设计概述

设计并实现一个 Pascal-语言的编译系统，掌握编译原理的各个环节：词法分析、语法分析、语义分析、代码生成，以及实现所需的数据结构：语法树、符号表等。

1. 词法分析

1.1 词法分析模块功能

输入	输出
Pascal 源码	识别出每个 token，并对每个 token 返回相应的 token 信息

1.2 Pascal 代码词法分析

(1)大小写无关性

Pascal 代码中，每个 identifier(变量名、函数名、保留字等)都是大小写无关的，也就是说”aBcDefG”和”AbCdefG”这两个 identifier 在 pascal 中是等价的。

我们使用 lex 中提供的%option caseless 这个选项来实现这个特性。

(2)整数、实数

Lex 代码:

```
DIGITS ([0-9])+
    INTEGER_10 {DIGITS}
    INTEGER $?{DIGITS}
    REAL {INTEGER_10}"."{DIGITS}(E{INTEGER_10})?
…
%%
…

{REAL} {
    yylval = NEWNODE(TK_REAL);
    yylval->lineno = yylineno;
    double tmp;
```

```
    sscanf(yytext, "%lf", &tmp);
    yylval->value = Value(tmp);
    return TK_REAL;
}  /*  实数类型应写在整数类型之上  */
{INTEGER} {
    yylval = NEWNODE(TK_INTEGER);
    yylval->lineno = yylineno;
    char *ts = yytext;

    int tmp;
    if (*ts == '$') sscanf(++ts, "%x", &tmp);
    else            sscanf(ts, "%d", &tmp);
    yylval->value = Value(tmp);
    return TK_INTEGER;
}
…
```

整数、实数分析时需要注意的有:

➢pascal 中 16 进制整数是以”$”后面跟一个 16 进制整数的形式表示，在解析的时候需要注意。

➢在词法分析解析时，实数的解析优先级应该高于整数的解析优先级。这是因为实数的前端实质上就是一个整数，如果将整数放在前面，那实数的前端就会被解析成一个整数，然后词法分析就会发生错误。如：对于”1.2”这个浮点数，如果整数优先级高于实数优先级，那么我们就会得到整数”1”、操作符”.”和整数”2”这三个 token，这明显是不对的。

➢词法分析中不能翻译整数的正负号。这是因为词法分析无法分清”减去一个数”和”负数”的区别。比如，若是词法分析中加上了负数的判断，那么”1-2”中的减号就会被翻译成负号。这种情况下，即使是将负号的解析优先级放到整数、实数的前面也是不行的，这样的话，对于”-2”这种负数，就会被解析成”减号”和”2”了。事实上，正负号的解析是语法分析做的事。

(3)字符与字符串

Lex 代码:

```
CHAR      '([^'\n]|'')'
STRING    '([^'\n]|'')*'
```

字符、字符串分析时需要注意的有:

➢在 Pascal 中，单引号字符以两个单引号字符表示，在 Pascal 代码中写为''''。

➢对于空字符，Pascal 解析成字符串而不是单字符变量。

(4)标识符

Lex 代码:

```
ID   [A-Z]([_A-Z0-9])*
```

标识符分析时需要注意的有:

➢标识符的解析优先级应低于所有关键字的优先级。

➢在真正的 pascal 中，program_head 中的标识符是不能以下划线开头的，而变量里的标识符是可以的。

(5)Pascal 常量

Lex 代码:

```
SYS_CON "maxint"|"maxlongint"|"true"|"false"
```

//注：因为 boolean 类型常量就只有两种值”true”和”false”，所以我直接将其写在 SYS_CON 中进行翻译。

(6)注释

Lex 代码:

```
"{" {
    char c;
    while ((c=yyinput())!='}') {
        if (!(~c)) endlessComment();
        if (c=='\n') yylineno++;
    }
}
"(*" {
    char c;
    do {
        while ((c=yyinput())!='*') {
            if (!(~c)) endlessComment();
            if (c=='\n') yylineno++;
        }
        while ((c=yyinput())=='*');
        if (!(~c)) endlessComment();
    if (c=='\n') yylineno++;
        if (c==')') break;
    } while (1);
}
```

解析注释时需要注意的有:

➢注释解析中需要处理行号的增加。

➢注释解析中需要判断是否是读到了文件尾而注释还没有结束，并在这种情况发生时进行报错。

上面代码中，endlessComment 函数就是对这种情况进行报错处理。

(7)其他

Lex 代码:

```
\n {
    ++yylineno;
}
\r {}
^[ \t]+ { /*  缩进  */ }
[ \t\f]+ { /*  skip whitespace  */ }
. { return ERROR; }
```

注意:对于无法识别，而又不是空白符的字符，我们需要进行报错。

2. 语法分析

语法分析需要做的事情有下面几个:

➢首先需要根据文法进行语法树的构建，在建树的过程当中，首先需要对每个节点都进行标识，这样方便后面进行语法树的遍历，同时在建树的过程中标识节点也能够对后面的错误检查和符号表建立有帮助。

➢进行语法的错误检查，要是程序的语法有误那么需要提醒进行相应的错误。这里主要处理缺少符号不对的错误。

➢节点压缩，这里为了能让遍历的节点尽量的少，需要对语法树的节点进行压缩处理。这里的压缩处理除了 expression, expr, term 的文法意外，其余的只要是同级的就需要进行节点并列排放，进行节点的压缩。

2.1 节点信息

首先列出节点的类型，如下所示:

```
%token     TK_AND TK_ARRAY TK_ASSIGN TK_CASE TK_TYPE TK_SYS_TYPE
%token     TK_COLON TK_COMMA TK_CONST TK_DIGITS TK_DIV  TK_REM TK_DO TK_DOT TK_DOTDOT
%token     TK_DOWNTO TK_ELSE  TK_ELSE_NULL TK_END TK_EQUAL TK_FOR
%token     TK_FUNCTION TK_GE TK_GOTO TK_GT TK_ID TK_REF TK_IF TK_IN TK_LB
%token     TK_LE TK_LP TK_LT TK_MINUS TK_MOD TK_UNEQUAL TK_OF TK_OR
%token     TK_OTHERWISE TK_BEGIN TK_PLUS TK_PROCEDURE
%token     TK_PROGRAM TK_RB TK_REAL TK_RECORD TK_REPEAT TK_RP TK_SYS_PROC TK_READ TK_READLN
%token     TK_SEMI TK_MUL TK_THEN TK_NOT
%token     TK_TO TK_UNTIL TK_UPARROW TK_VAR TK_WHILE TK_SET TK_STARSTAR
%token     TK_CHAR TK_STRING TK_INTEGER TK_SYS_CON TK_SYS_FUNCT TK_WITH TK_NIL
%token     ERROR

%token     TK_PROGRAM_HEAD TK_ROUTINE TK_ROUTINE_PART TK_ROUTINE_HEAD TK_ROUTINE_BODY TK_CONST_PART TK_CONST_PART_END
       TK_TYPE_PART TK_TYPE_PART_END TK_VAR_PART TK_VAR_PART_END TK_ROUTINE_PART_RF TK_ROUTINE_PART_RP TK_ROUTINE_PART_FUNC TK_ROUTINE_PART_PROC
TK_ROUTINE_PART_NULL TK_STMT_LIST TK_STMT_LIST_NULL TK_STMT_LABEL TK_STMT TK_CP_STMT

%token     TK_VAL_PARA_LIST TK_NON_LABEL_STMT_ASSIGN TK_NON_LABEL_STMT_PROC TK_NON_LABEL_STMT_CP
               TK_NON_LABEL_STMT_IF TK_NON_LABEL_STMT_REP TK_NON_LABEL_STMT_WHILE TK_NON_LABEL_STMT_FOR
               TK_NON_LABEL_STMT_CASE TK_NON_LABEL_STMT_GOTO
               TK_EXP_LIST TK_EXP_LIST_END TK_EXP TK_TERM  TK_CONST_MINUS TK_ID_MINUS
       TK_FACTOR_ID TK_FACTOR_ID_ARGS TK_FACTOR_SYS_FUNCT TK_FACTOR_CONST TK_FACTOR_EXP TK_FACTOR_NOT
       TK_FACTOR_MINUS TK_FACTOR_ID_EXP TK_FACTOR_DD TK_ARGS_LIST TK_ARGS_LIST_END
       TK_CONST_DL TK_CONST_DL_END TK_TYPE_DL TK_TYPE_DL_END TK_TYPE_DEF TK_TYPE_DECL TK_TYPE_DECL_SIM TK_TYPE_DECL_ARR TK_TYPE_DECL_REC
       TK_FIELD_DL TK_FIELD_DL_END TK_FIELD_DECL TK_NL TK_NL_END
       TK_STD_SYS_TYPE TK_STD_ID TK_STD_NL TK_STD_DD TK_STD_DD_M TK_STD_DD_MM TK_STD_DD_ID TK_DL TK_DL_END TK_VAR_DECL TK_FUNC_DECL TK_FUNC_HEAD
       TK_PROC_DECL TK_PROC_HEAD TK_PARA  TK_PARA_TL TK_PARA_NULL TK_PARA_DL TK_PARA_DL_END TK_PARA_TL_VAR TK_PARA_TL_VAL TK_PARA_TL_END TK_PROC
       TK_CASE_EL TK_CASE_EL_END TK_CASE_EXPR TK_CASE_EXPR_END TK_EXPR TK_ASSIGN_ID TK_ASSIGN_ID_EXPR TK_ASSIGN_DD
       TK_PROC_ID TK_PROC_ID_ARGS TK_PROC_SYS TK_PROC_SYS_ARGS TK_PROC_READ TK_PROC_READLN
       TK_STMT_ASSIGN TK_STMT_PROC TK_STMT_CP

%%
```

