CGTM #105

David Gries

March 1968

Compiler Implementation System

Code brackets and statements for controlling code generation

1. CODE AREAs, DATA AREAs and addresses
2. Some assumptions about DESCRIPTORS and code brackets
3. Runtime general register usage
4. Compile time description of runtime registers
5. Generating code for expressions
6. Generating code for statements
7. Runtime labels and branches to them
8. Generating procedure or function calls
9. Using CODE AREAs and DATA AREAs
10. Generating procedure entrances, exits and ends
11. Temporary storage at runtime

1. CODE AREAs, DATA AREAs and addresses.

The compiler writer may generate code into several different "CODEAREAs". At any time, one CODEAREA is designated as the current CODEAREA; code is generated into this particular one until another CODEAREA becomes current. Nested subroutines should be generated into different CODEAREAs. CODEAREAs are numbered 1, 3, 5,...,199. The initial current CODEAREA is number 1. This use of different CODEAREAs will be explained at the appropriate time; from section 1 to section 9 we assume we are generating code into one single CODEAREA.

The compiler writer may allocate storage for runtime variables in any number of DATAAREAs. These DATAAREAs can be STATIC, which means they exist during all of runtime — or DYNAMIC, which means they will be present from time-to-time. In the latter case, the compiler writer must generate runtime calls on subroutines which allocate and release storage for such DYNAMIC DATAAREAs. During compile time, there is always one current DATAAREA in which storage for variables will be allocated. DATAAREAs are numbered 2, 4, ..., 200. DATAAREA 2 is STATIC, is the initial current DATAAREA, and is moreover used to store all constants used at runtime.

The compiler writer may consider the current CODEAREA to be a dynamic table of elements of type BYTES. The name of the table is CODETABLE; CODEAD is a pointer to (the address of) the next free byte in the table. CODELOC always contains the runtime offset of CODEAD from the beginning of the table.

Thus one can enter a byte whose name is A into the table with

the usual ENTER command:

P ← ENTER(CODETABLE,A) .

Both CODEAD and CODELOC are altered by this function.

DATALOC contains the runtime offset of the next free byte in the current DATAAREA.

The address of a runtime operand is usually given at compile-time by the CODEAREA or DATAAREA number to which it has been assigned and the offset for that operand in the AREA (distance in bytes from the beginning of the AREA to the leftmost byte assigned to the operand). To reference the operand at runtime, the runtime address of the base of the AREA must be known and in a register. The system automatically takes care of generating and loading these "address constants". A DESCRIPTOR, with KIND ℰADDCON, is generated and storage is allocated in DATAAREA 2 for each such address constant.

In some instances the compiler writer must give such address constants for dynamic DATAAREAs to the system. This topic we leave to section 10 .

2. Some assumptions about DESCRIPTORs and code brackets.

A code statement looks like

code(...) or P = code(...)

where P is of type DESCRIPTOR or a pointer variable. The part within the code brackets "code(" and ")" is one or more statements or expressions to be executed at runtime. Execution of a code statement causes 360 machine language code to be generated for these statements or expressions and perhaps a DESCRIPTOR to be built which describes the result. If P has type DESCRIPTOR, the resulting DESCRIPTOR is stored in P. If P is a pointer, then the system allocates storage to the result and puts the address of the result into P.

In most circumstances, the system will delete a DESCRIPTOR to which it allocated storage when that DESCRIPTOR is used again within code brackets. The compiler writer may override this option by setting a bit in the DESCRIPTOR. If the DESCRIPTOR itself is stored in P, then the compiler writer himself has control over it and can release it when no longer needed. However, there is one restriction on the use of the DESCRIPTOR. The code generation system maintains pointers to DESCRIPTORs in which it has an interest while code is being generated. These include DESCRIPTORs of operands in a register, address constants, labels whose addresses are not yet defined but referenced, etc. It is therefore important that no DESCRIPTOR involved in code generation be moved to another storage location.

