# Recitation 8: Procedure Code Generation 2

COP3402 FALL 2015 – ARYA POURTABATABAIE

FROM EURIPIDES MONTAGNE, FALL 2014

# Procedure Address

There are two possible addresses to use for a procedure:

- The initial JMP address
- The INC address

Both are valid to use, because the program will behave exactly in same way.

```
00 jmp 0 6
01 jmp 0 2
02 inc 0 5
03 lit 0 2
04 sto 0 5
05 opr 0 0
06 inc 0 4
07 cal 0 2
08 rtn 0 0
```

# Procedure Address (JMP)

Here, we use the procedure declaration to store the address of the procedure:

procedure-declaration ::= { "**procedure**" ident "**;**" block "**;**" }

```
procedure PROC-DECL;
begin
    if TOKEN <> IDENTIFIER then ERROR() ;
    enter(PROCEDURE, ident, 0, level, NEXT_CODE_ADDR);
    GET_TOKEN();
    if TOKEN <> “;” then ERROR();
    GET_TOKEN();
    BLOCK();
    if TOKEN <> “;” then ERROR();
    GET_TOKEN();
end
```

This works because the first code instruction that block() generates is the JMP for this procedure.

# Procedure Address (INC)

If we want to use INC, then we must obtain the INC address from the BLOCK() function:

procedure-declaration ::= { "**procedure**" ident "**;**" block "**;**" }

```
procedure PROC-DECL;
begin
    if TOKEN <> IDENTIFIER then ERROR() ;
    enter(PROCEDURE, ident, 0, level, 888);
    GET_TOKEN();
    if TOKEN <> ";" then ERROR();
    GET_TOKEN();
    proc_addr = BLOCK();
    update(ident, proc_addr);
    if TOKEN <> ";" then ERROR();
    GET_TOKEN();
end
```

bogus address

Updates address in symbol table.

# Procedure Address (INC)

If we want to use INC, then we must obtain the INC address from the BLOCK() function:

procedure-declaration ::= { "**procedure**" ident "**;**" block "**;**" }

```
procedure BLOCK();
begin
        space = 4;
        jmpaddr = gen(JMP, 0, 0);
        if TOKEN = “const” then CONST-DECL();
        if TOKEN = “var” then space += VAR-DECL();
        if TOKEN = “procedure” then  PROC-DECL();
        code[jmpaddr].m = NEXT_CODE_ADDR;
        proc_addr = gen(INC, 0, space);
        STATEMENT();
        gen(RTN, 0, 0);
        return proc_addr;
end;
```

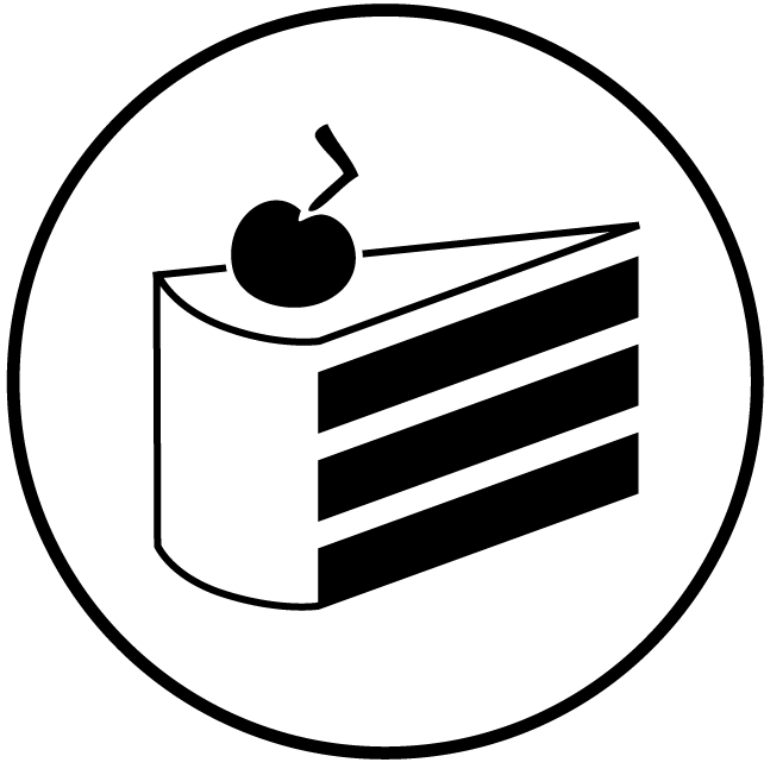
Returns the INC address

# Call

To generate a call, we must verify that we're calling a procedure, and use the correct level:

```
 procedure STATEMENT;
begin
      …
      else if TOKEN = "call" then begin
        GET_TOKEN();
        if TOKEN <> IDENT then ERROR (missing identifier);
        i = find(TOKEN);
        if i == 0 then ERROR (Undeclared identifier);
         if symboltype(i) == PROCEDURE then gen(CAL, level – symbollevel(i), symboladdr(i));
        else ERROR(call must be followed by a procedure identifier);
        GET_TOKEN();
      end
…
```

# What is wrong?



the cake is a lie !