首先需要将 common.h，stdio.h 以及 stdlib.h 的头文件包含在.y 文件里面。其中 common.h 是组内设计的通用头文件，里面包含了节点结构以及其他的一些信息。

```c
#include "common.h"
#include <stdio.h>
#include <stdlib.h>
int DEBUG = 0;
int IS_SYNTAX_ERROR=0;
int syntax_const_error = 0;
NODE* ROOT;
extern int yylineno;
```

下面是节点 NODE 的结构，节点的各种信息如下所示:

```c
typedef struct NODE{
    string name;
        Value value;
        int type;
    int child_number;
    struct NODE** child;
    struct NODE* record;
        Type dataType;
    SymbolTable* symbolTable;
        int lineno; //output the error line number
} NODE;
```

➢**name** - 节点名字，可以方便调试信息当中显示节点信息，同时也能够作为结点的标识符

➢**value** - 如果节点是一个表达式或者是一个变量、常量，那么 value 存储它相对应的值，例如 bool, int, real 等等

➢**type** - 节点的标识

➢child_number – 该节点的子节点个数，如果没有则为 0

➢child – 该节点的子节点数组

➢record – id1.id2 中 id2 是 id1 的 record，其中 id1 是要给记录类型

➢dataType – 数据类型

➢symbolTable – 符号表

➢lineno – 该节点所在的行号，如果它不是叶子节点，那么他的行号是所有子节点的最小行号

2.2 建树

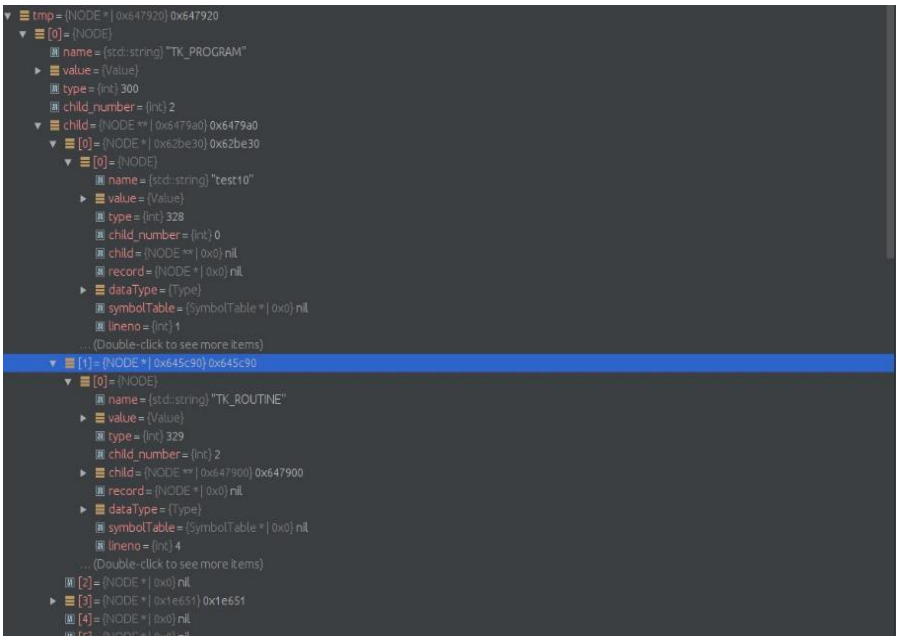
建立语法树的过程如下所示，在每一个文法后面都进行语法树节点的构建，当然在构建的时候一些可以推导并且对于后面的语义分析和代码生成无意义的会在构建树的时候忽略掉，如下所示，TK_DOT 这个终结符在进行建树的过程被忽略掉了。另外，在建树的时候首先使用 NEWNODE 函数进行节点的动态分配，并且将节点设置为 TK_PROGRAM 的标识进行标记，这是 program 的节点。

```
program : program_head routine TK_DOT{
            if(DEBUG){
                printf("PARSING PROGRAM\n");
            }
            $$ = NEWNODE(TK_PROGRAM);
            $$->child = MALLOC($$,2);
            $$->child[0] = $1;
            $$->child[1] = $2;

            $$->lineno = MIN($1,$2);

            if(syntax_const_error)
                ROOT = NULL;
            else
                ROOT = $$;
        }
```

除了要设置 program 的节点，还需要将它的两个儿子依序放到儿子节点数组里面，并且需要设置 child_number 为相应的值。这里使用 MALLOC 宏定义。

```
#define MALLOC(pointer,num) new NODE*[pointer->child number=num]
```

如下所示就是建树的节点在 DEBUG 模式下的截图，可以看到 TK_PROGRAM 下面的有两个儿子，第一个儿子是 test10，即程序名字，第二节点是 TK_ROUTINE:



2.3 节点压缩

如下所示，stmt_list 的各个子节点都是 stmt,都是同级的，这里就需要将所有的 stmt 挂在 stmt_list 的下面。

```
stmt_list : stmt_list stmt TK_SEMI{
            if(DEBUG){
                printf("PARSING STMT LIST\n");
            }
            if($1==NULL){
                        $$ = NEWNODE(TK_STMT_LIST);
                        $$->child = MALLOC($$, 1);
                        $$->child[0] = $2;

                        $$->lineno = $2->lineno;
            }
                else{
                        $$ = $1;
                    int old_child_number = $$->child_number;
                    NODE** tmp = $$->child;
                    $$->child = MALLOC($$, (1+old_child_number));
                    int i;
                    for(i=0;i<old_child_number;i++){
                            $$->child[i] = tmp[i];
                    }
                    $$->child[i] = $2;

                    free(tmp);
                    }
        }
```

首先，在 stmt_list : stmt_list stmt TK_SEMI 这里，需要首先判断后面的 stmt_list 是否一个 NULL 节点，如果是那么新建一个几点然后把 stmt 放入到这个节点当中。否则前面的 stmt_list 的节点就等于后面的 stmt_list 几点，并且把 stmt 添加到 stmt_list 的子节点当

中。这样子就可以把所有的 stmt 节点归于 stmt_list 下。另外对于一些无用的节点可以直接越级进行节点的压缩。

2.4 错误处理

错误处理拿 stmt_list 的例子作为补充和解释，如下所示，下面使用了 error 这个伪记号进行使用，如果后面是一个 error 的 token 话，那么就需要进行报错，但是不终止建树，而是将全局变量 IS_SYNTAX_ERROR 置为一：

```
| stmt_list stmt error{
    if(DEBUG){
        printf("PARSING STMT LIST\n");
    }
    if($1==NULL){
        $$ = NEWNODE(TK_STMT_LIST);
        $$->child = MALLOC($$, 1);
        $$->child[0] = $2;

        $$->lineno = $2->lineno;
    }
    else{
        $$ = $1;
        int old_child_number = $$->child_number;
        NODE** tmp = $$->child;
        $$->child = MALLOC($$, (1+old_child_number));
        int i;
        for(i=0;i<old_child_number;i++){
            $$->child[i] = tmp[i];
        }
        $$->child[i] = $2;

        free(tmp);
    }

    IS_SYNTAX_ERROR = 1;
    LOG_ERROR(STR_SEMI, $2->lineno);
}
```

2.5 接口预览

NEWNODE 函数，进行节点的构造：

```
NODE* NEWNODE(int type){
        NODE* node = new NODE();
        node->type = type;
        node->child_number = 0;
        node->child = NULL;
        if(NODE_NAMES.find(type)!=NODE_NAMES.end()){

        node->name = NODE_NAMES[type];
        }
        return node;
}
```

genString 以及 setName 对于节点名字的设置:

```
string genString(char *s, int len) {
        s[len] = 0; string ss(s);
        transform(ss.begin(), ss.end(), ss.begin(), ::tolower);
    return string(ss);
}

void setName(NODE* node, int type){
        if(NODE_NAMES.find(type)!=NODE_NAMES.end()){

        node->name = NODE_NAMES[type];
        }
}
```

