Access to Nonlocal Data on the Stack

It is very important to know that how procedures access their data, specially the mechanism for finding data used within a procedure p but that does not belong to p. Access becomes more complicated in languages where procedures can be declared inside other procedures. We therefore begin with the simple case of C functions, and then introduce a language, ML, that permits both nested function declarations and functions as "first-class objects;" that is, functions can take functions as arguments and return functions as values. This capability can be supported by modifying the implementation of the run-time stack.

Data Access without Nested Procedures: In the C family of languages, all variables are defined either within a single function or outside any function ("globally"). Most importantly, it is impossible to declare one procedure whose scope is entirely within another procedure. Rather, a global variable v has a scope consisting of all the functions that follow the declaration of v, except where there is a local definition of the identifier v.

Variables declared within a function have a scope consisting of that function only, or part of it, if the function has nested blocks. For languages that do not allow nested procedure declarations, allocation of storage for variables and access to those variables is simple:

1. Global variables are allocated static storage. The locations of these variables remain fixed and are known at compile time. So to access any variable that is not local to the currently executing procedure, we simply use the statically determined address.

2. Any other name must be local to the activation at the top of the stack. We may access these variables through the topsp pointer of the stack.

An important benefit of static allocation for globals is that declared procedures may be passed as parameters or returned as results (in C, a pointer to the function is passed), with no substantial change in the data-access strategy. With the C static-scoping rule, and without nested procedures, any name nonlocal to one procedure is nonlocal to all procedures, regardless of how they are activated. Similarly, if a procedure is returned as a result, then any nonlocal name refers to the storage statically allocated for it.

Issues with Nested Procedures: Access becomes far more complicated when a language allows procedure declarations to be nested and also uses the normal static scoping rule; that is, a procedure can access variables of the procedures whose declarations surround its own declaration, The reason is that knowing at compile time that the declaration of p is immediately nested within q does not tell us the relative positions of their activation records at run time. In fact, since either p or q or both may be recursive, there may be several activation records of p and/or q on the stack.

Finding the declaration that applies to a nonlocal name x in a nested procedure p is a static decision; it can be done by an extension of the static-scope rule for blocks. Suppose x is declared in the enclosing procedure q. finding the relevant activation of q from an activation of p is a dynamic decision; it requires additional run-time information about activations. One possible solution to this problem is to use "access links".

A Language with Nested Procedure Declarations: The C family of languages and many other familiar languages do not support nested procedures, so we introduce one that does. The history of nested procedures in languages is long. Algol 60, an ancestor of C, had this capability, as did its descendant Pascal, a once-popular teaching language. Of the laterlanguages with nested procedures, one of the most influential is ML, and it is this language whose syntax and semantics we shall borrow .ML is a functional language, meaning that variables, once declared and initialized, are not changed. There are only a few exceptions, such as the array, whose elements can be changed by special function calls.

Variables are defined, and have their unchangeable values initialized, by a statement of the form:

val (name) = (expression)

Functions are defined using the syntax:

fun (name) ((arguments)) = (body)

For function bodies we shall use let-statements of the form:

let (list of definitions) in (statements) end

The definitions are normally v a l or fun statements. The scope of each such definition consists of all following definitions, up to the in, and all the statements up to the end. Most importantly, function definitions can be nested. For example, the body of a function p can contain a let-statement that includes the definition of another (nested) function q. Similarly, q can have function definitions within its own body, leading to arbitrarily deep nesting of functions.

A code-generation algorithm

The algorithm takes as input a sequence of three-address statements constituting a basic block. For each three-address statement of the form x : = y op z, perform the following actions:

1. Invoke a function getreg to determine the location L where the result of the computation y op z should be stored.

2. Consult the address descriptor for y to determine y', the current location of y. Prefer the register for y' if the value of y is currently both in memory and a register. If the value of y is not already in L, generate the instruction MOV y' , L to place a copy of y in L.

3. Generate the instruction OP z' , L where z' is a current location of z. Prefer a register to a memory location if z is in both. Update the address descriptor of x to indicate that x is in location L. If x is in L, update its descriptor and remove x from all other descriptors.

