Programming in the GalaxC Language

John F. Beetem

Chapters 1-8 are derived from *The Galaxy Programming Language* by John F. Beetem, with changes and new material. Chapters 9-10 and Appendix A are new.

Revision 0.0c, 6 November 2011.

Note to the Reader

Thank you for your interest in the XXICC and GalaxC research projects. This document is the official reference and user guide for the GalaxC programming language. It assumes you have previous programming experience. Since many GalaxC features are derived from the C programming language, it is most helpful to have C experience but not required. Experience with object-oriented languages like C++ and Java is not necessary. GalaxC has some features that correspond to object-oriented languages, but GalaxC uses quite different notations and terminology.

This document is somewhat unusual for a programming language book since it is trying to do several things at once. It is primarily the reference for the GalaxC language, but tries to provide enough tutorial examples so you can get started. But it also provides considerable detail of how GalaxC's compiler is implemented. This is necessary for an extensible language, since to add your own features to the language you need to understand how the default features are described and implemented. Plus, GalaxC is not implemented like a conventional language: many features that are built into a conventional language's compiler are viewed as language extensions in GalaxC. In principle, the entire language can be built as mutually-recursive language extensions, though it is not currently implemented that way.

GalaxC is a research work in progress, and there are plenty of rough edges at this time. These are being addressed by the author, with priority given to those things he needs most. Users must realize that at the present time GalaxC is still in the experimental stages and while most things work, it has never been stress-tested or even tested by anyone other than the author. You are very likely the second GalaxC programmer. Thus GalaxC comes "as is" with no warranty whatsover. But that doesn't stop GalaxC from being a lot of fun.

John F. Beetem, May 2011

Acknowledgements

The author would like to renew thanks to the talented individuals who helped make the original version of Galaxy circa 1988. Foremost he wishes to thank Anne Beetem for her ideas, inspiration, support, and scholarly collaborations. He would also like to thank Jong-Min Park and Jim Rose for their contributions to the original implementation of the Galaxy compiler and its environment, and Monty Denneau for asking The Question which led to the original conception of Galaxy.

The author would also like to thank his family and friends for their support and encouragement over the years which led to GalaxC and XXICC reaching this point, with special thanks to Joni Beetem. He would also like to acknowledge the literary inspirations of Miguel de Cervantes and Edmond Rostand towards putting Quixotic idealism ahead of practicality, and Kurt Vonnegut for *Cat's Cradle.*

Revision History

8 August 2011: Revision 0.0a added unary operator '`?`' and binary operators '`&?`' and '`!&`'. It also added single character relational symbols '≡', '≠', '≤', and '≥', which are equivalent to '`==`', '`!=`', '`<=`', and '`>=`'.

27 August 2011: Revision 0.0b removed the `MULU` operator from PSI §6.3.1 and changed pointer arithmetic and array access to use a `ulong` index (§8.3.3 and §8.4). It also added type conversions `int(x)` along with the `signed` and `unsigned` operators (§6.4.8).

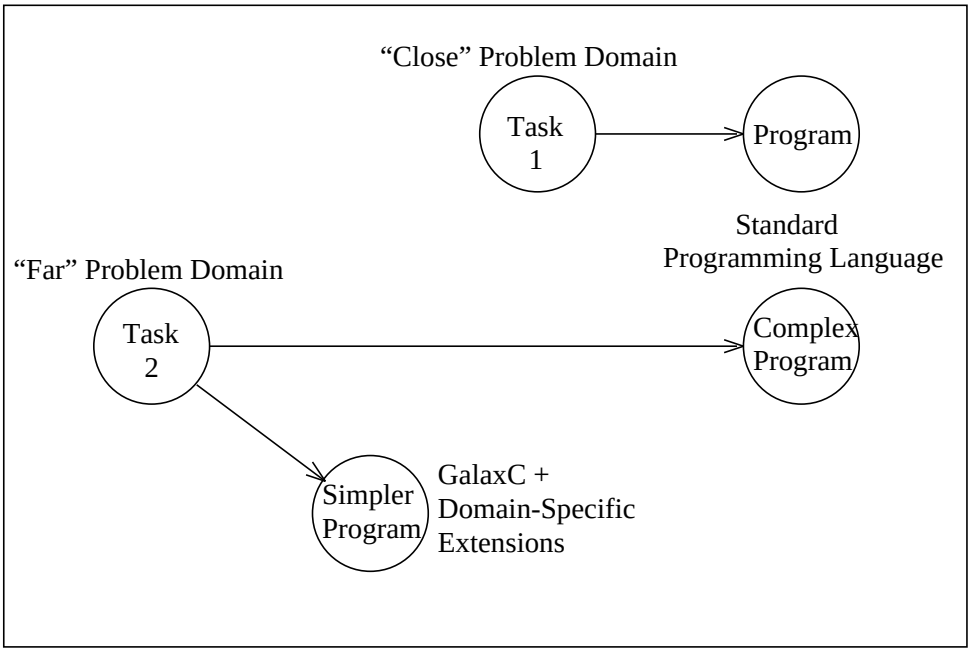
6 November 2011: Revision 0.0c makes major changes to variable attributes -- `const`, `volatile`, etc. -- with details in new Appendix A. Vattr `register` has been removed from the main text. Variable attributes are no longer reserved words, so there are minor changes to token and expression syntax (§2.6 and §3.3). Type `string` is now read-only. For writable strings, use `wstring` (§8.4.1). PSI now has opcodes for `volatile` load, store, and increment (§6.3.1, §6.3.2 and §6.4.11).

Chapter 1
Introduction to XXICC and GalaxC

Krazy Kat: Why is "Lenguage", Ignatz?
Ignatz Mouse: "Language" **is**, that we may understand one another.
KK (pensive): Is that so?
IM: Yes, that's so.
KK (suddenly agressive): Can you unda-stend a Finn, or a Leplender, or a Oshkosher, huh?
IM (cowed): No.
KK (more agressive): Can a Finn, or a Leplender, or a Oshkosher unda-stend **you**?
IM (more cowed): No.
KK (now relaxed): Then I say lenguage **is** that we may *mis*-unda-stend each udda.
[George Herriman, *Krazy Kat*, 1920]

XXICC (*21st Century Computing*) is an attempt to bring software design into the 21st Century using an improved programming language and a Reduced Software Complexity philososphy [JFB 11: Reducing Software Complexity]. Its goal is to improve software productivity by a factor of 2-10 (or more) by making it easier and more fun to write and maintain software. XXICC is pronounced "Chicken Coop", so-called because it has so many layers.