Node_init 函数对节点的 TK 以及相应的字符串映射(只显示部份):

```
void node_init(){

        NODE_NAMES[TK_PROGRAM] = "TK_PROGRAM";
        NODE_NAMES[TK_PROGRAM_HEAD] = "TK_PROGRAM_HEAD";
        NODE_NAMES[TK_ROUTINE] = "TK_ROUTINE";
        NODE_NAMES[TK_ROUTINE_HEAD] = "TK_ROUTINE_HEAD";
        NODE_NAMES[TK_CONST_PART] = "TK_CONST_PART";
        NODE_NAMES[TK_CONST_DL] = "TK_CONST_DL";
        NODE_NAMES[TK_CONST_DL_END] = "TK_CONST_DL_END";
        NODE_NAMES[TK_CONST_MINUS] = "TK_CONST_MINUS";
        NODE_NAMES[TK_TYPE_PART] = "TK_TYPE_PART";
        NODE_NAMES[TK_TYPE_DL] = "TK_TYPE_DL";
        NODE_NAMES[TK_TYPE_DL_END] = "TK_TYPE_DL_END";
        NODE_NAMES[TK_TYPE_DEF] = "TK_TYPE_DEF";
        NODE_NAMES[TK_TYPE_DECL_SIM] = "TK_TYPE_DECL_SIM";
        NODE_NAMES[TK_TYPE_DECL_ARR] = "TK_TYPE_DECL_ARR";
        NODE_NAMES[TK_TYPE_DECL_REC] = "TK_TYPE_DECL_REC";
        NODE_NAMES[TK_ARRAY] = "TK_ARRAY";
        NODE_NAMES[TK_RECORD] = "TK_RECORD";
        NODE_NAMES[TK_FIELD_DL] = "TK_FIELD_DL";
        NODE_NAMES[TK_FIELD_DL_END] = "TK_FIELD_DL_END";
        NODE_NAMES[TK_FIELD_DECL] = "TK_FIELD_DECL";
        NODE_NAMES[TK_NL] = "TK_NL";
        NODE_NAMES[TK_NL_END] = "TK_NL_END";
        NODE_NAMES[TK_STD_SYS_TYPE] = "TK_STD_SYS_TYPE";
        NODE_NAMES[TK_STD_ID] = "TK_STD_ID";
```

3. 符号表

在语义分析与符号表构建阶段，所做的工作主要有三点:

(1)检查每个表达式的类型，并在合适的时候自动提升类型

(2)对于每个作用域（对于 Pascal 而言是函数以及子函数）构建一张符号表，将符号（常量/

类型定义/变量/函数/过程等）插入到其对应作用域的符号表中。对于符号的引用，给出一个对应符号表的链接。

(3)在类型不匹配或符号引用缺失的情况下提示错误，并中止后续任务（因为不可能再生成合法的中间代码）。

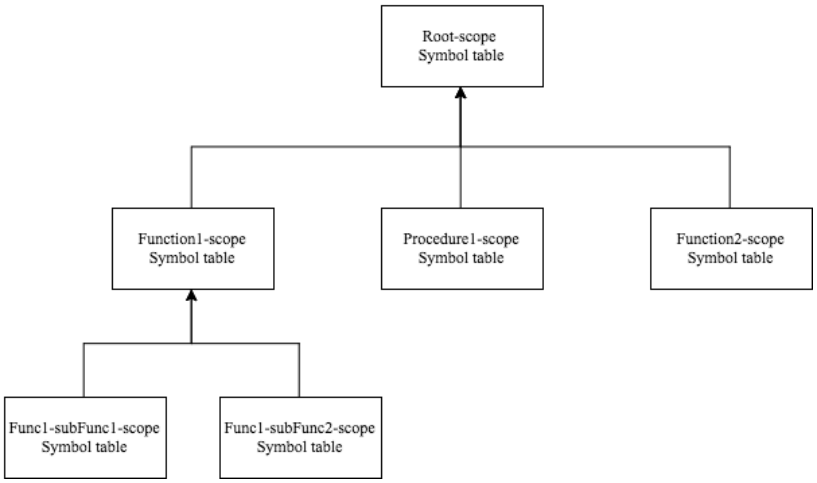
为简单起见，我们的 Pascal 编译器使用的是静态作用域的设计。

语义分析主要基于符号表，因此下面给出符号表与符号表所使用的类型系统的设计:

我们定义了 SymbolTable 作为符号表的类，其中包含若干个 unordered_map（hashmap），给出符号名即可在常数时间找到对应的符号，相比于手动实现的 hash 表更加方便快捷。

```
class SymbolTable {
public:
    SymbolTable* nextSymbolTable;
    unordered_map<string, Value> constSymbolTable;
    unordered_map<string, Type> varSymbolTable;
    vector<string> varSequence;
    vector<string> paraSequence;
    unordered_map<string, Type> typeSymbolTable;
    unordered_map<string, vector<Type>> funcSymbolTable;
    // the following data structures are used to check goto labels' validity
    unordered_map<int, int> labelRef;
    unordered_map<int, NODE*> labelMap;
    // for DEBUG
    int enumCount;
    string name;
```

我们采用了每个作用域一张符号表的做法，使用一个 list<SymbolTable>将其串接起来，每次使用 list.front()获取到当前作用域的符号表，使用 list.pop()在离开作用域时弹出当前符号表，符号表本身也有一个 nextSymbolTable，指向它上一级作用域的符号表，这一过程可以在 list.push 时向 SymbolTable 的构造函数传入 list.front()来完成。不难看出，整个 SymbolTable 依托于 AST 形成一个树状结构，每个 SymbolTable 保留指向父亲的指针，如图 1 所示。这样的做法可以保证在知道当前作用域的符号表后可以访问到当前作用域所能访问到的所有符号，而“屏蔽”其不应该访问到的符号。



SymbolTable 类中包含若干子表，其中包括 constSymbolTable, varSymbolTable, typeSymbolTable, funcSymbolTable。constSymbolTable 所存放的是利用 const 关键字所声明的常量符号，在引用了对应的标识符时（例如数组声明时利用了之前定义的常量标识符）需要查找这张表，由于常量在编译期即可确定，这部分存放的是具体的值，所以constSymbolTable的类型是string->Value。我们使用Value类将所有简单类型的值统一装箱，在需要使用时再进行拆箱，Value 类定义如下:

```
class Value {
public:
        bool operator < (const Value &b) const;
        bool invalid;
        SimpleTypeEnum type;
        int ival;
        double dval;
        bool bval;
        char cval;
        string sval;
```

其中利用 type 指明了 Value 的具体类型，ival，dval，bval，cval，sval 分别存放 integer，real，boolean，char，string 类型的数据。

在介绍后面的符号表之前，我们需要先介绍 Type 类，Type 类是我们所定义的类型系统，在经过合适的设计之后，一个 Type 类的变量即可表达任意本编译器可处理的类型。

```
class Type {
public:
    bool operator <(const Type &o) const;
    bool operator ==(const Type &o) const;
        int size();

        bool null;
        bool isSimpleType;
        SimpleType *simpleType;
        ComplexType *complexType;
```

Type 类实际上是对 SimpleType 类与 ComplexType 类的简单封装，分别代表系统中的标准类型与构造类型，由于 ComplexType 类中某些部分又引用了 Type 类型，为了避免无限循环引用，我将 SimpleType 与 complexType 类定义为指针类型。下面介绍 SimpleType 与 ComplexType 类。

```
class SimpleType {

public:
        SimpleTypeEnum simpleType;
        IntType intType;
        RealType realType;
        int size();

    SimpleType(){}
    SimpleType(const string &x);
};
```

SimpleType 类中包括了所有的系统简单类型，包括 integer，real，boolean，char，string 五种，利用 simpleType 指示出具体是哪一种类型，当是 int 或者 real 时，再通过 intType，realType 来细分。

```
class ComplexType {
public:
        ComplexTypeEnum complexType;
        EnumType enumType;
        RecordType recordType;
        ArrayType arrayType;
        RangeType rangeType;
        FPType fpType;
    int size();
```

ComplexType 类包含了一些较为复杂的类型，例如枚举类型，record 类型，数组类型，range 类型，函数/过程类型等。

```
class EnumType {
        // NOTE: one enum item cannot be in different enumType
public:
        EnumType(){}
        EnumType(const vector<string> &x): enumList(x.begin(), x.end()){}
        set<string> enumList;
};
```

EnumType 类包含了该枚举类型所有的标识符，需要注意的是，将这些值存下来是基于最大化保留信息量的考虑。实际上除了语义分析阶段（检查当时定义的枚举类型中是否有重复的标识符）以外这个类型并不会被用到，因为我们实际对枚举类型的实现是为其分配连续的常量值，将标识符插入到常量表中来实现的。为了不排除进一步改进的可能性，还是将所有的信息都保留下来了。

```
class RecordType {
public:
        // NOTE: here we assume string in different unordered_map should have the same order
        unordered_map<string, Type> attrType;
        unordered_map<string, int> offset;
```

RecordType 类实现了 record 类型的定义，利用一个 unordered_map<string, Type>，来记录

每个域的类型，注意到由于这里递归的使用了 Type 类型，我们的类型系统是支持多重嵌套的 record 类型定义的，虽然实际上由于文法的限制我们的编译器实际只能支持一层 record 访问。

```
class ArrayType {
public:
        ArrayType(){}
        ArrayType(int _start, int _end, const Type &_elementType);
        int start, end, elementSize;
        Type elementType;
};
```

ArrayType 类通过记录开始，结束的下标以及对应元素的类型来定义一个数组，类型系统支持定义一个元素类型为复杂类型的数组，但受限于文法，我们实际上不支持数组元素为 record 类型下 record 类型的域的访问。对于 a[i].b 这种写法在语法分析阶段就会报错。

