The research reported in this paper was sponsored by the Advanced Research Projects Agency Information Processing Techniques Office and was monitored by the Electronic Systems Division, Air Force Systems Command under contract F1962867C0004, Information Processing Techniques, with the System Development Corporation.

TECH MEMO *a working paper*

System Development Corporation / 2500 Colorado Avenue / Santa Monica, California 90406

Information International Inc. / 11161 Pico Boulevard / Los Angeles, California 90064

TM- 3417/385/00

AUTHOR D. Anschultz

TECHNICAL J. Barnett

RELEASE C. Weissman, S.D.C.
D. Anschultz, I.I.I.
for J. I. Schwartz

DATE 4/26/67



(Page 2 is blank)

# LISP 2 Compiler Register Counter and Code Generator Specifications

## ABSTRACT

This document describes functions performed during the Register Counter and Code Generator passes of the LISP 2 compiler proposed for the IBM S/360 computer. The Register Counter (pass V of the LISP 2 compiler) counts and remembers register needs of subexpressions. The Code Generator (pass VI of the LISP 2 compiler) compiles Register-Counted Interlude Language (RCIL) into LAP assembly language.

## 1. INTRODUCTION

The first pass of the LISP 2 compiler that is concerned with register allocation is the Machine Link pass. At this time, information about the references to lexical variables generated in bindings is collected and inserted into the declarations for these variables. Those lexical variables that have no reference made to their locations are established as candidates to be LEXREG's. Those variables later assigned as LEXREG's will have their values maintained in a register (or registers) throughout their scope, rather than having their values stored.

The real concern about register allocation begins in the next pass, the Register Counter pass. Here, those LEXREG candidates whose references are numerous enough to warrant it are assigned as LEXREG's. Basically, this pass counts the number of registers that will be required to evaluate each expression. Information about such items as which registers will be used in evaluating an expression and where the value will be found is placed into the expression for use by the following pass, the Code Generator pass.

During the Code Generator pass, final register assignments are made. Cognizance of parallel register usage* is used to allow more LEXREG candidates to be assigned and to allow more intermediate results to be kept in registers.

As can be seen, a dominant dictum has been to keep all registers containing meaningful information whenever possible, and provide those variables whose utility is greatest with the most convenient access. Adherence to these precepts should generate code that is both shorter and faster.

## 2. CONVENTIONS

The assignment of functions for all four floating-point and M general-purpose registers resides in the compiler. For purposes of discussion, the general-purpose registers will be considered to be M contiguous registers whose names are the digits 1 through M. (A likely value for M is 8.)

N (probably 3 or 4) of these registers are used in the transmission of arguments to functions. Assignment of argument registers is from the last argument to the first, and from register 1 to register N. Values are returned in register 1. (For values requiring more than one register, the registers are assigned consecutively from 1.)

---

*"Parallel register usage" refers to registers that may be allocated at this time because a preceding or subsequent statement must have them available.

Functions whose value-type is REAL return their value in F1. The final REAL argument (if any) of any function is passed in F1. Indef args are treated as though they were the first and second arguments of a function, except that the first is never passed in a register. The second argument is the integer count of elements of the first, and may be passed in a register as any other integer argument.

Those arguments that require more than one register to contain their values are passed in registers if enough registers remain, otherwise earlier arguments (if any) may be passed in registers instead.

At the return from a function call, any general registers not used for values of arguments will be unchanged.

## 3. PROGRAM STRUCTURE

During the Register Counter pass, one basic switch functions as a nucleus for recursion: REGCOUNT. This function binds some public variables used for register counting and calls a function appropriate to the MSIL form-name to perform the counting. REGCOUNT then puts the results of this resolution into the expression. Updating of other public variables is also done to reflect increased parallel usage requirements.

REGCOUNT binds and puts into the listing: TRC, FRC, TFC, FFC, AR, and AF. It updates PUC, PUCF, TRC, FRC, TFC, and FFC.

## 4. PUBLIC VARIABLES

| Name | Type | Class | Range | Meaning |
|---|---|---|---|---|
| N | Integer | Parameter | Invariant | Number of general registers allotted to be used to pass arguments of functions |
| M | Integer | Parameter | Invariant | Number of general registers allotted to be assigned functions by the computer |
| THRESHOLDS | List | Parameter | No more than M-N elements | The first element is the threshold requirement for LEXREG assignment when M registers are available; the next for M-1, etc. |
| THRESHOLDF | List | Parameter | No more than 3 elements | Corresponds to THRESHOLDS for floating-point registers |

