

编译实验三实验报告——Parser of PCAT

田应涛 10302010029 王曦 10300240014
陈济凡 10300240076 承沐南 10300240003

June 22, 2013

Contents

1	概述	2
2	抽象语法树 (Abstract Syntax Tree)	2
3	类型检查 (Type Checking)	5
3.1	定义的类型检查	5
3.2	算数运算的类型检查	5
3.3	逻辑运算的类型检查	6
3.4	赋值的类型检查	7
3.5	函数调用的类型检查	7
4	栈帧 (Frames)	8
4.1	样例	9
5	中间表示 (Intermediate Representation)	9
5.1	暂存器传递语言	9
5.2	静态单赋值	10
5.3	注意要点	10
6	机器代码 (Machine Code)	10
6.1	数据传送	10
6.2	算术运算	10
6.3	逻辑运算	11
6.4	过程调用	11
6.5	函数进入/退出	12
7	实验结果	12
7.1	IF 语句实验	12
7.2	循环语句实验	17
7.3	命名空间实验	22
7.4	简单嵌套实验	26
7.5	复杂嵌套实验	32

1 概述

本次 Project 包含五个部分

1. 抽象语法树 (Abstract Syntax Tree)
2. 类型检查 (Type Checking)
3. 栈帧 (Frames)
4. 中间表示 (Intermediate Representation)
5. 机器代码

本报告分为两大部分。前一部分分别讲述 Project 所做的五个部分，后一部分为实验结果。

2 抽象语法树 (Abstract Syntax Tree)

抽象语法树即使源代码的抽象语法结构的树状表现形式。树上的每个节点都表示源代码中的一种结构。之所以说语法是“抽象”的，是因为这里的语法并不会表示出真实语法中出现的每个细节。比如，嵌套括号被隐含在树的结构中，并没有以节点的形式呈现；而类似于 if-condition-then 这样的条件跳转语句，可以使用带有两个分支的节点来表示。

我们实现的语法规则参照了 http://web.cecs.pdx.edu/apt/cs302_1999/pcat99/pcat99.html 中给出的 PCAT 标准, 并根据该标准写出了 EBNF 语法, 之后我们将该 EBNF 语法转换至 BNF 语法, 即具体语法树。最后依据具体语法树制定了抽象语法树生成规则。

```
=====
Abstract Syntax - Modified
=====
```

```
program          -> ( Program ( body ) )
body              -> ( BodyDef ( declarations-list
                           statements-list ) )
declarations-list -> ( DeclareList ( {declarations
                                   } ) )
declarations      -> ( VarDecs ( {var-dec} ) )
                  -> ( TypeDecs ( {type-dec} ) )
                  -> ( ProcDecs ( {proc-dec} ) )
var-dec           -> ( VarDec ( ID type expression
                           ) )
type-dec          -> ( TypeDec ( ID type ) )
proc-dec          -> ( ProcDec ( ID formal-param-
                           list type body ) )
type              -> ( NamedTyp ( ID ) )
                  -> ( ArrayTyp ( type ) )
```

	-> (RecordTyp (component-list))
	-> (NoTyp ())
component-list	-> (CompList ({ component }))
)	
component	-> (Comp (ID type))
formal-param-list	-> (FormalParamList ({formal-
param }))	
formal-param	-> (Param (ID type))
statements-list	-> (SeqSt ({ statements-list }
)	
statement	-> (AssignSt (lvalue
expression))	
	-> (CallSt (ID expression-list
))
	-> (ReadSt (lvalue-list))
	-> (WriteSt (expression-list
))
	-> (IfSt (expression statement
	statement-else))
	-> (WhileSt (expression
	statement))
	-> (LoopSt (statement))
	-> (ForSt (ID expression-from
	expression-to expression-by
	statement))
	-> (ExitSt ())
	-> (RetSt (expression))
	-> (RetSt ())
	-> (SeqSt ({ statements-list }
))
expression-list	-> (ExprList ({ expression })
)	
expression	-> (BinOpExp (binop expression
-left expression-right))	
	-> (UnOpExp (unop expression
))
	-> (LvalExp (lvalue))
	-> (CallExp (ID expression-
	list))
	-> (RecordExp (ID record-init-
	list))
	-> (ArrayExp (ID array-init-
	list))
	-> (IntConst (INTEGER))
	-> (RealConst (REAL))

```

record-init-list      -> ( StringConst ( STRING ) )
      init } ) )
record-init           -> ( RecordInit ( ID expression
      ) )
array-init-list       -> ( ArrayInitList ( { array-
      init-list } ) )
array-init            -> ( ArrayInit ( expression-
      count expression-instance ) )
lvalue-list           -> ( LvalList ( { lvalue } ) )
lvalue                -> ( Var ( ID ) )
                     -> ( ArrayDeref ( lvalue
      expression ) )
                     -> ( RecordDeref ( lvalue ID ) )
binop                 -> GT | LT | EQ | GE | LE | NE |
      PLUS | MINUS | TIMES | SLASH
                     -> DIV | MOD | AND | OR
unop                  -> UPLUS | UMINUS | NOT

