

类Rust编译器实现



Rust

性能

Rust速度极快，性能比肩C语言，内存效率极高，无需运行时或垃圾收集器，可以为性能关键型服务提供支持。

可靠性

Rust丰富的类型系统和所有权模型保证了内存安全和线程安全，使编程人员能够在编译时消除许多类别的错误。

生态性

Rust拥有出色的文档、带有有用错误消息的友好编译器、集成的包管理器和构建工具、自动格式化程序等。

Rust是一门能够安全高效地编写系统级程序的语言，在内存安全方面具有显著优势。



知名问答平台StackOverflow的调查表明，自2015年以来，Rust一直是开发者最爱的编程语言。

nature

《Nature》杂志2020年尾的文章《Why Scientists are Turning to Rust》中也强调：科学家极为推崇Rust。

华为是五位Rust基金会创始白金会员成员之一，引领了Rust语言的发展。

华为内部使用

- Rust 广泛用于嵌入式系统开发、系统驱动、云计算、虚拟存储、网络传输协议、并发编程框架基础库等产品中。
- 华为实验室正基于 Rust 探索先进的代码解析、安全分析等工具。

Rust社区贡献

- 华为深度参与了在Rust社区中，贡献了包括C到Rust转换、内联汇编、交叉编译、Parking Lot并发库、SIMD 基础库、文档导航、代码多态化、热补丁、AOP等特性。

Rust中国推广

- 华为战略支持了在中国举办的Rust China Conf大会，并推行多项社区活动。
- 华也为中国的开发者提供Rust教程和Rust编码规范等。

大作业2：中间代码生成器

- 了解类Rust语义规则，根据源程序给出语义分析结果，实现中间代码生成（建议以四元式的形式作为中间代码）；
- 注意静态语义错误的诊断和处理；
- 在此基础上，考虑更为通行的高级语言的语义检查和中间代码生成所需要注意的内容，并给出解决方案。
- 分组，保持大作业1的分组
- 撰写设计和说明文档

