浙江大学

本科实验报告

课程名称：	编译原理
姓　　名：	李易非、王丹尧、陈俊儒
学　　院：	计算机科学与技术学院
系：	
专　　业：	计算机（统计交叉创新平台）
学　　号：	3160105705、3160104814、3160104341
指导教师：	冯雁

编译原理实验报告

序言

概述

pascal 语言由 Niklaus Wirth 教授于六十年代末设计并创立。 Pascal具有简洁的语法，丰富完备的数据类型，结构化的程序结构。在本项目中，我们基于 pascal 的子集语法实现了一个简单的 pascal 编译器，完成了词法分析、语法分析、语义分析以及最后的中间代码生成。在各个阶段中，我们均进行了一定程度的优化处理，缩减抽象语法树以及最终生成的中间代码；其次，我们实现了精确到行与语句的报错机制，并采取 resynchronization 错误处理方法，让编译器发现尽可能多的语法错误与语义错误；同时，编译器还支持输出可视化的抽象语法树图像与符号表的图像，便于各个阶段的开发。

提交文件描述

所有代码均在 `soup` 文件夹下，其中 `!README.md` 中介绍了编译器的使用方法和测试方法：

- `Lexer.py` 包含词法分析过程中的 token list 以及 token rules， `Parser.py` 中包含语法分析过程中的 grammar rules 定义， `Semantic.py` 中包含语义分析的数据结构与方法函数， `CodeGenerator.py` 中包含中间代码生成的函数方法， `AST.py` 中包含语法树的数据结构定义以及语义分析中所需的各类 utility function， `SymbolTable.py` 包含符号表的数据结构定义， `ErrorHandler.py` 包含项目报错信息的 logging 处理机制， `utils.py` 包含各种辅助数据结构；
- `soup.py` 为编译器运行文件， `test_script.py` 为批量测试脚步，具体使用方法见 `!README.md`
- `soup/test` 文件夹包含我们在报告以及 slides 中提到的所有测试文件，通过 `test_script.py` 可以进行批量测试，观察输出结果；
- `soup/visualization` 文件夹包含一个 demo visualization 输出，其中 *original_ast.png* 为 Parser 生成的语法树， `final_ast.png` 为经过语义分析缩减后的语法树， `symb_tab.png` 包含符号表的可视化表示；

分工

李易非：词法分析，语法分析，语义分析，测试

陈俊儒：代码生成，语法分析，测试

王丹尧：词法分析，语法分析，测试

1、词法分析

1.1 关键字

根据我们定义的pascal语法子集，对所需关键字进行了如下分类并将它们保存在reserved中。

```
sys_con = ("false", "maxint", "true")
sys_funct = ("abs", "chr", "odd", "ord", "pred", "sqr", "sqrt", "succ")
sys_proc = ("write", "writeln")
sys_type = ("boolean", "char", "integer", "real")
key_word = (
    "and", "array", "begin", "case", "const", "do", "downto", "else",
"end", "for", "function", "goto", "if", "in",
    "label", "mod", "not", "of", "or", "packed", "procedure", "program",
"read", "record", "repeat", "set", "then",
    "to", "type", "until", "var", "while", "with", 'div')
```

1.2 符号标记

根据ply的语法规则，将语法中所用到的符号定义对应的标记符号TOKENS。标记TOKENS定义在最前面，以列表的形式存储。每种TOKEN用一个正则表达式规则来表示，每个规则需要以"t_"开头声明，表示该声明是对标记的规则定义。对于简单的标记，可以直接定义：

```
t_STRING = r'\".*\"'
t_ASSIGN = r':='
t_EQUAL = r'='
t_UNEQUAL = r'<>'
t_ADD = r'\+'
t_SUBTRACT = r'-'
t_MUL = r'\*'
t_DIV = r'/'
t_LB = r'\['
t_RB = r'\]'
t_LP = r'\('
t_RP = r'\)'
t_GE = r'>='
t_LE = r'<='
t_GT = r'>'
t_LT = r'<'
t_COMMA = r','
t_COLON = r':'
t_SEMICON = r';'
t_DOT = r'\.'
t_DOUBLEDOT = r'\.\.'
t_ignore = ' \t\r'
```

对于需要执行动作的符号标记，如整数、实数、字符串和ID等TOKENS，将规则写成一个方法。方法总是需要接受一个LexToken实例的参数，该实例有一个t.type的属性（字符串表示）来表示标记的类型名称，t.value是标记值（匹配的实际的字符串）。方法可以在方法体里面修改这些属性。但是，如果这样做，应该返回结果token，否则，标记将被丢弃。在这里我们使用了@TOKEN装饰器来引用已有的变量并定义规则。

定义如下：

```
char = r'(\'([^\\\'\.]?)\')|(\"([^\\\"\.]?)\")'
identifier = r'[_a-zA-Z][_a-zA-Z0-9]*'
interger = r'\d+'
real = r'\d+\.\d+'
newline = r'\n+'
comment = r'{[\S\s\n]*?}'

@TOKEN(identifier)
def t_ID(t):
    # check for the reserved word
    t.type = reserved.get(t.value, 'ID')
    return t

@TOKEN(char)
def t_CHAR(t):
    t.value = t.value[1:-1]
    return t

@TOKEN(real)
def t_REAL(t):
    t.value = float(t.value)
    return t

@TOKEN(interger)
def t_INTEGER(t):
    t.value = int(t.value)
    return t

@TOKEN(newline)
def t_newline(t):
    t.lexer.lineno += len(t.value)

@TOKEN(comment)
def t_comment(t):
    # escape the comment
    pass

def t_error(t):
    print("Illegal char: `%s` at line %d" % (t.value[0], t.lexer.lineno))
    exit(-1)

# EOF handling rule
def t_eof(t):
    pass
```

1.3 匹配规则顺序

在lex内部，lex.py用re模块处理匹配模式，匹配顺序如下：

1. 所有由方法定义的标记规则，按照他们的出现顺序依次加入

2. 由字符串变量定义的标记规则按照其正则表达式长度倒序后，依次加入

2、语法分析

2.1 语法规则

我们对pascal定义了一个语法子集，下面是主要的文法规则：

2.1.1 程序整体框架

```
program : program_head routine DOT
program_head : PROGRAM ID SEMI
routine : routine_head routine_body
routine_head : const_part type_part var_part routine_part
routine_body : compound_stmt
compound_stmt : BEGIN stmt_list END
```

我们用下面这个例子来解释上面的文法:

program由program_head和routine构成，其中Program_head包括程序第一行的program关键字和ID信息(例子中是“if_statement”)。routine则又包括routine_head和routine_body。其中routine_head可能包有const_part, type_part, var_part和routine_part这四个部分中的一个或多个，之后会对这几个部分进行进一步的文法规则阐释（例子中包含了var_part）。routine_body则是begin开始end结束的程序过程。

```
program if_statement;
var x, y: integer;
begin
    x := 2;
    y := 15;
end.
```

2.1.2 常量 const_part

```
const_part : CONST const_expr_list | ε
const_expr_list : const_expr_list NAME EQUAL const_value SEMI
| NAME EQUAL const_value SEMI
const_value : INTEGER | REAL | CHAR | SYS_CON

```

常量部分可能为空，非空时由const关键字开头，一串常量定义由分号连接，支持整数、实数、字符等类型，实例如下：

```
const
    maxn=10000;
```

2.1.3 类型 type_part

```
type_part : TYPE type_decl_list | ε
type_decl_list : type_decl_list type_definition | type_definition
type_definition : NAME EQUAL type_decl SEMI
type_decl : simple_type_decl | array_type_decl | record_type_decl
simple_type_decl : SYS_TYPE | ID | LP name_list RP
                 | const_value DOTDOT const_value
array_type_decl : ARRAY LB simple_type_decl RB OF type_decl
record_type_decl : RECORD field_decl_list END
field_decl_list : field_decl_list field_decl | field_decl
field_decl : name_list COLON type_decl SEMI
name_list : name_list COMMA ID | ID
```

type部分可以为空，非空时由关键字type开头，后面是一串type定义。type定义的格式为 “变量名 = 类型声明”。类型声明又分为简单类型，array和record。其中简单类型包括系统已有类型，enum类型，range和自定义的ID。下面是一个array_type的实例：

```
type
    real_arr=array[2..10] of real;

```

2.1.4 变量 var_part

```
var_part : VAR var_decl_list | ε
var_decl_list : var_decl_list var_decl | var_decl
var_decl : name_list COLON type_decl SEMI

```

变量部分可以为空，非空时由var关键字开头，后面是一串变量定义，格式为" 变量名 : 类型名 " ，实例如下：

```
var
    i,j,n,p,q,ans,pos,sum           :integer;
    x, y: array[1..100] of integer;

```

2.1.5 函数和过程 routine_part

```
routine_part: routine_part function_decl | routine_part procedure_decl
            | function_decl | procedure_decl | ε
function_decl : function_head SEMI sub_routine SEMI
function_head : FUNCTION NAME parameters COLON simple_type_decl
procedure_decl : procedure_head SEMI sub_routine SEMI
procedure_head : PROCEDURE NAME parameters
parameters : LP para_decl_list RP | ε
para_decl_list : para_decl_list SEMI para_type_list | para_type_list
para_type_list : var_para_list COLON simple_type_decl
| val_para_list COLON simple_type_decl
var_para_list : VAR name_list
val_para_list : name_list