The purpose of software is to tell a computer how to perform a task. The task is initially conceived of and described using the terminology of the *problem domain*, which may be computer graphics, electronic CAD, data communications, numerical analysis, business, etc. Then the task must be coded into a form that the computer can understand, usually a standard programming language such as C, C++, or Java. If the problem domain notations and concepts are close to the programming language -- as with Task 1 in the figure -- this is relatively straightforward. On the other hand, if the problem domain is far from the language (Task 2), coding often ends up being *encryption* where clean problem domain notations must be twisted to fit the constraints of the language. The wider the gap, the harder it is to write and maintain software.



XXICC's **GalaxC** programming language tries to narrow the gap between problem domain and programming language by allowing programmers to *extend* GalaxC by adding problem domain notations. Instead of

adapting Task2 to the language, you adapt GalaxC to Task2, resulting in a simpler, easier-to-maintain program.

GalaxC programs may consist of ordinary ASCII characters and white space, like C. However, GalaxC programs may also have executable tables, comment blocks containing formatted text and figures, variable names in different fonts, special symbols, string literals containing formatting, mathematical formulas, and WYSIWYG dialog boxes. This eliminates the need for separate documentation files (which are very hard to keep synchronized with a program) as well as a separate "resource editor".

These are all edited using the XXICC Object Editor (XOE), a unified program and document editor which combines the features of a document editor, spreadsheet program, figure editor, dialog box editor, and more into a small, easy-to-use program with a consistent user interface. XOE is the front end of an integrated development environment (IDE). XOE is written entirely in GalaxC and is used for editing all XXICC software and documentation. 21st Century Computing believes in the "take your own medicine" approach to software engineering.

1.1. The GalaxC Programming Language

GalaxC is a continuation of the Galaxy Programming Language [B&B 89, JFB 91], an experimental extensible programming language initially used for writing electronic computer-aided design tools. To provide the flexibility to add problem domain notations, Galaxy introduced *separation of syntax from semantics*. In most computer languages, there is a one-to-one correspondence between language features and their implementation. For example, variables are always denoted by simple identifiers: `Cat`, `Dog`, `VeryLongVariableName`, etc. Array access is always denoted using a bracket notation, e.g., `A[i,j]` or `A(i,j)`. Structure access is often denoted using a dot notation, e.g., `z.Re` and `z.Im`. Function calls are denoted `F(x,y,z)` whether or not that's a good notation for the problem domain. This simplifies life for the compiler, since it can determine the kind of object by merely looking at the object's syntax. Unfortunately, this is precisely why typical programming language notations are inflexible.

To avoid this restriction, Galaxy expanded the syntax of program components such as variables, types, function calls, array access, and record access in a very simple way: *any of these can have of any syntax that is legal in Galaxy*. This gives us a vast range of possibilities for program constructs, e.g.,

variables: V_{CES} V_{th} y_0 x^0

constants: `kt/q` $e^{i\ pi/4}$ K_{blue}

functions: e^x $\sin^2 x$ `z1+z2` `insert x into sorted list L`

macros: a_i `x + i y` `Re z` `Im z` `a[i, j]`

For example, we can define the notations "`Re z`" and "`Im z`" to access the fields of a complex number, and the notation "`z1+z2`" to call a function which adds two complex numbers. The same notation "`z1+z2`" can also be used to add two `Points`: Galaxy uses the types of the arguments to determine which function to call.

Galaxy's philosophy is that users teach the compiler their notations instead of encrypting their problems for the convenience of the compiler. Galaxy's approach does require the compiler do more work: since a given expression can be any kind of program construct, Galaxy must *pattern match* each expression against all possible function, macro, and variable definitions. This is more complex than analyzing a C program, but can be done very efficiently using an incremental compiler. Computers are now much faster than when Galaxy

was developed and we no longer need an incremental compiler for reasonable size programs.

Galaxy's second important contribution is *orthophrase extensibility*, which is the ability to add new kinds of constructs to the language beyond functions and macros [BBP 92]. Galaxy provides *special functions* which it calls at compile time instead of at run time. Special functions have access to compile-time variables and can define how to generate code for a new construct. Galaxy defines basic constructs such as *if-then-else*, function calls, and type definitions as special functions: Galaxy is actually defined at its core as a set of mutually-recursive special functions and primitive types. With special functions, the programmer has -- in principle -- the same power as the Galaxy compiler's writer.

GalaxC is a new language which borrows most of its concepts from Galaxy, but is closer to the syntax and semantics of the C Programming Language [K&R 78]. Specifically, GalaxC expression and comment syntax is almost the same as C, and code produced by GalaxC can be optimized in the same way. There are some important differences between GalaxC and C:

1. GalaxC control structures are modeled after Pascal rather than C: it uses *while-do* and *repeat-until* in place of *while()* and *do-while*. GalaxC loops use `break` and `continue` the same way as C.

2. GalaxC has a *switch* statement similar to C, but does not require the infuriating `break` for each clause. Instead, it provides a `fall through` statement to continue to the next case.

3. GalaxC macros are very different from C. C expands macro in a text-based pre-processor, while GalaxC expands macros as part of the regular compilation process. This allows GalaxC macros to have typed arguments like GalaxC functions so that the same macro syntax can be used for different purposes depending on the types of the arguments. If no macro has the correct syntax, GalaxC generates an error instead of blindly making a text substitution like the C preprocessor. C macros are quite powerful and are an important tool for getting around limitations of the base language. GalaxC macros are powerful in a different way, and are used cleverly to define the language itself. For example, all the looping constructs are defined as macros.

4. GalaxC provides a *for* statement that is the same as C. It also has statements like "`for i = 1 upto 10 do` *stmt*" and "`for i = 20 downto 0 by 2 do` *stmt*".

5. GalaxC allows local variables to be declared anywhere in a block, instead of only at the beginning of the block as in C. Some programmers use C++ instead of C only so they can use this feature.

GalaxC programs can be written entirely with ASCII text and white space, and may look very much like C except for the differences just mentioned. Such GalaxC programs can be edited using any text editor. However, the real power of the language becomes apparent when one adds XXICC objects into a GalaxC program using XOE. Here are some examples:

1. Literal text strings can contain formatting such as bold, italic, and underline. One place this is used in the XOE source code is for menu item and button text, for example, "<u>O</u>pen..." instead of a character sequence like "&Open" or "<u>O</u>pen".

2. Variable names can use formatting to distinguish between variables, e.g., "*a*" versus "**a**" or "*e* = 2.718281828".

3. XOE supports the Latin-1 character set, so it is possible to have string literals with accented text such as "Çui-là" or variable/function names with accented characters such as "año = 2009".

4. XOE supports the Symbol font, which permits Greek variable names "π = 3.14159265" and expressions with proper comparison operators "*while i ≤ 10 do...*".

5. XOE allows subscripts and superscripts, which can then be included in GalaxC programs. Galaxy also had this capability.