```

```

=====
Abstract Syntax - Extension
=====

```

in `get_comp_id`, # is ommited

```

program               -> ( Program ( body #local-
      offset) )
proc-dec              -> ( ProcDec ( ID formal-param-
      list type body #level #local-offset ) )
var-dec               -> ( VarDec ( ID type expression
      #level #offset ) )
formal-param          -> ( Param ( ID type #level #
      offset ) )
statement              ->
      -> ( CallSt ( ID expression-list
      #type #level-diff ) )
      -> ( ForSt ( ID expression-from
      expression-to expression-by
      statement #offset ) )
expression            -> ( BinOpExp ( binop expression
      -left expression-right #type #offset ) )
      -> ( UnOpExp ( unop expression
      #type #offset) )
      -> ( LvalExp ( lvalue #type #
      offset ) )

```

```

-> ( CallExp ( ID expression-
    list #type #level-diff #
    offset) )
-> ( IntConst ( INTEGER #type )
    )
-> ( RealConst ( REAL #type ) )
-> ( StringConst ( STRING #type
    ) )
lvalue
    offset ) )
-> ( Var ( ID type #level-diff #
    offset ) )
-> ( Var ( ID type) )    !! for
    TRUE/FALSE

```

3 类型检查 (Type Checking)

类型检查指的是对程序中的类型以及相关信息进行检查的一种程序分析过程。严格意义上的类型检查将会确保被检查的程序在执行时候不会有任何类型错误（即程序是类型安全的），但是更为通常意义上的类型检查提供了一定程度的类型安全性，但是不一定保证执行时候没有任何类型错误。

3.1 定义的类型检查

对于每个新的定义，我们将其加入当前空间之中，并检查是否有冲突。如果有的话，则需要报告错误。

```

VAR i : INTEGER := 10; (* OK *)
VAR i : INTEGER := 20; (* Error: Name Conflict *)

```

鉴于 PCAT 要求在定义变量的时候赋上初始值，我们同样对此进行检查，要求初始值的类型必须不矛盾。

```

VAR i : INTEGER := 10;    (* OK *)
VAR j : INTEGER := 20.0; (* Error: Type Conflict *)

```

由于在有初始值的时候可以省略类型名，因此需要进行类型推断。

```

VAR i := 10;    (* i is INTEGER *)
VAR j := 20.0; (* j is REAL *)

```

3.2 算数运算的类型检查

算数运算符号包括如下

```

a + b    (* PLUS *)
a - b    (* MINUS *)
a * b    (* MULTIPLICATION *)
a / b    (* DIVISION *)

```

```

a DIV b    (* INTEGER DEVISION *)
a MOD b    (* INTEGER MOD *)
a < b      (* LESS THAN *)
a <= b     (* LESS THAN OR EQUAL *)
a <> b     (* NOT EQUAL *)
a > b      (* LARGER THAN *)
a >= b     (* LARGER THAN OR EQUAL *)

```

对于算数运算表达式，我们将会对其检查。如果类型不符合，则会报错。

```

VAR i : INTEGER := 10;
VAR b : BOOLEAN := TRUE;
...
i + b; (* Error: Invalid Arithmetic Operation *)

```

对于二元或者一元算数运算，我们会根据两个操作数判断表达式的类型

```

VAR i : INTEGER := 10;
VAR j : REAL := 1.0;
...
(* INTEGER *)
i + i ;
i - i ;
i * i ;
i DIV i;
i MOD i;
+ i;
- i;

```

```

(* REAL *)
j + j ;
j - j ;
j * j ;
j / j ;
+ j ;
- j ;

```

```

(* REAL *)
i + j ;
i - j ;
i * j ;
i / j ;

```

3.3 逻辑运算的类型检查

对于算数运算表达式，同样会对其检查。如果类型不符合，则会报错。对于二元或者一元逻辑运算，可以判断表达式的类型。

3.4 赋值的类型检查

赋值的时候，要求被赋值的必须为左值

```
VAR x : INTEGER := 10;
...
x := 10; (* OK *)
x + x := 10; (* Error: LVALUE required *)
```

且类型必须相同

```
VAR x : INTEGER := 10;
...
x := 10; (* OK *)
x := TRUE; (* Error: Type Conflict *)
```

3.5 函数调用的类型检查

函数的调用必须满足形式参数和实际参数的数量相同

```
PROCEDURE FACTORIAL (A : INTEGER; B : INTEGER) : INTEGER
IS
...
END
...
y := FACTORIAL(1,2); (* OK *)
y := FACTORIAL(1,2,3); (* Error: Wrong number of
parameters *)
```

函数的调用同时要求形式参数和实际参数的类型匹配

```
PROCEDURE FACTORIAL (A : INTEGER; B : INTEGER) : INTEGER
IS
...
END
...
y := FACTORIAL(1,2);
(* OK *)

y := FACTORIAL(1.0,2.0);
(* Error: Wrong type of actual/formal parameters *)
```

返回值类型也需要匹配

```
PROCEDURE FACTORIAL (A : INTEGER; B : INTEGER) : INTEGER
IS
...
END
...
```

```

VAR y : INTEGER := 1;
VAR z : BOOLEAN := FALSE;

y := FACTORIAL(1, 2);
    (* OK *)

z := FACTORIAL(1.0, 2.0);
    (* Error: Wrong type of returned value *)

```

4 栈帧 (Frames)

系统栈对动态分配存储空间提供了方便的机制，使得在函数调用之中存储以下数据非常方便：

1. 参数
2. 保存的寄存器值
3. 本地变量
4. 返回值

严格来说，只要调用和被调用的程序在栈帧的使用上一致即可。但是，考虑到现有系统库的存在，一般使用的调用规范都和系统库一致。

当调用一个过程时候，系统在栈上分配空间；当此过程结束时候，栈上的空间将会被释放。在栈上存储的，用于实现过程调用和返回的数据，就是栈帧，也成为活动记录。通常而言，一个过程的栈帧包括了保存和回复该过程所有数据的信息每一个栈帧对应了一个还没有返回的过程的调用。

栈帧对应了两个最为重要的指针，ESP 和 EBP。ESP 表示当前栈帧的顶部，EBP 表示当前栈帧的界限，用以区分这当前栈帧和上一个栈帧。具体数据如下

- ...
- EBP-8 第二个本地变量
- EBP-4 第一个本地变量
- EBP 指向上一个栈帧的 EBP
- EBP+4 指向返回地址
- EBP+8 静态链接
- EBP+12 第一个参数
- EBP+16 第二个参数
- ...