3. Runtime general register usage.

We indicate here the way the registers will be used at runtime; this should help the reader understand how code is generated. In general, we use the standard OS 360 conventions. Registers 0 and 1 may be used for parameter linkage; register 13 must always contain the address of a SAVE AREA for the subroutine being executed. This the compiler writer may provide for and will be explained in full detail in the section on subroutines. Registers 14 and 15 are used as usual for subroutine calls. In addition we require 3 other registers. Register 12 contains the address of DATAAREA 2 — the place where constants, etc., are stored. Register 10 contains the address of the subroutine being executed (for the main program this is address of the CODEAREA into which it is being generated.) Register 8 will be used, if the code for the subroutine needs over 4096 bytes, to provide addressibility for the extra bytes of code. Summing up, we have

Register	Use
0,1	parameter linkage, temporary storage
2-7	temporary storage
8	base register, temporary storage
9	temporary storage
10	base register for the address of a subroutine or main program
11	base register for current DATAAREA, if not number 2, temporary storage and if not in register 13.
12	base register for DATAAREA 2

Register	Use
13	address of a SAVEAREA
14	return address, temporary storage
15	address of called subroutine, temporary storage

4. Manipulating the compile time description of runtime registers.

The code generation system maintains a description of the runtime contents of registers for the current CODE AREA. This description changes as code is generated which changes the registers at runtime. The description of a register contains information such as a pointer to a DESCRIPTOR of the information in the register, whether the information is temporary or must be kept in the register, etc.

In order to produce code which does not execute unnecessary loading and string of registers, it is necessary to keep track of their contents. For most purposes however it is not necessary to use most of the material in this section. We have tried to make the scheme fairly general so that the writer of quick and dirty compilers can leave most of the details to the system, while the writer of an efficient compiler can handle the details himself.

This section contains a description of most of the methods of handling runtime registers and their descriptions. The use of these methods are explained in the different sections about code generation in detail – labels, procedure calls, entries, etc.

I. There are essentially two uses for a register

1. The register(s) holds an intermediate, temporary result. This may be a temporary value from an arithmetic expression, the base address (address constant) of a CODE or DATA AREA used to address some value, or some calculated BASIC ADDRESS of some value (see DESCRIPTOR writeup).

The normal state is for such a temporary value to be "used up" when it is first used in generating code. However it is usually still in the register and can be used again, for instance, if it is an address constant.

Moreover, the compiler writer may want to indicate that a value must remain in existence until he says it can be released.

2. The register is to act as the location of the variable defined by the DESCRIPTOR describing the value. This might be the case, for instance, for a loop variable which will be accessed very often. Another example is register 12, which always contains the address of DATA AREA 2.

The register description contains a pointer to a DESCRIPTOR of the value in the register. If empty, this pointer is 0 (EMPTYREG). If not empty the register description also contains 2 USE bits to indicate just what the value is:

USE bits		Meaning
0	0	the register contains a value which can be discarded on first use (TEMPREG).
0	1	the register contains a value which was used once and can be thrown away at any time (USEDREG).
1	0	the value in the register must be saved until further notice (SAVEREG).

USE bits		Meaning
1	1	the register is being used as a "fast" location for a variable (FASTREG).

The usual values for the USE bits are 00 and 01; the code generation system continually generates new intermediate results for expressions, uses them, and then releases them automatically.

II Suppose a register is needed to hold some new result. The following table indicates which register will be used, and what will happen (at runtime) to its present contents. The first register found which satisfies the lowest possible of the following priorities will be used.

Priority	USE bits	Code is generated to
1	empty	
2	01	(the value is lost)
3	00	store the value in a temporary location
4	10	store the value in a temporary location
5	11	store the value into its real location

In all cases, the DESCRIPTOR is changed (or destroyed) to reflect the change.

In general, we should say, the compiler write need not worry about what each register is being used for. Code for the usual assignment statements and other statements can be generated without

worrying about the registers. The compiler writer must only know the details of the allocation scheme when he wants to code conditional expressions, and for loops, etc. very efficiently.

III It is sometimes advantageous to save this register description for later use. For instance, when generating code for a statement

if E then <statement>$_1$ else <statement>$_2$,

the contents of the registers will be exactly the same when beginning both <statement>$_1$ and <statement>$_2$. Thus one can save the register descriptions just before executing <statement>$_1$ and restore them just before executing <statement>$_2$. Moreover, we should be able to "join" the register descriptions after <statement>$_1$ with those describing the results of executing <statement>$_2$, since the statement following this conditional statement is executed next in both cases. Note that this causes no code to be generated; it is only a means of aiding the code generation routines.

We thus have the following procedures. In all cases <data ref> must be a reference to a pointer variable.