| Name | Type | Class | Range | Meaning |
|---|---|---|---|---|
| C | Integer | Context | N<C≤M | Number of registers currently assignable |
| CF | Integer | Context | 1<CF≤4 | Number of floating-point registers currently assignable |
| PUC | Integer | Context | 0≤POC≤C | Parallel usage count |
| PUCF | Integer | Context | 0≤PUCF≤CF | Parallel usage count of floating-point registers |
| REGUSE | List | Context | | List of current register assignments |
| XREG | Name | Context | | Register in which answer desired |
| TRC | Integer | Requirement | 0≤TRC≤M | Total register count |
| FRC | Integer | Requirement | 0≤FRC≤N | Fixed register count (argument regs) |
| TFC | Integer | Requirement | 0≤TFC≤4 | Total floating register count |
| FFC | Integer | Requirement | 0≤FFC≤1 | Fixed floating register count |
| AR | Name | Requirement | Numbers 1-M, ARB | Answer register |
| AF | Name | Requirement | TRUE, FALSE, COM, PART | Availability flag |
| NEED | List | Requirement | | Fixed register usage requirement |

## 5. RULES FOR REGISTER ALLOCATION

### 5.1 BIND FORM FOR REGISTER COUNTER PASS

(BIND ($V_1$ ... $V_k$) S)

(1) Bind i to 1, x to NIL, y to NIL.

(2) If i GR k, go to (6).

(3) If $V_i$ is not a lexreg-candidate, go to (4).

(4) If the type of $V_i$ is REAL, put reference-count of $V_i$ in list y, else in list x. (Lists x and y are generated in sorted order, i.e., $x_i$ LQ $x_{i+1}$.)

(5) Set i to i+1; go to (2).

(6) Set x to PICKEM(x NOFF(M-C THRESHOLDS)).

(7) Set i to k.

(8) If x is NIL, go to (13).

(9) If $V_i$ is not a non-REAL, lexreg-candidate, go to (12).

(10) If reference-count of $V_i$ NQ CAR(x), go to (12).

(11) Change $V_i$ from lexreg-candidate to LEXREG; set x to CDR(x); go to (7).

(12) Set i to i - 1; go to (9).

(13) Set y to PICKEM(y NOFF(4-CF THRESHOLDF)).

(14) Set i to k.

(15) If y is NIL, go to (20).

(16) If $V_i$ is not a REAL, lexreg-candidate, go to (19).

(17) If reference-count of $V_i$ is NQ CAR(y), go to (19).

(18) Change $V_i$ from lexreg-candidate to LEXREG; set y to CDR(y); go to (14).

(19) Set i to i - 1; go to (16).

(20) Bind C to C, PUC to PUC; set i to 1; bind CF to CF, PUCF to PUCF.

(21) If i GR k, go to (24).

(22) Compile (i.e., count registers for) $V_i$; if $V_i$ is LEXREG, then:
if REAL, set CF to CF-1; set PUCF to MIN(CF PUCF);
if not REAL, set C to C-1; set PUC to MIN(C PUC).

(23) Set i to i + 1; go to (21).

(24) Compile S.

PICKEM: Args: (x y)

(1) Set x to NOFF(LENGTH(x)-LENGTH(y) x).

(2) If x is NIL, return NIL.

(3) If there exists in x an element whose value is less than the corresponding element in y, set x to CDR(x) and go to (2).

(4) Return x.

5.2 BIND FORM FOR CODE GENERATOR PASS

(BIND ($V_1$ ... $V_k$) S)

(1) Bind PUC ← PUC, PUCF ← PUCF; set i ← 1.

(2) If i > k, go to (7).

(3) If $V_i$ is lexreg-candidate, then:
if type is REAL and PUCF > $TFC^*_{subsequent}$, or
type is not REAL and PUC > $TRC^*_{subsequent}$, make $V_i$ LEXREG.

(4) Compile $V_i$.

(5) If $V_i$ is LEXREG, set PUC ← PUC - 1.

(6) Set i ← i + 1; go to (2).

(7) Compile S.

5.3 IPLUS AND APLUS FORMS FOR REGISTER COUNTER PASS

(IPLUS $A_1$ ... $A_k$) k > 1

(APLUS $A_1$ $A_2$)

(1) Bind double = off, PUC ← PUC, $TRC_m$ ← 0.

(2) Compile $A_i$ (pass 1).

(3) If $TRC_i \geq$ PUC, go to (7).

(4) If $TRC_i = TRC_m$, set double on; go to (6).

(5) If $TRC_i > TRC_m$, set double off; TRCM ← $TRC_i$.

(6) If i = k, go to (12); else i ← i + 1 and go to (2).

(7) $TRC_m$ ← $TRC_i$, PUC ← $TRC_m$ - 1.

---

*TFC subsequent is the maximum value found by adding to the value of TFC for the subsequent binding (or statement) the number of LEXREG bindings intervening between it and $V_i$.

(8) If i = k, go to (13).

(9) Set i ← i+1.