由此可知，调用函数时候应该按照从后向左的顺序，把参数压入栈中。

4.1 样例

对于过程

```
PROCEDURE MULT (A, B : INTEGER) : INTEGER IS
    VAR I, P : INTEGER := 0;
BEGIN
    I := 1;
    FOR P:= 0 TO A BY 0 DO
        P := ADD (P, B);
        I := I + 1;
    END;
    RETURN P;
END;
```

其栈帧是

```
Frame for routine "MULT"
  formal parameters:
    [static link]      @ (%esp+8)
    A                  @ (%esp+12)
    B                  @ (%esp+16)
  local variables:
    I                  @ (%esp-4)
    P                  @ (%esp-8)
  frame size:
    stack allocated = 40 bytes
```

5 中间表示 (Intermediate Representation)

中间表示，是指一种应用于抽象机器的编程语言，设计用于，是用来帮助分析计算机程序。在编译器将源代码编译为目的码的过程中，会先将源代码转换为一个或多个的中间表示，以方便编译器进行最佳化，并产生出目的机器的机器语言。通常，中间表示之中每个指令代表仅有一个基本的操作，指令集内可能不会包含控制流程的资讯，暂存器可用的数量可能会很大，甚至没有限制。

最常见的中间表示形式，是三元码或者四元码。作为语言，有暂存器传递语言，静态单赋值形式，或者直接使用汇编语言。

5.1 暂存器传递语言

暂存器传递是一种中间语言，和汇编语言很接近，曾是 GCC 的中间语言，也被称为暂存器传递语言，风格类似于 LISP。GCC 的前端会先将编程语言转译成暂存器传递语言，之后再利用后端转化成机器码。

5.2 静态单赋值

静态单赋值形式是中间表示的特性，即每个变量仅被赋值一次。这个特性可是实现很多编译器最佳化的算法。

5.3 注意要点

逻辑运算符具有短路的特性，因此对于

```
IF a > 0 OR f(x) > 0 THEN
```

在 $a > 0$ 的时候，不应该调用 $f(x)$ ，以免语义出错。

6 机器代码 (Machine Code)

我们使用了 x86 汇编作为机器代码的生成。x86 汇编指令范围非常之大，我们使用了其一个子集。

6.1 数据传送

将数据从内存之中取出，放入寄存器。例如取得第一个参数就是

```
movl 12(%ebp), %ecx
```

或者将数据从寄存器中放回内存。例如存倒第一个本地变量就是

```
movl %ecx, -4(%ebp)
```

6.2 算术运算

加法使用

```
add src, dest
```

减法使用

```
sub src, dest
```

乘法使用

```
imul src, dest
```

除法使用

```
idev arg
```

6.3 逻辑运算

若要比較两个参数，应该使用

```
test arg1, arg2
cmp arg1, arg2
```

根据不同的结果，可以有条件或者无条件地转跳

```
jmp loc ; unconditional jumps
je loc ; jump on Equality
jne loc ; jump on Inequality
```

更大时候转跳，应该使用

```
jg loc
jge loc
ja loc
jae loc
```

更小时候转跳，应该使用

```
j1 loc
jle loc
jb loc
jbe loc
```

参数为 0 时候转跳，应该使用

```
jz loc
jnz loc
```

6.4 过程调用

调用应该按照从左到右压栈。例如

```
A := FACTORIAL (1, 4);
```

其调用形式应该是

```
movl $4, %eax
pushl %eax
movl $1, %eax
pushl %eax
pushl %ebp
call FACTORIAL
addl $12, %esp
```

注意到存入 EBP 是为了保存静态链接

6.5 函数进入/退出

进入函数时候，应该在栈上做好准备，代码如下

```
pushl %ebp
movl %esp, %ebp
subl $32, %esp
andl $-16, %esp
```

注意到其中的 32 是分配的空间，最后一句是为了栈上对其 16B 的界限。

退出函数时候，应该做好整理，代码如下

```
leave
ret
```

7 实验结果

我们采取若干程序作为样例，用来演示 Project 的效果。

7.1 IF 语句实验

本样例的目的在于测验 IF 语句
PCAT 代码为

```
(*   logic   *)
PROGRAM IS
    VAR i : INTEGER := 2;
    VAR j : INTEGER := 1;
BEGIN

    IF i < j THEN
        WRITE("less");
    ELSIF i = j THEN
        WRITE("equal");
    ELSE
        WRITE("larger");
    END;
    WRITE();

    i := 0;
    WHILE i < 5 DO
        WRITE(i);
        i := i + 1;
    END;
    WRITE();

    FOR i := 1 TO 6 DO
```

```

        WRITE(i);
    END;
    WRITE();

    FOR i := 1 TO 6 BY 2 DO
        WRITE(i);
    END;
    WRITE();
END;