# Recursive Programs

Consider this PL0 code:

```
procedure fooA;
      procedure fooB;
        begin
              call fooA;
        end;
      begin
        call fooB;
      end;
begin
      call fooA;
end.
```

What is going to happen if we generate code using INC addresses???

# Recursive Programs

Consider this PL0 code:

```
procedure fooA;
      procedure fooB;
        begin
              call fooA;
        end;
      begin
        call fooB;
      end;
begin
      call fooA;
end.
```

Here, we enter the procedure into the symbol table using a bogus address (888).

# Recursive Programs

Consider this PL0 code:

```
procedure fooA;
      procedure fooB;
        begin
              call fooA;
        end;
      begin
        call fooB;
      end;
begin
      call fooA;
end.
```

Here, we enter the procedure into the symbol table using a bogus address (888).

And here, we generate a CAL operation using fooA's bogus address:
CAL 0 2 888

# Recursive Programs

Consider this PL0 code:

```
procedure fooA;
    procedure fooB;
      begin
            call fooA;
      end;
    begin
      call fooB;
    end;
begin
    call fooA;
end.
```

Here, we enter the procedure into the symbol table using a bogus address (888).

And here, we generate a CAL operation using fooA's bogus address:
CAL 0 2 888

Here we know fooA's real address, but it's too late, we already generated who knows how many wrong CALs.

# Recursive Programs

Using the INC address works in some programs, but fails for some recursive programs.

Using the INC address for procedures is a lie!!!

# Scope

The scope of a symbol refers to where it is valid to use some given symbol.

The scope begins with the symbol declaration.

When the lexicographical level is less than the level at which the symbol was declared, the scope of that symbol ends.

Variables declared at level 0 have global scope: can be used anywhere.

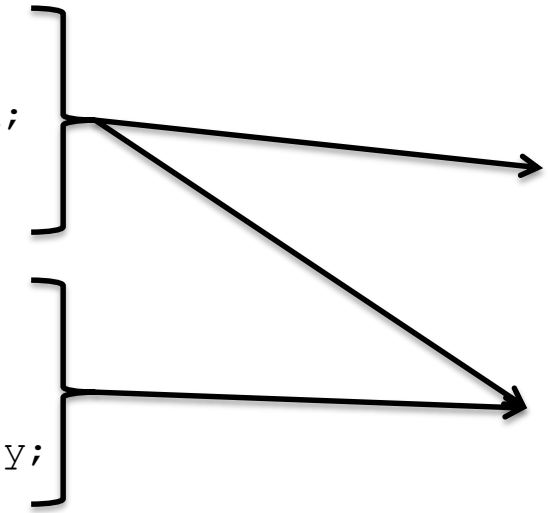
# Scope

```
0 var a,b;
0 procedure foo;
1    var x;
1    begin
1         read x;
1         a:=x;
1    end;
0 procedure lol;
1    var y;
1    begin
1         y:=b;
1         write y;
1    end;
0 begin
0    call foo;
0    b:=a;
0    call lol;
0 end.
```

- A symbol can be used only after it is declared.
- Access to: a,b,x,foo
- Access to: a,b,y,foo,lol
- Access to: a,b,foo,lol

# Scope

```
0 var a,b;
0 procedure foo;
1     var x;
1     begin
1           read x;
1           a:=x;
1     end;
0 procedure lol;
1     var y;
1     begin
1           y:=b;
1           write y;
1     end;
0 begin
0     call foo;
0     b:=a;
0     call lol;
0 end.
```



- foo can’t access lol: it has not been declared.
- lol can’t access x: the scope of x has ended.
- Main can’t access x,y: both scopes have ended.

# How to handle symbol's scope?

Using the Symbol Table!

When the scope begins, add the symbol to the symbol table.

When the scope ends, delete the symbol from the symbol table (it can't be used anymore).

How to delete symbols depends on implementation, but here is one example.