```

routine_part可以为空，非空时主要包括函数和过程。下面的函数和过程格式可以对应上面的文法。

```
function <函数名> (<形式参数表>):<类型>;
<说明部分>
begin
    <语句>;
    ...
    <语句>;
end;

procedure <过程名> (<形式参数表>);
<说明部分>
begin
    <语句>;
    ...
    <语句>;
end;

形式参数表：[var]变量名list:类型;...;[var]变量名list:类型。
其中带var的变量名list为变量形参，没有var的为值形参。

```

2.1.6 语句

语句是一种执行一串操作但是没有返回值的语法元素。我们的语言中，语句包含这几类：条件语句，while语句，repeat语句，for语句，赋值语句，case语句，goto语句。

```
stmt_list : stmt_list stmt SEMI | ε
stmt : INTEGER COLON non_label_stmt | non_label_stmt
non_label_stmt : assign_stmt | proc_stmt | compound_stmt | if_stmt | repeat_stmt | while_stmt | for_stmt | case_stmt | goto_stmt

```

赋值语句

赋值语句的左边是一个标识符，右边是一个表达式，左边可以是普通的ID、array或record的成员。

```
assign_stmt : ID ASSIGN expression
            | ID LB expression RB ASSIGN expression
            | ID DOT ID ASSIGN expression

```

proc语句

proc语句包括系统的函数和自定义的过程调用。

```
proc_stmt : ID
          | ID LP args_list RP
          | SYS_PROC
          | SYS_PROC LP expression_list RP
          | READ LP factor RP

```

条件语句

```
if_stmt : IF expression THEN stmt else_clause
else_clause : ELSE stmt |

```

```
if ... then ...;
if ... then ... else ...;

```

repeat和while语句

```
repeat_stmt : REPEAT stmt_list UNTIL expression
while_stmt : WHILE expression DO stmt

```

```
while <布尔表达式> do <语句>;

repeat
    <语句1>;
    <语句2>;
    ...
until <布尔表达式>

```

for语句

```
for_stmt : FOR ID ASSIGN expression direction expression DO stmt
direction : TO | DOWNTO

```

```
for <控制变量> := <初值> to <终值> do <语句>;
for <控制变量> := <初值> to <终值> downto <语句>;

```

case语句

```
case_stmt : CASE expression OF case_expr_list END
case_expr_list : case_expr_list case_expr | case_expr
case_expr : const_value COLON stmt SEMI
          | ID COLON stmt SEMI

```

```
case <表达式> of
    <情况表达式>: 语句1;
    ...
    ...
    [else 语句;]
end;

```

2.1.7 表达式

表达式list由一系列的表达式构成

```
expression_list : expression_list COMMA expression | expression

```

比较表达式

表达式可以细化到两个表达式之间的比较关系

```
expression : expression GE expr | expression GT expr | expression
LE expr | expression LT expr | expression EQUAL expr
| expression UNEQUAL expr | expr

```

二元表达式

再进一步，表达式可以细化到加减乘除，取模等二元运算

```
expr : expr PLUS term | expr MINUS term | expr OR term | term
term : term MUL factor | term DIV factor | term MOD factor
 | term AND factor | factor

```

facor

factor是表达式的最小单位，包括常量、变量、record的成员、函数调用的返回值等。

```
factor : ID | ID LP args_list RP | SYS_FUNCT |
SYS_FUNCT LP args_list RP | const_value | LP expression RP
| NOT factor | MINUS factor | ID LB expression RB
| ID DOT ID

```

2.2 实现方法

默认情况下，yacc.py 依赖 lex.py 产生的标记，默认的分析方法是 LALR，在 yacc 中的第一条规则是起始语法规则（在我们的程序中是program规则）。一旦起始规则被分析器归约，而且再无其他输入，分析器终止，最后的值将返回（这个值将是起始规则的p[0]）。

2.2.1 语法树结点

为了构建语法树，我们创建了一个通用的树节点结构：

```
class Node(object):

    def __init__(self, t, *args):
        self._type = t
        self._children = args

```

2.2.2 文法实现

每个语法规则被定义为一个python的方法，方法的文档字符串描述了相应的上下文无关文法，方法的语句实现了对应规则的语义行为。每个方法接受一个单独的 p 参数，p 是一个包含有当前匹配语法的符号的序列，p[i] 与语法符号一一对应。其中，p[i] 的值相当于词法分析模块中对 p.value 属性赋的值，对于非终结符的值，将在归约时由 p[0] 的赋值决定。如下，我们就建立了一个'program'的Node。

```
def p_program(p):
    'program : program_head routine DOT'
    p[0] = Node('program', p[1], p[2])

```

对于产生式右边只有一个所需参数的文法，我们也可以不建立Node，而是直接赋值，如下：

```
def p_program_head(p):
    'program_head : kPROGRAM ID SEMICON'
    p[0] = p[2]

```

如果所有的规则都有相似的结构，那么我们可以将语法规则合并（比如，产生式的项数相同）。不然，语义动作可能会变得复杂。简单情况下，可以使用`len()`方法区分，复杂情况下则可以根据语法的具体内容进行区分，比如：

```
def p_term(p):
    '''term :  term  MUL  factor
                    |  term  kDIV factor
                    |  term  DIV  factor
                    |  term  kMOD  factor
                    |  term  kAND  factor
                    |  factor'''
    if len(p) == 2:
        p[0] = p[1]
    elif p[2] == '*':
        p[0] = Node("term-MUL", p[1], p[3])
    elif p[2] == '/':
        p[0] = Node("term-DIV", p[1], p[3])
    elif p[2] == 'div':
        p[0] = Node("term-INTDIV", p[1], p[3])
    elif p[2] == 'mod':
        p[0] = Node("term-MOD", p[1], p[3])
    elif p[2] == 'and':
        p[0] = Node("term-AND", p[1], p[3])

```

2.2.3 二义性

如果在 yacc.py 中存在二义文法，会输出"移进归约冲突"或者"归约归约冲突"。在分析器无法确定是将下一个符号移进栈还是将当前栈中的符号归约时会产生移进归约冲突。为了解决二义文法，尤其是对表达式文法，yacc.py 允许为标记单独指定优先级和结合性。我们像下面这样增加一个 precedence 变量，这样的定义说明 ADD/SUBTRACT 标记具有相同的优先级和左结合性，MUL/DIV/kDIV/kMOD 具有相同的优先级和左结合性。在 precedence 声明中，标记的优先级从低到高。因此，这个声明表明 MUL/DIV/kDIV/kMOD（他们较晚加入 precedence）的优先级高于 ADD/SUBTRACT，这样就解决了算术运算中的二义性问题。

```
precedence = (
    ('left', 'ADD', 'SUBTRACT'),
    ('left', 'MUL', 'DIV', 'kDIV', 'kMOD')
)

```

2.3 错误处理

对于文法规则部分出现的错误，一般而言，最简单的处理方式就是在遇到错误的时候就抛出异常并终止。但我们希望它能报告错误并尽可能的恢复并继续分析。

2.3.1 panic mode

一种处理方式是panic mode，该模式下，开始放弃剩余的标记，直到能够达到一个合适的恢复机会。我们一开始实现的panic mode如下，它会简单的抛弃错误的标记，并告知分析器错误被接受了。

```
def p_error(p):
    if p:
        SemanticLogger.error(p.lineno,
                             "syntax error at token {}".format(p.value))
        # Just discard the token and tell the parser it's okay.
        parser.errok()
    else:
        print("Syntax error at EOF")
```

2.3.2 resynchronization

优化后采用的错误处理方式是根据 error 规则恢复和再同步，通过在语法规则中包含 error 标记来实现，例如：

```
def p_type_definition(p):
    '''type_definition :  ID  EQUAL  type_decl  SEMICON'''
    p[0] = Node("type_definition", p.lexer.lineno, p[1], p[3])

#type定义出错
def p_type_definition_error(p):
    'type_definition :  ID  EQUAL  error  SEMICON'
    SemanticLogger.error(p[3].lineno,
                         f"Syntax error at token `{p[3].value}` in type
definition.")
```

当type中有内容出错时，error 标记会匹配任意多个分号之前的标记（分号是`SEMI`指代的字符）。一旦找到分号，规则将被匹配，这样 error 标记就被归约了。我们针对不同的文法规则添加了类似的error标记，优化错误处理。

2.3.3 错误处理实例

通过下面这个例子可以看到我们的错误处理方式resynchronization的优势：

```
type
    arr = array [050] of integer;
    MailingListRecord = record
        FirstName: string;
        LastName: string;
        Address: string;
```