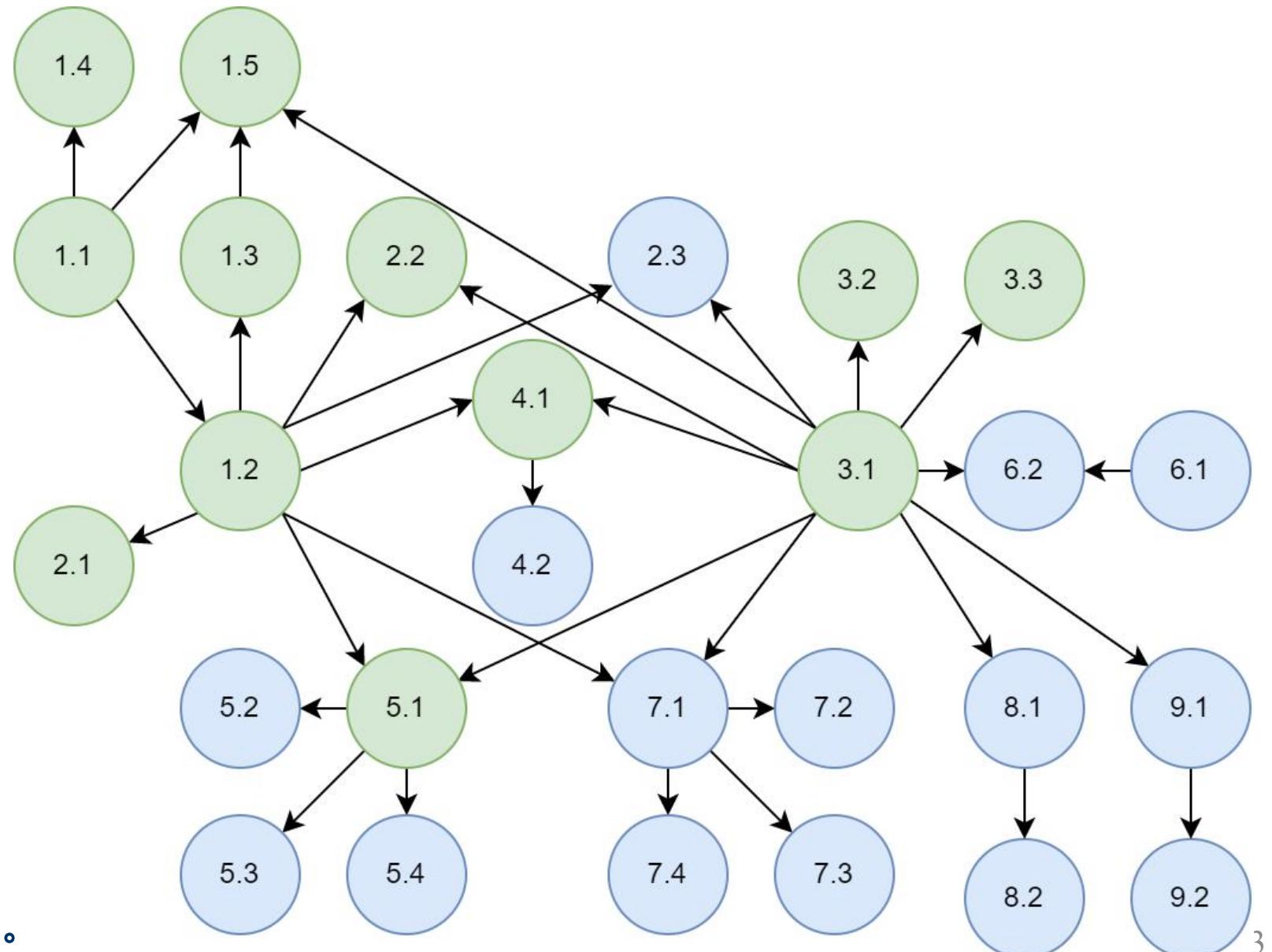
类Rust语法和语义规则

绿色节点为基础规则：

- 0.1、0.2、0.3、
- 1.1、1.2、1.3、1.4、1.5、
- 2.1、2.2、3.1、3.2、3.3、
- 4.1、5.1。

蓝色节点为拓展规则：

- 2.3、4.2、5.2、5.3、5.4、
- 6.1、6.2、7.1、7.2、7.3、7.4
- 8.1、8.2、9.1、9.2。



注：0.1、0.2、0.3未在图中画出。

类Rust语法和语义规则

➤ 1.1 基础程序

- Program -> <声明串>
- <声明串> -> 空 | <声明> <声明串>
- <声明> -> <函数声明>
- <函数声明> -> <函数头声明> <语句块>
- <函数头声明> -> fn <ID> '(' <形参列表> ')'
- <形参列表> -> 空
- <语句块> -> '{' <语句串> '}'
- <语句串> -> 空

语义说明：定义了一个最基础的程序结构，支持声明若干个函数，每个函数无参数且函数体为空。

```
fn program_1_10 {  
}  
}
```

➤ 0.1 变量声明内部

- <变量声明内部> -> mut <ID>

语义说明：声明变量为可变变量，如果变量为不可变变量，则不可以二次赋值。

➤ 0.2 类型

- <类型> -> i32

语义说明：支持基础类型i32，表示32位的整型。

➤ 0.3 可赋值元素

- <可赋值元素> -> <ID>

语义说明：标识符可以作为左值，被用于进行赋值操作。

类Rust语法和语义规则

➤ 1.2 语句 (前置规则1.1)

- <语句串> -> <语句> <语句串>
- <语句> -> ;

语义说明：允许函数体内放置任意数量的语句，单个分号可以作为一个语句。

➤ 1.3 返回语句 (前置规则1.2)