4. If the current values of y or z have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution of x : = y op z , those registers will no longer contain y or z

Generating Code for Assignment Statements:

• The assignment d : = (a-b) + (a-c) + (a-c) might be translated into the following three-address code sequence:

Code sequence for the example is:

```
t : = a - b
u : = a - c
v:= t+ u
d : = v + u
```

with d live at the end.
Code sequence for the example is:

Statements	Code Generated	Register descriptor	Address descriptor
		Register empty	
t : = a - b	MOV a, R0 SUB b, R0	R0 contains t	t in R0
u : = a - c	MOV a , R1 SUB c , R1	R0 contains t R1 contains u	t in R0 u in R1
v:=t+ u	ADD R1, R0	R0 contains v R1 contains u	u in R1 v in R0
d : = v + u	ADD R1, R0 MOV R0, d	R0 contains d	d in R0 d in R0 and memory

Generating Code for Indexed Assignments

The table shows the code sequences generated for the indexed assignmen a:= b[i] and a[i]:= b

Statements	Code Generated	Cost
a : = b[i]	MOV b(Ri), R	2
a[i] : = b	MOV b, a(Ri)	3

Generating Code for Pointer Assignments

The table shows the code sequences generated for the pointer assignments a : = *p and *p : = a

Statements	Code Generated	Cost
a : = *p	MOV *Rp, a	2
*p : = a	MOV a, *Rp	2

Generating Code for Conditional Statements

Statement	Code
if x < y goto z	CMP x, y CJ< z /* jump to z if condition code is negative */
x : = y +z	MOV y, R0

```
if x < 0 goto z ADD z, R0
                    MOV R0,x
                    CJ< z
```

CODE GENERATION

The final phase in compiler model is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program. The code generation techniques presented below can be used whether or not an optimizing phase occurs before code generation.

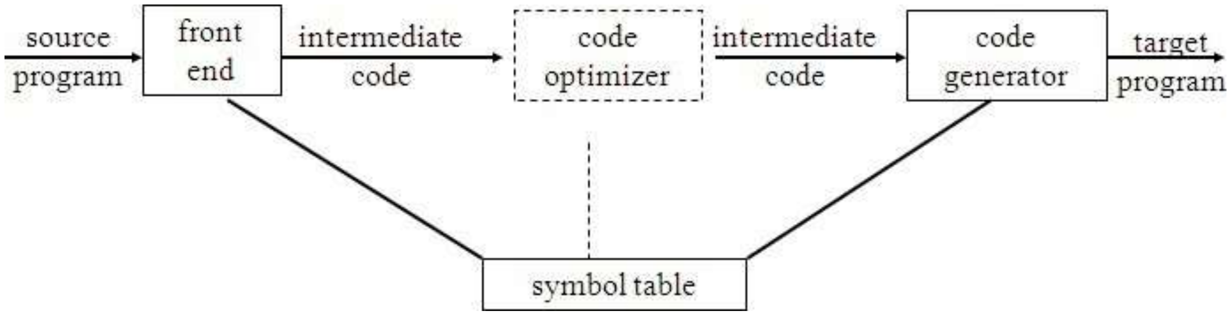


Fig. Position of code generator

ISSUES IN THE DESIGN OF A CODE GENERATOR

The following issues arise during the code generation phase:

1. Input to code generator
2. Target program
3. Memory management
4. Instruction selection
5. Register allocation
6. Evaluation order

1. Input to code generator:

• The input to the code generation consists of the intermediate representation of the source program produced by front end , together with information in the symbol table to determine run-time addresses of the data objects denoted by the names in the intermediate representation.

• Intermediate representation can be :

a. Linear representation such as postfix notation
b. Three address representation such as quadruples
c. Virtual machine representation such as stack machine code
d. Graphical representations such as syntax trees and dags.

• Prior to code generation, the front end must be scanned, parsed and translated into intermediate representation along with necessary type checking. Therefore, input to code generation is assumed to be error-free.

2. Target program:

• The output of the code generator is the target program. The output may be :

a. Absolute machine language

- It can be placed in a fixed memory location and can be executed immediately.

b. Relocatable machine language

- It allows subprograms to be compiled separately.

c. Assembly language

- Code generation is made easier.

3. Memory management:

• Names in the source program are mapped to addresses of data objects in run-time memory by the front end and code generator.

• It makes use of symbol table, that is, a name in a three-address statement refers to a symbol- table entry for the name.

• Labels in three-address statements have to be converted to addresses of instructions. For example,

j:gotoigenerates jump instruction as follows:

* if i < j, a backward jump instruction with target address equal to location of code for quadruple i is generated.