6. JOINREG (<data ref>$_1$, <data ref>$_2$)

The register descriptions are joined, as in 4., and stored in <data ref>$_1$. The storage pointed at by <data ref>$_2$ is released and <data ref>$_2$ is set to zero.

1. SAVEREG (<data ref>)

   Storage is allocated for a set of register descriptions. The address is stored in <data ref>. The current register descriptions are copied into the allocated storage.

2. USEREG (<data ref>)

   The register descriptions pointed at by <data ref> are stored into the current register description area.

3. RESTOREREG (<data ref>)

   Same as USEREG, except that the storage pointed at by <data ref> is released and <data ref> is set to zero.

4. JOINREG (<data ref>)

   The current register description is "joined" with the one pointed at by <data ref>: those register descriptions which are the same in both sets are kept, the others are set to "empty". The storage pointed at by <data ref> is released and <data ref> is set to zero.

5. EXCH (<data ref>)

   The register descriptions pointed at by <data ref> become the current ones, while <data ref> is changed to point at the old current descriptions.

When the current register descriptor is changed by the above statements, the system will check to make sure that all register values are consistent with the normal usage.

IV Register names

$\mathcal{E}$GREG denotes any general register — to be determined by the system, while $\mathcal{E}$FREG denotes any floating pointer register.

In addition one may use

$\mathcal{E}$REG(i)

to represent a specific register, where i is a BYTE expression and has the same meaning as in component INREG of the DESCRIPTOR:

i	register
0	none
1 - 15	general 1-15
16	general 0
18,20,22,24	floating 0,2,4,6

V Loading registers

1. A register name may appear on the left side of an assignment statement within code brackets. Thus one can write

P = CODE($\mathcal{E}$GREG = D) and P = CODE($\mathcal{E}$REG(2) = D)

In the first case, the system determines the register to use, in the second, general register 2 is used. P will contain the address of the descriptor describing the result (which is not D). If necessary, code will first be generated to store the register. The DESCRIPTOR

D is not changed (but may be destroyed if temporary storage is allocated by the system.) USE bits are set to 00.

2. The statement &INREG (D, <regis>) specifies that the value described by D is in the specific register described by D. D is changed to reflect this fact, but no code is actually generated. (The system automatically performs a &EMPTYREG (<regis>) before changing the DESCRIPTOR (see Section VI) USE bits are set to 00.

VI Emptying registers

1. The operand &DUMPREG (<regis>) may appear with code brackets

P = CODE (&DUMPREG(<regis>))

Code is generated to store the contents of <regis> in a location. If no address was defined, a temporary location is allocated. The address of the DESCRIPTOR which described the value in the register is stored in P. This DESCRIPTOR is changed to reflect the change. The register description is set to empty. A register which had use bits 10 or 11 loses this status of course.

The following table describes what happens for &DUMPREG, depending on the USE bits of the register description

USE bits	Code generated to
empty	none
00	store register in a location
01	none

USE bits	Code generated to
10	store register in a location
11	store register in its location

2. Execution of the statement

P = &EMPTYREG (<regis>)

generates no code, but changes the register description for<regis> to empty. The DESCRIPTOR is changed to reflect the fact it is no longer in a register and is destroyed if allowed. P will contain the address of the DESCRIPTOR (or 0 if it has been destroyed).

VII Altering and testing a register description

The following functions change the value of the register description USE bits as described. Errors may occur if things are not consistent

function	Use bits changed to
EMPTYREG(<regis>)	empty
FASTREG(<regis>)	11
SAVEREG(<regis>)	10
USEDREG(<regis>)	01
TEMPREG(<regis>)	00

The result of the functions is the pointer to the DESCRIPTOR of the register value (0 if none).

The following functions yield the value true if the register description contains the information in column 2, false otherwise.

function	true if Use bits are
ISEMPTYREG(<regis>)	empty
ISFASTREG(<regis>)	11
ISSAVEREG(<regis>)	10
ISUSEDREG(<regis>)	01
ISTEMPREG(<regis>)	00

5. Generating code for expressions

We assume in this section that runtime storage has been allocated and the addresses have been filled in in the DESCRIPTORS. We first indicate the simple operands one can put in code brackets.