对于上面的内容，type中的arr定义出错，但后面的record定义完全正确。

如果是panic mode处理模式，出现的结果是从出错位置一直到最后的程序都会被标记错误，最终无法建立分析树。因为程序始终无法找到合理的同步点，导致丢弃了大量的输入：

```
(python36) F:\Yapc\yapc>yacc_pas.py
[line 4 ERROR] syntax error at token ]
[line 4 ERROR] syntax error at token of
[line 4 ERROR] syntax error at token integer
[line 4 ERROR] syntax error at token ;
[line 5 ERROR] syntax error at token MailingListRecord
[line 5 ERROR] syntax error at token =
[line 5 ERROR] syntax error at token record
[line 6 ERROR] syntax error at token FirstName
[line 6 ERROR] syntax error at token :
[line 6 ERROR] syntax error at token string
[line 6 ERROR] syntax error at token ;
[line 7 ERROR] syntax error at token LastName
[line 7 ERROR] syntax error at token :
[line 7 ERROR] syntax error at token string
[line 7 ERROR] syntax error at token ;
[line 8 ERROR] syntax error at token Address
[line 8 ERROR] syntax error at token :
[line 8 ERROR] syntax error at token string
[line 8 ERROR] syntax error at token ;
[line 9 ERROR] syntax error at token City
[line 9 ERROR] syntax error at token :
[line 9 ERROR] syntax error at token string
[line 9 ERROR] syntax error at token ;
[line 10 ERROR] syntax error at token State
[line 10 ERROR] syntax error at token :
[line 10 ERROR] syntax error at token string
```

然而，当采用resynchronization错误恢复模式时，只要从出错位置开始匹配到下一个分号，我们就能将error规约并继续分析输入，语法树也将成功建立，唯一的不同只是少了出错的arr：

```
(python36) F:\Yapc\yapc>yacc_pas.py
yacc: Syntax error at line 3, token=RB
[line 3 ERROR] Syntax error at token `]` in type definition.
```

3、语义分析

3.1 overview

在 static semantic analysis 阶段，遍历基于 syntax analysis 得到的语法树，编译器将 *构建 symbol table*、*检查类型声明*、*检查变量声明*、*检查函数/过程声明*、*检查各类 statement 语句的变量定义*、*constant folding 以及 类型检查 (type checking)*，经过语义分析之后，原始语法树将会进行一定程度的缩减，便于的后续代码生成流程。在接下来的小节中，**3.2** 将描述语义分析中构建的 symbol table 的数据结构与常用操作；**3.3** 将描述在语义分析过程中对语法树进行了两种优化改动；**3.4** 将具体描述在各类声明、语句中，语义分析所做的语义检查；在上述所有小节中，我们都配以相应的例子与输出结果以便理解。

3.2 symbol table

在整个语义分析过程中，编译器将动态构建 symbol table，在遍历树的整个过程中不断插入新的 symbol table 项，并利用 symbol table 进行 overview 中描述的各类检查。考虑到 pascal 语言支持嵌套的 procedure, function 定义，我们需要支持不同 scope 的 symbol table, 由于嵌套的 scope 有逻辑上的父子关系，不同 scope 的 symbol table 由树形结构进行表达，在进行诸如 *变量是否定义* 的检查过程中，可以通过当前 symbol table 结点不断上溯 parent 节点进行跨 scope 检查。

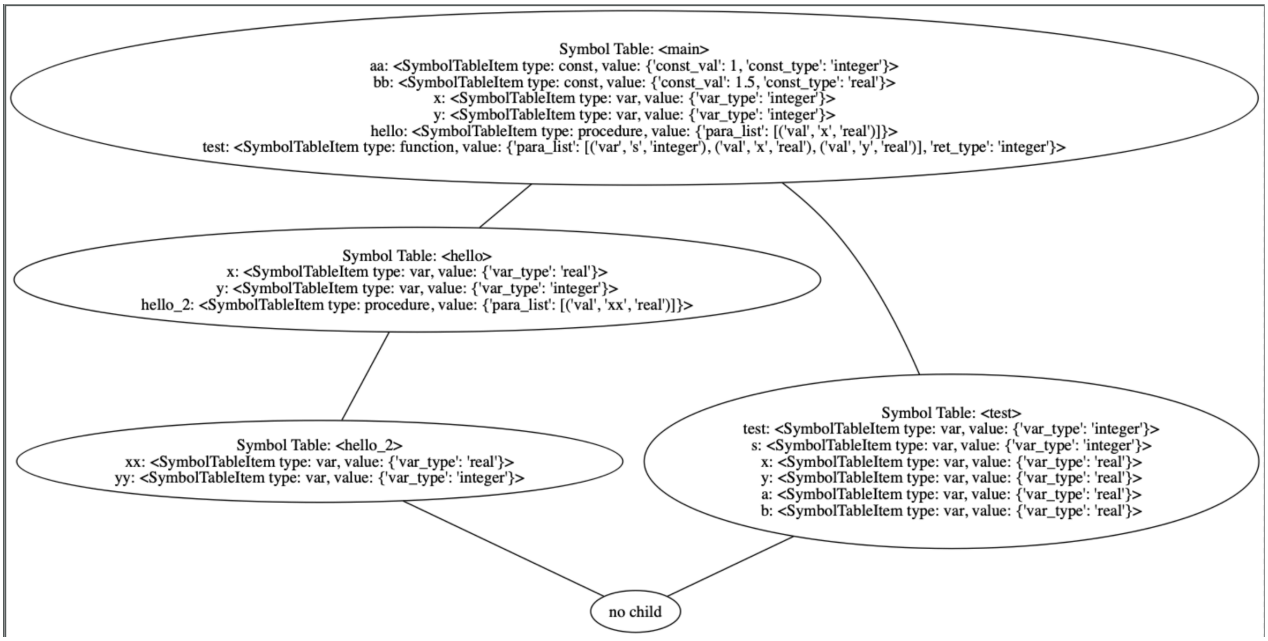


Figure 1: symbol table 树形结构示意图

3.2.1 数据结构与操作

symbol table 支持基础的 *insert*, *look up*, *delete* 操作，insert 操作将 key, value 对插入 symbol table 中，若 key 已经存在，则返回 None, 否则返回 value；lookup 操作在 symbol table 中搜索 key 所对应的 value，若 key 不存在返回 None；delete 操作删除 symbol table 中的 key, value 对，若 key 不存在，返回 None。

```
class SymbolTable(object):

    def insert(self, key, value):
        """ insert (key, value) into symbol table
        if the key is already in the symbol table, return None
        else return value
        """
    def lookup(self, key):
        """ lookup a key in the symbol table
        if the key is in the symbol table, return corresponding value
        else return None
        """

    def delete(self, key):
        """ delete a key in the symbol table
        if the key is in the symbol table, delete (key, value) and return
value
```

```
        else return None
        """


```

以上 symbol table 数据结构仅能处理单一 scope 的情形，为了支持上文中描述的树形结构 symbol table，symbol table node 继承 symbol table，并增加操作 *chain_lookup* ，chain_lookup 逐级向上搜索 key, 若 key 不存在于任何一个 scope 则返回 None.

```
class SymbolTableNode(SymbolTable):

    def chain_look_up(self, key):
        """ 逐层向上搜索 key
        if the key does not exist in every symbol table on the path, return None
        """
```

3.3 语义分析过程中对语法树的缩减与更改

为了方便后续代码生成，在语义分析过程中将对语法分析产生的语法树进行一定的缩减与更改，首先是 constant folding 操作，该操作会将可以推断结果的算式提前计算出结果，并将结果作为对应节点的子节点，在 **3.4** 我们将详细叙述这一操作；第二，由于在语法规则中有大量的左递归语法，将导致语法树的深度过高，在语义分析中 “压平” 语法树，增加其宽度，减小其深度；

3.3.1 constant folding

grammar rules 中 expression 语法包含了所有的关系运算与算术运算，语义分析将 *后序遍历* **expression** 节点，进行最大程度的 constant 推断（部分运算中包含 var 无法进行 const folding）以及 type 检查；

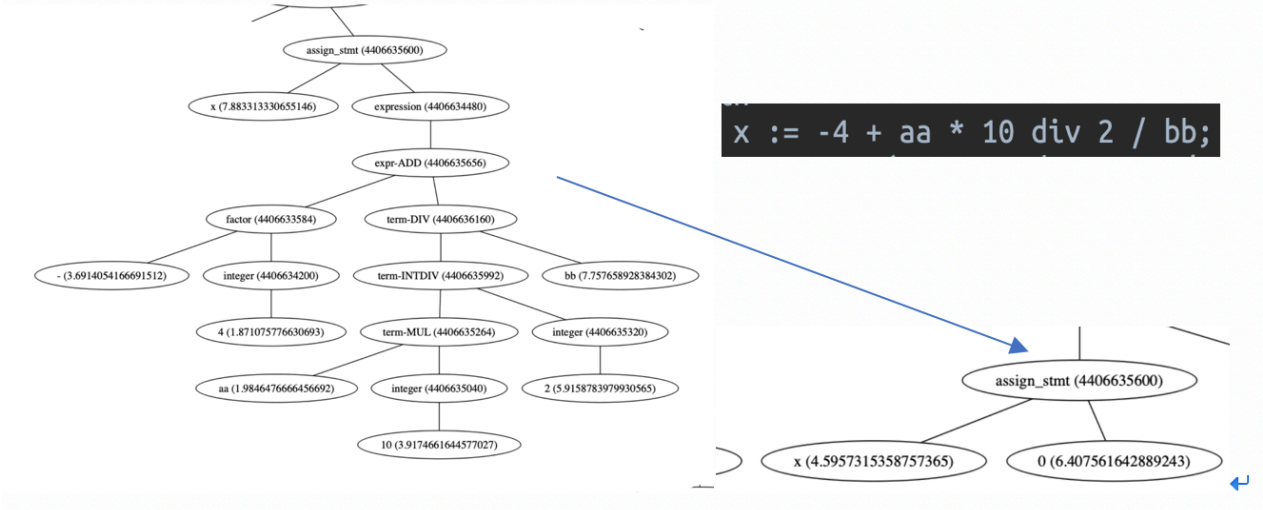


Figure 2: constant folding demo 展示 (1)