```

其编译记录为

```

=====+=====
1 | PROGRAM IS
2 |   VAR i : INTEGER = 2;
3 |   VAR j : INTEGER = 1;
4 | BEGIN
5 |   IF (i < j) THEN
6 |     WRITE("less");
7 |   ELSE
8 |     IF (i = j) THEN
9 |       WRITE("equal");
10 |    ELSE
11 |      WRITE("larger");
12 |    END;
13 |  END;
14 |  WRITE([Empty Expression]);
15 |  i := 0;
16 |  WHILE (i < 5) DO
17 |    WRITE(i);
18 |    i := (i + 1);
19 |  END;
20 |  WRITE([Empty Expression]);
21 |  FOR i := 1 TO 6 BY 1 DO
22 |    WRITE(i);
23 |  END;
24 |  WRITE([Empty Expression]);
25 |  FOR i := 1 TO 6 BY 2 DO
26 |    WRITE(i);
27 |  END;
28 |  WRITE([Empty Expression]);
29 | END;
=====+=====

```

Type Checking finished *SUCCESSFULLY*

其 Frame 记录为

Frame for routine "Main Routine"

```

formal parameters:
    (not applicable for Main Routine)
local variables:
    i                                @ (%esp-4)
    j                                @ (%esp-8)
frame size:
    stack allocated = 64 bytes

```

其代码为

```

.text
.globl MainEntry

MainEntry:
    pushl %ebp
    movl %esp, %ebp
    subl $64, %esp
    andl $-16, %esp
    movl $2, %eax
    movl %eax, -4(%ebp)
    movl $1, %eax
    movl %eax, -8(%ebp)
    movl -4(%ebp), %eax
    movl %eax, -12(%ebp)
    movl -8(%ebp), %eax
    movl %eax, -16(%ebp)
    movl -12(%ebp), %eax
    movl -16(%ebp), %ecx
    cmpl %ecx, %eax
    setl %al
    movzbl %al, %eax
    movl %eax, -20(%ebp)
    movl -20(%ebp), %eax
    cmpl $0, %eax
    jne label_1
    movl -4(%ebp), %eax
    movl %eax, -24(%ebp)
    movl -8(%ebp), %eax
    movl %eax, -28(%ebp)
    movl -24(%ebp), %eax
    movl -28(%ebp), %ecx
    cmpl %ecx, %eax
    sete %al
    movzbl %al, %eax
    movl %eax, -32(%ebp)
    movl -32(%ebp), %eax
    cmpl $0, %eax

```

```

        jne label_3
        movl $label_5, %eax
        push %eax
        call print_str
        addl $4, %esp
        call print_line
        jmp label_4
label_3:
        movl $label_6, %eax
        push %eax
        call print_str
        addl $4, %esp
        call print_line
label_4:
        jmp label_2
label_1:
        movl $label_7, %eax
        push %eax
        call print_str
        addl $4, %esp
        call print_line
label_2:
        call print_line
        movl $0, %eax
        movl %eax, -4(%ebp)
label_8:
        movl -4(%ebp), %eax
        movl %eax, -36(%ebp)
        movl -36(%ebp), %eax
        movl $5, %ecx
        cmpl %ecx, %eax
        setl %al
        movzbl %al, %eax
        movl %eax, -40(%ebp)
        movl -40(%ebp), %eax
        cmpl $0, %eax
        je label_9
        movl -4(%ebp), %eax
        movl %eax, -44(%ebp)
        movl -44(%ebp), %eax
        push %eax
        call print_int
        addl $4, %esp
        call print_line
        movl -4(%ebp), %eax
        movl %eax, -48(%ebp)

```

```

        movl -48(%ebp), %eax
        movl $1, %ecx
        addl %ecx, %eax
        movl %eax, -52(%ebp)
        movl -52(%ebp), %eax
        movl %eax, -4(%ebp)
        jmp label_8
label_9:
        call print_line
        movl $1, %eax
        movl %eax, -4(%ebp)
label_10:
        movl -4(%ebp), %eax
        movl $6, %ecx
        cmpl %ecx, %eax
        jg label_11
        movl -4(%ebp), %eax
        movl %eax, -56(%ebp)
        movl -56(%ebp), %eax
        push %eax
        call print_int
        addl $4, %esp
        call print_line
        movl -4(%ebp), %eax
        movl $1, %ecx
        addl %ecx, %eax
        movl %eax, -4(%ebp)
        jmp label_10
label_11:
        call print_line
        movl $1, %eax
        movl %eax, -4(%ebp)
label_12:
        movl -4(%ebp), %eax
        movl $6, %ecx
        cmpl %ecx, %eax
        jg label_13
        movl -4(%ebp), %eax
        movl %eax, -60(%ebp)
        movl -60(%ebp), %eax
        push %eax
        call print_int
        addl $4, %esp
        call print_line
        movl -4(%ebp), %eax
        movl $2, %ecx

```



```

        addl %ecx, %eax
        movl %eax, -4(%ebp)
        jmp label_12
label_13:
        call print_line
        leave
        ret

        .section .rodata
label_5:
        .string "larger\0"

        .section .rodata
label_6:
        .string "equal\0"

        .section .rodata
label_7:
        .string "less\0"