I. <operand>s and <simple operand>s

<simple operand> ::= <DESCR> | <RUN VAR> | <constant>

<operand> ::= <simple operand> | &INDIR(<descriptor expression>) |
<DESCR> (<descriptor expression>) | <DESCR> (*<descriptor-
expression>) |
<DESCR> &LENGTH(<descriptor expression>) | &EA (<des-
criptor expression>)
&EFAD(<descriptor expression>)

A <DESCR> is a reference to a compiletime element of type DESCRIPTOR or a reference to a pointer variable which points to a DESCRIPTOR. It thus describes some runtime variable. A <RUN VAR> is a variable declared to be valid at runtime. If a <simple operand> is both a <DESCR> and a <RUN VAR>, it is assumed to be a <DESCR>.

An operand may appear alone in code brackets:

CODE(<operand>) or P = CODE(<operand>)

The appearance of an <operand> within code brackets (either alone or within an expression or statement) may cause code to be generated into the current CODE AREA and a new DESCRIPTOR to be built which describes the result of execution of that code, as follows:

A) <operand> is a <DESCR>. No code is generated. The result is the <DESCR> itself.

B) <operand> is a <RUN VAR>. No code is generated. The result is a DESCRIPTOR which describes the <RUN VAR>. If the <RUN VAR> is an element of some structure type, the resulting DESCRIPTOR KIND is &BYTES, with LENGTH equal to the number of bytes needed for the element.

C) <operand> is <constant>. No code is generated. The result is a DESCRIPTOR for the <constant>.

The next three type of operands are used to indicate indirect addressing and subscripting. Code is in general not generated, but may be in certain circumstances. Note that these <operand>s indicate only an alteration of the effective address of a runtime value.

D) <operand> is &INDIR(<descriptor expression>). See CGTM 104, Section 3.IV.A.

E) <operand> is <DESCR> (<descriptor expression>). This is a subscripting process. See CGTM 104, Section 3.IV.B.

F) <operand> is <DESCR> (*<descriptor expression>). See CGTM 104, Section 3.IV.C.

G) <operand> is <DESCR>&LENGTH (<descriptor expression>) See CGTM 104, Section 3.IV.

H) <operand> is &EA (<descriptor expression>). The types of operands which alter effective addresses (D,E, and F) do not in general emit code to calculate these effective addresses; this is usually done when the operand is actually used in some operation (like +,=,etc). However, it may be necessary at times to calculate this effective address completely, before it is used in an operation. The resulting DESCRIPTOR of an operand &EA (<descriptor expression>) has all the characteristics of the DESCRIPTOR of <descriptor expression>

except that the effective address has been calculated so that no subscripting and at most one level of indirectness is used (the address is in a register or location).

I) <operand> is &EFAD (<descriptor expression>). The resulting DESCRIPTOR has KIND &POINTER. The value described is the address of the value of the <descriptor expression>. (If the value of the <descriptor expression> is only in a register, code is generated to store it in a temporary location.)

II. <descriptor expression>s

This section describes the use of expressions within code brackets. The syntax is the same as the syntax of expressions outside code brackets.

<descr primary> ::= <operand> | (<descriptor expression>)

<descr unary exp> ::= [+ | - | NOT] <descr primary>

<descr power exp> ::= <descr unary exp> [power **]*

<descr mult exp> ::= <descr power exp> [/ | *]

<descr add exp> ::= <descr mult exp> [+ | -]

<descr rel exp> ::= <descr add exp> [<rel operator> <descr add exp>]

<descr and exp> ::= <descr rel exp> AND

<descriptor expression> ::= <descr and exp> OR

<rel operator> ::= [EQUAL =] | [LESS <] | [GREATER >] | [NOTLESS GE] | [NOTGREATER LE]

When a <descriptor expression> within code brackets is evaluated, code is generated into the current CODE AREA which, when executed at runtime, will perform the desired calculations. For instance, execution of

P= CODE(D1 + D2 * D3)

accomplishes the following:

1. Code is generated to multiply the runtime operands whose DESCRIPTORS are D2 and D3 (if the KINDs of D2 and D3 are not the same, code may first be generated to convert one or both of them as necessary.) A new DESCRIPTOR, say D4, will be built to describe the result of this runtime multiplication. The code generation system will be changed to describe the new runtime state of the machine (contents of registers, condition code, D2 or D3 may be released if temporary DESCRIPTORS, etc.).
2. Code is generated to add the runtime operands whose DESCRIPTORs are D1 and D4; again code for any necessary conversions is also generated. A new DESCRIPTOR; say D5, is built to describe the result of this runtime addition, and the code generation system is again changed to reflect the new runtime state of the machine.
3. The address of the DESCRIPTOR D5 of the result is stored in the pointer variable P.