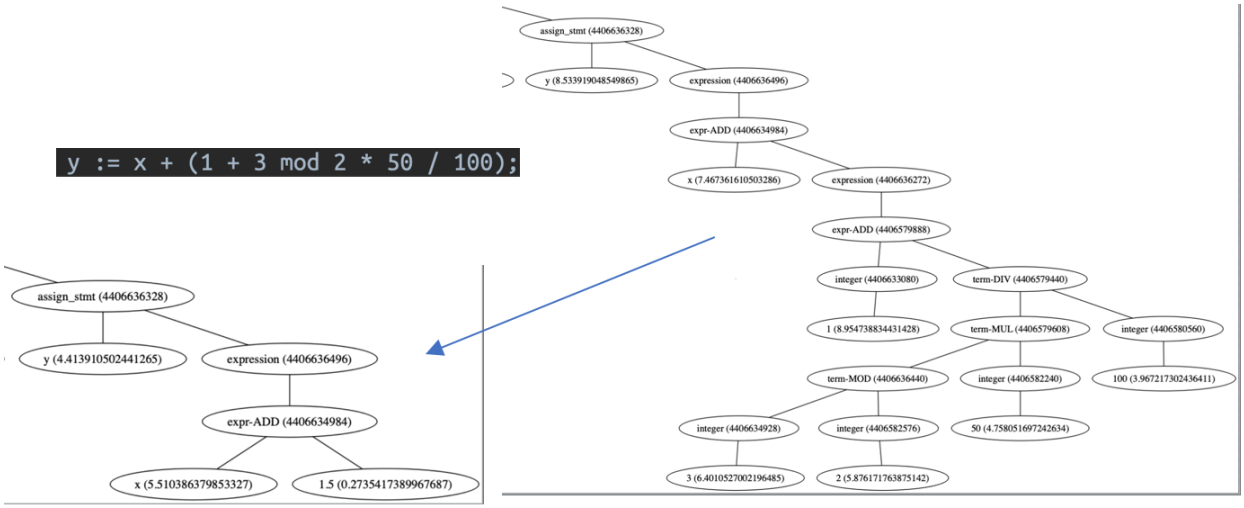


Figure 3: constant folding demo 展示 (2)

同时，在 constant folding 过程中，语义分析将推断 expression/expr/term/factor 节点的数据类型，从而进行不同运算符的类型检查（如 *div* 运算符只能接受两个整数），以及后续的变量类型检查，type casting，我们将在 **TODO** 中详述这一过程。

3.3.2 “压平”语法树

由于在语法规则中有大量的左递归语法，将导致语法树的深度过高，在语义分析中 “压平” 语法树，增加其宽度，减小其深度。以 stmt_list 语法为例，该语法面对大量的 statement 语句将导致树的深度过大，通过语义分析遍历树的过程中，我们可以将各个有效的后裔节点作为 stmt_list 的孩子，缩减树的深度；

```
1  stmt_list :  stmt_list  stmt  SEMICON
2            |  empty
```

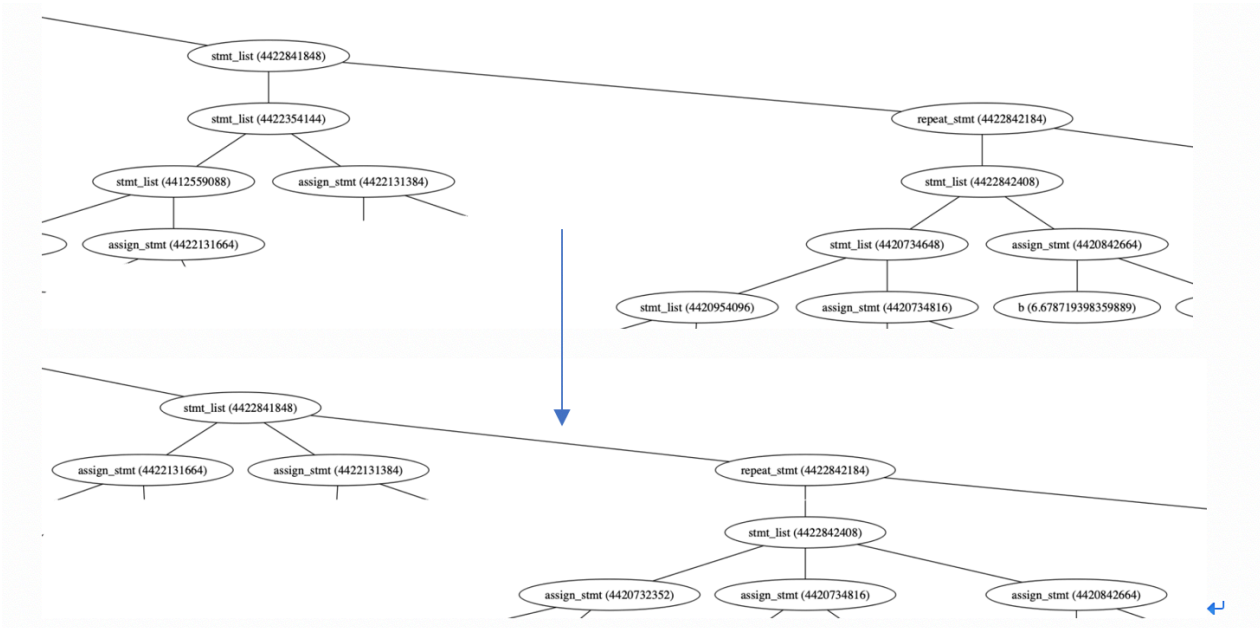


Figure 4: 压平语法树 展示

3.4 语义分析各类检查

这一部分将展示编译器在语义分析阶段进行的各类检查，并通过实际例子展示运行结果，完整的测试文件将在附录中提供；

3.4.1 类型声明检查

pascal 语言支持 alias type 定义，语义分析需要检查各类 type 定义是否 *重复定义* 、 *无中生有* 以及 *数组下标非法*，下面是一段正常的 type 定义，在这段定义中，`int` 为 `integer` 类型的 alias type，`double` 为 `real` 的 alias type，`int_arr` 为 `int` 类型的数组，`int_mat` 为 `int_arr` 类型的数组（矩阵），`int_cube` 为 `int_mat` 类型的数组（三维），`people` 为 record 类型变量，`people_arr` 为 `people` 类型的数组；

```
type
    int=integer;
    double=real;
    int_arr=array[1..3] of int;
    int_mat=array[1..4] of int_arr;
    int_cube=array[1..5] of int_mat;
    people=record
        score: integer;
        sex: char;
        score_arr: int_arr;
    end;
    people_arr=array[1..10] of people;
```

将其进行修改，产生 *重复定义* 、 *无中生有* 、*数组下标非法* 的错误

```
type
    int=integer;
    double=real;
    double=integer;
    int_alias_arr=array[1..3] of int_alias;
    int_arr=array[1..3] of int;
    int_arr=array[1..3] of char;
    int_mat=array[1..4] of int_arr;
    int_cube=array[5..1] of int_mat;
    people=record
        score: integer;
        sex: char;
        score_arr: int_arr;
    end;
    people_arr=array[1..10] of people;
```

```
[INFO] compiling const.pas
[line 21 ERROR] type declare `double` is duplicated
[line 21 ERROR] alias type: `int_alias` used before defined
[line 24 ERROR] type declare `int_arr` is duplicated
[line 25 ERROR] left range val `5` > right range val `1`
[line 26 ERROR] left range type: `char`, right range type: `integer`
[INFO] Find 0 warnings and 5 errors
```

Figure 5: type 无中生有、重复定义、数组下标非法错误

3.4.2 变量类型检查

在变量定义中，语义分析将检查变量是否 *重复定义* 以及变量类型是否 *无中生有*，以下是一段正常的变量声明，变量 `x, y, z` 均为 integer 类型，`arr2` 为 下标从 '3' 到 '5'，

```
var x, y, z: integer;
    arr2: array['3'..'5'] of integer;
    arr1: int_arr;
    mat1: int_mat;
    cube1: int_cube;
    Newton: people;
```

将其修改如下，产生 *重复定义* 、 *无中生有* 、*数组下标非法* 的错误

```
var x, y, z: integer;
    arr2: array['5'..'3'] of integer;
    arr1: int_arr;
    mat1: int_mat;
    cube1: int_cube;
    Newton: people;
    Newton: real;
    yeeef: man;
```

```
[line 43 ERROR] left range val `5` > right range val `3`
[line 49 ERROR] variable `Newton` is already declared
[line 50 ERROR] alias type: `man` used before defined
```

Figure 6: type 无中生有、重复定义、数组下标非法错误

3.4.3 赋值语句类型检查

在赋值语句中，对于左值，语义分析将检查该左值是否 *定义*，*是否为不可改变的 const value*, 具体来说，在我们的语法中，有三种 assign statement grammar：

```
assign_stmt : ID  ASSIGN  expression
            | ID  DOT  ID  ASSIGN  expression
            | ID LB expression RB ASSIGN expression
```