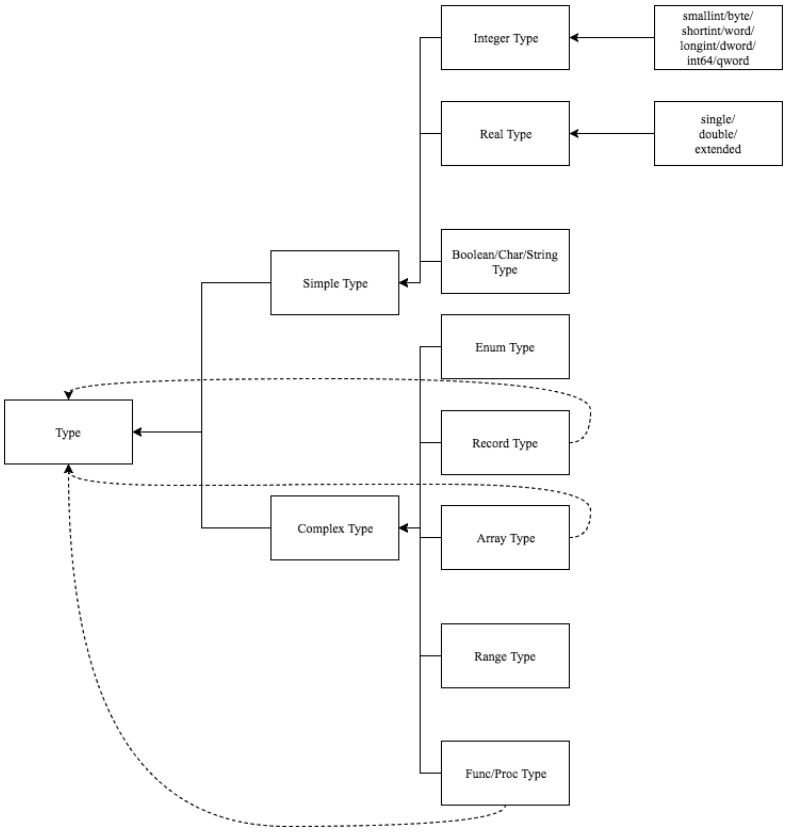
```
class RangeType {
        // NOTE: we assume we can determine the value in semantic analysis phase
public:
        SimpleTypeEnum rangeType;
        Value start, end;
```

RangeType 记录了 start，end 的值，在我们的编译器设计中该类型实际上只用于进行数组声明，因此目前的实现中后续阶段也不会用到。

```
class FPType {
public:
        vector<Type> argTypeList;
    vector<bool> argVarList;
        Type retType;
```

FPType 通过记录参数和返回值的类型定义一个函数和过程类型，其中 argVarList 记录的是对应位置的参数是否要求为左值，当 retType 为一个空 Type 时 FPType 表示一个过程，否则为一个函数。

Type 定义的示意图如下:



以上就是我们的类型系统的设计。下面让我们回到符号表本身的介绍。

typeSymbolTable 中存放的是利用 type 关键字所声明的类型，在变量声明部分如果使用了非预定义的标识符就需要查找此表，显然此表的类型应是 string->Type，因为 type 关键字所声明的类型一定能用一个 Type 变量来表达。只需要查找这张表就可以知道标识符所对应的类型。

varSymbolTable 中存放的是当前作用域中声明的变量的类型，用于类型检查以及后续的代码生成。

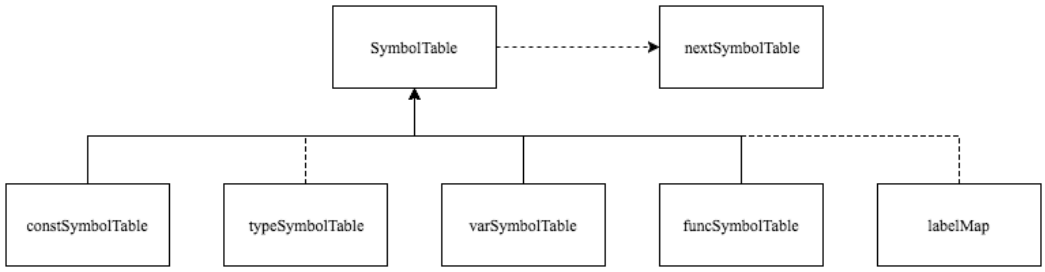
funcSymbolTable 中记录了当前作用域声明的函数与过程，用于函数调用的时候查找对应的符

号。由于需要支持函数的重载，此表的类型是 string->vector<Type>，因为一个标识符可以对应多个函数，我们需要进一步根据参数的类型确定具体是哪一个函数，注意此处并不考虑返回值类型，即不能同时插入两个名称一样，参数类型一样，但返回值不一样的函数/过程。

由于本编译器的文法不支持 goto Label 的统一声明，而是使用了 C-style 的方式。因此在语义分析时有可能发生引用了尚未“看到”的 label。因此我们需要使用两个 unordered_map 来记录 goto Label 所在的<label number,对应语句的 AST 节点>和引用 goto Label 所在的<label Number，对应语句的行号（方便生成错误信息）>，在离开一个作用域时统一进行检查，是否存在引用了无效 label 的情况。

varSequence/paraSequence 这两个 vector 主要方便后续代码生成，记录了变量/参数的声明/传递顺序，以便于计算栈上的 offset，与本阶段关系不大。

符号表的大致组成部分由下图所示，该图隐藏了 varSequence 等非关键成员，其中 typeSymbolTable 与 labelMap 由虚线所示，表示这两部分不会被后续代码生成阶段所使用。

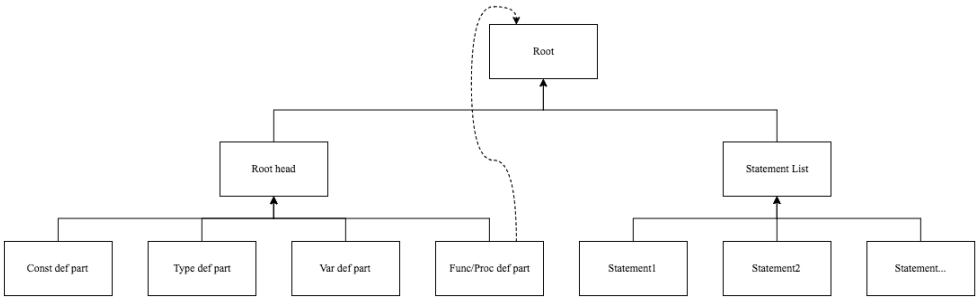


以上就是符号表的定义部分。

4. 语义分析

AST：抽象语法树

得益于 Pascal 的良好结构，每一部分有其对应的特点，在 Pascal 上进行语义分析是一件相对比较好完成的工作。根据其 AST 生成规则，我们可以将其大致分为如下图所示的几部分：



我们需要做的，就是定义一个 RoutineAnalysis 函数，分别解析其 const part/type part/var

part/func|proc part 还有 statement list 部分。对于 func|proc part 内部的解析，递归调用 RoutineAnalysis 函数即可，因为两者结构是类似的。

让我们按照上图的顺序一步步来说明:

semanticAnalysis 函数作为整个 Semantic Analysis 阶段的主函数，这里需要对最开始的根符号表做一些初始化的工作，例如，处理一些原生函数（由于 MIPS 模拟器功能所限，并不支持 Pascal 的诸多原生函数如 chr, ord 等，而这些函数的实现非本次大程的重点，所以我们只支持 read/readln, write/writeln 两组 IO 函数，其中 read 在符号表中插入对应元素，write 进行特殊判断（因为 write 支持任意多参数）来通过语义分析的检查阶段)。在准备好最开始的根符号表后调用 routineAnalysis 函数，该函数主要包含以下部分:

const Analysis:

首先是 constAnalysis 部分，该部分假定常量声明一定是以 a = b 的形式一条条给出，其中 a 是标识符，b 已经作为 Value 的实例被存放到 AST 的节点中，因此只需要从 AST 中取出 a 对应的字符串与 b 对应的 Value，将其插入到当前符号表即可。在插入时，需要进行常量标识符与变量标识符重名的判断，虽然此时还未进入当前作用于的变量定义阶段，但考虑到如果此时在函数内部，就存在虽然还未定义变量，但已经存在变量符号的情况，那就是函数的参数。

type Analysis:

typeAnalysis 与 constAnalysis 类似，类型定义也是以 a = b 的形式一条条给出，其中 a 是标识符，b 是一个类型，可以是简单类型也可以是复杂的数组或者 record 类型定义，因此我们需要使用一个 parseType 函数来解析出 b 的类型，最后将其插入当前符号表的 typeSymbolTable。此处我们需要考虑 a 有可能与当前符号表中的常量重名，因此在插入时需要查找当前的 constSymbolTable 中是否有同名常量。

Var Analysis:

varAnalysis 与前面类似，区别在于变量定义是以 namelist: type 的形式给出，我们调用 parseType 解析出 type 的具体类型后，遍历 namelist 将其插入 varSymbolTable，此处需要判断 namelist 中的标识符是否与当前作用域的 const 与 type 重名。

Function/Procedure Analysis:

如果存在函数/过程的定义，我们首先需要解析函数/过程头部，取得函数名，参数类型与返回类型（如果有的话），插入一个 FPType 到当前作用域的 funcSymbolTable 中，然后，我们新建一张符号表，插入到 symbolTableList 的头部作为当前符号表，将函数的参数插入到新的符号

表中的 varSymbolTable，如果是函数，还需插入一个与函数同名的变量。然后递归调用 routineAnalysis 函数进行分析，分析完成后，弹出 symbolTableList 的头部，恢复到原来的作用域。

Statement Analysis:

前面提到的都是定义部分，主要完成了符号表的构建工作，接下来的部分是对程序主体进行分析，基于之前定义的符号表进行符号和类型检查。

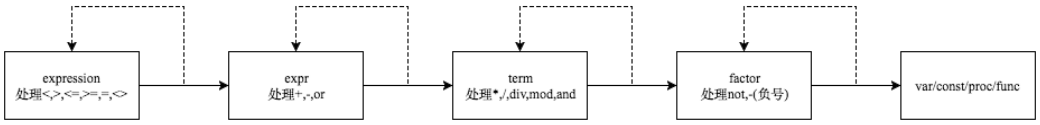
程序主体是一个包含了多个 Statement 的 Statement List，我们遍历这个 Statement List，一句句的对 Statement 作分析。

对于 Statement 的分析类似于递归下降，在去掉了 goto Label 后，statement 可分为如下几部分:赋值，函数调用，条件，循环，case，goto 语句。

赋值部分:

我们认为以下三种值为左值，可以进行赋值，以 identifier 形式表示的变量（类型为简单类型或者 record 及 enum 类型），以 identifer[expression]形式表示的数组元素（类型同上），以 identifier.identifier 形式表示的 record 成员（类型同上），如前文所述，受限于文法，我们不支持 record 的多层访问。

对于赋值号的右边，我们调用 expressionAnalysis 解析出其类型，expressionAnalysis 是一个递归下降的解析过程，其结构如下图所示:



其中每一层处理按照优先级处理其对应的符号，并完成类型的定义以及自动转换（如 expression，如果存在<,>,<=,>=,=,<>等符号，首先判断符号两边类型是否一致，如果一致，设置当前节点的类型为 boolean，否则报错）。

函数调用:

由于函数重载的原因，我们需要获取到所有参数的类型，然后在符号表中进行查找。这样一来对参数的匹配过程已经由符号表自动完成，如果符号表查找失败，则提示函数未定义。为简单起见，我们可以自动对传入参数进行 upcast 并且在有多个函数可以匹配成功时认为第一个匹配到的函数即为程序试图调用的函数而不是报错。特别需要注意的地方在于，如果函数声明时使用了 var 关键字，或者类似 read 等函数，函数的参数将会要求是一个左值，这种情况需要额外进行判断。我是用了一个 checklvalue 函数。

条件与循环:

If/repeat/while 的部分都很简单，分析条件部分的 Type 是否为 boolean，然后再调用 statementAnalysis 分析条件/循环的主体部分即可。稍复杂一点的是 for 循环，需要判断循环所使用的标识符代表一个变量以及变量类型为整数，同时判断起始值与终止值是 integer 类型（注意，枚举类型在我们的实现中也为 integer 类型）。然后再进行循环主体的分析。

Case 语句:

只需匹配好 case 中 expression 与分支里常量的类型即可。

Goto 语句:

需要将引用到的 label 放入当前的符号表中，在退出当前作用域时进行 label 合法性的检查。

错误提示信息:

在语义分析阶段，我们有可能提示如下错误:

➢Const Analysis/Type Analysis/Var Analysis 阶段:

Duplicate identifier: 重复的标识符

Undefined type: 未定义的类型

Invalid enum item identifier: 非法的枚举标识符（该标识符已被类型/常量/变量所使用）

Duplicate enum item: 重复的枚举标识符

Range data type mismatch: range 类型左右两边类型不一致

Illegal range: 非法的范围（左边值大于右边值）

Supported range data type: 不支持的范围类型（左右两边的值不能为 real 等类型）

Undefined const identifier: 未定义的常量标识符

Duplicate field name: 重复的 record 域名称

Duplicate function definition: 重复的函数声明

Duplicate procedure definition: 重复的过程声明

Duplicate parameter name: 重复的参数名

➢Routine Analysis 阶段:

Duplicate label: 重复的 goto label

Undefined variable: 未定义的变量引用

Unsupport assignment operator:变量不支持赋值语句

Type mismatch between assignment operator: 赋值号两边类型不一致

Cannot downcast data type automatically: 无法自动向下转换数据类型

Xx is not an array: xx 不是一个数组

Index must be integer: 数组的下标必须为整数

Xx is not a record: xx 不是一个 record

Invalid attribute xx: record 不存在域 xx

Undefined function or precedure: 未定义的函数或过程

xx needs a lvalue: xx 需要一个左值作为参数

the type of condition clause mys be boolean: 条件部分的类型需要是 boolean

xx must be a variable: xx 必须是一个变量

xx musb be integer: xx 必须是一个整型变量

for loop require integer type: for 循环的边界需要是整数类型

xx must be integer or char: xx 必须是一个整型或者字符变量

label type mismatch: case 语句的 label 类型不匹配

type mismatch for cmp operator: 比较操作符两边类型不匹配

type mismatch for + and -: +/-操作符两边类型不匹配

type mismatch for or: or 操作符两边类型不匹配

type mismatch for *: *操作符两边类型不匹配

type mismatch for /: /操作符两边类型不匹配

type mismatch for div: div 操作符两边类型不匹配

type mismatch for mod: mod 操作符两边类型不匹配

type mismatch for and: and 操作符两边类型不匹配

type mismatch for not: not 操作符两边类型不匹配

type mismatch for -: -（负号）操作符两边类型不匹配

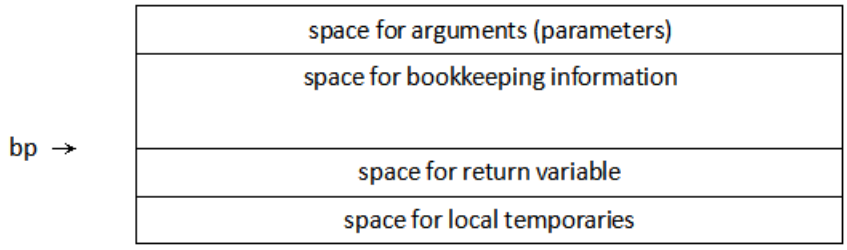
undefined variable or const or function: 未定义的常量或变量或函数

undefined function: 未定义的函数

use a invalid label: 引用了非法的 goto label

5. 运行环境设计

5.1 基于栈的运行时环境



//注:bp 即为函数体内的 control link。

5.2 参数传递机制

(1) 参数传递顺序

以从左到右的顺序计算参数值并传递。

(2) 值传递

在我们的编译器中，我们实现了值传递。例子如下。

Pascal 代码:

```
program test;
var a:integer;
function f(a:integer):integer;
begin
    f:=a*a;
end;
begin
    read(a);
    writeln(f(a));
end.
```

在上面的 Pascal 代码中，程序会将 a 的值传递进 f 函数中，并计算 a 的平方的值，然后返回输出。

(3) 引用传递

在我们的编译器中，我们实现了值传递。例子如下。

Pascal 代码:

```
program test;
var a:integer;
function f(var a:integer):integer;
begin
    a:=10;
end;
begin
    f(a);
    writeln(a);
end.
```

在上面的 Pascal 代码中，程序会将 a 的引用传递进 f 函数中，并将 a 的引用赋值为 10，然后返回。

在我们的编译器中，引用是使用指针的方式实现的。

6. 中间代码(三地址码)生成

6.1 三地址码模块功能

输入	输出
带语法表信息的语法树	三地址码

6.2 三地址码语法规定

(1)临时变量和常量

A.所有计算用到的临时变量(中间变量)都以 t 开头，后面跟一个数字。

B.临时(中间)变量定义:

- 指针临时变量定义:point {type} t0，这里{type}可以是 int, double 等类型
- 普通临时变量定义:var {type} t0，这里{type}可以是 int, double 等类型

C.所有字符串常量，如”Hello World”，都会赋给一个以 s 开头且后面跟一个数字的变量，以方便操作。

D.对于整数常量、字符常量、浮点数常量，我们采用直接输出的方式。对于布尔型的常量，遇到 true 时输出 1，遇到 false 时输出 0。

E.临时变量(以 t 开头的变量)支持的类型：

```
var int, var double, var char
point int, point double, point char
```

F.boolean 类型的两个常量 true, false 会直接被翻译为 int 型的 1, 0。

(2)特殊操作符

A.在一个变量前加上”*”号，表示取这个指针变量所指向空间中所存储的值。

比如:”a = *t0 + 1”表示将 t0 所指向空间的变量加上 1 后赋给变量 a。

而”*t0 = 1”则表示将 1 的值保存到 t0 所指向的内存空间。

B.在一个全局变量前加上”&”号，表示取这个全局变量的地址值。

比如:“t0 = &a”表示把 a 的地址赋给临时变量 t0。

(3)Pascal 源代码中的变量处理

A.在输出时，所有全局变量名和函数名前都加上一个下划线”_”，以避免和临时变量、字符串常量重名。

B.局部变量直接以 bp-{integer}的形式赋给一个临时变量，这里{integer}是一个整数。

假设该临时变量为 t0，那么 t0 就是该局部变量的指针。在需要访问局部变量或需要对它进行赋值的时候，我们就可以用*t0 的形式进行访问或对它进行赋值。我们通过这样的方式，实现了前端和后端的分离，简化了汇编代码生成部分的工作。

访问实例:

```
point int t0
t0 = bp - 4
*t0 = 4
```

C.函数参数直接以 bp+{integer}的形式赋给一个临时变量，这里{integer}是一个整数。假设该临时变量为 t0，那么 t0 就是该函数参数的指针。在需要访问函数参数或需要对它进行赋值的时候，我们就可以用*t0 的形式进行访问或对它进行赋值。我们通过这样的方式，实现了前端和后端的分离，简化了汇编代码生成部分的工作。