* if i > j, the jump is forward. We must store on a list for quadruple i the location of the first machine instruction generated for quadruple j. When i is processed, the machine locations for all instructions that forward jumps to i are filled.

4. Instruction selection:

• The instructions of target machine should be complete and uniform.

• Instruction speeds and machine idioms are important factors when efficiency of target program is considered.

• The quality of the generated code is determined by its speed and size.

• The former statement can be translated into the latter statement as shown below:

```
a:=b+c
d:=a+e          (a)

MOV b,R0 ADD
c,R0
MOV R0,a        (b)
MOV a,R0
ADD e,R0 MOV
R0,d
```

5. Register allocation

• Instructions involving register operands are shorter and faster than those involving operands in memory. The use of registers is subdivided into two subproblems :

1. Register allocation - the set of variables that will reside in registers at a point in the program is selected.

2. Register assignment - the specific register that a value picked.

• Certain machine requires even-odd register pairs for some operands and results.
For example , consider the division instruction of the form :D x, y where, x - dividend even register in even/odd register pair y-divisor even register holds the remainder odd register holds the quotient

6. Evaluation order

• The order in which the computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others.

RUN-TIME STORAGE MANAGEMENT

• Information needed during an execution of a procedure is kept in a block of storage called an activation record, which includes storage for names local to the procedure. The two standard storage allocation strategiesare:

1. Static allocation 2. Stack allocation

• In static allocation, the position of an activation record in memory is fixed at compile time.
• In stack allocation, a new activation record is pushed onto the stack for each execution of a procedure. The record is popped when the activationends.

• The following three-address statements are associated with the run-time allocation and de allocation of activation records:

1. Call,
2. Return,
3. Halt, and
4. Action, a placeholder for otherstatements.

• We assume that the run-time memory is divided into areas for:

1. Code
2. Static data
3. Stack

Static allocation

- In this allocation scheme, the compilation data is bound to a fixed location in the memory andit does not change when the program executes.
- As the memory requirement and storage locations are known in advance, runtime support package for memory allocation and de-allocation is not required.
- In a static storage-allocation strategy, it is necessary to be able to decide at compile time exactly where each data object will reside at run time. In order to make such a decision, at least two criteria must be met:

    1. The size of each object must be known at compile time.
    2. Only one occurrence of each object is allowable at a given moment during program execution.

- Because of the first criterion, variable-length strings are disallowed, since their length cannot be established at compile time. Similarly dynamic arrays are disallowed, since their bounds are not known at compile time and hence the size of the data object is unknown.
- Because of the second criterion, nested procedures are not possible in a static storage-allocation scheme. This is the case because it is not known at compile time which or how many nested procedures, and hence their local variables, will be active at execution time.

Implementation of call statement:

The codes needed to implement static allocation are as follows: MOV #here + 20, callee.static_area /*It saves return address*/
GOTO callee.code_area /*It transfers control to the target code for the called procedure */
where,

callee.static_area - Address of the activation record
callee.code_area - Address of the first instruction for called procedure
#here + 20 - Literal return address which is the address of the instruction following GOTO.

Implementation of return statement:

A return from procedure callee is implemented by : GOTO *callee.static_area
This transfers control to the address saved at the beginning of the activation record.

Implementation of action statement:

The instruction ACTION is used to implement action statement.

Implementation of halt statement:

The statement HALT is the final instruction that returns control to the operating system.

Dynamic Allocation

- The allocation can be varied during the execution
- It makes the use of recursive function.
- **In a dynamic storage-allocation strategy**, the data area requirements for a program are not known entirely at compilation time.
- In particular, the two criteria that were given in the previous section as necessary for static storage allocation do not apply for a dynamic storage-allocation scheme.
- The size and number of each object need not be known at compile time; however, they must be known at run time when a block is entered.
- Similarly more than one occurrence of a data object is allowed, provided that each new occurrence is initiated at run time when a block is entered.

Stack Allocation

- Procedure calls and their activations are managed by means of stack memory allocation.
- It works in last-in-first-out (LIFO) method and this allocation strategy is very useful for recursive procedure calls.

Static allocation can become stack allocation by using relative addresses for storage in activation records. In stack allocation, the position of activation record is stored in register so words in activation records can be accessed as offsets from the value in this register.