(10) Compile (count registers for) A (pass 1) in context of PUC.

(11) If $TRC_i > TRC_m$, go to (7); else go to (8).

(12) If double = on TRC ← $TRC_m + 1$.

(13) Set AF ← FALSE, AR ← ARB.

## 5.4 IPLUS AND APLUS FORMS FOR CODE GENERATOR PASS

(IPLUS $A_1 \ldots A_k$) $k \geq 2$

(APLUS $A_1$ $A_2$)

(1) Regroup arguments into two lists: $(B_1 \ldots B_n)$ $(C_1 \ldots C_m)$ where $C_i$ are all $A_i$ which have AF = TRUE or AF = COM, and TRC of $B_i \geq$ TRC of $B_{i+1}$.

(2) If no $B_1$, then compile load of $C_1$ to XREG and go to (8).

(3) If no $B_2$, then compile $B_1$ into XREG and go to (6).

(4) If $TRC_1 \neq TRC_2$, then:

(4.1) If $AR_1$ = ARB, then

(4.1.1) compile $B_1$ into partial-sum-accumulator, else

(4.1.2) compile $B_1$, and if there exists $TRC_j$ such that $j \neq 1$, and $AR_j \subset TRC_j$, then

(4.1.2.1) move $AR_1$ into partial-sum-accumulator,* else

(4.1.2.2) call $AR_1$ partial-sum-accumulator.

---

*Partial-sum-accumulator = if XREG is a LEXREG or there exists $FRC_j$ such that $j \neq 1$, and XREG $\subset FRC_j$, then last arbitrary register allotted, else XREG.

(4.2) If partial-sum-accumulator ≠ XREG and there exists $AR_i$ = XREG or $AR_i$ = ARB ($i \neq 1$), then move $B_i$ to end of list.

(4.3) Compile $B_2 \ldots B_{n-1}$ (if any) into:
if AR is arbitrary, then the last arbitrary reg allotted it,
else the fixed register it desired.

Add the results to partial-sum-accumulator as they are encountered.

(4.4) If partial-sum-accumulator = XREG,

then compile $B_n$ into last-arb-reg-allotted it, and add to XREG,

else compile $B_n$ into XREG and add partial-sum-accumulator.

(4.5) Go to (5), else

(4.6) If $TRC_1$ = PUC, then

(4.6.1) if there exists $B_i$ such that $TRC_i$ = PUC and $AR_i$ ≠ ARB, then

compile $B_i$; remove it from the list;
if something already pushed,
then add to it,
else start push; go to (4)
else set i to 1 and go to (4.6.1); else

(4.6.2) if there exists $B_i$ such that $TRC_i$ = $TRC_1$ and $AR_i$ = ARB, then

(4.6.2.1) $TRC_i \leftarrow TRC_i+1$; move $B_i$ to head of list and go to (4.1.1), else

(4.6.2.2) if there exists $B_i$ such that $TRC_i$ = $TRC_1$ and $AR_i \neq$ XREG, then go to (4.6.2.1);

else set $i \leftarrow 1$ and go to (4.6.2.1).

(5) If pushed sum, add it to XREG.

(6) If no $C_1$, exit.

(7) Add $C_1$ to XREG.

(8) Add $C_2$ through $C_m$ (if any) to XREG.

(9) Exit.

5.5 FNCALL FORM FOR REGISTER COUNTER PASS

(FNCALL full-type form-name $arg_i$ ... $arg_k$)

(1) From the full-type, determine and mark those arguments to be passed in registers and determine the number of registers required to hold the value.

(2) Set TRC ← Set FRC ← Max (argument-register-count value-reg-count).
Set TFC ← Set FFC ← Set PUCF ← 4.
Set PUC ← MAX (PUC TRC).

(3) Compile all arguments not being passed in registers.

(4) Bind PUC ← 0 PUCF ← 0 to compile all args being passed in register.

(5) Set PUC ← MAX (PUC $TRC_i$ (for all args passed in regs)).

5.6 FNCALL FORM FOR CODE GENERATOR PASS

(FNCALL full-type form-name $arg_1$ ... $arg_k$)

(1) Compile all arguments not being passed in registers onto the pushdown stack in sequence.

(2) Form a list of those arguments whose AF is FALSE.

(3) If none in list, go to (6).

(4) If any argument requires all the registers currently available, compile it now. (Destination is the stack if another argument requires all the registers or the use of the register in which this argument will be passed. Otherwise, compile into its proper argument register.) Remove this argument from list and go to (3).

(5) Choose the argument from the list that has the highest FRC and compile it next. (Destination is the stack if the other argument on the list requires the use of the argument register; otherwise compile to target register.) Go to (3).

(6) If any register-passed arguments are on the stack, compile loads for them.

(7) Compile loads for those arguments whose AF was not FALSE.