访问实例:

```
point int t0
t0 = bp + 4
*t0 = 4
```

D.特殊变量

- 特殊变量 bp 即为 Control Link。
- 特殊变量 sp 即为栈顶指针。

(4)函数相关

A.函数入口以 entry {函数名}的形式给出。如”entry _f”。比较特殊的是，为了配合汇编代码生成那边的工作，对于 pascal 中的入口函数，在函数体前会加上”entry main”。

B.函数结束时，输出”ret”。

C.传递函数参数：以”arg {变量/常量}”的形式给出。如”arg 4”、”arg t0”。

D.调用函数:”call {函数名}”。如”call _f”。

E.栈顶指针 sp 的相关规定

- 在进入函数时，三地址码会输出”sp = sp - {integer}”这样的语句，其中{integer}是局部变量和返回值所占空间的大小，在三地址码中会直接以整数的形式给出。这样做是为了使符号表和汇编代码生成分离，从而实现前端和后端的分离，简化汇编代码生成工作。
- 在退出函数时，三地址码会输出”sp = sp + {integer}”这样的语句，其中{integer}是

局部变量和返回值所占空间的大小，在三地址码中会直接以整数的形式给出。这样做是为了使符号表和汇编代码生成分离，从而实现前端和后端的分离，简化汇编代码生成工作。

F.函数返回值

在我们的运行环境中规定，函数的返回值和其他局部变量一同放在栈上。在调用完一个函数后，若是要取得函数的返回值，我们可以直接索引到栈上保存临时变量的地方，并新开辟一部分栈空间以生成一个和函数返回值类型相同的变量，然后将返回值复刻给新生成的变量。

(6)运算符

原 pascal 代码与三地址码符号对应关系:

Pascal 代码	三地址码	含义
+	+	加法
-	-	减法
-	-	负号
*	*	乘法
DIV	DIV	整数除
/	/	浮点除
MOD	%	模除取余
AND	&	位与
OR	|	位或
NOT	~	逻辑非
NOT	~	整数取非
>=	>=	大于等于
>	>	大于
<=	<=	小于等于
<	<	小于
=	==	等于
<>	!=	不等于
:=	=	赋值

(7)逻辑语句

➢if a then goto L1

此语句表示若 a 值为真(即 a 不等于 0)，则跳到 label L1 处。

➢if a relop b then goto L1

此语句表示若 a relop b 值为真(即 a relop b 不等于 0)，则跳到 label L1 处。这里 relop 表示关系运算符，可以是”>=”，“>”，“<=”，“<”，“==”，“!=”。

➢if_false a goto L1

此语句表示若 a 为假(即 a 等于 0)，则跳到 label L1 处。

(8)跳转命令

A.“goto L1”表示程序跳转到 label L1 处继续执行。

B.“label L1”表示定义 L1 这个 label。

(9)其他规定

A.在输出时，为方便汇编代码生成，所有的单变量和操作符，两两之间都要用空格隔开。

6.3 逻辑语句的三地址码翻译

(1)数组

Pascal 代码	三地址码
t2 = a[t1]	t3 = t1 - {start_position_of(a)}; t3 = t3 * {elem_size(a)}; t3 = &a + t3 t2 = *t3

上面的 start_position_of(a)和{elem_size(a)}在三地址码中都会直接以一个整数的形式给出。直接输出整数而不是 start_position_of(a)、{elem_size(a)}，实现了前端和后端的分离，简化了汇编代码生成部分的工作。

(2) 结构

Pascal 代码	三地址码
x.j = x.i	t1 = &x + {field_offset (x,j)} t2 = &x + {field_offset (x,i)} *t1 = *t2

上面的{field_offset (x,j)}和{field_offset (x,i)}在三地址码中都会直接以一个整数的形式给出。直接输出整数而不是 start_position_of(a)、{elem_size(a)}，实现了前端和后端的分离，简化了汇编代码生成部分的工作。

(3) if 语句

Pascal 代码	三地址码

if E then
S1
else
S2
	<code to evaluate E to t1>
	if_false t1 goto L1
	<code for S1>
	goto L2
	label L1
	<code for S2>
	label L2

(4)while 语句

Pascal 代码	三地址码

while E do
begin
S
end
	label L1
	<code to evaluate E to t1>
	if_false t1 goto L2
	<code for S>
	goto L1
	label L2

(5)repeat 语句

Pascal 代码	三地址码
repeat S until E;	label L1 <code for S> a = result<code for expression> if_false a goto L1

(6)for 语句

Pascal 代码	三地址码

for x := A direction B do
stmt
	a = <code for calculate A>
	b = <code for calculate B>
	x = a;
	<direction == “to” ? d=1 : d=-1>
	while (x!=b) {
	stmt;
	x=x+d;
	}
	while (x==b) {
	stmt;
	x=x+d;
	}

上面的三地址码最后之所以要写成 while (x==b) {}的形式，是因为程序在运行过程中 x 变量加到某个值后有可能出现饱和的现象，即 x=x+1 后，x 变量的值不发生变换。这在基本类型中或许不会发生，不过在对于我们自己定义的结构是有可能发生的。

(7)Case 语句

case 语句使用跳转表实现。由于在某个特定时刻，我们只能得到语法树的某一个节点，而不能得到 case 语句相关的所有节点，所有我们需要**使用自己定义的结构来维护跳转表**。具体的维护方法就是先把所有底层的 case 节点跑一遍，将它们的 label 和条件记录下来，然后再在 case 语法最上层的语法树节点进行输出。

A.用于维护跳转表的结构定义

```
struct CaseExpr {
        piv a;
        int label;
        CaseExpr() {}
        CaseExpr(piv a, int label):a(a), label(label) {}
};
struct CaseParse {
        static vector<vector<CaseExpr> > scases;
        static int endLabel;

        static void setEndLabel(int a) { endLabel = a; }
        static int getEndLabel() { return endLabel; }

        static void addNew() { scases.push_back(vector<CaseExpr>()); }
        static vector<string> pop(piv E) {
                if (scases.size() == 0) throw Error("Unexpected end of 'switch..case'");
                vector<string> ret;
                vector<CaseExpr> &cases = *scases.rbegin();
                for (int i=0; i<cases.size(); i++)
                        ret.push_back("if " + getName(E) + " == " + getName(cases[i].a) + " then goto L" +
toString(cases[i].label));
                scases.pop_back();
                return ret;
        }

        static void addCase(piv a, int label) {
                if (scases.size() == 0) throw Error("Unexpected case of 'switch..case'");
                scases[scases.size() - 1].push_back(CaseExpr(a, label));
        }
};
vector<vector<CaseExpr> > CaseParse::scases;
int CaseParse::endLabel;
```

B.具体维护方法

➢在遇到一个新的 case 时，使用 CaseParse::addNew()函数在 scases 列表中增加一个空的 vector<CaseExpr>()类型变量，这里 CaseExpr 是用于记录 case 语句中每个条件的值以及相应语句起始处的 label。由于可能出现 case 语句套 case 语句的情况，所以我们这里的 scases 使用 vector<vector<CaseExpr> >类型来模拟一个栈，以解决 case 语句套 case 语句的情况。

➢对于 case 每个条件下的代码，生成相应的三地址码和 label，并记录条件的值和 label。

➢ 在将 case 语句中所有子条件下的代码对应的三地址码输出之后，使用在 scases 变量中记录的子条件值和相应语句起始处的 label，输出 case 语句的跳转表。

C. case 语句与对应的三地址码

Pascal 代码	三地址码
case E of a1: S1; a2: S2; ... end;	x = <code for calculate E> goto caseL1(CaseTableStart) Label L1 <code for S1> goto caseL2 (CaseEnd) Label L2 <code for S2>

	goto caseL2 (CaseEnd) ... label caseL1 (CaseTableStart) {if E==a1 then goto L1} {if E==a2 then goto L2} ... label caseL2(CaseEnd)

6.4 其他三地址码处理

(1)临时变量回收机制

三地址码的临时变量，当不再使用后，是可以被回收再利用的。通过这个机制，我们可以避免临时变量后的整数无限增大。(注：临时变量以 t 开头，后面跟一个整数，如”t0”、”t5”)

(2)label 维护

A.为实现逻辑语句所产生的 label，以”L”后面跟一个整数的形式输出。

B.Pascal 语言中的 label 我们使用老师文法里的定义，是一个整数。在翻译成三地址码时，我们会在这个整数前加上”LUSER”，如”LUSER1”、”LUSER2”，来达到和我们为实现逻辑语句时产生的 label 区分的目的。

(3)特殊函数处理

由于 read 和 writeln 需要和系统打交道，需要用到 MIPS 自带的函数 read 和 println，和普通的函数不同，所以需要特殊处理。

A. Read

Pascal 代码	三地址码
read(x);	point int t0 t0 = bp - {offset(x)} read t0

B. Writeln

Pascal 代码	三地址码
writeln(x);	point int t0 t0 = bp - {offset(x)} #此处 offset(x)

	为数字 var int t1 t1 = *t0 println t1

(4) 三地址报错机制

由于前面词法、语法、语义部分都已经做了错误分析，所以三地址码部分不再冗余地对 pascal 代码语义进行纠错，而只是对一些在编写程序时可能出现的 bug 进行报错，以提高 bug 修复效率。