6. At an even later point XOE, will permit complex mathematical expressions such as [quadratic formula]. This is already supported in GalaxC's parser, but requires implementation in XOE.

7. XOE has paragraph boxes containing formatted text with an optional frame.

   Inserting a PARABOX is a great way to add program documentation as a large comment block with word wrap, text and paragraph styles, and figures.

8. XOE tables allow formulas to be entered and calculated as spreadsheets. Unlike a stand-alone spreadsheet, XOE tables can share variables and functions with the rest of a GalaxC program.

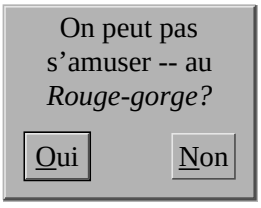
   Here is a simple table that calculates squares of numbers:

i+1	(i+1)*(i+1)
"	"
"	"
"	"

   "i" is the row, numbered from 0. A cell with a "ditto" mark uses the same formula as the cell above it. Compiling and executing the table results in:

1	1
2	4
3	9
4	16

9. XOE supports WYSIWYG dialog box editing so that dialogs can be incorporated into a GalaxC source file instead of being in a separately-compiled resource file.

Most modern programming languages use ASCII characters and white space. This was required by the technology available for source code entry at the time the language was invented, viz., punched cards and teletypes. FORTRAN originally required "<" and "≥" to be entered as ".LT." and ".GE." because the "<" and "≥" characters were not available on IBM 026 keypunches. Given that FORTRAN was created by mathematicians to translate formulas, doesn't it make sense that they would have used proper mathematical

notations such as real subscripts and superscripts if they were available instead of “A(I,J)” and “X**3”?

GalaxC recognizes that times have changed since the 1960’s and 1970’s and that technology no longer requires expressions to be encoded as ASCII strings. The fact that old languages still require it and new languages for the most part still have teletype compatibility is (in the author’s opinion) silly and sad .

The key to GalaxC’s flexibility is that XOE documents are stored internally as XXICC Object Trees, which will be described in the next section. Since they are already in tree form, GalaxC does not have to parse the tree structure itself, though it does need to parse textual expressions within the tree. Tables and dialogs are treated as statements.

1.2. XXICC Object Editor (XOE)

Text-only GalaxC programs can be edited using any ASCII text editor. However, to take full advantage of GalaxC you must use XOE so you can create and edit tables, dialogs, and figures.

Traditionally, personal computers have a “suite” of productivity tools consisting of a document editor, a spreadsheet program, and a drawing program. For example, Microsoft Office has Word, Excel, PowerPoint, and Visio. The OpenOffice suite has Writer, Calc, Impress, and Draw. While there is a capability to cut and paste from one tool to another, it’s awkward and requires that documents be edited separately from drawings. In addition, while each tool has a similar user interface there are significant differences in capability from one tool to another. For example, the author was able to create accented words easily in Microsoft Word, but not able to do so in PowerPoint. Yes, he could have cut and pasted from Word, but that gets quite cumbersome.

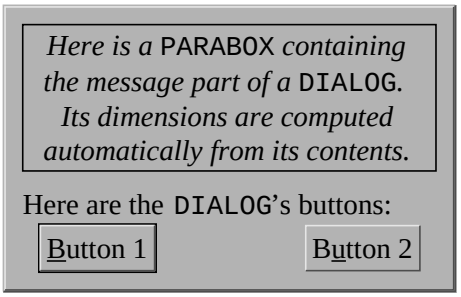
It’s unclear to the author why after so many years these are still separate tools. It made sense to have separate MacWrite and MacDraw tools on a 1984 Macintosh with 128KB of RAM and floppy disk storage, but that’s clearly no longer necessary. It also made sense for Microsoft Word and Excel to be separate programs originally since they were purchased from different companies. However, that was decades ago and you would think a company with as many employees as Microsoft could have a few of them integrate the tools. Perhaps in the case of Microsoft they want to have the flexibility to sell single tools as well as suites. More likely it’s just that the programs have become so complex that nobody knows enough about two of them at once to be able to integrate them.

XOE takes a different approach: all the necessary capabilities of a word processor, spreadsheet, and drawing editor are integrated into a single tool with a uniform user interface. Exactly the same commands are used to edit formatted text in documents, tables, and figures.

The key to making this work is to reduce the complexity of the software down to the basic features that are truly needed. Over the years productivity suites have become increasingly complex, largely for no good reason. The author’s own experience is that new versions do not add any new features he really needs, and often screw up existing features that he does need. In many cases, it seems that the new features are there solely so there is a reason “on paper” to upgrade to the new version. Technology magazines are complicit in this, since they tend to celebrate the new features without considering whether they are useful or merely “newsworthy”. The author’s own experience is that LaTeX was quite sufficient for his own documents except for two omissions: WYSIWYG editing and incorporation of figures. XOE solves both of these problems.

There are some users who want all the latest features and enjoy using software that is the equivalent of flying a jet airliner with 100,000 switches. That’s fine, there are plenty of commercial and FLOSS alternatives to support them. XOE follows a Reduced Software Complexity philosophy [JFB 11: Reducing Software Complexity], which targets an entirely different audience.

The key to XOE's simplicity is that it treats all documents internally as *XXICC Object Trees* (XOTs), built out of *XXICC Objects* (XOs). An XO may be a primitive such as TEXT or RECT, or else is an *XO Subtree* (XOST) containing XOs that may be nested XOSTs. An XOST may be a large object such as a TABLE, DIALOG, PARABOX (a rectangle containing formatted paragraphs), or FIGURE. These in turn may contain smaller XOSTs such as horizontal and vertical stacks (H/VSTACKs) which stack their components horizontally or vertically. For example, a DIALOG may stack a message contained within a PARABOX on top of an HSTACK containing several BUTTONs. Each BUTTON is a special kind of HSTACK that includes text (usually with an underlined character) along with an optional check box.



Similarly, a menu is a VSTACK of BUTTONs, each with an associated action. A complex mathematical expression may also be an XOST. For example, a ratio may be shown as a VSTACK with a horizontal separator line between textual numerator and denominator expressions.

Basic editor operations such as select, insert, delete, copy, and paste are essentially the same whether they're applied to TEXT or to non-textual XOSTs such as dialog boxes and tables.

While this takes care of editing all sorts of objects, spreadsheets add the additional complexity of compiling and executing code. XOE's approach to this is to integrate XOE tables with the GalaxC compiler so that all the capabilities of GalaxC, including nice mathematical formulas [some day] are available within spreadsheets. The syntax of XOE formulas is very different from VisiCalc and its successors and takes a very different approach to naming cells so that XOE tables integrate with the rest of a GalaxC program.