In general, any<descriptor expression> may appear alone in code brackets. Since the compile-time result of the execution of the code

brackets is usually a DESCRIPTOR or the address of a DESCRIPTOR of some runtime result, it is usual to save this DESCRIPTOR or its address in some compile-time variable (as above). Any conversions or operations which include only constants will be done at compile-time.

The execution of these binary and unary operations at runtime is exactly the same as the execution of these operations outside code brackets at compile-time. (Incomplete) tables are given in section 5.2 of CGTM 101, except for operands whose KIND is &BYTES.

Runtime operans of KIND &BYTES may only be used in the operations NOT (complement), AND, OR, and assignment (described later). If two operands of type &BYTES are ANDed or ORed together, the length of the result is the length of the longer operand; the shortest is padded on the left with leading zeroes. Better code will be generated for statements of the form

$$\text{CODE}\ (D_A = D_A \boxed{\begin{matrix}\text{AND}\\ \text{OR}\end{matrix}} D_B) \quad \text{or} \quad \text{CODE}\ (D_A = D_B \boxed{\begin{matrix}\text{AND}\\ \text{OR}\end{matrix}} D_A)$$

than if it is broken up into

$$P = \text{CODE}\ (D_A \boxed{\begin{matrix}\text{AND}\\ \text{OR}\end{matrix}} D_B); \quad \text{CODE}\ (D_A = P)$$

6. Generating code for statements

<run state seq> ::= <run state> | <run state seq> ; <run state>

<run state> ::= <open run state> | <closed run state>

<open run state> ::= <label decl> <open run state>

::= <iter open run state>

::= IF <descriptor expression> THEN <closed run state>
ELSE <open run state>

::= IF <descriptor expression> THEN <run state>

<closed run state> ::= <label decl> <closed run state>

::= <iter closed run state> | <run assign state>

::= <compound run state> | <case run state>

::= IF <descriptor expression> THEN <closed run state>
ELSE <closed run state>

::= <run branch state> | <run proc call>

<label decl> ::= (see section 7a)

<iter open run state> ::= FOR <run assign var> = <fle> DO <open run state>

::= WHILE <descriptor expression> DO <open run state>

<iter closed run state> ::= FOR <run assign var> = <fle> DO <closed run state>

::= WHILE <descriptor expression> DO <closed run state>

<run assign var> ::= <DESCR> | <RUN VAR> (see section 5)

<fle> ::= <descriptor expression> [STEP <descriptor expression>]
UNTIL <descriptor expression>

<run assign state> ::= <run assign var> = <descriptor expression>

<compound run state> ::= BEGIN <run state seq> END

<case run state> ::= CASE <descriptor expression> OF <run state seq> ENDCASE

<run branch state> ::= (see section 7b)

<run proc call> ::= (see section 8)

For purposes of understanding, in the above nonterminals run stands for runtime, state for statement and assign for assignment.

Any <run state seq> may appear within code brackets. It causes code to be generated for that sequence of statements.

The following statements all have their counterparts in the semantic language outside code brackets and need no further explanation:

<iter open run state>, <iter closed run state>, <run assign state>, <compound run state> and <case run state>.

The usual conditional statements also need no explanation, However, the <run branch state> is more likely to be used than the usual conditional statement. It is suggested that Section 7 be read carefully.

7. Runtime labels and branches to them.

a) Generating labels

<run label> ::= <DESCR> (with component KIND = £LABEL)

<label decl> ::= <run label>: | <run label> (<data ref>):

A <label declaration> may appear alone or in code brackets:

CODE(<run label>:) or CODE(<run label> (<data ref>):)

or along with other labels and statements. It declares the runtime address of the label <run label> to be the address of the next free byte in the current CODEAREA. This address is inserted into the DESCRIPTOR. Previous branches to this label will also be "fixed up" — the address will be inserted into the branch instruction.

Note that no DESCRIPTOR results from a label within code brackets. If a statement P=CODE(<run label>:) is executed, P will be set to zero.

With the first type of <label decl>, CODE(<run label>:), the register descriptions will be set as follows:

1. Register descriptions with USE bits 01 are set to empty.
2. Register descriptions with USE bits 10 and 11 are left alone.
3. Register descriptions with USE bits 00 yield an error message and are set to empty. This is because the value has not been used and there is probably a mistake. Note that with USE bits 10 and 11, it is up to the compiler writer to make sure these values are in the register from wherever he branches to this label (at runtime).