The codes needed to implement stack allocation are as follows:

Initialization of stack:

```
MOV #stackstart , SP /*
initializes stack */ Code for
the first procedure
HALT /* terminate execution */
```

Implementation of Call statement:

```
ADD #caller.recordsize, SP          /*
increment stack pointer */ MOV #here + 16,
*SP/*Save return address */
GOTO
callee.c
ode_ar
ea
where,
caller.recordsize            - size of the activation record
#here + 16 - address of the instruction following the GOTO
```

Implementation of Return statement:

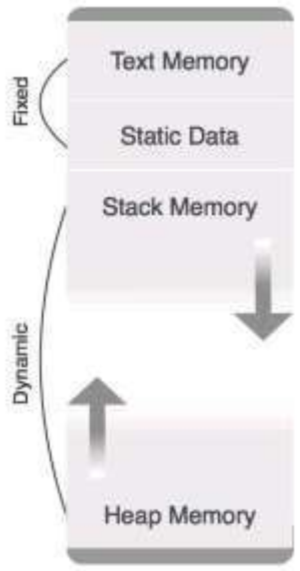
```
GOTO *0 ( SP )               /*return to the caller */
SUB #caller.recordsize, SP          /* decrement SP and restore to previous value */
```

Runtime environment manages runtime memory requirements for the following entities:

- **Code:** It is known as the text part of a program that does not change at runtime. Its memory requirements are known at the compile time.
- **Procedures:** Their text part is static but they are called in a random manner. That is why, stack storage is used to manage procedure calls and activations.
- **Variables:** Variables are known at the runtime only, unless they are global or constant. Heap memory allocation scheme is used for managing allocation and de-allocation of memory for variables in runtime.

Heap Allocation

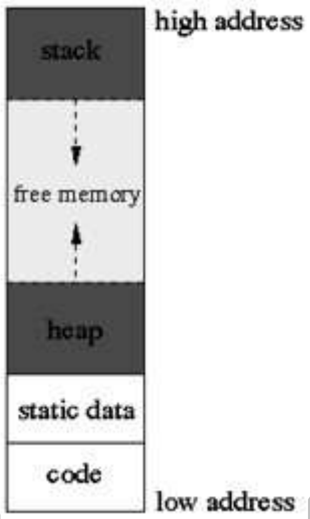
- Variables local to a procedure are allocated and de-allocated only at runtime.
- Heap allocation is used to dynamically allocate memory to the variables and claim it back when the variables are no more required.
- Except statically allocated memory area, both stack and heap memory can grow and shrink dynamically and unexpectedly.
- Therefore, they cannot be provided with a fixed amount of memory in the system.



- As shown in the image above, the text part of the code is allocated a fixed am unt of memory.
- Stack and heap memory are arranged at the extremes of total memory allocated to the program. Both shrink and grow against each other.

STORAGE ORGANIZATION

- An executable program generated by a compiler will have the following organization in memory on a typical architecture (such as on MIPS):



- This is the layout in memory of an executable program.
- Note that in a virtual memory architecture (which is the case for any modern operating system), some parts of the memory layout may in fact be located on disk blocks and they are retrieved in memory by demand (lazily).
- The machine code of the program is typically located at the lowest part of the layout.
- Then, after the code, there is a section to keep all the fixed size static data in the program.
- The dynamically allocated data (ie. the data created using malloc in C) as well as the static data without a fixed size (such as arrays of variable size) are created and kept in the heap. The heap grows from low to high addresses.
- When you call malloc in C to create a dynamically allocated structure, the program tries to find an empty place in the heap with sufficient space to insert the new data; if it can't do that, it puts the data at the end of the heap and increases the heap size.
- The focus of this section is the stack in the memory layout. It is called the run-time stack.
- The stack, in contrast to the heap, grows in the opposite direction (upside-down): from high to low addresses, which is a bit counterintuitive. The stack is not only used to push the return

address when a function is called, but it is also used for allocating some of the local variables of a function during the function call, as well as for some bookkeeping.

Activate Record

- It is used to store the current record and the record is been stored in the stack.
- It contains return value .After the execution the value is been return.
- It can be called as return value.

Parameter

- It specifies the number of parameters used in functions.

Local Data

- The data that is been used inside the function is called as local address

Temporary Data

- It is used to store the data in temporary variables.

Links

- It specifies the additional links that are required by the program.

Status

- It specifies the status of program that is the flag used.