1.2.1 Decoded XXICC Objects (DXOs)

A XOE document is a sequence of XOs, i.e., an *XO list*. As mentioned earlier, XOs may be primitives such as TEXT and RECT, or may be subtrees bracketed with an XOST opcode and ENDOBJ. Except inside FIGUREs, XO lists do not usually contain placement information (x/y coordinates), but they do contain formatting instructions encoded as control characters in TEXT strings or as XOs. XOE produces a formatted version appropriate to the output device using well-known document formatting algorithms used by any document editor. The XO list also contains data for reversing changes using the UNDO command, i.e., it marks XOs that have been deleted (so they can be restored) and XOs that have been inserted (so they can be deleted).

Formatting a large document from scratch takes considerable computing time. It's not a good idea to reformat each time part of a document needs to be refreshed on the screen. So XOE transforms an XO list into a *Decoded XO list* (*DXO list*) which is easy to render on the screen. For example, a TEXT XO may contain many lines of text with various formatting styles. Decoding this results in multiple DXOs, each confined to a single line with a uniform formatting style (font, size, color). DXOs also have placement coordinates so they are easily rendered.

A large document can have a huge number of DXOs. There is no point in saving all of them, since only a fraction will be visible in a window or on a single printed page. XOE therefore *abridges* the DXO list, omitting DXOs that are not currently visible. However, it retains enough information so that if scrolling exposes DXOs they can be regenerated incrementally instead of reformatting the entire document from scratch. Abridgement follows the tree structure of the DXO list, i.e., XOE can omit the contents of a subtree if that entire subtree is currently invisible. This approach is loosely based on outline editors with elision capability and also on incremental processing algorithms in general.

1.3. The Postfix Stack Intepreter (PSI or Ψ)

PSI is a 32-bit stack-oriented interpreter that executes PSI-code, a portable 16-bit representation similar to Pascal p-code and Java bytecode. The GalaxC compiler generates PSI-code, which can either be interpreted or translated into native code using the PSI code optimizer (PSI-CO) and PSI code generator (PSI-*x*CG, where *x* = architecture). Since PSI-code represents both code and data, it is also used as the internal representation for XOTs by augmenting the basic PSI instruction set with instructions representing each kind of XO. For example, the `TEXT` instruction forms a `TEXT` XO using the string operand at the top of the stack as string data. This operand may be a string literal, or it may be the value of a string expression, e.g., the current date or time. An XOST begins with an XOST opcode, e.g., `HSTACK`, and ends with the `ENDOBJ` instruction. An entire document is a paragraph box beginning with `PARABOX` and ending with `ENDOBJ`.

To process a XOE document, PSI simply executes the PSI code for that document. XO opcodes are interpreted using a vector of function pointers, with a different vector for each kind of processing. PSI also has instructions to tag incremental changes to the memory copy of a XOE document so they can be reversed using the UNDO command. These tags are cleaned out when storing a XOE document to a file. XOE documents in files are “data only”, i.e., they only contain PSI data instructions and XO opcodes. Otherwise it would be easy to transfer a virus as part of a XOE document.

Using a single representation for code and data is an old idea, mostly explored using LISP. The Galaxy CAD System [JFB 92] used the executable data concept extensively for processing CAD data. A particularly nice example was the DEC GT40 Graphics Display System [DEC 73] which represented both instructions and graphics data as 16-bit words which were interpreted by hardware to produce vector graphics.

1.4. Summary

This chapter provided a brief overview of XXICC and what’s different about the GalaxC language. The remaining chapters describe the syntax and semantics of GalaxC in detail.

Chapter 2
Tokens

> Good spelling is very important. I knew a man who spent a night in a Warehouse because he couldn't spell.
>
> *Mark Twain*

In this chapter, we define and discuss **tokens**, the smallest meaningful program units in the GalaxC language, and the separators used to delimit them. Tokens include such objects as numbers, identifiers (alphanumeric names), quoted characters, quoted strings, and special characters such as ':', '+', '<', etc. Tokens correspond to the words and punctuation marks in written text, and are combined to form larger structures called **expressions**.

Separators prevent consecutive tokens from being erroneously merged into a single token and also to make program text more readable. They include spaces, tabs, new-line characters, and comments. In many cases, separators are not actually needed to delimit tokens: for example, the string `a+b` is considered to be three tokens `a`, `+`, and `b`. A program can be viewed equivalently as a sequence of characters or as a sequence of tokens and separators.

There are six kinds of tokens in the current version of GalaxC. These are integer numbers, identifiers, characters, strings, floating-point numbers, and special tokens. Numbers, characters, and strings are also called **literals**. For the most part they are the same as in C. In principle, tokens can be any length. In practice, implementations limit token length to some reasonable value. Currently GalaxC limits tokens to 255 characters.

2.1. Numbers

Numbers are unsigned integers containing one or more digits. As far as the scanner is concerned, numbers can contain any number of digits up to the maximum token length. In practice, the number of digits is limited by the word size of the machine. GalaxC numbers can be decimal, octal, binary or hexadecimal and mostly use C syntax:

[fn: {*X*} means any number of instances of *X*, including zero. [*X*] means *X* is optional. *X* | *Y* means to use either *X* or *Y*. (*X*) is used for grouping.*]*

digit = `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
dec_num = *digit* {*digit*} [First decimal digit ≠ 0].

oct_digit = `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`
oct_num = `0` {*oct_digit*} [First octal digit = 0].

bin_digit = `0` | `1`
bin_num = '`0b`' *bin_digit* {*bin_digit*} | '`0B`' *bin_digit* {*bin_digit*}

hex_digit = *digit* | `A` | `B` | `C` | `D` | `E` | `F` | `a` | `b` | `c` | `d` | `e` | `f`
hex_num = '`0x`' *hex_digit* {*hex_digit*} | '`0X`' *hex_digit* {*hex_digit*}

number = *dec_num* | *oct_num* | *bin_num* | *hex_num*

Note that hexadecimal digits and the `0x` and `0b` prefixes may be upper or lower case.

Valid examples: `0`, `1234`, `042` (octal), `0xBeef` (hexadecimal), `0b01001101` (binary).

Invalid number tokens:

`-4`, `+32`	Signs are not part of number tokens.
`1.5`, `6.023E23`	Not integers.
`123A7`, `0afb`	Letters not allowed in decimal and octal numbers.
`0x`, `0b`	Need at least one hexadecimal or binary digit.
`0x02gh`	Invalid hexadecimal digits.

In addition, underscore ('_') characters may be used after the first digit to group digits into a more readable form, e.g.,

`0xfeed_beef`, `0b0110_1001`, `123_456_789`

Underscores do not affect the value of a number and are not checked in any way.