报错方式:因为我们是使用 C++语言来实现编译器的，所以我使用 C++的 throw、try、catch 机制来实现遇到错误停止程序以及输出错误信息。

报错举例:当语法树出现不应该出现的空指针时报错。

6.5 三地址码优化

(1)常数计算优化

在将语法树翻译成三地址码时，对于常数加减这样的语句，我们显然是可以进行优化的。

A.优化方法

自己维护一个用于常量计算的结构，并重载所有可能出现的操作符。

已经支持的在 pascal 语法中的计算运算符有：+、-、-(负号)、*、DIV、/、MOD、AND、OR、NOT(逻辑非)、NOT(位非)、>=、>、<=、<、=、<>。

已经支持的常数计算类型有：integer, char, double, string。

这些不同类型的常数和不同运算符的组合约有 220 种左右，都是支持的。

7. 代码生成

7.1 三地址码的格式

为了代码优化的方便，和生成汇编代码的方便，三地址址有些和课件上不太一样。

7.1.1 赋值

格式	含义	实例

Ptr = Ptr2	直接让 Ptr1 和 Ptr2 指向同一地址	t0 = bp
*Ptr1 = *Ptr2	让 Ptr1 的值等于 Ptr2 的值	*t0 = *t1
*Ptr1 = 123	让 Ptr1 的值等于立即数	*t0 = 100

7.1.2 运算

//注:有的格式两个源操作数交换一下又是一个新的格式，不再赘述。

格式	含义	实例
Ptr1 = Ptr2 + Imm	Ptr1 是一个指针，地址是 Ptr2 增加	t0 = sp + 4
Ptr1 = Ptr2 - Imm	Ptr1 是一个指针，地址是 Ptr2 增加	t0 = sp - 8
*Ptr1 = *Ptr2 + Imm	和是指针的值，加数是指针的值和一个立即数	*t0 = *t1 + 1
*Ptr1 = *Ptr2 - Imm	差是指针的值，减数是指针的值和一个立即数	*t0 = *t1 - 1
*Ptr1 = *Ptr2 * Imm	积是指针的值，乘数数是指针的值和一个立即数	*t0 = *t1 * 1
*Ptr1 = *Ptr2 / Imm	商是指针的值，被除数和除数是指针的值和一个立即数	*t0 = *t1 / 1
*Ptr1 = *Ptr2 + *Ptr3	和是指针的值，加数是指针的值和另一个指针的值	*t0 = *t1 + *t2
*Ptr1 = *Ptr2 - *Ptr3	差是指针的值，减数是指针的值和另一个指针的值	*t0 = *t1 - *t2
*Ptr1 = *Ptr2 * *Ptr3	积是指针的值，乘数数是指针的值另一个指针的值	*t0 = *t1 * *t2
*Ptr1 = *Ptr2 / *Ptr3	商是指针的值，被除数和除数是指针的值和另一个指针的值	*t0 = *t1 / *t2

7.1.3 比较

格式	含义	实例

*Ptr1 = *Ptr2 > Imm	判断 Ptr2 的值是否大于立即数，是的话把 0 或 1 赋给 Ptr1 的值。	*t0 = *t1 > 1
*Ptr1 = *Ptr2 < Imm	判断 Ptr2 的值是否小于立即数，是的话把 0 或 1 赋给 Ptr1 的值。	*t0 = *t1 < 1
*Ptr1 = *Ptr2 == Imm	判断 Ptr2 的值是否等于立即数，是的话把 0 或 1 赋给 Ptr1 的值。	*t0 = *t1 == 1
*Ptr1 = *Ptr2 != Imm	判断 Ptr2 的值是否不等于立即数，是的话把 0 或 1 赋给 Ptr1 的值。	*t0 = *t1 != 1
*Ptr1 = *Ptr2 > *Ptr3	判断 Ptr2 的值是否大于 Ptr3 的值，是的话把 0 或 1 赋给 Ptr1 的值。	*t0 = *t1 > *t2
*Ptr1 = *Ptr2 < *Ptr3	判断 Ptr2 的值是否小于 Ptr3 的值，是的话把 0 或 1 赋给 Ptr1 的值。	*t0 = *t1 < *t2
*Ptr1 = *Ptr2 == *Ptr3	判断 Ptr2 的值是否等于 Ptr3 的值，是的话把 0 或 1 赋给 Ptr1 的值。	*t0 = *t1 == *t2
*Ptr1 = *Ptr2 != *Ptr3	判断 Ptr2 的值是否不等于 Ptr3 的值，是的话把 0 或 1 赋给 Ptr1 的值。	*t0 = *t1 != *t2

7.1.4 跳转

格式	含义	实例
Label lb	定义一个名为 lb 的标签	label loop
Goto lb	跳转到名为 lb 的标签	Goto loop
If f_false t0 goto lb	如果 t0 的值为 0，那么就跳转到标签为 lb 的地方	If f_false t0 goto lb
If a op b then goto lb	这是为了上层方便定义的跳转指令，a 可以是立即数或者指针，b 也可以是立即数或者指针，lb 是标签	If *t1 == 2 then goto loop

7.1.5 函数调用

格式	含义	实例

Print Ptr	利用系统函数，进行输出，不换行	Print t0
Println Ptr	利用系统函数，进行输出，换行	Println t0
Read Ptr	利用系统函数，进行读入	Read t0
Arg Ptr	调用参数前，声明一位值为 t0 的参数，方便后期进行堆栈维护	Arg t0
Call func	调用名为 func 的函数	Call calc
Ret	函数跑完了，直接返回	Ret

7.2 数据结构一览

7.2.1 变量结构

变量结构用于保存三地址码已经声明了的变量，一般来说都是临时变量。三地址码上层处理的过程中，实际上下层已经没有办法太了解变量的名字，但是基础变量名字的类型和值仍然是非常有必要的。因为譬如当三地址码中两次提到 t0 的时候，它的含义可能已经发生了变化。

```
typedef struct {
    char name[200];
    union {
        char initval[200]; //初始值
        char floatval[200];
        char doubleval[200];
    };
    type type; //指向的类型
    bool isVar;
} Variable;
```

7.2.2 三地址结构

三地址码的结构跟书本上的介绍非常类似，我们先用一个结构详细说明每个地址，再用一个结构把三个地址集合起来，并指明类型。

```
typedef struct {
    //地址的类型，是立即数还是非立即数。是整形还是浮点数。
    AddrKind kind;
    //如果是立即数，要存储值。如果是非立即数，要存储名字
    union {
        int intVal;
        float floatVal;
        double doubleVal;
        char name[20];
    } contents;
} Address;

//三地址码有三个地址，op是类型
typedef struct {
    Address addr1, addr2, addr3;
    OpKind op;
} Quad;
```

7.2.3 其他结构

把变量的个数和变量本身集合起来，形成一个变量表，方便查询相关的信息。把所有的三地址码存起来做离线的处理，而不是读一句生成一句。

```
typedef struct {
    int num;
    Variable var[200];
} VariableMap;

typedef struct {
    int num;
    Quad quad[1000];
} QuadTable;
```

7.3 MIPS 规范

7.3.1 通用寄存器格式

32 个通用寄存器我们非常熟悉，在计组和体系中我们都了解过，实际上，我频繁使用的只有$t0~$t7 和$ra 和$sp。我并没有过于强求充分利用寄存器，而是求得一个代码运行效率和我写代码效率的平衡。

Register Number 寄存器编号	Alternative Name 寄存器名	Description 寄存器用途
0	zero	永远返回零
1	$at	汇编保留寄存器

2-3	$v0 - $v1	存储表达式或者是函数的返回值
4-7	$a0 - $a3	存储子程序的前 4 个参数，在子程序调用过程中释放
8-15	$t0 - $t7	临时变量，同上调用时不保存
16-23	$s0 - $s7	非临时的寄存器
24-25	$t8 - $t9	临时寄存器算是前面$0~$7 的一个继续，属性同$t0~$t7
26-27	$k0 - $k1	中断函数返回值，不可做其他用途
28	$gp	指向 64k(2^16)大小的静态数据块的中间地址块
29	$sp	（S tack P ointer 简写）栈指针，指向的是栈顶
30	$s8/$fp	帧指针
31	$ra	return a ddress 返回地址，目测也是不可做其他用途

7.3.2 浮点寄存器和特殊寄存器

Register Number 寄存器编号	Alternative Name 寄存器名	Description 寄存器用途
$f0 ~ $f31	?	永远返回零
特殊的$f10、$f12	?	
特殊的$f0	?	
特殊的偶数寄存器	?	偶数开头的连续两个寄

		存器可以存双精度浮点数
FCCR	浮点数控制寄存器	用于浮点寄存器比较时候。比较的结果 0 或者 1 会在 FCC 的某一个位置上置 1。

7.3.3 指令格式

格式	含义	实例
LW / L.S / L.D LW $rt, offset (base)	GPR[rt] = memory[GPR[base] + offset] LW L.S L.D 的区别是这个三个指令是给整形、浮点数、双精度浮点数使用的，下面同理	LW $t1, 0($t0)
SW / S.S / S.D SW $rt, offset (base)	memory[GPR[base] + offset] = GPR[rt]	SW $t1, 0($t0)
LA $rt, name	伪指令,$RT = 名字为 name 的全局量的地址	LA $t0, a
ADD / ADD.S / ADD.D ADD $rd, $rs, $rt	GPR[rd] = GPR[rs] + GPR[rt]	ADD $t0, $t1, $t2
SUB / SUB.S / SUB.D SUB $rd, $rs, $rt	GPR[rd] = GPR[rs] - GPR[rt]	SUB $t0, $t1, $t2
MUL / MUL.S / MUL.D SUB $rd, $rs, $rt	GPR[rd] = GPR[rs] - GPR[rt]	SUB $t0, $t1, $t2
DIV DIV $rs, $rt	(HI, LO) = GPR[rs] / GPR[rt] 其中 HI 为余数，LO 为商	DIV $t0, $t1
DIV.S / DIV.D DIV.S fd, fs, ft	FPR[fd] = FPR[fs] / FPR[ft]	DIV.s $f0, $f1, $f2
MTC1 MTC1 rt, fs	FPR[fs] = GPR[rt] 整形的值放到浮点寄存器里面	Mtc1 $r0, $f0

