

UC Berkeley Computer Science

Subject: Some Ideas About Notation
Date: September 13, 2005
From: Ivan Sutherland
UCIES #2005-is06

References:

UCIES# 2005-is02: FLEET – A One-Instruction Computer, Ivan Sutherland, 24 August 2005
UCIES# 2005-is03: Defining Some SHIPs, Ivan Sutherland, 24 August 2005
UCIES# 2005-is04: A Dozen Problems, Ivan Sutherland, 6 September 2005
UCIES# 2005-is05: Notation Questions, Ivan Sutherland, 12 September 2005

PURPOSE

This memo records some agreements made in class on 12 September. It also offers some ideas about comments and renaming that might be useful.

Reader feedback is important to me. Send email to ivan.sutherland@sun.com.

I thank the students of CS 294-4 for the start they provided. I'm counting on them to improve this document. I also thank my associate, Igor, and my son, Dean, for their early critique and suggestions.

PRINCIPLES

Programs get read lots more than they get written. Readability is, therefore, more important than easy composition.

This language is likely to grow over time and get more complex. Let's do the simple things well so we have a sound foundation on which to build.

COMMENTS

It's tempting to adopt the comment conventions from Java, but I suggest a slight modification that uses `/* */` only as a way to put a short comment inside a line. To "comment out" code one must put `//` on every line. Any smart editor can do so easily.

`//` marks a comment to the end of the current line

`/*` marks a comment to either at the next `*/` or to the end of the current line

`*/` without a preceding `/*` is an error.

MOVE INSTRUCTIONS

Because all instructions are MOVE instructions, an infix notation seems better than endless repetition of the word "move." "Guzzinta" folks dominate the 294-4 class rather than "gets" folks, so we'll use the right pointing arrow as our infix. We can spell it in

This document is a product of a collaboration between Sun Microsystems and the University of California at Berkeley.. The ideas contained herein are freely available for any academic purpose.

any convenient way, including minus greater, or a single or double right arrow:

-> or **→** or **⇒**

The source, of which there must be one and only one, appears on the left and the destinations on the right. We don't yet know how many destinations the hardware will give to each MOVE instruction, called the "fan-out limit." Nor do we know how code can replicate data to expand the hardware fan-out limit. Nevertheless, let us assume that the assembler can map statements with any number of destinations into multiple MOVE instructions if necessary. Thus, any number of destinations are permitted in source code. Destination names may be separated by commas or white space. An instruction may be terminated by semicolon or white space. For example:

```
tom -> pete, george, david; mary → sue, jane;     // two instructions
tom -> pete george david mary -> sue jane       // also two instructions
```

The two instructions in the second line are distinct because **mary**, followed by **->** must be a source. I very much prefer the upper form because it is easier to read, albeit verbose.

SYMBOLS

Names for sources and destinations may be any combination of upper or lower case letters, numbers, square brackets and decimal points including at least one letter. Upper case letters are distinguished from lower case letters. White space and punctuation marks other than square bracket and period are forbidden in names.

Names may apply either to SHIPs or to their inputs or outputs or to a particular input or output of a particular ship. If a SHIP's inputs or outputs are separately named, the period should be used as separator with the SHIP name first. For example:

```
ADDER.inputX   or   COMPARE.input[1]   or   READ[2].tokenIN
```

As used above, **input[1]** is the destination corresponding to the first input of the COMPARE SHIP, as opposed to other inputs, e.g. input[2]. Similarly **READ[2]** refers to the second READ SHIP, suggesting that there are more than one. However, these indices are each merely a part of a name and are NOT numbers that we can calculate.

CODE BAGS

Code bags are contained within curly brackets **{ }**. Code bags may extend over as many lines of text as necessary. A semicolon after the closing curly bracket is optional, as is the semicolon before the closing curly bracket.

Code bags may be named by preceding them with a name followed by colon. For example, here is a code bag called "LOUISE" that contains three instructions:

```
LOUISE: { a -> B; c-> d, e; pete -> john, tom; } // three instructions
```

Curly brackets may be nested. Nesting serves two purposes. First, it confines the lexical scope of names. A name defined within curly brackets overrides the same name as defined outside the brackets. A name defined only within curly brackets is undefined outside the brackets.