2.2. Identifiers

Identifiers are used to name objects such as variables, arguments, and function names. An identifier consists of a letter or underscore followed by zero or more letters, digits, and underscores. GalaxC is *case-sensitive*, i.e., lower case and upper case letters are considered to be distinct. For example, the identifiers `Cat` and `CAT` are different. Identifiers can have any length up to the maximum token length and all characters are significant. The syntax of an identifier is the same as C:

letter = `A` | `B` | ... | `Z` | `a` | `b` | ... | `z` | `_`
identifier = *letter* {*letter* | *digit*}

Valid examples: `x`, `y`, `A123`, `Very_long_identifier`, `_exit`

Invalid Identifier Tokens:

`1ABC`, `$total`	Must start with a letter or underscore.
`Very long name`	Cannot have separators in an identifier.
`a[3]`	'`[`' is not a valid identifier character.

2.3. Characters

A **character** literal consists of 1-4 characters delimited by single quotes, e.g., `'a'` or `'bc'`. A character may be a printing character or a non-printable character such as CR or LF specified as a C *escape sequence*:

`\`*nnn*	*nnn* = 1-3 octal digits, for example `'\015'` is the ASCII code CR = 13 decimal, and `'\0'` is `NUL`.
`\x`*nn*	*nn* = 1-2 hexadecimal digits, for example `'\x7F'` is the ASCII code DEL.
`\X`*nn*	Same as `\x`*nn*.
`\b`	BS = backspace = `'\10'` = `'\x08'`.
`\f`	FF = form feed = `'\14'` = `'\x0C'`.
`\n`	LF = line feed = new line = `'\12'` = `'\x0A'`.
`\r`	CR = carriage return = `'\15'` = `'\x0D'`.
`\t`	HT = horizontal tab = `'\11'` = `'\x09'`.

Any other character preceded by a backslash is itself. For example, `'\''` is a single quote and `'\\'` is the backslash character. Here is the formal syntax for a character token:

char = \ *oct_digit* [*oct_digit*] [*oct_digit*] | \x *hex_digit* [*hex_digit*] | \X *hex_digit* [*hex_digit*]
| \ *any printable character* | *any printable character*
character = ' *char* [*char*] [*char*] [*char*] '

The single quotes and the backslash are considered to be part of the character token, but not its value.

Valid examples: `'a'`, `'b'`, `'#'`, `'\''`, `'\n'`, `'"'`

Invalid character tokens:

`'abcde'`	More than four characters.
`'a`	Closing quote missing.

Character tokens are always integer values of type `char` (if single character) or `ulong` (if multiple). This is different from a string, which is a sequence of single characters accessed through a pointer.

2.3.1 Multi-character Tokens

A character token normally has a single character and fits in an 8-bit byte. ANSI C introduced the idea of a multi-character token, fitting 2 characters in a `short` and 3-4 characters in a `long`. Multi-character tokens are not the same thing as strings (see next section) since they can be assigned as integers and do not have implicit `NUL` termination. However, they are occasionally useful as a way to pass short strings as integers instead of worrying about allocating and deallocating strings.

One aspect of multi-character tokens which is not well defined in C is the placement order of characters in an integer, i.e., should they be placed in MSB to LSB order or vice-versa, or should it depend on the byte order of the target architecture? Since this was not well defined, different C compilers may implement them differently. For example, GNU C always generates multi-character constants in MSB to LSB order: the last character of the multi-byte constant is the LSB.

GalaxC matches the byte order of the target architecture: for a big-endian target (PowerPC, 68000, Coldfire) the last character is the LSB, whereas for a little-endian target (Intel Architecture, most ARM implementations) the first character is the LSB. This puts the characters in the same order as if they were in a string; in fact, if the multi-character constant is stored in memory and ends with a `NUL` it can be treated as a string. This always works for a little-endian target, but with a big-endian target it's usually necessary to pad the character to be exactly 2 or 4 bytes long, depending on whether it is stored in a `short` or a `long`.

Here is how to calculate the value x for an n character token ($1 \le n \le 4$) with left-to-right characters c_1, c_2, c_3, and c_4, assuming 8-bit bytes and unused characters are `NUL`:

- For a big-endian target, $x = (c_1 << 8*(n-1)) + (c_2 << 8*(n-2)) + (c_3 << 8*(n-3)) + (c_4 << 8*(n-4))$.
- For a little-endian target, $x = c_1 + (c_2 << 8) + (c_3 << 16) + (c_4 << 24)$.

For example, `'abc'` is `0x00616263` on a big-endian target and `0x00636261` on a little-endian target.

2.4. Strings

A **string** literal is a sequence of zero or more characters delimited by double quote characters, e.g., `"cat"`. As in C, a string is stored as an array of characters terminated with `NUL` (not shown) and accessed using a pointer. Strings are not the same as characters, even if the string is one character long: `'a'` is different from `"a"`. Strings use the above backslash notation for special characters. Strings can have any length as long as the token including quotes and escape sequences fits in the maximum token length. The syntax of a string is:

string = " [{*char*}] "

Valid examples:

```
"cat", "Hello, world!\n", "He said \"Hello\".",
"He said 'hello'.", ""
```

Invalid String tokens:

`"He said "Hello""`	Use `\"` to put double quote inside a string.
`"Now is the time` `for all good men"`	Use `\n` to put new line in middle of a string.
`"But now for...`	Closing quote is missing.

The double quotes and backslashes are considered to be part of the string token but not its value. When we speak of the length of a string, we normally mean its compiled length after escape sequences have been interpreted. For example, the string `"Hi!\n"` is 4 characters long since `\n` is a single character. String length does not include the `NUL` terminator: the empty string `""` is 0 characters long.

A string literal is of type `string`, which is a read-only array of `char`. For details, see §8.4.1.

2.4.1 String Concatenation

As with ANSI C, you can break up a long string into smaller strings and let the compiler combine them. There can be separators between the individual strings, e.g., you can put them on separate lines:

```
"Now is the time "
"for all good men"
```

is the same as the single string `"Now is the time for all good men"`. C concatenates strings during pre-processing. GalaxC concatenates them using an invisible concatenation operator during analysis and code generation. The current version of GalaxC limits combined strings to 16K characters.

If the strings contain printable characters there are no restrictions on combining them. However, it is possible to create strings of arbitrary binary data using `\`*nnn* and `\x`*nn*. In this case, it can be unclear whether `\0` is the end of the string or a data byte of the string. This requires the following rule:

> If a string containing binary data ends in `\0`, the string must have even length to be concatenated with the following string.

2.5. Floating-Point Numbers

A **floating-point number** must have a decimal point '`.`' to distinguish it from an integer and/or have a base 10 exponent. The exact syntax is:

signed_dec = [`+` | `-`] *dec_num*
exponent = (`e` | `E` | `d` | `D`) *signed_dec*
float_num = *dec_num* [`.` *dec_num*] [*exponent*]

A *dec_num* by itself is a number token (an integer). The exponent characters '`e`' and '`E`' imply a single-precision `float` value, while '`d`' and '`D`' imply a `double` value.