```

7.2 循环语句实验

本样例的目的在于测验循环语句
PCAT 代码为

```

(*    call    *)
PROGRAM IS

    VAR n : INTEGER := 0;
    (*
    PROCEDURE A () : INTEGER IS
    BEGIN
        RETURN 1;
    END;

    PROCEDURE B (a:INTEGER; b:INTEGER) : INTEGER IS
    BEGIN
        RETURN a+b+1;
    END;

    PROCEDURE C (a:INTEGER; b:INTEGER; c:INTEGER) :
        INTEGER IS
    BEGIN
        RETURN b;
    END;

```

```

PROCEDURE D (a:INTEGER) : INTEGER IS
    VAR x : INTEGER := 0;
BEGIN
    x := 1;
    RETURN a+x;
END;
*)
PROCEDURE E (a:INTEGER) : INTEGER IS
    VAR x : INTEGER := 0;
BEGIN
    x := 1;
    RETURN n+a+x;
END;

PROCEDURE F () : INTEGER IS
BEGIN
    n := n * 10;
END;
BEGIN
    (*
    WRITE( A() , " 1");
    WRITE( B(100,10) , " 111" );
    WRITE( C(1,2,3) , " 2" );
    WRITE( D(10) , " 11" );

    n := 100;
    WRITE( E(10) , " 111" );
    *)

    n := 1;
    WRITE(n, " 1");
    F();
    WRITE(n, " 10");

END;

```

其编译记录为

```

=====+=====
1 | PROGRAM IS
2 |   VAR n : INTEGER = 0;
3 |   PROCEDURE E (a : INTEGER) : INTEGER
4 |     VAR x : INTEGER = 0;
5 |   BEGIN
6 |     x := 1;
7 |     RETURN ((n + a) + x);
8 |   END;

```

```

9 |   PROCEDURE F () : INTEGER
10 |   BEGIN
11 |       n := (n * 10);
12 |   END;
13 | BEGIN
14 |     n := 1;
15 |     WRITE(n, " 1");
16 |     F();
17 |     WRITE(n, " 10");
18 | END;

```

====+=====

Type Checking finished *SUCCESSFULLY*

其 Frame 记录为

Frame for routine "E"

```

    formal parameters:
        [static link]          @ (%esp+8)
        a                      @ (%esp+12)
    local variables:
        x                      @ (%esp-4)
    frame size:
        stack allocated = 28 bytes

```

Frame for routine "F"

```

    formal parameters:
        [static link]          @ (%esp+8)
    local variables:
        (No local variables)
    frame size:
        stack allocated = 12 bytes

```

Frame for routine "Main Routine"

```

    formal parameters:
        (not applicable for Main Routine)
    local variables:
        n                      @ (%esp-4)
    frame size:
        stack allocated = 16 bytes

```

其代码为

```

.text
.globl MainEntry

```

```

MainEntry:
    pushl %ebp
    movl %esp, %ebp

```

```

subl $16, %esp
andl $-16, %esp
movl $0, %eax
movl %eax, -4(%ebp)
movl $1, %eax
movl %eax, -4(%ebp)
movl -4(%ebp), %eax
movl %eax, -8(%ebp)
movl -8(%ebp), %eax
push %eax
call print_int
addl $4, %esp
movl $label_1, %eax
push %eax
call print_str
addl $4, %esp
call print_line
pushl %ebp
call F
addl $4, %esp
movl -4(%ebp), %eax
movl %eax, -12(%ebp)
movl -12(%ebp), %eax
push %eax
call print_int
addl $4, %esp
movl $label_2, %eax
push %eax
call print_str
addl $4, %esp
call print_line
leave
ret

```

E:

```

pushl %ebp
movl %esp, %ebp
subl $28, %esp
andl $-16, %esp
movl $0, %eax
movl %eax, -4(%ebp)
movl $1, %eax
movl %eax, -4(%ebp)
movl 8(%ebp), %edx
movl -4(%edx), %eax
movl %eax, -8(%ebp)

```

```

movl 12(%ebp), %eax
movl %eax, -12(%ebp)
movl -8(%ebp), %eax
movl -12(%ebp), %ecx
addl %ecx, %eax
movl %eax, -16(%ebp)
movl -4(%ebp), %eax
movl %eax, -20(%ebp)
movl -16(%ebp), %eax
movl -20(%ebp), %ecx
addl %ecx, %eax
movl %eax, -24(%ebp)
movl -24(%ebp), %eax
leave
ret
leave
ret

```

F:

```

pushl %ebp
movl %esp, %ebp
subl $12, %esp
andl $-16, %esp
movl 8(%ebp), %edx
movl -4(%edx), %eax
movl %eax, -4(%ebp)
movl -4(%ebp), %eax
movl $10, %ecx
imull %ecx, %eax
movl %eax, -8(%ebp)
movl -8(%ebp), %eax
movl 8(%ebp), %edx
movl %eax, -4(%edx)
leave
ret

```

```

.section .rodata
label_1:
.string " 1\0"

.section .rodata
label_2:
.string " 10\0"