With the second <label declaration>, CODE(<run label>(<data ref>:)), the <data ref> must be a pointer to a set of register descriptions. These register descriptions become the current ones. The storage is released and <data ref> is set to zero.

b) Generation of transfers

<run branch state> ::= <go to operator> <DESCR>

::= IF <descriptor expression> <go to operator> <DESCR>

::= IFNOT <descriptor expression> <go to operator> <DESCR>

<go to operator> ::= GO | GOTO | GO TO

A <run branch state> inside code brackets indicates a branch or a conditional branch to the runtime address specified by the DESCRIPTOR <DESCR> which is KIND &LABEL. The address of <DESCR> need not be defined yet. If not yet defined, the code generation system will automatically fill in the address when it becomes defined.

<DESCR> may specify an operand of type &POINTER, in which case it is assumed that the address of the label is the value of the runtime pointer operand. This address must be in this current CODEAREA into which the branch is being generated.

With a <conditional goto statement>, the branch will occur if the value defined by <descriptor expression> is not equal to zero (IF) or zero (IFNOT).

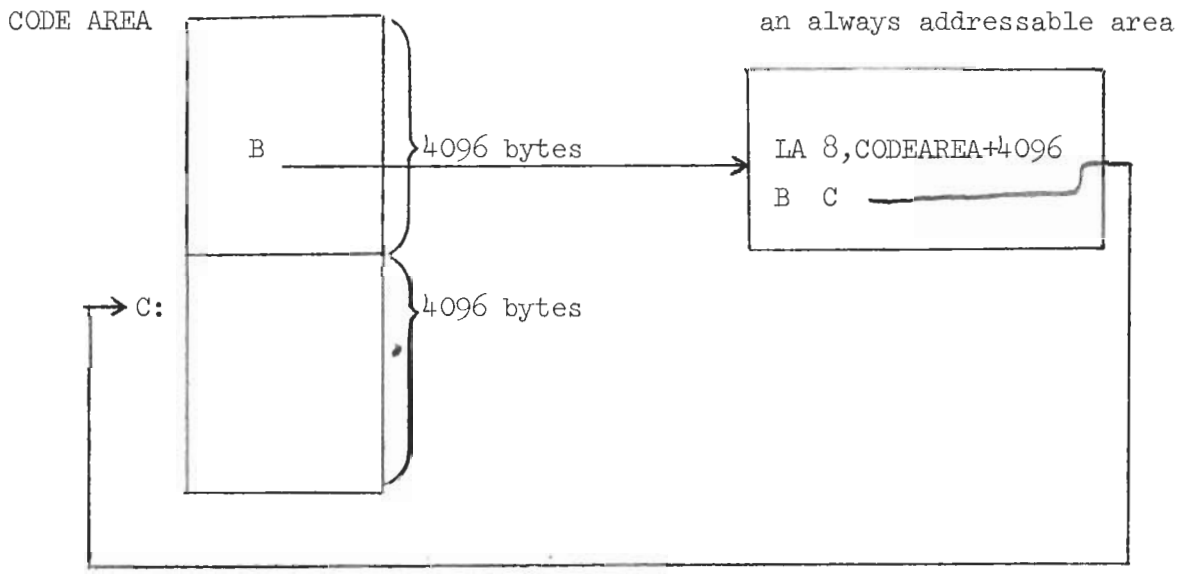
Efficient code will be produced if the <descriptor expression> has the form

<descr add exp> <rel operator> <descr add exp> (see Section 3b)

Jumps accross CODE AREAs are in general not allowed. Since each CODE AREA is used for a subroutine (or a set of parallel subroutines) such a jump could easily cause an error with the SAVE AREAs, etc. A provision for such jumps may be included at a later date if necessary.