# How to handle symbol’s scope?

Suppose that the symbol table is a simple array of symbols, where each new symbol is added at the end.

'symbol_table‘ denotes the array of symbols.

‘sx' denotes the amount of symbols in the symbol table. It starts at sx=0.

Both are global variables.

# Add a new symbol

Just add the symbol at the end of the array.

```
void enter(kind,name,val,level,addr){
   table[sx].kind = kind;
   strcpy(table[sx].name, name);
   table[sx].val = val;
   table[sx].level = level;
   table[sx].addr = addr;
   sx++;
}
```

# Delete symbols

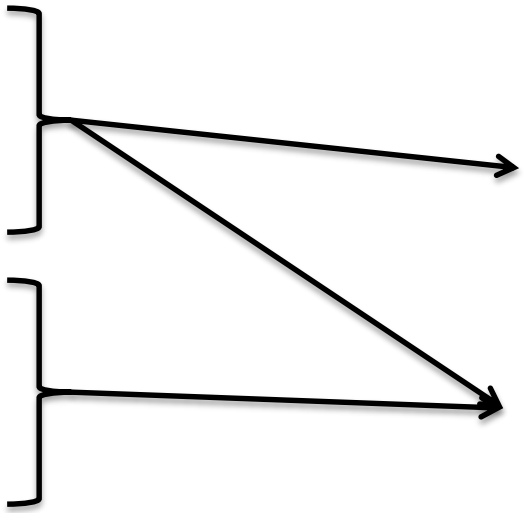
When we leave level 1, we don’t need the symbols declared at level 1 anymore.

One option is to loop through the whole array, and delete the symbols with level 1 or more.

Cumbersome: deleting arbitrary symbols from an array requires to move the symbols after it.

# Scope

```
var a,b;  0
procedure foo;
   var x; 1
   begin  1
        read x;
        a:=x;
   end;   1
procedure lol;
   var y; 1
   begin  1
        y:=b;
        write y;
   end;   1
begin     0
   call foo;
   b:=a;  0
   call lol;
end.      0
```



- foo can’t access lol: it has not been declared.
- lol can’t access x: the scope of x has ended.
- Main can’t access x,y: both scopes have ended.

# How to handle symbol's scope?

Using the Symbol Table!

When the scope begins, add the symbol to the symbol table.

When the scope ends, delete the symbol from the symbol table (it can't be used anymore).

How to delete symbols depends on implementation, but here is one example.

# How to handle symbol’s scope?

Suppose that the symbol table is a simple array of symbols, where each new symbol is added at the end.

'symbol_table‘ denotes the array of symbols.

‘sx' denotes the amount of symbols in the symbol table. It starts at sx=0.

Both are global variables.

# Add a new symbol

Just add the symbol at the end of the array.

```
void enter(kind,name,val,level,addr){
   table[sx].kind = kind;
   strcpy(table[sx].name, name);
   table[sx].val = val;
   table[sx].level = level;
   table[sx].addr = addr;
   sx++;
}
```

# Delete symbols

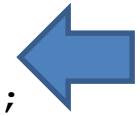
When we leave level 1, we don’t need the symbols declared at level 1 anymore.

One option is to loop through the whole array, and delete the symbols with level 1 or more.

Cumbersome: deleting arbitrary symbols from an array requires to move the symbols after it.

# Delete Symbols

```
0 var a,b;
0 procedure foo;
1    var x;
1    begin
1         read x;
1         a:=x;
1    end;
0 procedure lol;
1    var y;
1    begin
1         y:=b;
1         write y;
1    end;
0 begin
0    call foo;
0    b:=a;
0    call lol;
0 end.
```

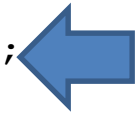


Symbol Table

- 0 a
- 0 b
- 0 foo
- 1 x

# Delete Symbols

```
0 var a,b;
0 procedure foo;
1    var x;
1    begin
1         read x;
1         a:=x;
1    end;
0 procedure lol;
1    var y;
1    begin
1         y:=b;
1         write y;
1    end;
0 begin
0    call foo;
0    b:=a;
0    call lol;
0 end.
```

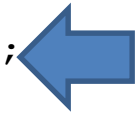


Symbol Table

- 0 a
- 0 b
- 0 foo

# Delete Symbols

```
0 var a,b;
0 procedure foo;
1    var x;
1    begin
1         read x;
1         a:=x;
1    end;
0 procedure lol;
1    var y;
1    begin
1         y:=b;
1         write y;
1    end;
0 begin
0    call foo;
0    b:=a;
0    call lol;
0 end.
```