```

7.3 命名空间实验

本样例的目的在于测验命名空间

PCAT 代码为

```
(*  call *)
PROGRAM IS

    VAR n : INTEGER := 0;
    (*
    PROCEDURE A () : INTEGER IS
    BEGIN
        RETURN 1;
    END;

    PROCEDURE B (a:INTEGER; b:INTEGER) : INTEGER IS
    BEGIN
        RETURN a+b+1;
    END;

    PROCEDURE C (a:INTEGER; b:INTEGER; c:INTEGER) :
        INTEGER IS
    BEGIN
        RETURN b;
    END;

    PROCEDURE D (a:INTEGER) : INTEGER IS
        VAR x : INTEGER := 0;
    BEGIN
        x := 1;
        RETURN a+x;
    END;
    *)
    PROCEDURE E (a:INTEGER) : INTEGER IS
        VAR x : INTEGER := 0;
    BEGIN
        x := 1;
        RETURN n+a+x;
    END;

    PROCEDURE F () : INTEGER IS
    BEGIN
        n := n * 10;
    END;
BEGIN
    (*
    WRITE ( A() , " 1");
```

```

WRITE( B(100,10), " 111" );
WRITE( C(1,2,3), " 2" );
WRITE( D(10), " 11" );

n := 100;
WRITE( E(10), " 111" );
*)

n := 1;
WRITE(n, " 1");
F();
WRITE(n, " 10");

```

END;

其编译记录为

```

=====+=====
1 | PROGRAM IS
2 |   VAR n : INTEGER = 0;
3 |   PROCEDURE E (a : INTEGER) : INTEGER
4 |     VAR x : INTEGER = 0;
5 |   BEGIN
6 |     x := 1;
7 |     RETURN ((n + a) + x);
8 |   END;
9 |   PROCEDURE F () : INTEGER
10 |  BEGIN
11 |    n := (n * 10);
12 |  END;
13 | BEGIN
14 |   n := 1;
15 |   WRITE(n, " 1");
16 |   F();
17 |   WRITE(n, " 10");
18 | END;
=====+=====
Type Checking finished *SUCCESSFULLY*

```

其 Frame 记录为

```

Frame for routine "E"
  formal parameters:
    [static link]      @ (%esp+8)
    a                  @ (%esp+12)
  local variables:
    x                  @ (%esp-4)
  frame size:

```

```

stack allocated = 28 bytes

Frame for routine "F"
  formal parameters:
    [static link]      @ (%esp+8)
  local variables:
    (No local variables)
  frame size:
    stack allocated = 12 bytes

Frame for routine "Main Routine"
  formal parameters:
    (not applicable for Main Routine)
  local variables:
    n                  @ (%esp-4)
  frame size:
    stack allocated = 16 bytes

```

其代码为

```

.text
.globl MainEntry

MainEntry:
    pushl %ebp
    movl %esp, %ebp
    subl $16, %esp
    andl $-16, %esp
    movl $0, %eax
    movl %eax, -4(%ebp)
    movl $1, %eax
    movl %eax, -4(%ebp)
    movl -4(%ebp), %eax
    movl %eax, -8(%ebp)
    movl -8(%ebp), %eax
    push %eax
    call print_int
    addl $4, %esp
    movl $label_1, %eax
    push %eax
    call print_str
    addl $4, %esp
    call print_line
    pushl %ebp
    call F
    addl $4, %esp
    movl -4(%ebp), %eax

```



```

movl %eax, -12(%ebp)
movl -12(%ebp), %eax
push %eax
call print_int
addl $4, %esp
movl $label_2, %eax
push %eax
call print_str
addl $4, %esp
call print_line
leave
ret

```

E:

```

pushl %ebp
movl %esp, %ebp
subl $28, %esp
andl $-16, %esp
movl $0, %eax
movl %eax, -4(%ebp)
movl $1, %eax
movl %eax, -4(%ebp)
movl 8(%ebp), %edx
movl -4(%edx), %eax
movl %eax, -8(%ebp)
movl 12(%ebp), %eax
movl %eax, -12(%ebp)
movl -8(%ebp), %eax
movl -12(%ebp), %ecx
addl %ecx, %eax
movl %eax, -16(%ebp)
movl -4(%ebp), %eax
movl %eax, -20(%ebp)
movl -16(%ebp), %eax
movl -20(%ebp), %ecx
addl %ecx, %eax
movl %eax, -24(%ebp)
movl -24(%ebp), %eax
leave
ret
leave
ret

```

F:

```

pushl %ebp
movl %esp, %ebp

```

```

        subl $12, %esp
        andl $-16, %esp
        movl 8(%ebp), %edx
        movl -4(%edx), %eax
        movl %eax, -4(%ebp)
        movl -4(%ebp), %eax
        movl $10, %ecx
        imull %ecx, %eax
        movl %eax, -8(%ebp)
        movl -8(%ebp), %eax
        movl 8(%ebp), %edx
        movl %eax, -4(%edx)
        leave
        ret

        .section .rodata
label_1:
        .string " 1\0"

        .section .rodata
label_2:
        .string " 10\0"

```

7.4 简单嵌套实验

本样例同样目的在于检测嵌套过程中的静态链接和动态链接的正确使用，这一点对于正确地实现嵌套的过程，包括其调用以及变量引用至关重要。

PCAT 代码为

```

(* This is a test of nested recursive parameterless *)
(* procedure calls with no local variables.           *)

PROGRAM IS
  VAR I, J, ANSWER : INTEGER := 0;
  PROCEDURE FACTORIAL() IS
    PROCEDURE
      MULT() IS BEGIN
        ANSWER := ANSWER * J;
      END;
    FACT() IS BEGIN
      MULT();
      IF J <> I THEN J := J + 1; FACT(); END;
    END;
  BEGIN
    ANSWER := 1;
    J := 1;