- <语句> -> <返回语句>
- <返回语句> -> return ;

语义说明：<返回语句> 用于从函数调用中退出，并将控制权交还给调用者。

当前仅支持不返回任何结果的返回语句。

➤ 1.4 函数输入 (前置规则0.1、0.2、1.1)

- <形参列表> -> <形参> | <形参> ; <形参列表>
- <形参> -> <变量声明内部> :: <类型>

语义说明：允许函数声明时接收参数，参数具有指定的类型信息。

➤ 1.5 函数输出 (前置规则0.2、1.3、3.1)

- <函数头声明> -> fn <ID> '(' <形参列表> ')' '->' <类型>
- <返回语句> -> return <表达式> ;

语义说明：<函数头声明> 中支持标注返回类型，并将 <返回语句> 增强为支持返回一个表达式的值。

```
fn program_1_2() {  
    .....  
}  
  
fn program_1_3() {  
    return ;  
}  
  
fn program_1_4(mut a:i32) {  
}  
  
fn program_1_5_1() -> i32 {  
    return 1;  
}  
  
fn program_1_5_2() -> i32 {  
    return ;  × 返回语句的类型 (空) 和函  
              数声明返回类型 (i32) 不一致  
}  
  
fn program_1_5_3() {  
    return 1;  × 返回语句的类型 (i32) 和函  
              数声明返回类型 (空) 不一致  
}
```

类Rust语法和语义规则

➤ 2.1 变量声明语句 (前置规则0.1、0.2、1.2)

- <语句> -> <变量声明语句>
- <变量声明语句> -> let <变量声明内部> '::' <类型> ';
- <变量声明语句> -> let <变量声明内部> ';

语义说明：允许两种变量声明语句：

- 显式声明一个具有指定类型的变量，必须与后续赋值或使用的表达式类型一致（**若不一致，则报错**）；
- 声明一个变量但不显式标注其类型，编译器需根据后续使用的上下文进行类型推导（**若无法推导，则报错**）。

变量允许二次声明，每次声明**隐藏之前的绑定**，称为**重影**。

```
fn program_2_1_10 {  
    let mut a:i32;  
}
```

```
fn program_2_1_20 {  
    let mut b; × 后续无语句,  
}                                无法推断b的类型
```

```
fn program_2_1_30 {  
    let mut a:i32; ✓ 可以二次声明,  
    let mut a;      称为重影  
    let mut a:i32;  
}
```

```
fn program_2_2_1(mut a:i32) {  
    a=32;  
}
```

```
fn program_2_2_20 {  
    a=32; × 变量未声明  
}
```

➤ 2.2 赋值语句 (前置规则0.3、1.2、3.1)

- <语句>-> <赋值语句>
- <赋值语句> -> <可赋值元素> '=' <表达式> ';

语义说明：允许将一个 <表达式> 的计算结果赋值给某个左值（L-value），从而实现变量值的修改。该左值需要提前声明，若未声明，则报错。

类Rust语法和语义规则

➤ 2.3 变量声明赋值语句 (前置规则0.1、0.2、0.3、1.2、3.1)

- <语句>-> <变量声明赋值语句>
- <变量声明赋值语句> -> let <变量声明内部> ':' <类型> '=' <表达式> ;
- <变量声明赋值语句> -> let <变量声明内部> '=' <表达式> ;

语义说明：允许两种变量声明并赋值语句：

- 显式声明一个变量，并指定其类型，立即使用右侧 <表达式> 的值对其进行初始化，右侧 <表达式> 必须能求值且与 <类型> 类型相同（**若不能求值、或类型不同，则报错**）；
- 声明一个变量但不显式标注其类型，类型由右侧 <表达式> 推导得出，且需与后续赋值或使用的表达式的类型一致（**若不一致，则报错**）。