Each generated branch consists of a single four byte (perhaps conditional) branch instruction. If the label is not addressed at this

point, the branch will be a set of instructions which will load register 8 with the necessary address and then branch to the label (see the section on runtime general register usage), as the following diagram shows for a branch to label C



8. Generating procedure or function calls.

<runtime proc call> ::= <DESCR>

The <DESCR> must have component KIND = &PROC. The address-defining components need not yet be filled in — the branch address will be filled in when the address components are defined — as with labels. A <runtime proc call> in code brackets results in the code

```
<dump register 14 and 15 if necessary>
L     15,=address of procedure
BR    14,15
```

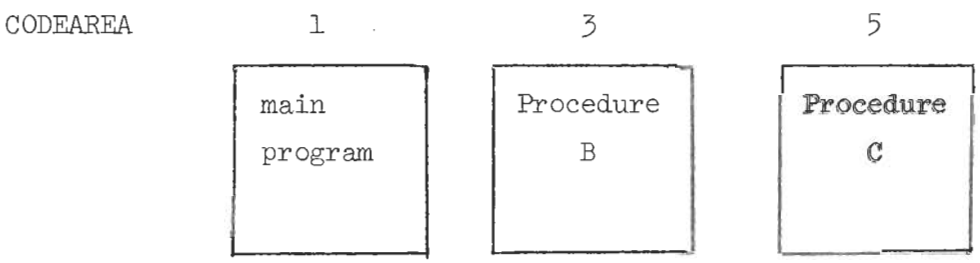
This is the usual OS 360 calling sequence. It is up to the compiler writer to generate the instructions to load registers for parameter linkage, etc., and any values returned in registers. It is also assumed that the other registers are not changed by the procedure or function at runtime. If they are changed, the compiler writer must change the resgister descriptions.

9. Using CODE AREAs and DATA AREAs.

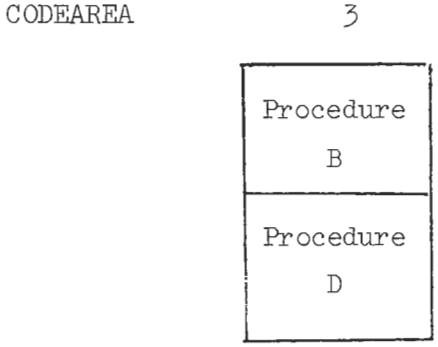
As explained before in Section 1, nested procedures should be generated into different CODE AREAs. For example, if we are translating an ALGOL program with structure

```
A: begin     procedure B
                begin
                   procedure C...
                   begin
                        .
                        .
                        .
                   end;
                end;
     end
```

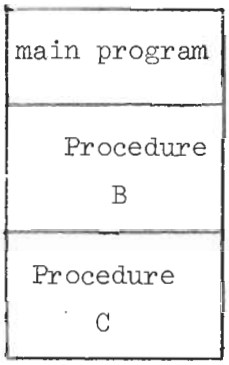
we would generate code as follows:



If another procedure D is adjacent to B and not nested within D, we could generate D into CODEAREA 3 also



The point is that it is easier to keep track of register descriptions and base registers in this manner. It is just compile time organizational detail. When translation is finished these code areas will automatically be concatenated and executed immediately or made into an OS load module:



The code generation system will automatically switch CODEAREAs, relieving the compiler writer of this responsibility, when a runtime procedure begin and end is declared. A stack is kept of the CODEAREAs into which code is being generated. When procedure begin is declared, the current CODEAREA is pushed onto this stack and a new CODEAREA is started. At a procedure end the current CODEAREA is "closed" and the top one on this stack becomes the current one. See Section 10 for complete details.

The compiler writer may wish to handle these details himself because of his source language requirements. Execution of the statement

USECODEAREA(<expression>)

makes the CODEAREA <expression> the current one. Note that the register description used is the one assigned to the new CODEAREA. These specify empty if the CODEAREA hasn't been used yet. It is up to the compiler writer to indicate what is in the registers before generating code into

a new CODEAREA. The initial register descriptions for the initial CODEAREA are: registers 0-9, 11, 14, 15-empty; register 10 – base of CODEAREA 1; register 12 – base of DATAAREA 2; register 13 – address of SAVEAREA (in DATAAREA 2). These will be loaded correctly when the main program execution is begun.

The system BYTE variable &CODEAREA always contains the number of the current CODEAREA.

Storage is always automatically allocated for temporary locations, and storage is usually allocated for runtime variables, in the current DATAAREA. This DATAAREA may be static – in which case it exists throughout runtime – or dynamic. In the latter case the compiler writer must allocate and release storage for the DATAAREA at runtime himself, by calling (at runtime) routines of his own or system routines, which will be described in another report. Byte &DATAAREA always contains the current DATAAREA number.