```

```

        FACT();
    END;
BEGIN
    WRITE ("The first 5 factorials are (in descending
        order):");
    FOR I := 5 TO 0 BY -1 DO
        FACTORIAL();
        WRITE("FACTORIAL(", I, ") = ", ANSWER);
        I := I - 1;
    END;
END;

```

其编译记录为

```

=====+=====
1 | PROGRAM IS
2 |   VAR I : INTEGER = 0;
3 |   VAR J : INTEGER = 0;
4 |   VAR ANSWER : INTEGER = 0;
5 |   PROCEDURE FACTORIAL () : [Void Type]
6 |     PROCEDURE MULT () : [Void Type]
7 |     BEGIN
8 |       ANSWER := (ANSWER * J);
9 |     END;
10 |   PROCEDURE FACT () : [Void Type]
11 |   BEGIN
12 |     MULT();
13 |     IF (J <> I) THEN
14 |       J := (J + 1);
15 |       FACT();
16 |     ELSE
17 |       [Empty Statement];
18 |     END;
19 |   END;
20 | BEGIN
21 |   ANSWER := 1;
22 |   J := 1;
23 |   FACT();
24 | END;
25 | BEGIN
26 |   WRITE("The first 5 factorials are (in descending
    order):");
27 |   FOR I := 5 TO 0 BY (- 1) DO
28 |     FACTORIAL();
29 |     WRITE("FACTORIAL(", I, ") = ", ANSWER);
30 |     I := (I - 1);
31 |   END;

```

32 | END;

====+=====
Type Checking finished *SUCCESSFULLY*

其 Frame 记录为

```
Frame for routine "MULT"
  formal parameters:
    [static link]          @ (%esp+8)
  local variables:
    (No local variables)
  frame size:
    stack allocated = 16 bytes

Frame for routine "FACT"
  formal parameters:
    [static link]          @ (%esp+8)
  local variables:
    (No local variables)
  frame size:
    stack allocated = 24 bytes

Frame for routine "FACTORIAL"
  formal parameters:
    [static link]          @ (%esp+8)
  local variables:
    (No local variables)
  frame size:
    stack allocated = 4 bytes

Frame for routine "Main Routine"
  formal parameters:
    (not applicable for Main Routine)
  local variables:
    I                      @ (%esp-4)
    J                      @ (%esp-8)
    ANSWER                 @ (%esp-12)
  frame size:
    stack allocated = 36 bytes
```

其代码为

```
.text
.globl MainEntry

MainEntry:
  pushl %ebp
  movl %esp, %ebp
```

```

    subl $36, %esp
    andl $-16, %esp
    movl $0, %eax
    movl %eax, -4(%ebp)
    movl $0, %eax
    movl %eax, -8(%ebp)
    movl $0, %eax
    movl %eax, -12(%ebp)
    movl $label_1, %eax
    push %eax
    call print_str
    addl $4, %esp
    call print_line
    movl $5, %eax
    movl %eax, -4(%ebp)
label_2:
    movl -4(%ebp), %eax
    movl $0, %ecx
    cmpl %ecx, %eax
    jg label_3
    pushl %ebp
    call FACTORIAL
    addl $4, %esp
    movl $label_4, %eax
    push %eax
    call print_str
    addl $4, %esp
    movl -4(%ebp), %eax
    movl %eax, -20(%ebp)
    movl -20(%ebp), %eax
    push %eax
    call print_int
    addl $4, %esp
    movl $label_5, %eax
    push %eax
    call print_str
    addl $4, %esp
    movl -12(%ebp), %eax
    movl %eax, -24(%ebp)
    movl -24(%ebp), %eax
    push %eax
    call print_int
    addl $4, %esp
    call print_line
    movl -4(%ebp), %eax
    movl %eax, -28(%ebp)

```

```

        movl -28(%ebp), %eax
        movl $1, %ecx
        subl %ecx, %eax
        movl %eax, -32(%ebp)
        movl -32(%ebp), %eax
        movl %eax, -4(%ebp)
        movl $1, %eax
        negl %eax
        movl %eax, -16(%ebp)
        movl -4(%ebp), %eax
        movl -16(%ebp), %ecx
        addl %ecx, %eax
        movl %eax, -4(%ebp)
        jmp label_2
label_3:
        leave
        ret

```

```

FACTORIAL:
        pushl %ebp
        movl %esp, %ebp
        subl $4, %esp
        andl $-16, %esp
        movl $1, %eax
        movl 8(%ebp), %edx
        movl %eax, -12(%edx)
        movl $1, %eax
        movl 8(%ebp), %edx
        movl %eax, -8(%edx)
        pushl %ebp
        call FACT
        addl $4, %esp
        leave
        ret

```

```

MULT:
        pushl %ebp
        movl %esp, %ebp
        subl $16, %esp
        andl $-16, %esp
        movl 8(%ebp), %edx
        movl 8(%edx), %edx
        movl -12(%edx), %eax
        movl %eax, -4(%ebp)
        movl 8(%ebp), %edx
        movl 8(%edx), %edx

```

```

movl -8(%edx), %eax
movl %eax, -8(%ebp)
movl -4(%ebp), %eax
movl -8(%ebp), %ecx
imull %ecx, %eax
movl %eax, -12(%ebp)
movl -12(%ebp), %eax
movl 8(%ebp), %edx
movl 8(%edx), %edx
movl %eax, -12(%edx)
leave
ret