和2.1一样，变量允许二次声明，每次声明隐藏之前的绑定，称为重影。

```
fn program_2_3_10 {  
    let mut a:i32=1;  
    let mut b=1;  
}
```

```
fn program_2_3_20 {  
    let mut b:i32=a;  × 右值求值时发  
}                                现变量a未声明
```

```
fn program_2_3_30 {  
    let mut a:i32;      × 右值求值时发  
    let mut b:i32=a;  现变量a未赋值  
}
```

```
fn program_2_3_40 {  
    let mut a:i32=1;  ✓ 可以二次声明,  
    let mut a=2;       称为重影  
    let mut a:i32=3;  
}
```

类Rust语法和语义规则

➤ 3.1 基本表达式 (前置规则0.3)

- <语句> -> <表达式> `;`
- <表达式> -> <加法表达式>
- <加法表达式> -> <项>
- <项> -> <因子>
- <因子> -> <元素>
- <元素> -> <NUM> | <可赋值元素> | `(` <表达式> `)`

```
fn program_3_1_10 {  
    0;  
    (1);  
    ((2));  
    (((3)));  
}
```

```
fn program_3_1_2(mut a:i32) {  
    a;  
    (a);  
    ((a));  
    (((a)));  
}
```

语义说明：提供**表达式求值的能力**，能支持常量、变量以及带括号的嵌套表达式。

➤ 3.2 表达式增加计算和比较 (前置规则3.1)

- <表达式> -> <表达式> <比较运算符> <加法表达式>
- <加法表达式> -> <加法表达式> <加减运算符> <项>
- <项> -> <项> <乘除运算符> <因子>
- <比较运算符> -> `<` | `<=` | `>` | `>=` | `==` | `!=`
- <加减运算符> -> `+` | `-`
- <乘除运算符> -> `*` | `/`

语义说明：支持**比较、加减、乘除三种运算**，且优先级**乘除>加减>比较**。

```
fn program_3_2() {  
    1*2/3;  
    4+5/6;  
    7<8;  
    9>10;  
    11==12;  
    13!=14;  
    1*2+3*4!=4/2-3/1;  
}
```

*注：对比于大作业1文档，3.1的语法进行了修正

类Rust语法和语义规则

➤ 3.3 函数调用 (前置规则3.1)

- <元素> -> <ID> '(' <实参列表> ')'
- <实参列表>-> 空 | <表达式> | <表达式> ',' <实参列表>

语义说明：提供函数调用表达式的能力，可以将函数作为运算的一部分参与整个程序逻辑。

- 实参数量与形参数量需一致；
- <实参列表> 中的每个 <表达式> 的类型必须与对应位置形参的类型一致；
- 若函数无返回值，则不能出现在需要右值的上下文中。

```
fn program_3_3_3_a() {  
}  
fn program_3_3_3_b() {  
    program_3_3_3_a(1);  
}  
    × 实参数量与形参数量不一致
```

```
fn program_3_3_4_a(mut a:i32) {  
}  
fn program_3_3_4_b() {  
    program_3_3_4_a(program_3_3_4_a);  
}  
    × 实参类型与形参类型不一致
```

```
fn program_3_3_1_a() {  
}  
fn program_3_3_1_b() {  
    program_3_3_1_a();  
}  
  
fn program_3_3_2_a(mut a:i32) {  
}  
fn program_3_3_2_b() {  
    program_3_3_2_a(1+2);  
}  
  
fn program_3_3_5_a() {  
}  
fn program_3_3_5_b() {  
    let mut a=program_3_3_5_a();  
}  
    × 无返回值函数不能作为右值
```

类Rust语法和语义规则

➤ 4.1 选择结构 (前置规则1.2、3.1)

- <语句> -> <if语句>
- <if语句> -> if <表达式> <语句块> <else部分>
- <else部分> -> 空 | else <语句块>

语义说明：支持 if 条件分支语句及其可选的 else 分支，可根据 <表达式> 的真假执行不同的代码路径：若为真，则执行 if 后的 <语句块>，否则若有 else 分支则执行 else 后的 <语句块>。