第一种对应简单的变量赋值、第二种对应数组元素赋值、第三种对应 record field 赋值；在三种 assignment 语句中，都出现了 expression 语法，在语义分析中我们针对 expression 做了 constant folding, 将能够提前计算出运算结果的 expression 对应的语法树节点替换为运算结果，**3.3.1** 着重描述了这一过程。

在 constant folding 过程中，语义分析将推断 expression/expr/term/factor 节点的数据类型，从而进行不同运算符的类型检查（如 *div* 运算符只能接受两个整数），以及后续的变量类型检查，type casting。在运算符类型检查上，语义分析采取 **weak consistency**, 不强制要求两个算子一定是同一类型（real 类型和 integer 类型的运算）；

3.4.3.1 运算符算子类型检查与结果类型推断

部分运算符对运算算子有类型要求，在 pascal 语法中，`div`, `mod` 运算符要求两个运算符均为整数，对于 `+`, `-`, `*`, `/` 运算符，语义分析不进行太过严格的类型检查，仅要求两个算子不能为 `char` 类型（我们的编译器不支持 string 类型）；对于关系运算符 `and`, `or`, `not`，语义分析仍不进行太过严格的类型检查，仅要求若一个算子为 `char` 类型，则另一个算子也需要为 `char` 类型；

结果类型推断与运算符与两个算子的类型均有关系，对于 `/` 运算符，运算结果一定为 `real` 类型；对于 `+`, `-`, `*`, 运算符，只要有一个算子为 `real` 类型，则结果为 `real` 类型；对于 `div`, `mod` 运算符，运算结果一定为 `integer` 类型；对于关系运算符 `and`, `or`, `not`, 运算结果一定为 `bool` 类型。

如果变量赋值语句的右值可以完全的 const fold, 语义分析还将依据变量的类型对计算结果进行 type casting, 例如，`x := 1 / 2;`, x 为 integer 变量，则语义分析将会把 0.5 cast 为 0，并在语法树中替换 expression 节点；

例如下面的程序：

```
program ConstFold;
const aa = 1; bb = 1.5; flag = true;
var x, y: integer; z: real;
begin
    x := -3 + aa * 10 div 2 / bb;
    y := x + (1 + 3 mod (2 * 50 / 7));
    z := not flag and aa = 1;
end.

```

```
[line 5 WARN] cast `-0.6666666666666665` to `0` for variable `x`
[line 6 ERROR] div and mod expect 2 integer, but got `integer` and `real`
[line 7 WARN] cast `False` to `0.0` for variable `z`
```

Figure 7: constant folding demo 展示 (3)

- 变量 `x` 为 integer 类型，语义分析将自动将计算结果 -0.666 cast 为 0，并在语法树中做相应替换，见 **Figure 4**
- 变量 `y` 赋值语句中，`mod` 运算符接受了 `real` 运算符，则爆出相应错误；
- 变量 `z` 为 `real` 类型，语义分析自动将计算结果 `False` cast 为 `0.0`, 并在语法树中做相应替换；

3.4.3.2 变量赋值语句左值检查

在 *3.2.1* 以及上小节，我们着重讲述了 const folding 以及 type inference, type cast, type check 的过程，在本部分中，我们将着重讲述变量赋值语句对左值的检查；

```
assign_stmt : ID  ASSIGN  expression

```

对于简单的变量赋值语句，语义分析首先通过 `chain_lookup` （变量可能存在于外 scope 中）判断变量是否存在，再如 3.2.1 所描述的 对 expression 节点进行 const fold 与 type inference，并根据 lookup 到的 variable type 来进行 type checking 与 type casting。

```
assign_stmt : ID  DOT  ID  ASSIGN  expression

```

对于 record 赋值语句，语义分析首先通过 `chain_lookup` 判断该 record 变量是否存在，再判断该 record 变量是否存在该 field，再如 3.2.1 所描述的 对 expression 节点进行 const fold 与 type inference，并根据 lookup 到的 variable type 来进行 type checking 与 type casting。

```
assign_stmt : ID LB expression RB ASSIGN expression

```

对于 array 赋值语句，语义分析首先通过 `chain_lookup` 判断该 array 变量是否存在，再解析中间的 expression 节点（对应下标），检查下标是否在该变量的下标范围之内，最后解析末尾的 expression 节点，进行 const fold 和 type casting。

例如下面的程序：

```
program Arithmetic;
const a = 2; b = 3.4; c = 'l'; flag=true;
type
    int=integer;
    people=record
        score: integer;
        sex: char;
    end;
    people_arr=array [1..3] of people;
var x, y, z: integer; q:boolean; newton: people; peoples: people_arr;
begin
    q := true and true and true and not flag;
    x := (a + 13) div 5 mod 1;
    eistein.sex := 'm';
    newton.sex := 'm';
    newton.not_score := 100;
    peoples[1] := newton;
    peoples[10] := eistein;
end.

```

```
[line 14 ERROR] `eistein` is not a record variable
[line 16 ERROR] `not_score` is not a valid field for record variable `newton`
[line 16 ERROR] `eistein` is not a const or variable or func
[line 18 ERROR] arr `peoples` expect `{'score': 'integer', 'sex': 'char'}` element but got `real`
[line 18 ERROR] arr `peoples` has range `(1, 3)`, but got index 10
```

Figure 8: assignment 语句常见语义错误

3.4.3 if / while / repeat / for / case 语句语义分析与检查

在这一部分中，我们将讨论编译器对于 `if_stmt`, `while_stmt` , `repeat_stmt`, `for_stmt`, `for_stmt`, `case_stmt` 的语义分析与检查；

```
if_stmt : kIF expression kTHEN stmt else_clause
repeat_stmt : kREPEAT stmt_list kUNTIL expression
while_stmt : kWHILE expression kDO stmt
for_stmt : kFOR ID ASSIGN expression direction expression kDO stmt
case_stmt : kCASE expression kOF case_expr_list kEND

```

从他们的语法规则中我们可以看到，每一个 production 所涉及的语义分析已经在之前小节基本提及，比如 if_stmt, 条件判断式为 expression 语句，语义分析只需如 *3.3.3.1* 所述的将 expression node 进行 const fold 与 type inference 即可，stmt 和 else_clause 中则包含 *3.3.2* 中所提及的各类赋值语句的语义分析；所以，对于 if / while / repeat / for / case 语句语义分析与检查本质上是在对其内部的 statement 和 expression 检查，故不做更多赘述；

3.4.4 procedure / function 定义、调用语句语义分析与检查

procedure / func 较之前小节提到的语句更加特殊，因为 procedure 与 function 的定义语句将产生一个新的 scope, 一个新的 symble table 节点，*3.3.4.1* 将着重于 procedure / function 在定义过程中涉及的语义分析，*3.3.4.2* 将着重于 procedure / function 在调用过程中的涉及的语义分析。

3.4.4.1 定义过程中涉及的语义分析

procedure

在 procedure 定义中，首先定义 procedure 传入的参数，在下面的例子中，`print` procedure 共有 6 个参数，分别为 s, x, y, xx, yy, zz，语义分析将在当前 symbol table 中插入该 procedure 的定义描述项；每个 procedure 会产生一个新的 scope, 在这个 scope 中可以定义属于这个 scope 的 constant, type, variable, procedure, function 以及后续 begin end 中的各类语句，语义分析将创建一个新的 symbol table，并将其作为之前 symbol table 的子节点；

对于参数定义以及声明，语义分析将进行仅限于该 scope 的语义检查（对应 symbol table 的 lookup 操作），对参数和变量声明进行 *重复定义* 与 *无中生有* 的检查；

如下面的例子，在该例子中，涉及到了两个 scope, 一个是 program 对应的 scope, 一个是 procedure print 所对应的 scope，在 print procedure 的参数定义中，出现了和主 scope 同名的变量与常数，但是语义分析仅会检查当前 scope，所以并不会报错；而在 procedure 的函数体中，出现了当前 scope 不存在的 `a` 变量，语义分析会一路上溯父亲 symbol table (chain_lookup)，检查该变量是否定义过；

```
program Procedure;
const a = 1; b = 1.5; flag=true;
type
    real_arr=array[2..10] of real;
var x, y, s: integer; arr1: real_arr;

procedure print(var s: real; x, y: integer; xx,yy,zz: boolean);
var aa,bb,cc: real;
begin
    s := x * x + y * y + a;
    writeln(s);
end;