```

FACT:

```

pushl %ebp
movl %esp, %ebp
subl $24, %esp
andl $-16, %esp
movl 8(%ebp), %edx
pushl %edx
call MULT
addl $4, %esp
movl 8(%ebp), %edx
movl 8(%edx), %edx
movl -8(%edx), %eax
movl %eax, -4(%ebp)
movl 8(%ebp), %edx
movl 8(%edx), %edx
movl -4(%edx), %eax
movl %eax, -8(%ebp)
movl -4(%ebp), %eax
movl -8(%ebp), %ecx
cmpl %ecx, %eax
setne %al
movzbl %al, %eax
movl %eax, -12(%ebp)
movl -12(%ebp), %eax
cmpl $0, %eax
jne label_6
jmp label_7

```

label_6:

```

movl 8(%ebp), %edx
movl 8(%edx), %edx
movl -8(%edx), %eax
movl %eax, -16(%ebp)
movl -16(%ebp), %eax

```

```

        movl $1, %ecx
        addl %ecx, %eax
        movl %eax, -20(%ebp)
        movl -20(%ebp), %eax
        movl 8(%ebp), %edx
        movl 8(%edx), %edx
        movl %eax, -8(%edx)
        movl 8(%ebp), %edx
        pushl %edx
        call FACT
        addl $4, %esp
label_7:
        leave
        ret

        .section .rodata
label_1:
        .string "The first 5 factorials are (in
                descending order):\0"

        .section .rodata
label_4:
        .string "FACTORIAL(\0"

        .section .rodata
label_5:
        .string ") = \0"

```

7.5 复杂嵌套实验

本样例的目的在于检测嵌套过程中的静态链接和动态链接的正确使用，以便能够正确地实现嵌套的过程，包括其调用以及变量引用。

PCAT 代码为

```

(* This is a test of nested recursive procedures with *)
(* parameters, local variables, and return values.      *)

PROGRAM IS
  VAR A : INTEGER := 0;
  PROCEDURE FACTORIAL (A : INTEGER; B : INTEGER) :
    INTEGER IS
    VAR T : INTEGER := 0;
    PROCEDURE ADD (A, B : INTEGER) : INTEGER IS BEGIN
      RETURN A + B;
    END;
    PROCEDURE MULT (A, B : INTEGER) : INTEGER IS

```



```

        VAR I, P : INTEGER := 0;
    BEGIN
        I := 1;
        FOR P:= 0 TO A BY 0 DO
            P := ADD (P, B);
            I := I + 1;
        END;
        RETURN P;
    END;
BEGIN
    WRITE ("IN FACTORIAL", "A=", A, ", B=", B);
    IF B > 0 THEN
        T := MULT (A, B);
        RETURN FACTORIAL (T, B - 1);
    END;
END;
BEGIN
    A := FACTORIAL (1, 4);
    WRITE ("FACTORIAL(4) = ", A, " (SHOULD BE 24)");
END;

```

其编译记录为

```

=====+=====
1 | PROGRAM IS
2 |   VAR A : INTEGER = 0;
3 |   PROCEDURE FACTORIAL (A : INTEGER, B : INTEGER) :
   INTEGER
4 |     VAR T : INTEGER = 0;
5 |     PROCEDURE ADD (A : INTEGER, B : INTEGER) :
   INTEGER
6 |       BEGIN
7 |         RETURN (A + B);
8 |       END;
9 |     PROCEDURE MULT (A : INTEGER, B : INTEGER) :
   INTEGER
10 |       VAR I : INTEGER = 0;
11 |       VAR P : INTEGER = 0;
12 |       BEGIN
13 |         I := 1;
14 |         FOR P := 0 TO A BY 0 DO
15 |           P := ADD(P, B);
16 |           I := (I + 1);
17 |         END;
18 |         RETURN P;
19 |       END;
20 |     BEGIN

```

```

21 |      WRITE("IN FACTORIAL", "A=", A, ", B=", B);
22 |      IF (B > 0) THEN
23 |          T := MULT(A, B);
24 |          RETURN FACTORIAL(T, (B - 1));
25 |      ELSE
26 |          [Empty Statement];
27 |      END;
28 |  END;
29 | BEGIN
30 |  A := FACTORIAL(1, 4);
31 |  WRITE("FACTORIAL(4) = ", A, " (SHOULD BE 24)");
32 | END;

```

====+=====

Type Checking finished *SUCCESSFULLY*

其 Frame 记录为

Frame for routine "ADD"

```

formal parameters:
    [static link]          @ (%esp+8)
    A                      @ (%esp+12)
    B                      @ (%esp+16)
local variables:
    (No local variables)
frame size:
    stack allocated = 16 bytes

```

Frame for routine "MULT"

```

formal parameters:
    [static link]          @ (%esp+8)
    A                      @ (%esp+12)
    B                      @ (%esp+16)
local variables:
    I                      @ (%esp-4)
    P                      @ (%esp-8)
frame size:
    stack allocated = 40 bytes

```

Frame for routine "FACTORIAL"

```

formal parameters:
    [static link]          @ (%esp+8)
    A                      @ (%esp+12)
    B                      @ (%esp+16)
local variables:
    T                      @ (%esp-4)
frame size:
    stack allocated = 52 bytes

```

```
Frame for routine "Main Routine"
  formal parameters:
    (not applicable for Main Routine)
  local variables:
    A                                @ (%esp-4)
  frame size:
    stack allocated = 16 bytes
```