➤ 4.2 增加else if (前置规则4.1)

- <else部分> -> else if <表达式> <语句块> <else部分>

语义说明：拓展 else if 分支，若前一个 if 或 else if 条件不成立，则评估此分支的 <表达式>，若为真则执行该 else if 后的 <语句块>。

```
fn program_4_1_1(mut a:i32) -> i32
{
    if a>0 {
        return 1;
    }
}
fn program_4_1_2(mut a:i32) -> i32
{
    if a>0 {
        return 1;
    } else {
        return 0;
    }
}

fn program_4_2(mut a:i32) -> i32 {
    if a>0 {
        return a+1;
    } else if a<0 {
        return a-1;
    } else {
        return 0;
    }
}
```

类Rust语法和语义规则

➤ 5.1 while循环结构 (前置规则1.2、3.1)

- <语句> -> <循环语句>
- <循环语句> -> <while语句>
- <while语句> -> while <表达式> <语句块>

语义说明：拓展循环控制能力，实现基于条件的多次重复执行逻辑。每次循环开始前求表达式的值，若结果为真，则执行 <语句块>；若为假，则跳过整个循环。

```
fn program_5_1(mut n:i32) {  
    while n>0 {  
        n=n-1;  
    }  
}
```

➤ 5.2 for循环结构 (前置规则5.1)

- <循环语句> -> <for语句>
- <for语句>-> for <变量声明内部> in <可迭代结构> <语句块>
- <可迭代结构> -> <表达式> '..' <表达式>

语义说明：拓展基于范围的迭代能力，对可迭代结构中的每一个元素进行遍历，
<可迭代结构> -> <表达式> '..' <表达式> 的第一个<表达式>是闭区间，第二个
<表达式>是开区间，即包含第一个<表达式>的值，不包含第二个<表达式>的值。

```
fn program_5_2(mut n:i32) {  
    for mut i in 1..n+1 {  
        n=n-1;  
    }  
}
```

类Rust语法和语义规则

➤ 5.3 loop循环结构 (前置规则5.1)

- <循环语句> -> <loop语句>
- <loop语句>->loop <语句块>

语义说明：拓展无条件无限循环的能力，定义一个无限循环，每次迭代时执行 <语句块> 中的内容，只有通过显式退出指令（如 break）才能终止循环。

```
fn program_5_3() {  
    loop {  
        //  
    }  
}  
  
fn program_5_4_10 {  
    while 1==1 {  
        break;  
    }  
}  
  
fn program_5_4_20 {  
    break;      ✖ break; 必须出现  
在循环体内  
}  
  
fn program_5_4_30 {  
    while 1==0 {  
        continue;  
    }  
}  
  
fn program_5_4_40 {  
    continue; ✖ continue; 必须  
出现在循环体内  
}
```

➤ 5.4 增加break和continue (前置规则5.1)

- <语句> -> break ';' | continue ;'

语义说明：增强循环逻辑的灵活性和表达能力。

- break ';' 表示立即终止最内层当前所在的循环，控制流将跳转到该循环之后的第一条语句；必须出现在循环体内，否则应报错；若在嵌套循环中使用，仅跳出当前最内层循环。
- continue ';' 表示跳过当前循环体中剩余的代码，直接进入下一次循环判断；必须出现在循环体内，否则应报错；在嵌套循环中，仅影响当前所在循环。

类Rust语法和语义规则

➤ 6.1 声明不可变变量 (前置规则0.2)

- <变量声明内部> -> <ID>

语义说明：表示一个变量名 <ID> 被用作不可变变量的声明；该变量在赋值之后不能通过赋值语句再次修改其值；若变量是复合类型（如数组），其实体内容也不应被更改。

➤ 6.2 借用和引用 (前置规则3.1、6.1)