```

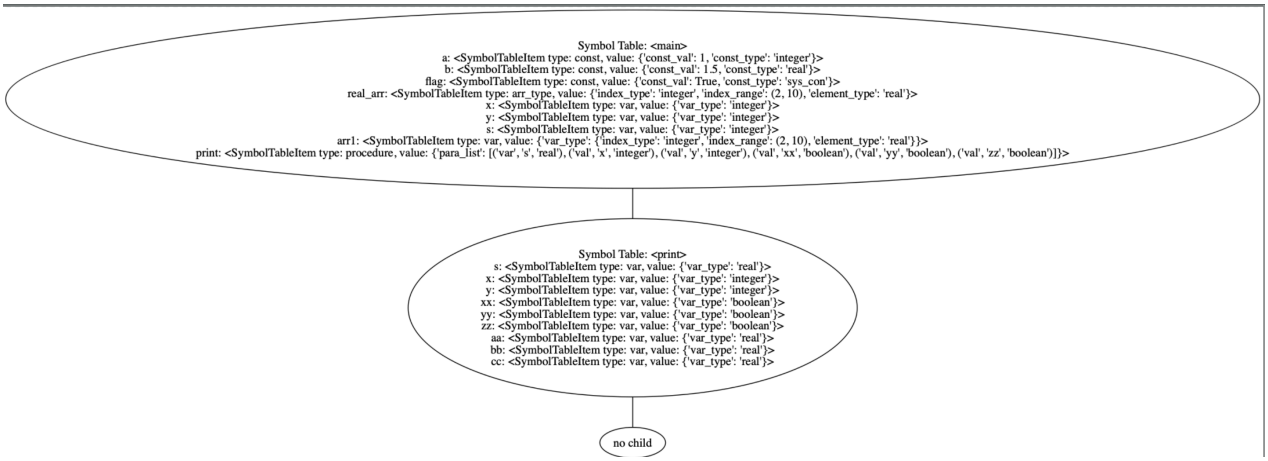


Figure 9:上述例子对应的 symbol table

function

对于 function 定义，情况基本与 procedure 相同，不同点在于 function 自己的 scope 对应的 symbol table 会事先插入与 function 同名的变量（作为返回值变量），防止后续语义分析认定返回值是未经定义的变量；

3.3.4.2 调用过程中涉及的语义分析

procedure

在 procedure 的调用过程中，语义分析首先从当前 symbol table 出发，chain_lookup 检查该 procedure 是否定义，再对传入的参数进行 type 检查以及数量检查，继续从之前的 `print` 例子出发；

```
program Simple;
```

```
const a = 1; b = 1.5; flag=true;
type
    real_arr=array[2..10] of real;
var x, y, s: integer; arr1: real_arr;

procedure print(var s: real; x, y: integer; xx,yy,zz: boolean);
var aa,bb,cc: real;
begin
    s := x * x + y * y + a;
    writeln(s);
end;
begin
print(a, true, false);
print(b, a, a * a, true, flag, not flag);
end.

```

```
[line 14 ERROR] procedure `print` expect 6 args, got 3 args
[line 14 WARN] procedure `print` arg `s` expect `real` got `integer`
[line 14 WARN] procedure `print` arg `x` expect `integer` got `boolean`
[line 14 WARN] procedure `print` arg `y` expect `integer` got `boolean`
```

Figure 10:上述例子输出结果

在上述例子中，我们对 `print` procedure 进行了两次调用，第一次是错误调用，仅提供了三个参数，并且参数类型有 inconsistency，由于在我们的编译器中实行 weak consistency 的机制，参数类型错误仅会报出 warning。

function

function 调用过程涉及的语义检查与 procedure 类似，不再赘述；

4、优化考虑

4.1 错误处理优化

在语法分析部分，我们用 resynchronization 错误恢复机制对错误处理进行了优化，这样可以将语法设计成在一个相对容易恢复和继续分析的点捕获错误。相对于panic mode更有优势。具体细节在语法分析部分的**2.3节错误处理** 部分进行了详细阐述。

4.2 语义分析中对语法树的缩减优化

在语义分析阶段，我们通过 *constant folding* 以及 *压平 语法树* 的操作，一定程度的缩减了由语法分析提供的原始语法树，在 `visualization` 文件夹中我们提供了 `original_ast.png` 以及 `final_ast.png` 作为对比；具体 *constant folding* 以及 *压平 语法树* 已在 **3.2.1** 及 **3.2.2** 进行过详细叙述，不在此重新赘述；

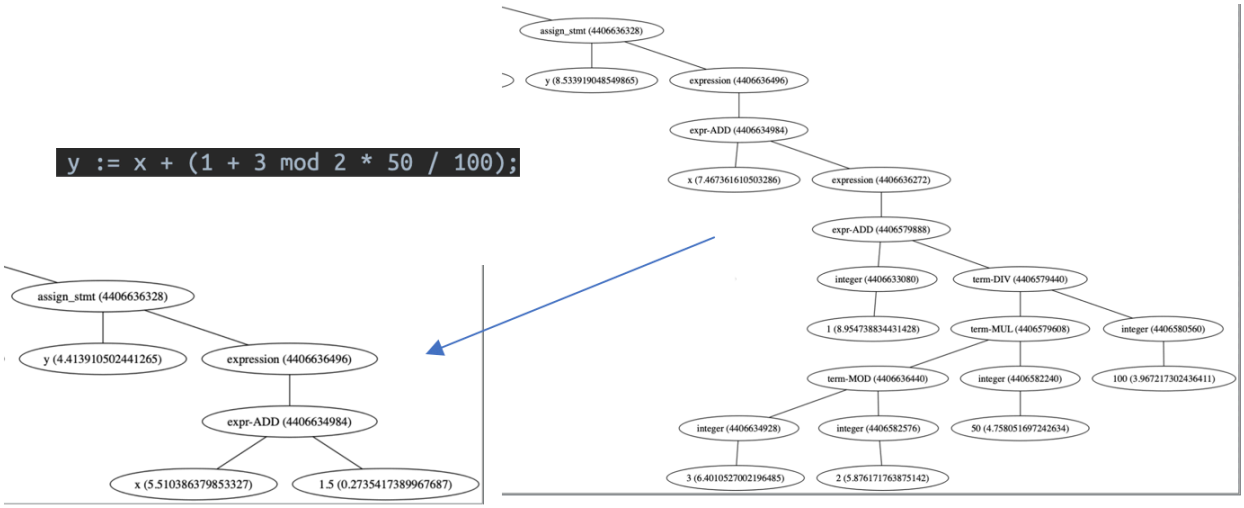


Figure 11: constant folding demo 展示 (2)

5、代码生成

5.1 Overview

为了考虑开发效率，我们将语义分析和代码生成完全分开，语义分析负责所有的正确性检测和常数折叠。代码生成则是基于语义分析后新产生的语法树进行的，主要分为几大部分`control statement`、`assignment statement`、`routine statement`、`expression statement`，分别对应条件语句、赋值语句、函数及过程语句、算术表达式语句。

我们使用的语言为`Pascal`语法子集，代码生成为`three-address code`中间代码，符号基本沿用教材定义，对于支持的功能、语法原生不支持的功能、语法支持但未实现的功能，会在以下展示出来，报告顺序基本按照我们定义的`yacc`语句顺序展开。

关于主要类和函数：

- **class Quadruple(object)** — 四元组数据结构，存储形如(op, target, right_1, right_2)四元组，根据需要任何一个都可以为`None`。
- **class CodeGenerator(object)** — 供主函数调用的类。
- **def _add_new_quad(...)** — 添加新的四元组。
- **def _traverse_ast_gen_code(self, root_node)** — 主循环，几乎所有的判断都在这里实现，不断被递归调用。
- **def gen_quad_list_in_expression_node(self, expression_node)** — 专门为`expression`结点设计处理函数，这是由于其他语句不需要返回值，但这里作为算术运算部分，自成生态，这里面的所有语句都需要拥有返回值，要么直接是值本身，要么是存储这个值的临时变量名，供其他语句调用，这个函数使用非常频繁。
- **def gen_quad_list_from_expression_node(self, expression_node)** — 为`expression`结点下设计的处理函数，主要包含`expr/term/factor`语句。被上面的函数调用。
- **def traverse_skew_tree_gen(node, stop_node_type=None)** — 将`node`结点左递归展平，按照`stop_node_type`为准则决定何时停止进程。

- **def traverse_skew_tree_bool(node, stop_node_type, target_node_type)** — 此处和上面不同，专用于`term-AND/expr-OR`条件语句，这里是为了辅助从左至右只要有一个语句满足了退出条件，则需要退出，因此，此处是按照连续的同名结点展开，当结点为下一级或同级非同名时，需要退出递归过程。

5.2 Routine Statement

此部分展示关于`function/procedure`部分，包含了定义和调用模块，实现严格按照教材示例用法，由于只生成中间代码，因此此处没有考虑返回值及返回地址的传递和堆栈。

5.2.1 定义部分

宏观调度

```
# routine : routine_head routine_body
```

```
            if root_node.type == 'routine':
                self._traverse_ast_gen_code(root_node.children[0])
                self._traverse_ast_gen_code(root_node.children[1])
                if len(self._routine_stack) > 0:
                    if self._routine_stack[-1][1]:
                        self._add_new_quad('return',
self._routine_stack[-1][0])
                    self._add_new_quad('end_define',
self._routine_stack[-1][0])
                    self._routine_stack.pop()
```

> `routine`是声明与执行语句的连接点，Pascal语言要求声明和定义必须在执行语句之前，而`routine_head`则包含了基本声明（名称、参数、返回值？，局部临时变量？），`routine_body`则是执行代码部分，如果`routine_head`的儿子全为空，则说明这是主函数，而非声明或定义。

此处由于函数嵌套定义的存在，因此必须借助堆栈来记录函数名称，这是由于Pascal语言没有`return`这样的保留字，默认为将函数名称视为返回值，因此，我必须记录返回值（即函数名）来完成返回语句。加上嵌套，因此选用堆栈，处理完声明部分，再处理执行语句，最后加一个返回语句。