Empty code bags are allowed and may be useful for lexical scoping. In particular, surrounding any valid program with an additional matching pair of curly brackets should not change the meaning of the program.

The second purpose for nesting curly brackets is to define code bags in place. The name of an inner code bag is valid only within the code bag in which it appears, i.e. within the surrounding curly brackets. For example in:

MARY: { a -> B; sue:{ x -> y, z;} c -> d, e; sue -> john, tom;}
code bag **MARY** includes exactly three instructions and the inner code bag **sue**. The instruction in code bag **sue**, namely **x -> y, z;** is not a part of **MARY**. The last instruction in code bag **MARY**, namely **sue -> john, tom;** uses the code bag descriptor of **sue** as a literal and moves that code bag descriptor into destinations **john** and **tom**. Moreover, the definition of code bag **sue** could follow rather than precede its use:

MARY: { a -> B; c -> d, e; sue -> john, tom; sue:{ x -> y, z;} }

Unnamed code bags are allowed. Thus the code bag **MARY** might have been written without naming code bag **sue**.

MARY: { a -> B; c -> d, e; { x -> y, z;} -> john, tom;}
which still contains three instructions, the last of which moves a descriptor of the unnamed code bag to destinations **john** and **tom**.

Please note, however, that the curly brackets of unnamed code bags in sequence may serve only for lexical scoping and carry no other meaning. For example:

ANN: { a -> B; {c-> d, e; } pete -> john, tom; } // three instructions
has exactly the same meaning as

ANN: { a -> B; c-> d, e; pete -> john, tom; } // same three
albeit different scoping, because the inner curly brackets provide lexical nesting only.

Question: Do we need some additional way to distinguish the literal use of **sue** from the use of a source that happens to have the same name? Remember the **sue** may have another meaning outside code bag **MARY**.

How code maps into instruction memory is totally divorced from the sequence and nesting of the code bag definitions. The sequence of code in memory is divorced from the program listing because all instructions in each code bag are concurrent. The location of the code is divorced from the definition sequence and nesting because the code in a bag is delimited entirely from outside the bag; a code bag descriptor tells both where to find the code bag and how big it is.

Divorcing the sequence and nesting of code bags in memory from the program listing frees the assembler to reuse code. The assembler is free to implement a code bag by reusing any matching code it finds anywhere. If a code bag defined inside code bag X has the same implementation as that of a code bag defined inside code bag Y, the compiler may allocate only a single implementation of that code for both uses. Moreover, the assembler is free to use any sub-part of the code for one code bag as the code for another code bag or to overlap parts of code bags at will. Most interesting of all, the assembler is free to compact code further by rearranging the sequence of instructions in any code bag to make some part of that code fit additional uses. There's a nice code-packing task here for the assembler.

DECLARATIONS

Feel free to declare a fresh name for any SHIP or any input or output of a SHIP or any combination of a SHIP name and input or output name. Thus if one is tired of writing:

ADDER.inputX

one can instead use a renaming statement:

tom RENAMES ADDER.inputX ENDRENAME;

terminated by an optional semicolon. You may substitute the shorter name for the longer

string anywhere in the bag containing the **RENAME** statement. This is a lexical substitution only, strictly a character string substitution, but defines only full fledged names that serve as parsing tokens. Thus the following string concatenation is disallowed:

qqq RENAMES COMPARE.in ENDRENAME;

qqqput[1] is meaningless and does **not** mean **COMPARE.input[1]**

Similarly, this sequence is also disallowed:

qqq RENAMES COMPARE.input ENDRENAME;

qqq[1] is meaningless and does **not** mean **COMPARE.input[1]**

because **[1]** is just a character sting as is **put[1]**.

The impact of **RENAME** statements is felt only within the bag containing the statement. **RENAME** statements made prior to the first opening curly bracket apply to the entire text.

MORE STUFF

There are lots of things we need that don't appear in this document. For example, we will want assembly-time arithmetic to calculate constants of various kinds. Thus we must be able to define numeric values and do ordinary arithmetic on their values. I've omitted further comment in this document to let it focus discussion on the things that are unique to FLEET.