- <因子> -> '*' <因子> | '&' mut <因子> | '&' <因子>
- <类型>-> '&' mut <类型> | '&' <类型>

语义说明：支持引用语义与解引用访问能力。

- '*' <因子> 表示对一个引用类型的值进行解引用操作，访问其指向的数据；不允许对非引用类型进行解引用；
- '&' mut <因子> 表示创建一个指向某个变量的可变引用；允许通过该引用修改原始数据；当前不允许存在其他的不可变或可变引用；仅允许从可变变量创建可变引用；
- '&' <因子> 表示创建一个指向某个变量的不可变引用；允许读取但不允许修改原始数据；可以同时存在多个不可变引用；
- '&' mut <类型> 表示一个指向特定数据类型的可变引用类型；
- '&' <类型> 表示一个指向特定数据类型的不可变引用类型。

```
fn program_6_1_10 {  
    let a:i32=1;  
    let b=2;  
}  
  
fn program_6_1_20 {  
    let c:i32=1;     x 不可变变量不可  
    c=2;                 二次赋值  
}  
  
fn program_6_2_10 {  
    let mut a:i32=1;  
    let mut b:&mut i32=&mut a;  
    let mut c:i32=*b;  
}  
  
fn program_6_2_20 {  
    let a:i32=1;  
    let b:& i32=&a;  
    let c:i32=*b;  
}
```

类Rust语法和语义规则

➤ 6.1 声明不可变变量 (前置规则0.2)

- <变量声明内部> -> <ID>

语义说明：表示一个变量名 <ID> 被用作不可变变量的声明；该变量在赋值之后不能通过赋值语句再次修改其值；若变量是复合类型（如数组），其实体内容也不应被更改。

➤ 6.2 借用和引用 (前置规则3.1、6.1)

- <因子> -> '*' <因子> | '&' mut <因子> | '&' <因子>
- <类型>-> '&' mut <类型> | '&' <类型>

语义说明：支持引用语义与解引用访问能力。

- '*' <因子> 表示对一个引用类型的值进行解引用操作，访问其指向的数据；不允许对非引用类型进行解引用；
- '&' mut <因子> 表示创建一个指向某个变量的可变引用；允许通过该引用修改原始数据；当前不允许存在其他的不可变或可变引用；仅允许从可变变量创建可变引用；
- '&' <因子> 表示创建一个指向某个变量的不可变引用；允许读取但不允许修改原始数据；可以同时存在多个不可变引用；
- '&' mut <类型> 表示一个指向特定数据类型的可变引用类型；
- '&' <类型> 表示一个指向特定数据类型的不可变引用类型。

```
fn program_6_2_3() {  
    let mut a:i32=1;  
    let mut b=*a;      × 不允许对非引用  
}                                类型进行解引用
```

```
fn program_6_2_4() {  
    let mut a:i32=1;  
    let b=&a;          × 可变引用不能和  
    let mut c=&mut a;  其他的引用共存  
}                                
```

```
fn program_6_2_5() {  
    let a:i32=1;        × 仅支持从可变变  
    let mut b=&mut a;  量创建可变引用  
}                                
```

```
fn program_6_2_6() {  
    let mut a:i32=1;  
    let b=&a;          ✓ 可以存在多个不  
    let c=&a;          可变引用  
}                                
```

类Rust语法和语义规则

➤ 7.1 函数表达式块 (前置规则1.2、3.1)

- <表达式> -> <函数表达式语句块>
- <函数表达式语句块>->'{' <函数表达式语句串> '}'
- <函数表达式语句串>-> <表达式> | <语句> <函数表达式语句串>

语义说明：提供在表达式中嵌入代码块的能力，一个合法的 <表达式> 可以是一个包含一系列 <语句> 和一个末尾的 <表达式> 的代码块，末尾的 <表达式> 的值作为返回值。

➤ 7.2 函数表达式块作为函数体 (前置规则7.1)