Examples: `1.3`, `6.023E23`, `1.602e-19`, `0.4`, `1D6`

Invalid floating-point number tokens:

`.5`	Must start with a digit.
`1.`	Must have a digit after the decimal point.
`1.5E`	Exponent number is missing.

GalaxC floating point numbers are slightly different from C: they must begin with a digit and if there is a decimal point it must be followed by a digit. In addition, GalaxC allows underscore characters as with integer number tokens.

2.6. Special Tokens

All other tokens are **special tokens**. These include special characters such as '`+`', '`-`', '`*`', and '`/`'; pairs of special characters such as '`<=`', '`>=`', and '`&=`'; a few triples such as "`<<=`" and "`>>=`"; and certain identifiers that are *reserved words*. Any character that is not part of a separator or another kind of token is or is part of a special token.

There are 28 special tokens that are pairs or triples of special characters:

```
==   !=   <=   >=   ++   --   <<   >>   &&   ||   ^^   ->   &?   !&
+=   -=   *=   /=   %=   &=   #=   |=   ^=   :=   ::=  <<=  >>=  ^^=
```

There are 20 **reserved words** in GalaxC. While these look like identifiers, they are treated as special tokens rather than normal identifiers and can never be used as identifiers. They are:

```
and  def  else  for  inline  repeat  spclarg  then     until  while
arg  do   fn    if   or      spcl    switch   typedef  var    with
```

Some reserved words may not be used in the current implementation: they are for planned features.

Since GalaxC is case-sensitive only lower-case identifiers match these reserved words. This means `If`, `And`, and `FOR` are not reserved words -- they are ordinary identifiers.

All other characters that are not part of another kind of token or a separator are *special characters*. They include:

```
+    -    *    /    ^    !    @    #    $    %    &    |    ~    ?
```

```
(    )    ~    [    ]    {    }    <    =    >    ,    :    ;    .
```

In summary, the syntax for a special token is:

reserved_word = `and` | `arg` | `def` | `do` | `else` | `fn` | `for` | `if` | `inline` | `or` | `repeat` | `spcl` | `spclarg` | `switch` | `then` | `typedef` | `until` | `var` | `while` | `with`

special_pair = '`==`' | `!=` | `<=` | `>=` | `++` | `--` | `<<` | `>>` | `&&` | '`||`' | `^^` | `->` | `&?` | `!&` | `+=` | `-=` | `*=` | `/=` | `%=` | `&=` | `#=` | '`|=`' | `^=` | `:=`

special_triple = `<<=` | `>>=` | `^^=` | `::=`

special_char = any non-identifier non-separator character

SPF = style prefix, described later

SCH = style change, described later

XO = XXICC object, described later

special = reserved_word | *special_char* | *special_pair* | *special_triple* | *SPF* | *SCH* | *XO*

2.7. Summary of Tokens

A token is either a number, an identifier, a character, a string, a floating-point number, or a special token. The formal syntax is:

token = number | *identifier* | *character* | *string* | *float_num* | *special*

2.8. Separators

Separators are used to punctuate tokens that would otherwise be considered a single token and are also used to enhance readability of program text. Separators are not needed when token distinction is clear without them. For example, scanning the text "`x+3`" produces the tokens `x`, `+`, and `3` since they are clearly distinct kinds of tokens. It is usually correct to insert separators between tokens: "`x+3`", "`x + 3`", and "`x  +  3`" are lexically equivalent. However, it is incorrect to insert separators inside a token: "`catdogbird`" and "`cat dog bird`" are not lexically equivalent. In a few cases inserting a separator between two distinct tokens changes the meaning. For example, "`f(x, y)`" is different from "`f (x, y)`", and "`a[i]`" is different from "`a [i]`". These will be discussed in Chapter 3.

GalaxC tries to find the longest tokens possible. It scans a number until it finds a non-digit, it scans an identifier until it finds a non-identifier character, and it scans a string until it finds the closing quote. The character that delimits a token can be a separator or the first character of the next token if that is possible. For example, scanning "`10m`" produces the tokens `10` and `m`, since `m` is not a digit. Separators are only strictly required when two tokens would otherwise be joined into a single token, e.g., "`if a then`" requires separators between the three tokens since "`ifathen`" is a single identifier. Inside a string, separators are treated as normal characters.

A separator may be a space or any control character other than `NUL`. If GalaxC is edited using a conventional text editor, the only control characters possible are `CR`, `LF`, and `FF`. However, if GalaxC is edited using XOE then control characters may be *format control* (FC) characters which change the font and style of the

following token(s). See Section 2.11.

A separator may also be a **comment**. GalaxC supports both kinds of ANSI C comments:

- Any sequence of characters beginning with '`/*`' and ending with '`*/`'. Such a comment can span any number of lines. They cannot be nested, i.e., inside a comment '`/*`' and '`//`' are ignored.

- Any sequence of characters beginning with '`//`' and ending at the end of the line.

Comments edited using XOE may contain format control, e.g., italics and boldface. They have no special significance other than improved readability.

Here is the syntax for separators:

comment = `/*` [*any sequence of characters except* '`*/`'] `*/` |
`//` *any sequence of characters up and including the end of the line*
sep = *space* | *control character other than* `NUL` | *comment*
separator = [{*sep*}]

In summary, a separator can be any number (including zero) of spaces, control characters, and comments in any order.

We can now define the lexical syntax of a GalaxC program simply as an alternating sequence of separators and tokens:

program = separator token separator {*token separator*}

Note that a program must contain at least one token.

2.9. Example Program

In this example, we take a fragment of a GalaxC program and separate it into tokens using the token syntax described in this chapter. Here is the initial program text:

```
// Define construction of complex numbers.
float arg {x, y},
def x + i y = (x, y): complex;

// Define complex addition.
complex arg {z1, z2},
def z1 + z2 = (Re z1 + Re z2) + i (Im z1 + Im z2);
```

When scanned by the GalaxC compiler, the following tokens are produced:

```
float    arg  {   x   ,   y   }   ,
def      x    +   i   y   =   (   x   ,   y   )   :   complex  ;
complex  arg  {   z1  ,   z2  }   ,
def      z1   +   z2  =   (   Re  z1  +   Re  z2  )   +
                      i   (   Im  z1  +   Im  z2  )   ;
```

Note that the comments and the spacing structure are lost. Scanning has converted the program text from a string of characters into a sequence of tokens. Combining tokens to produce expressions will be described in

the next chapter.