The normal method is to have one DATAAREA for each subroutine or procedure, but this is not necessary.

The code generation system takes care of addressing static DATAAREAs itself; if such a DATAAREA is referenced and if no address constant exists for it yet, the system will construct one and put it in DATAAREA 2. However for dynamic DATAAREAs the system needs some help from the compiler writer, since it is not known until runtime just where the DATAAREA will be. In fact, there may be several copies of it if it is the DATAAREA of a recursive subroutine. Whether a certain DATAAREA is accessible may also depend on which CODEAREA code is being generated into.

Note that the size of a DATAAREA is known at the end of translation. Storage must be allocated to variable length arrays and tables as usual

through runtime routines and these arrays and tables must be referenced through pointers in some DATAAREA.

When generating code which references a dynamic DATAAREA into the current CODEAREA, the compiler writer must indicate where (at runtime) the address of the beginning byte of the DATAAREA may be found. This he does by giving the code generation system a DESCRIPTOR of type &ADDCON which contains the address of the address constant and its value. One can do this several ways: The function DYNADDCON when evaluated yields a pointer to a DESCRIPTOR of KIND &ADDCON. Its evaluation depends on the parameters, as follows.

1. DYNADDCON (D) where D is a DESCRIPTOR of KIND &ADDCON. The address and value of the address constant must already be filled in. The result is a pointer to D.

2. DYNADDCON(<DATAAREA>$_1$,<DATAAREA>$_2$, <offset>)
   <DATAAREA>$_1$ is the DATAAREA number of the DYNAMIC DATA AREA.
   <DATAAREA>$_2$ and <offset> give the location of the address constant.
   The result is the address of this DESCRIPTOR.

3. P ← DYNADDCON (<DATAAREA>, <regis>)
   The <DATAAREA> is the number of the DYNAMIC DATAAREA. Its base address is in the register <regis>. A DESCRIPTOR for the address constant will be built.

Note that the address constant is usable only when generating code in the current CODEAREA. Note also that only the address constant for the base of the DATAAREA is needed — others will be generated as necessary by the system.

Execution of the statement

DELETEADDCON (D)

where D is a DESCRIPTOR of an address constant for a DYNAMIC DATAAREA makes D unavailable in the current CODEAREA. If the value was in a register, that register is set to empty.

10. Generating procedure entrances, exits and ends.

I. Runtime Procedure begin

Generating a procedure begin is done by the following

CODE(PROCEDURE <DESCR>) .

The <DESCR> must of course be KIND &PROC and have its address as yet undefined. The code generation system does the following.

1. Push the current CODEAREA and DATAAREA number on an internal stack.
2. Make the next higher CODEAREA the current CODEAREA.
3. Assign the new CODEAD as the basic address in the <DESCR> and fix up all previously generated called to it.
4. Set the current register descriptions as follows:

   0-11, 13-15 empty 12 = address of DATAAREA 2

It is then up to the compiler writer to execute instructions which

1. indicate the contents of registers (parameter linkage, reg 13 = address of old SAVEAREA, etc.)
2. generate instructions to save registers and get a new SAVEAREA or to jump to a routine which will save them.
3. generate instructions which bring over parameters or jump to a routine which does this.
4. indicate a new current DATAAREA (if any) and whether it is static or dynamic, and if necessary an address constant for it.

II. Other procedure entry points

A procedure may have more than one entry point. Execution of

CODE (<DESCR>:)

where <DESCR> has KIND &PROC defines the runtime address of the <DESCR> to be the address of the next byte in the current CODEAREA and any previous references to <DESCR> to be filled in. It is up to the compiler writer to initialize the register descriptions, etc. for the new procedure entry. Note that the current CODE and DATAAREA are not changed.

III. Runtime procedure returns

At present we leave this up to the compiler writer, since there are several different methods.

IV. Runtime procedure ends.

Execution of

CODE(ENDPROCEDURE <DESCR>)

causes the current CODEAREA and DATAAREA to be "closed" and the current areas to become defined by the top element of the stack (see I).

11. Temporary Storage at runtime.

The code generation routines continually need new temporary storage to hold intermediate results. In general, if an intermediate result is in a register it need not be assigned a location. If the register is needed for some other purpose, temporary storage is then allocated in the current DATAAREA for the intermediate result. This location remains allocated to that result until the DESCRIPTOR is destroyed. The location is then available for other use.