- <函数声明> -> <函数头声明> <函数表达式语句块>

语义说明：提供使用结构化表达式块作为函数执行内容的能力，一个完整的函数可由 <函数头> 和 <函数表达式语句块> 组成，<函数表达式代码块> 的最后一个 <表达式> 的值作为返回值。

```
fn program_7_1(mut x:i32,mut y:i32) {  
    let mut z={  
        let mut t=x*x+x;  
        t=t+x*y;  
        t  
    };  
}
```

```
fn program_7_2(mut x:i32,mut y:i32) -> i32 {  
    let mut t=x*x+x;  
    t=t+x*y;  
    t  
}
```

类Rust语法和语义规则

➤ 7.3 选择表达式 (前置规则7.1)

- <表达式> -> <选择表达式>
- <选择表达式>-> if <表达式> <函数表达式语句块> else <函数表达式语句块>

语义说明：提供将 if-else 结构用作表达式并返回值的能力，一个 <表达式> 可以是 if-else 结构和 <函数表达式语句块> 的组合，根据 if 后的 <表达式> 的真假，决定执行哪一个 <函数表达式语句块>，该语句块的返回值作为该 <选择表达式> 的返回值。

➤ 7.4 循环表达式 (前置规则7.1)

- <表达式> -> <loop语句>
- <语句> -> break <表达式> `'

语义说明：提供将循环结构作表达式并返回值的能力，一个 <表达式> 可以是一个 <loop语句>，其最终返回值由某个分支中的 break <表达式> 决定。break 必须出现在循环体内，否则应报错；break 后的表达式必须能求值；若多个 break 存在于同一循环体中，它们必须返回相同类型。

```
fn program_7_3(mut a:i32) {  
    let mut b=if a>0 {  
        1  
    } else {  
        0  
    };  
}
```

```
fn program_7_4_1() {  
    let mut a=loop {  
        break 1;  
    };  
}
```

```
fn program_7_4_2() {  
    break 2;  
}
```

x break <表达式>;
必须出现在循环体内

类Rust语法和语义规则

➤ 8.1 数组 (前置规则0.2、3.1)

- <类型>-> '[' <类型> ';' <NUM> ']'
- <因子> -> '[' <数组元素列表> ']'
- <数组元素列表>-> 空 | <表达式> | <表达式> ';' <数组元素列表>

```
fn program_8_1_10 {  
    let mut a:[i32;3];  
    a=[1,2,3];  
}
```

语义说明：提供静态大小数组的声明与初始化能力。定义数组类型时，**<类型>** 是数组中每个元素的类型，**<NUM>** 是数组的长度，必须是一个正整数常量。创建并初始化一个数组时，**<表达式>** 的数量必须与数组的长度一致，每个**<表达式>** 的类型要与数组元素类型一致。

```
fn program_8_1_2(mut a:i32) {  
    let mut a:[i32;2];  
    a=1;  
}
```

× 变量赋值的类型与声明的类型不一致

```
fn program_8_1_3(mut a:i32) {  
    let mut a:[i32;2];  
    a=[1,2,3];  
}
```

× 初始化时的元素数量与数组长度不一致

```
fn program_8_1_4() {  
    let mut a:[[i32;1];1];  
    a=[1];  
}
```

× 初始化时元素的类型与数组元素类型不一致

本来写的=是错误的

类Rust语法和语义规则

➤ 8.2 数组元素 (前置规则8.1)

- <可赋值元素> -> <元素> '[' <表达式> ']'
- <可迭代结构> -> <元素>

```
fn program_8_2_1(mut a:[i32;3]) {  
    let mut b:i32=a[0];  
    a[0]=1;  
}
```