声明部分

```
# function_head :  kFUNCTION  ID  parameters  COLON  simple_type_decl
```

```
            elif root_node.type == 'function_head':
                self._add_new_quad('entry', root_node.children[0])
                self._routine_stack.append((root_node.children[0], True))
```

```
# procedure_head :  kPROCEDURE ID parameters
```

```
            elif root_node.type == 'procedure_head':
                self._add_new_quad('entry', root_node.children[0])
                self._routine_stack.append((root_node.children[0], False))
```

> 此部分就是输出函数入口语句，然后将函数名堆栈，这里只存在声明，不存在执行语句，因此与生成代码无关，主要是语义分析需要使用。
>
> - True -- 表示函数声明，需要加返回语句。
> - False -- 表示过程声明，不需要返回语句。

5.2.2 调用部分

procedure

```
# proc_stmt :  ID
#           |  SYS_PROC
```

```
                if root_node.type == 'proc_stmt-simple':
                    self._add_new_quad('call', children[0])
```

> 此处直接产生`call+name`即可，表示无条件跳转到procedure执行代码部分。

```
# proc_stmt :  ID  LP  args_list  RP
#           |  SYS_PROC  LP  expression_list  RP
#           |  kREAD  LP  factor  RP
```

```
              if children[0] == 'read':
                  self._add_new_quad('begin_args', None)
                  args_val =
self.gen_quad_list_from_expression_node(children[1])
                  self._add_new_quad('read', args_val)
```

> 这是专门给`read`函数的执行语句，形式为`read args`，由于这个语法本身拥有这个语句，所以单独处理了。

```
# expression_list :  expression_list  COMMA  expression
#                 |  expression
# args_list :  args_list  COMMA  expression
#           |  expression
```

```
                else:
                    self._add_new_quad('begin_args', None)
                    args_list = traverse_skew_tree_gen(children[1],
'expression')
                    for args in args_list:
                        if isinstance(args, Node):
                            ret_val =
self.gen_quad_list_in_expression_node(args)
                            self._add_new_quad('args', ret_val)
                        else:
                            self._add_new_quad('args', args)
                    self._add_new_quad('call', children[0])
```

> 这里主要处理前两种语句，由于`args_list`和`expression_list`本质是一样的，因此可以联合处理。先计算参数值，这里通过将左递归语法树展平，再来从左至右逐一扫描，每一个`expression`均代表一个参数的计算，最后调用即可，不存在返回值。

function

```
# factor  : SYS_FUNCT
#         | ID  LP  args_list  RP
#         | SYS_FUNCT  LP  args_list  RP
```

```
        elif expression_node.type == 'factor-func':
            target = self.new_tmp_var
            if len(expression_node.children) == 1:
                self._add_new_quad('call', expression_node.children[0],
target)
            else:
                func_name, args_list_node = expression_node.children
                self._add_new_quad('begin_args', None)
                args_list = traverse_skew_tree_gen(args_list_node,
'expression')
                for args in args_list:
                    ret_val = self.gen_quad_list_in_expression_node(args)
                    self._add_new_quad('args', ret_val)
                self._add_new_quad('call', func_name, target)
            return target
```

> 函数调用位于`expression`系列中，这是因为函数调用存在返回值，可以作为赋值语句的右端。基本思路和过程调用类似，区分处理了系统自带无参函数和其余有参函数。只不过在最后需要返回存储函数返回值的临时变量名。

5.3 Assignment Statement

这是比较简单的一部分，只需要分为几种不同的左值处理即可。

5.3.1 普通变量

```
# ID ASSIGN expression
```

```
  if root_node.type == 'assign_stmt':
      if len(children) == 2:
          id_, maybe_expression_node = children
          if not isinstance(maybe_expression_node, Node):
              self._add_new_quad(None, id_, maybe_expression_node)
           else:
              val =
self.gen_quad_list_in_expression_node(maybe_expression_node)
              self._add_new_quad(None, id_, val)
```

> 此处直接通过`expression`处理函数，当然首先是保证右值不是常数而是结点的前提下。最后直接赋值即可。

5.3.2 数组变量

```
# ID LB expression RB assign expression
```

```
    elif root_node.type == 'assign_stmt-arr':
        id_, index_expression_node, val_expression_node = children
        index_val =
self.gen_quad_list_in_expression_node(index_expression_node)
        assign_val =
self.gen_quad_list_in_expression_node(val_expression_node)
        self._add_new_quad(None, f'{id_}[{index_val}]', assign_val)
```

> 此处也是直接进行两次`expression`调用获取索引值和右端值，按照教材写法，这里直接使用数组写法作为三段码输出即可。

5.3.3 结构体变量

```
# assign_stmt : ID  DOT  ID  ASSIGN  expression
```

```
            else:
                record_name, field_name, val_expression = root_node.children
                address_var = self.new_tmp_var
                self._add_new_quad('address', address_var, record_name,
field_name)
                ret_val =
self.gen_quad_list_in_expression_node(val_expression)
                self._add_new_quad(None, '*' + address_var, ret_val)
```

> 此处唯一要讲的是按照教材关于结构体成员取值的写法：
>
> t = &x + field_offset(x, field_name)

5.4 Control Statement

这一部分算是生成代码比较核心的一部分，主要贯穿着与汇编基本相同的写法，处理情况较多，Pascal包含了很多种循环体结构语句。但一旦厘清顺序和逻辑，本身并不复杂，由于这部分主要是控制语句，下面不会贴上代码，因为冗杂而不易理解，只展示控制流。

5.4.1 if statement

```
# if_stmt :  kIF  expression  kTHEN  stmt  else_clause
# else_clause :  kELSE stmt
#             |  empty
```

```
t <- if_expression
if_false t goto else_label

{if_stmt part}

goto exit_label
else_label

{else_stmt part}?(if exist)

exit_label
```

5.4.2 repeat statement

```
# repeat_stmt :  kREPEAT  stmt_list  kUNTIL  expression
# stmt_list :  stmt_list  stmt  SEMICON
#           |  empty
```

```
enter_label

{repeat_stmt}

t <- repeat_expression
if_false t goto exit_label
goto enter_label
exit_label
```

5.4.3 while statement

```
# while_stmt :  kWHILE  expression  kDO stmt
```

```
judge_label
t <- while_expression
if_false t goto exit_label

{while_stmt}

goto judge_label
exit_label
```

5.4.4 for statement

```
# for_stmt :  kFOR  ID  ASSIGN  expression  direction  expression  kDO stmt
# direction :  kTO
#            | kDOWNTO
```

```
ID <- start_value_expression
bound_var <- end_value_expression

judge_label

*'to': t <- ID <= bound_var
*'downto': t <- ID >= bound_var

if_false t goto exit_label

{for_stmt}

*'to': ID <- ID + 1
*'downto': ID <- ID - 1

goto judge_label
```

```
exit_label
```

5.4.5 case statement

```
# case_stmt : kCASE expression kOF case_expr_list kEND
# case_expr_list :  case_expr_list  case_expr
#                |  case_expr
# case_expr :  const_value  COLON  stmt  SEMICON
#           |  ID  COLON  stmt  SEMICON
#           |  kELSE  COLON  stmt  SEMICON
```

```
choose_var <- case_expression
case_list <- flatten the case_expr_list tree
for case_expr in case_list

{judge_value <- case_expr.children[0]
 next_label <- new_label
 t <- choose_var == judge_value
 if_false t goto next_label
 {case_expr_stmt}
 goto exit_label
 next_label}

 exit_label
```

5.4.6 goto statement

> 我们没有支持`goto`指令，因为这对语义分析带来了麻烦。

5.5 Expression Statement

这部分也是非常重要的部分，涉及到所有的算术运算，也是非常基本的分析部分。这块内容主要包含两个函数

- **def gen_quad_list_in_expression_node(self, expression_node)**
- **def gen_quad_list_from_expression_node(self, expression_node)**

5.5.1 expression node

这个函数是专门针对expression结点进行的，由于yacc中我们要求expression必须有结点，因此这里本质是算术运算的入口。

```
# expression :  expression  GE  expr
#            |  expression  GT  expr
#            |  expression  LE  expr
#            |  expression  LT  expr
#            |  expression  EQUAL  expr
#            |  expression  UNEQUAL  expr
#            |  expr
```

```
def gen_quad_list_in_expression_node(self, expression_node):
  if not isinstance(expression_node, Node):
      return expression_node
  if len(expression_node.children) == 1:
      return
self.gen_quad_list_from_expression_node(expression_node.children[0])
  else:
 left_val =
self.gen_quad_list_in_expression_node(expression_node.children[0])
 right_val =
self.gen_quad_list_from_expression_node(expression_node.children[2])
      target = self.new_tmp_var
      op = expression_node.children[1]
      if op == '=':
          self._add_new_quad('==', target, left_val, right_val)
      elif op == '<>':
          self._add_new_quad('!=', target, left_val, right_val)
      else:
          self._add_new_quad(op, target, left_val, right_val)
      return target
```

> 此处只区分了是否含有一个以上的孩子结点，若没有，则根据是否为结点直接返回相应值或者临时变量；若有，则递归调用自己或者处理后部分的函数，得到后做判断即可。

5.5.2 expr/term/factor node

这里分为四大部分，分别为`internal expression node`、`factor node`、`expr-OR/term-AND node`、`left node`。

internal expression node

```
# factor : LP  expression  RP
```

```
            if expression_node.type == 'expression':
                return
self.gen_quad_list_in_expression_node(expression_node)
```