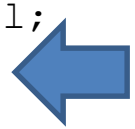


Symbol Table

- 0 a
- 0 b
- 0 foo
- 0 lol

# Delete Symbols

```
0 var a,b;
0 procedure foo;
1    var x;
1    begin
1         read x;
1         a:=x;
1    end;
0 procedure lol;
1    var y;
1    begin
1         y:=b;
1         write y;
1    end;
0 begin
0    call foo;
0    b:=a;
0    call lol;
0 end.
```

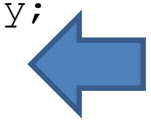


Symbol Table

- 0 a
- 0 b
- 0 foo
- 0 lol
- 1 y

# Delete Symbols

```
0 var a,b;
0 procedure foo;
1    var x;
1    begin
1         read x;
1         a:=x;
1    end;
0 procedure lol;
1    var y;
1    begin
1         y:=b;
1         write y;
1    end;
0 begin
0    call foo;
0    b:=a;
0    call lol;
0 end.
```

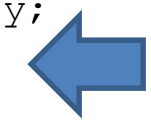


Symbol Table

- 0 a
- 0 b
- 0 foo
- 0 lol
- 1 y

# Delete Symbols

```
0 var a,b;
0 procedure foo;
1    var x;
1    begin
1         read x;
1         a:=x;
1    end;
0 procedure lol;
1    var y;
1    begin
1         y:=b;
1         write y;
1    end;
0 begin
0    call foo;
0    b:=a;
0    call lol;
0 end.
```

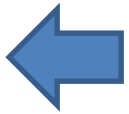


Symbol Table

- 0 a
- 0 b
- 0 foo
- 0 lol

# Delete Symbols

```
0 var a,b;
0 procedure foo;
1    var x;
1    begin
1         read x;
1         a:=x;
1    end;
0 procedure lol;
1    var y;
1    begin
1         y:=b;
1         write y;
1    end;
0 begin
0    call foo;
0    b:=a;
0    call lol;
0 end.
```



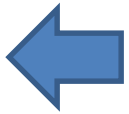
Symbol Table

- 0 a
- 0 b
- 0 foo
- 0 lol

# Delete Symbols

## Symbol Table

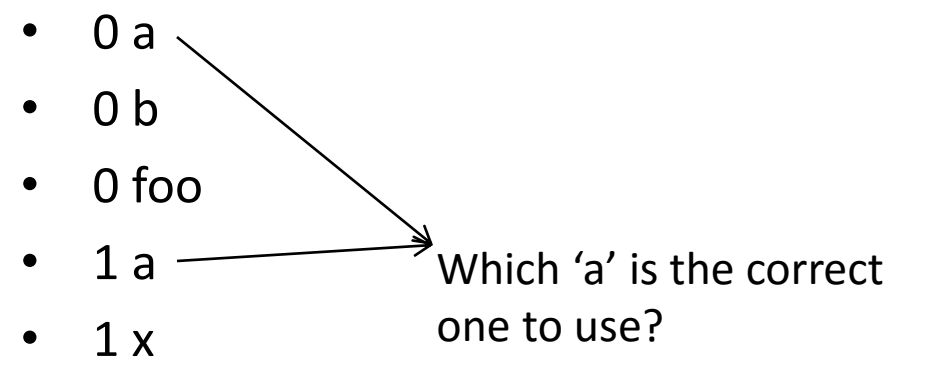
```
0 var a,b;
0 procedure foo;
1    var x;
1    begin
1         read x;
1         a:=x;
1    end;
0 procedure lol;
1    var y;
1    begin
1         y:=b;
1         write y;
1    end;
0 begin
0    call foo;
0    b:=a;
0    call lol;
0 end.
```



# Find symbols

When symbols have the same name, resolving the scope could be a problem.

```
var a,b;
procedure foo;
   var a,x;
   begin
       read x;
       a:=x;
   end;
begin
   call foo;
   b:=a;
   call lol;
end.
```



# Find Symbols

Return the last symbol found:

```
int find(ident){
   int index = -1;
   for(i = 0; i < sx; i++){
       if(strcmp(table[i].name,ident)==0){
              index = i;
       }
   }
   return index;
}
```

# Find Symbols

Search backwards:

```
int find(ident){
   for(i = sx-1; i >= 0; i--){
       if(strcmp(table[i].name,ident)==0){
             return i;
       }
   }
   return -1;
}
```

# Questions?