| FLOAT //32位浮点常量

| REAL //64位浮点常量

| USHORT //无符号16位整型常量

| SHORT //带符号16位整型常量

DIGIT10 = [ 0-9 ] //数字集合

EXPONENT = [ eE] [+-]? DIGIT10+ //科学计数法的指数

FLOAT = DIGIT10+ . DIGIT10\*[EXPONENT] “f” //32位浮点常量，以“f”结尾。

REAL = DIGIT10+ . DIGIT10\*[EXPONENT] //64位浮点常量

INTEGER = 0 | ( [ 1-9 ] DIGIT10\* ) //32位整型int常量

UINT = INTEGER “u” //无符号32位整型常量，以“u”结尾。

USHORT = INTEGER “us” //无符号16位整型常量，以“us”结尾。

SHORT = INTEGER “s” //带符号16位整型常量，以“s”结尾。

LETTER = | [ a-z ] | [ A-Z ] //字母集合

CHAR = LETTER | [+-] //字符常量

ID = [LETTER | \_] [DIGIT10 | LETTER | \_]\* //id定义，以字母和下划线开始