语义说明：提供随机访问数组元素的能力，支持在赋值语句左侧使用以修改指定位置的值。作为索引的<表达式>的类型必须是整数类型；索引值必须在运行时位于合法范围内 [0, len)，否则应视为越界；若数组本身不可变，则不能出现在赋值语句左侧。

```
fn program_8_2_2(mut a:i32) {  
    let mut a=[1,2,3];  × 数组的索引的类型  
    let mut b=a[0];    必须是整数类型  
}
```

```
fn program_8_2_3() {  
    let mut a=[1,2,3];  × 数组索引越界  
    let mut b=a[3];  
}
```

```
fn program_8_2_4() {  
    let a:[i32;3]=[1,2,3];  
    a[0]=4;  × 不可变数组的元素也不可变，不能作为左值  
}
```

类Rust语法和语义规则

➤ 9.1 元组 (前置规则0.2、3.1)

- <类型> -> '(' <元组类型内部> ')'
- <元组类型内部> -> 空 | <类型> ',' <类型列表>
- <类型列表> -> 空 | <类型> | <类型> ',' <类型列表>
- <因子> -> '(' <元组赋值内部> ')'
- <元组赋值内部> -> 空 | <表达式> ',' <元组元素列表>
- <元组元素列表> -> 空 | <表达式> | <表达式> ',' <元组元素列表>

```
fn program_9_1_1() {  
    let a:(i32,i32,i32);  
    a=(1,2,3);  
}
```

语义说明：提供静态大小元组的声明与初始化能力。定义数组类型时，支持空元组 () 和任意数量字段的元组。创建并初始化一个数组时，支持空元组和带元素的元组表达式，<表达式> 的数量必须与元组的长度一致，每个 <表达式> 的类型要与元组对应位置元素类型一致。

```
fn program_9_1_2(mut a:i32) {  
    let mut a:(i32,i32); × 变量赋值的类型与  
    a=1; 声明的类型不一致  
}  
}
```

```
fn program_9_1_3(mut a:i32) {  
    let mut a:(i32,i32); × 初始化时的元素数  
    a=(1,2,3); 量与数组长度不一致  
}  
}
```

```
fn program_9_1_4() {  
    let mut a:((i32,i32),); × 初始化时元素的类型  
    a=(1,); 与数组元素类型不一致  
}  
}
```

*注：仅有一个元素的元组的第一个元素后一定要加 ''，从而与加括号的表达式区分开

类Rust语法和语义规则

➤ 9.2 元组元素 (前置规则9.1)

- <可赋值元素>-><元素> ':' <NUM>

语义说明：提供按索引访问元组字段的能力，支持在赋值语句左侧使用以修改指定位置的值。作为索引的 <表达式> 的类型必须是整数类型；索引值必须在运行时位于合法范围内 [0, len)，否则应视为越界；若元组本身不可变，则不能出现在赋值语句左侧。

这个测试条件是语法错误

```
fn program_9_2_2(mut a:i32) {  
    let mut a=(1,2,3);  × 元组的索引的类型  
    let mut b=a.a;      必须是整数类型  
}
```

```
fn program_9_2_3() {  
    let mut a=(1,2,3);  × 元组索引越界  
    let mut b=a.3;  
}
```

```
fn program_9_2_4() {  
    let a:(i32,i32,i32)=(1,2,3);  
    a.0=4;  × 不可变元组的元素也  
}  不可变，不能作为左值
```

*注：对比于大作业1文档，9.2的语法进行了修正

评分标准

实验评价内容	所占比重	要求
问题分析能力	20%	说明语义分析和中间代码生成原理，绘制必要的状态转换图。
系统方案（算法）设计能力	20%	报告中体现系统各模块的总体设计和详细设计。
编程能力	20%	独立编程实现要求的全部功能，正确无误。
撰写报告能力	30%	表达通顺、结构清晰、内容完整、实验充分、提出个人想法，不存在抄袭。
查阅文献资料能力	10%	报告中列出所查阅的文献资料，含图书、论文、网络资源等。

报告要求

- 设计文档1份
- 程序源代码、可执行代码1份
- 程序实例与结果截屏
- 报告PPT1份