2.10. Extended Character Sets

GalaxC uses the 8-bit Windows-1252 ("ANSI") character encoding for most fonts. *[fn:* http://en.wikipedia.org/wiki/Code_page_1252*]* Windows-1252 includes the ASCII character set along with the ISO-8859-1 Latin-1 extensions to provide accented letters such as in *año*, *Çui-là*, and *Straße*, as well as additional special characters such as © and ®. Windows-1252 adds more letters such as in *œuf* and *Beneš* as well as "high-quality" quotes and daggers.

All Windows-1252 accented letters can be used in as letters in identifiers:

á	à	â	ä	ã	å	æ	Á	À	Â	Ä	Ã	Å	Æ
é	è	ê	ë				É	È	Ê	Ë			
í	ì	î	ï				Í	Ì	Î	Ï			
ó	ò	ô	ö	õ	ø	œ	Ó	Ò	Ô	Ö	Õ	Ø	Œ
ú	ù	û	ü	ý	ÿ		Ú	Ù	Û	Ü	Ý	Ÿ	
ç	ñ	š	ž	ß	ð	þ	Ç	Ñ	Š	Ž	Ð	Þ	

The remaining symbols are treated as special characters.

To use these characters, you must use an editor that supports them like XOE. Fonts should not be a problem, since Microsoft has made free Web fonts available which include Windows-1252. They are also fully supported by PDF.

GalaxC also supports the Symbol font and character set. *[fn:* http://en.wikipedia.org/wiki/Symbol_font*]* ASCII letters are mapped more or less to their Greek equivalents and can be used as single-character GalaxC tokens:

α	β	χ	δ	ε	ϕ	γ	η	ι	φ	κ	λ	μ
ν	ο	π	θ	ρ	σ	τ	υ	ϖ	ω	ξ	ψ	ζ
Α	Β	Χ	Δ	Ε	Φ	Γ	Η	Ι	ϑ	Κ	Λ	Μ
Ν	Ο	Π	Θ	Ρ	Σ	Τ	Υ	ς	Ω	Ξ	Ψ	Ζ

Symbol also includes numerous mathematical symbols such as ≤ and ≥. GalaxC treats them as special characters. Symbol includes ASCII digits 0-9, which GalaxC treats as normal *digit* characters. The Symbol font is a standard PDF font and is automatically included in Windows and in many Linux distributions. In other cases it may be necessary to purchase a copy of `symbol.ttf` to edit documents containing Symbol characters.

2.11. Format Control

When used with a conventional text editor, GalaxC programs have a single font with no style. Using XOE, GalaxC programs can have multiple fonts and styles. They can be used just for readability, or can change the meaning of tokens, for example, you can use '*a*' and '**a**' to represent different variables.

GalaxC encodes character formatting using invisible *format control* characters inserted in the program text. Lexically they are treated like separators, i.e., they are equivalent to inserting space or tab between tokens. Format control characters are single control characters sometimes followed by a non-zero argument byte. Format control characters are never 0: the only `NUL` character is the end of the file.

Here are the format control characters defined at this time:

ctl-B Set the current color to 0, which is a neutral color: black on while or white on black.

ctl-C Set the current color to the non-zero value in the next byte. The 256 possible colors are mapped into at most 16 distinct color classes to distinguish tokens.

ctl-E Set the current font number to 0. By default, this is a monospaced font like `Courier`.

ctl-F Set the font number to the non-zero value in the next byte. By default, 1 = serif (e.g., Times), 2 = sans-serif (e.g., Helvetica or Arial), and 3 = Symbol. Other values may be different type faces or different sizes of an existing type face. We have arbitrarily decided that 256 fonts (not including style) is more than enough to make a document look like a blackmail note. The 256 possible fonts are mapped into at most 16 distinct font families to distinguish tokens. Currently there are just four families, corresponding to the four default fonts.

ctl-N Set the current font style to normal.

ctl-S Set the current font style to the non-zero value in the next byte. The style bits include **bold**, *italics*, underline, overline, strike-through, subscript, and superscript.

Format control is treated as a separator, so it breaks up tokens. For example, "bi*rd*" is not a single token, since 'bi' and '*rd*' are different fonts.

Except for subscripts and superscripts, GalaxC only cares about character formatting for identifiers, and then only if a compiler option is enabled. Numbers, characters, strings, special characters, and reserved words mean the same thing no matter what format they have. If an identifier does not have the default style (which is set by the first token in the file), the GalaxC scanner inserts an invisible *style prefix token* (SPF) in front of it to specify the style.

Subscripts and superscripts are handled differently: if the subscript or superscript level changes, GalaxC inserts a *style change token* (SCH) at each change which indicates whether subscript or superscript level increased or decreased. They act as special kinds of parentheses during parsing.

2.12. XXICC Objects

If edited using XOE, a GalaxC program may contain *XXICC Objects* (XOs) such as dialogs, tables, and mathematical formulas *[future]*. An XO has invisible tags to show the structure of the XO. For example, the table:

a[0,0]	a[0,1]
a[1,0]	a[1,1]

has an internal representation of the form:

```
TABLE
  HSTACK    CELL  a[0,0]  ENDOBJ   CELL  a[0,1]  ENDOBJ    ENDOBJ
  HSTACK    CELL  a[1,0]  ENDOBJ   CELL  a[1,1]  ENDOBJ    ENDOBJ
ENDOBJ
```

where `TABLE`, `HSTACK` (horizontal stack), `CELL`, and `ENDOBJ` are XO tags. `TABLE`, `HSTACK`, and `CELL`

begin *XO subtrees* (XOSTs) each of which ends with `ENDOBJ`.

Scanning a GalaxC program with XOs simply converts XO tags to XO tokens. Parsing XOs is also simple since the XOST and `ENDOBJ` tokens already define the XO's tree structure.

XOs are described in detail in [JFB 11: XXICC Objects].

2.13. Implementation Notes

The lexical syntax described here is for the current GalaxC implementation. In this version, the lexical syntax is fixed and cannot be changed by the programmer. In future versions of GalaxC it may be possible to expand and modify the lexical syntax to create application-specific libraries.

Overline and strike-through have not been implemented yet.
Character color has not been implemented yet.

Chapter 3
Expressions

In this chapter, we will see how the tokens of Chapter 2 are combined into **expressions**. Tokens correspond to the words and punctuation marks of written language, i.e., they are sequences of characters. Expressions correspond to phrases, sentences, paragraphs, etc. formed from those words, i.e., they are sequences of tokens. Just like written language, GalaxC has rules that define what is a legal sequence of tokens.

Every GalaxC program construct is some kind of expression. Expressions are combined into larger expressions using **syntax rules** which define legal combinations as well as the precedence among operators. GalaxC syntax is both simple and general. It recognizes a wide variety of notations, with the goal of allowing users to express programs simply and clearly.