> 这是一种特殊情况，虽然实际应用中，这个内部的expression node会直接接在外层的expression node下，但为了保险起见，还是在这里加一句。

---

factor node

这里又分为几种不同的右值情况，与`assignment statement`左值情况类似。

```
# kNOT factor
# SUBSTRACT factor
```

```
            elif expression_node.type == 'factor':
                children = expression_node.children
                unary_op, right_child = children
                right_val =
self.gen_quad_list_from_expression_node(right_child)
                target = self.new_tmp_var
                self._add_new_quad(unary_op, target, right_val)
                return target
```

> 这是单操作符（not, -）运算。

```
# factor  : ID  LB  expression  RB
```

```
        elif expression_node.type == 'factor-arr':
            arr_id, right_child_node = expression_node.children
            index_val =
self.gen_quad_list_from_expression_node(right_child_node)
            target = self.new_tmp_var
            self._add_new_quad(None, target, f'{arr_id}[{index_val}]')
            return target
```

> 这是数组元素取值。

```
# factor : ID  DOT  ID
```

```
            elif expression_node.type == 'factor-member':
                record_name, field_name = expression_node.children
                target = self.new_tmp_var
                self._add_new_quad('address', target, record_name,
field_name)
                return '*' + target
```

> 这是结构体取成员变量。

> `factor-func`见函数调用部分。

---

expr-OR/term-AND node

这里要这么区分的原因在于，`and/or`操作连续出现时，我们必须按顺序从左往右逐一判断，对于`and`操作而言，本质是连续满足条件，一旦有一个条件不满足就需要直接结束判断，这是因为后续条件很可能需要这个前序条件成立。当然，这也是一种优化，我们不必一定要从头判断到尾，只要中间有一个打破或者满足条件了，就能离开或者进入。

```
# expr :  expr  kOR  term
```

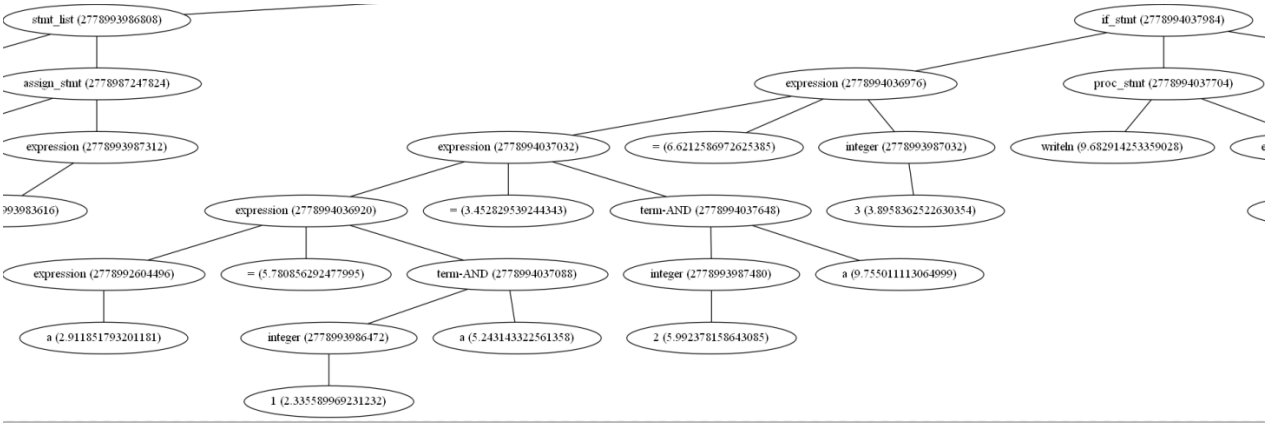
```
      if expression_node.type == 'expr-OR':
          bool_list = traverse_skew_tree_bool(expression_node, 'term',
'expr-OR')
          jump_label = self.new_label
          for or_node in bool_list:
              condition_value =
self.gen_quad_list_from_expression_node(or_node)
              self._add_new_quad('goto', jump_label, 'if',
condition_value)
          target = self.new_tmp_var
          self._add_new_quad(None, target, 0)
          exit_label = self.new_label
          self._add_new_quad('goto', exit_label)
          self._add_new_quad(jump_label, None)
          self._add_new_quad(None, target, 1)
          self._add_new_quad(exit_label, None)
          return target
```

> 这里最关键的就是
>
> - **def traverse_skew_tree_bool(node, stop_node_type, target_node_type)**
>
> 这个函数功能在`overview`部分已经有介绍，目的是根据连续的`and/or`结点将表达式打断，形成一条长串，然后按照之前说的情形进行判断。
>
> `term-AND`类似不再赘述。

⚠ 这里有一个语法本身的问题出现。

```
if (a = 1 and a = 2 and a = 3) then
```

在这个语句里，我们的语法图中为



我们可以看到，在我们的语法中，`expression`的优先级高于`term`，因此产生`shitf-reduce`冲突时，优先进行`shift`操作，比如栈中出现`$a=1`时，这里有两个选择，一个是规约产生`expression`，一个是继续移入再规约产生`term`，由于默认移入，因此右值就被`term`结点抢走了，没有达到我们的要求，我们应当优先规约。

但我们可以换一种写法，即每一个判断式都加上括号

```
if ((a = 1) and (a = 2) and (a = 3)) then
```

这样就会导致始终按照`(factor)`规约，保证`expression`的产生。

---

left node

这里指的是剩余的结点操作可以统一起来

```
# expr :  expr  ADD  term
#       |  expr  SUBTRACT  term
#       |  term
# term :  term  MUL  factor
#       |  term  kDIV factor
#       |  term  DIV  factor
#       |  term  kMOD  factor
#       |  factor
```

```
else:
   if len(expression_node.children) == 1:
      return
self.gen_quad_list_from_expression_node(expression_node.children[0])
   left_child, right_child = expression_node.children
   left_val, right_val =
self.gen_quad_list_from_expression_node(left_child), \

self.gen_quad_list_from_expression_node(right_child)

   bin_op = type_to_bin_op[expression_node.type]
   target = self.new_tmp_var
   self._add_new_quad(bin_op, target, left_val, right_val)
   return target
```

> 此处处理很基本，不再赘述。

5.6 Optimization

5.6.1 识别不可到达代码

这里主要针对bool条件判断，即`if/while/repeat/case/for`语句

if

```
            condition_value =
self.gen_quad_list_in_expression_node(if_expression)
            if condition_value is True:
                self._traverse_ast_gen_code(if_stmt)
                return
            elif condition_value is False:
                if isinstance(else_part, Node):
                    self._traverse_ast_gen_code(else_part)
                    return
                else:
                    return
```

> 此处如果if后面是`True`则直接进入if代码段执行，不考虑else部分，否则考虑else部分，如果有则执行，没有这个if语句就作废。

while

```
        condition_value = self.gen_quad_list_in_expression_node(while_expression)
        if condition_value is False:
            return
        if condition_value is not True:
            self._add_new_quad('goto', jump_loop_label, 'if_false', condition_value)
```

> 此处如果是False，则直接丢弃while循环，如果为True则不需要重复判断。

repeat

> 与while几乎一模一样。

for

```
                start_val = self.gen_quad_list_in_expression_node(start_expression)
                stop_val = self.gen_quad_list_in_expression_node(stop_expression)
                if not isinstance(start_val, str) and not isinstance(stop_val, str):
                    if op == 'to' and start_val > stop_val:
                        return
                    elif op == 'downto' and start_val < stop_val:
                        return
```

> 此处主要用于判断上下界是否反过来，如果反过来则直接舍弃for代码。

case

```
                const_flag = False
                ret_val = self.gen_quad_list_in_expression_node(case_expression)
                if not isinstance(ret_val, str) or len(ret_val) == 1:
                    const_flag = True
                ...
                    if const_flag and type(ret_val) == type(judge_val):
                        if ret_val == judge_val:
                            if next_entry_label != "":
                                self._add_new_quad(next_entry_label, None)
                            self._traverse_ast_gen_code(entry_stmt)
                            return
                        else:
                            continue
```

> 这里情况为如果判别值是一个常值且case语句也是常值，则可以直接比较，此时只需要保留相等的执行语句或者是参数的表达语句，其余不相等的常值直接舍弃。

5.6.2 识别尾递归

这里主要是用于函数和过程

```
                enter_label = self.new_label
                pos = len(self._quad_list)
                self._traverse_ast_gen_code(root_node.children[1])
                if len(self._routine_stack) > 0:
                    tail = self._quad_list[-1]
                    if tail.op == 'call' and tail.target ==
self._routine_stack[-1][0]:
                        self._quad_list.pop()
                        self._quad_list.insert(pos, Quadruple(enter_label,
None))
                        self._add_new_quad('goto', enter_label)
```

> 此处先遍历定义部分，记录下位置，在执行语句中如果最后是调用自己，都需要弹出这个调用，并且在之前记录的位置处加上标签，直接跳到起始标签处。

5.6.3 连续条件判断

见之前部分，此处不再赘述。

6、测试案例

测试部分案例较多，我们将所有的测试文件均放置在 `soup/test` 文件夹下，通过我们提供的 `test_script.py` 可以进行批量测试，重现报告中提到的所有结果；