	去，但是格式不会发生转变	
CVT.S.D CVT.W.S	FPR[fd] = convert_and_round (FPR[fs])。用于 upcast，把单精度浮点数转化为双精度，或者把整形转化为浮点数。	CVT.S.D $f0, $f2
SYSCALL	调用系统函数，具体可以看下一个小姐	syscall
J lb	跳转到 lb 标签	J finish
JAL lb	I: GPR[31] = PC + 8 I+1: PC = $PC_{GPRLEN-1..28}$ || instr_index || 0^2 调用函数	JAL sum
JR $rt	返回到$ra 地址的地方	JR $ra
SLT $rd, $rs, $rt	GPR[rd] = (GPR[rs] < GPR[rt]) Rs 和 rt 比较的值放入 rd	SLT $t0, t1, $t2
BEQ $rs, $rt, lb	if GPR[rs] == GPR[rt] then branch	BEQ $t0, $t1, finish
BNE $rs, $rt, lb	if GPR[rs] != GPR[rt] then branch	BNE $t0, $t1, finish

7.3.4 系统函数

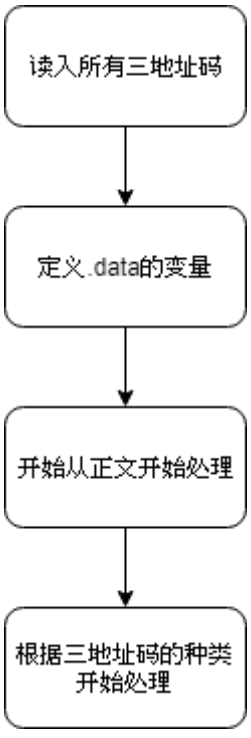
//注:跟整形有关的函数的返回值都在$v0，跟浮点数有关的函数的返回值都在$f0。

功能	调用的编号	所需参数
print_int 打印一个整型	$v0 = 1	将要打印的整型赋值给 $a0
print_float 打印一个浮点	$v0 = 2	将要打印的浮点赋值给 $f12
print_double 打印双精度	$v0 = 3	将要打印的双精度赋值给 $f12
print_string	$v0 = 4	将要打印的字符串的地址赋值给 $a0

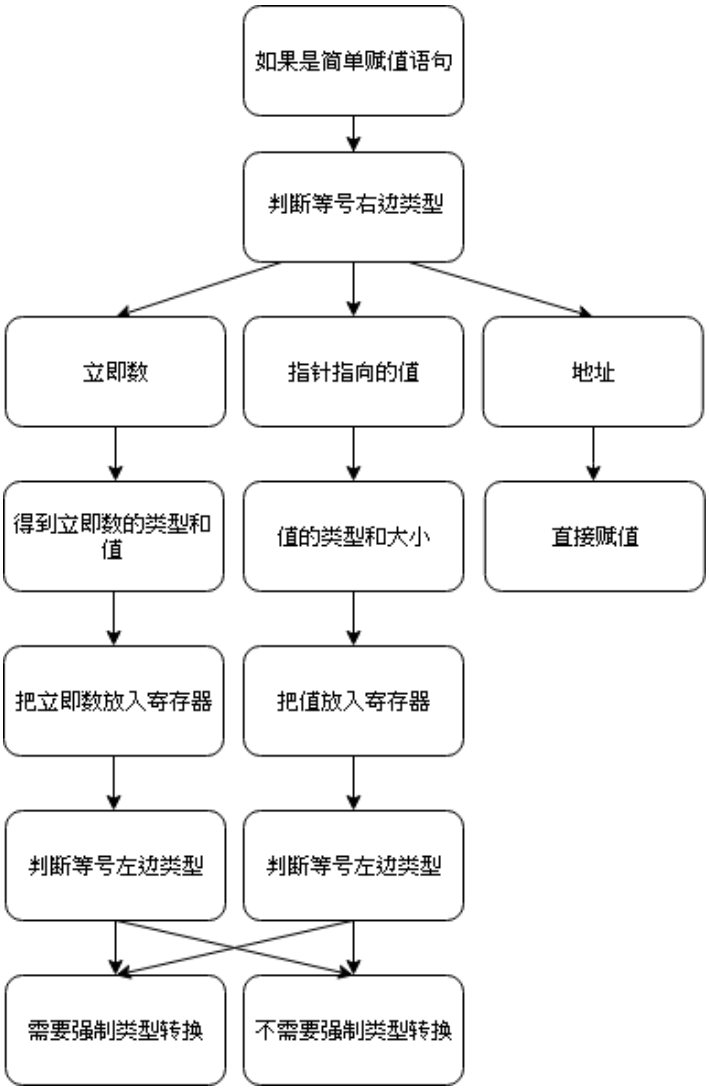
读取一个字符串		
read_int 读取一个整形	$v0 = 5	将读取的整型赋值给 $v0
read_float 读取浮点数	$v0 = 6	将读取的浮点赋值给 $v0
read_double 读取双精度	$v0 = 7	将读取的双精度赋值给 $v0
read_string 读取字符串	$v0 = 8	将读取的字符串地址赋值给 $a0 将读取的字符串长度赋值给 $a1
sbrk 分配内存空间	$v0 = 9	需要分配的空间大小
exit 退出	$v0 =10	退出了

7.4 处理流程

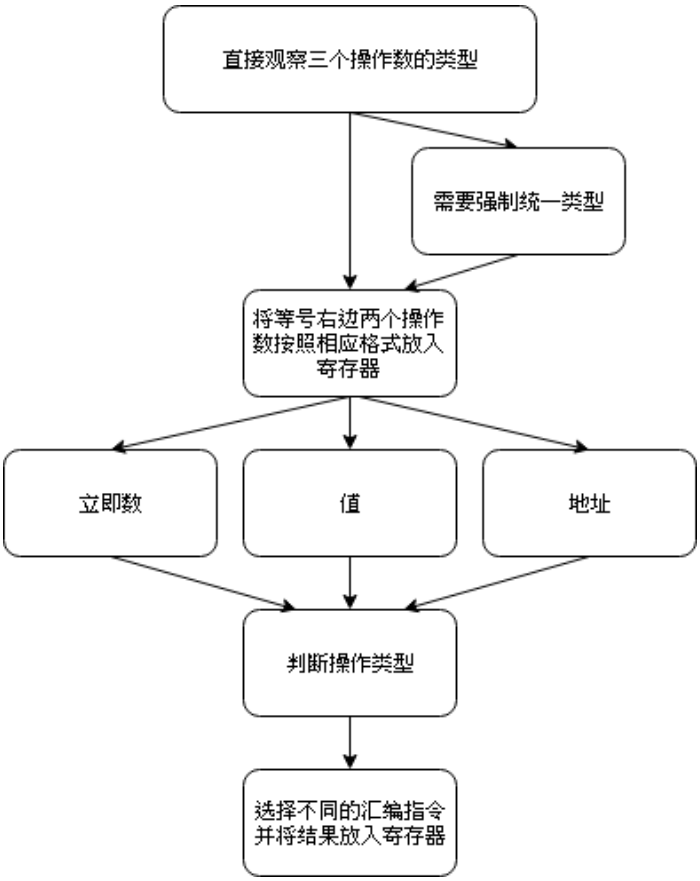
7.4.1 整体处理流程



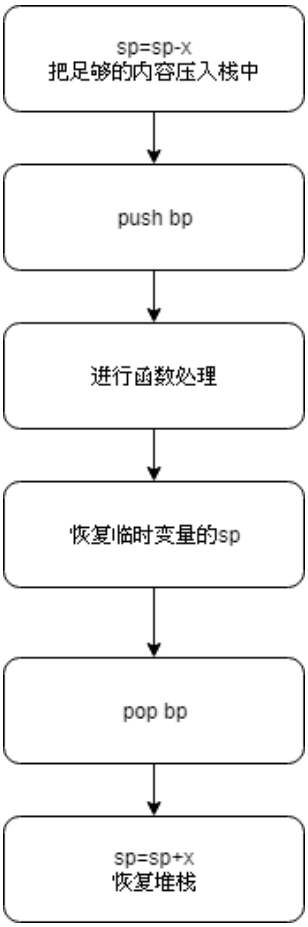
7.4.2 处理赋值



7.4.3 处理加减乘除运算


直接观察三个操作数的类型
需要强制统一类型
将等号右边两个操作数按照相应格式放入寄存器
立即数
值
地址
判断操作类型
选择不同的汇编指令并将结果放入寄存器

7.4.4 堆栈的维护


sp=sp-x
把足够的内容压入栈中
push bp
进行函数处理
恢复临时变量的sp
pop bp
sp=sp+x
恢复堆栈