One important use of expressions is for writing mathematical formulas. GalaxC allows C-like mathematical expressions and adds additional notations such as:

`sin x`	Function calls do not require parentheses.
`|a-b|`	Absolute value, also used for vector and string length.
`n!`	Factorial.
`x + i y`	Construction of a complex number.

In addition, GalaxC programs edited using XOE support subscript and superscript notations:

$\sin^2\alpha + \cos^2\beta$
$a_2x^2 + a_1x + a_0$
e^x

In those cases where standard notation is not possible due to the absence of special characters or graphical notations for the available hardware, GalaxC adopts familiar C-like notations. Future versions of GalaxC will allow more general mathematical notation.

Expressions are also used as programming constructs. These include *assignment expressions*, *conditional expressions*, and *iterative statements*. In GalaxC, a statement is just a kind of expression.

There are many kinds of expressions and specific syntax rules to define which expressions can be sub-expressions of other expressions. The syntax rules (§3.3) define what is a correct GalaxC expression and also establish the operator precedence, which is consistent with standard mathematical notation and C.

As we discussed in the introduction, GalaxC separates syntax from semantics. The syntax rules described in this chapter only define how a list of tokens is parsed in the strict syntactic sense. They do not define what the syntax means. In practice, actual implementations of GalaxC do assign semantic meanings to most basic syntactic constructs. For example, the *if-then-else* expression is defined to have the expected C-like behavior. Nevertheless, these are only default definitions: it is certainly possible, though perhaps ill-advised, to redefine such constructs to have some other behavior entirely.

This chapter defines the syntax of the current version of GalaxC. We also describe default semantics for some of the constructs to clarify their purpose. Semantics will be discussed in detail in later chapters.

3.1. Mathematical Expressions

Mathematical expressions in GalaxC have a rich syntax with many operators and are able to represent most

basic mathematical notations. Operator precedence is built into the syntax and is based on standard mathematics and C. Table 3-1 lists the types of mathematical expressions with operator precedence shown from top to bottom.

Table 3-1: Mathematical Expressions

Expression name	*Examples*	*Operators*
sub	`f(x), a[2],` a_2	`f(), a[]`
suffix	`n!, x^^2,` x^2	`!, ++, --, ^^`
prefix	~*x*	`~, ++, --`
unary	-*x*, |x|	`+, -,` |*x*|
concat	`sin x`	
factor	`!x, ?x`	`!, ?`
term	`x / y`	`*, /, %`
arith	`x + y`	`+, -`
shift	`x << 3`	`<<, >>`
relation	`x > 3`	<, >, >=, <=, ≥, ≤
equality	`x == y`	==, !=, ≡, ≠
bitand	`x & y, x # 3`	&, #
bitxor	`x ^ y`	^
bitor	`x | y`	|

Mathematical expressions are formed by combining subexpressions with mathematical operators. These include unary operators, e.g., '`+`' and '`-`', and binary operators, e.g., '`+`', '`-`', '`*`', '`/`', and '`%`'. The last three are the C operators for multiplication, division, and remainder (used for modulo arithmetic). The highest precedence expression is *sub*, which includes C-style function calls and subscripts. The second highest is *suffix*, which includes superscripted expressions, factorial (`n!`), and C post-increment and post-decrement (`pc++`, `sp--`). Next we have expressions with unary and multiplicative operators. Binary '`+`' and '`-`' have the lowest precedence of the arithmetic operators. Suffix and binary operators associate from left to right; prefix and unary operators associate from right to left. Use parentheses to establish different precedence or associativity.

Examples:

```
a*-b + c*d      Equivalent to: (a*(-b))+(c*d).
-a + b + c      Equivalent to: ((-a)+b)+c.
a + b % 10      Equivalent to: a + (b % 10).
```

The spaces in the above examples are unnecessary and are only present to improve readability. Except for "`a[i]`" (which is different from from "`a [i]`") and "`f(x)`" (which is different from "`f (x)`"), GalaxC ignores spaces except as token separators and does not consider them when parsing expressions: only the operator precedence matters. For example, consider the expression:

```
a+b * c+d
```

Since the spaces are ignored, GalaxC parses it as:

```
(a+(b*c))+d
```

To get any other result, use parentheses to circumvent the normal operator precedence, e.g.,

```
(a+b) * (c+d)
```

3.1.1 Subscripts and Superscripts

GalaxC allows expressions that contain subscripts and superscripts, such as:

$x_1 + x_2$ e^x $\log_2 N$

Subscripts and superscripts have the highest operator precedence, with subscripts higher than superscripts. This agrees with standard mathematical conventions. Here are some examples and how GalaxC parses them:

$a_2*x^2 + a_1*x + a_0$	$(((a_2)*(x^2)) + ((a_1)*x) + (a_0)$
$x_1^2 + x_2^2$	$((x_1)^2) + ((x_2)^2)$
$-x^3$	$-(x^3)$

XOE documents are normally limited to one subscript and one superscript level. *[footnote: We are planning to incorporate a general mathematical editor some day that will support arbitrary levels.]* Nested sub- and superscripts must be parenthesized explicitly, as shown in these examples:

$(x_1)_2$ $(y^3)^4$

The syntax for a subscript or superscript expression is quite general, as shown in these examples:

$x_{1,2}$ $e^{i*\omega*t}$ $a^{(b,c)}$

GalaxC programs may be written using an editor that does not support formatted subscripts and superscripts. In this case, the following alternative notations may be used:

subscripts:	`x[1,2]`	`log[2] N`
superscripts:	`e^^x`	`x^^2`
mixed:	`a[2]*x^^2 + a[1]*x + a[0]`	

Note that there is no space in front of bracket '`[`': in GalaxC "`a[i]`" is different from "`a [i]`". GalaxC uses `^^` for superscript because C already uses `^` for exclusive OR. As with formatted subscripts and superscripts, the superscript operator has higher precedence than prefix and most other operators, so that complex superscripts need to be put in parentheses. For example, $e^{i*\omega*t}$ must be entered as `e^^(i*ω*t)`. XOE parses `e^^i*ω*t` as `(e^^i)*ω*t`.

3.1.2 Suffix and Prefix Operators

GalaxC superscripts, both formatted and using the `^^` operator, are *suffix* expressions, as are expressions with the C post-increment and post-decrement operators, e.g., `i++` and `n--`. GalaxC also has the factorial operator, e.g., `n!` Suffix operators have higher precedence than prefix operators. Here are some examples and how GalaxC parses them:

`n! / r! (n-r)!`	`(n!) / ( (r!)((n-r)!) )`
$\log_2 n!$	$(\log_2)\ (n!)$

Suffix operators associate from left to right.

GalaxC's next higher operator is *prefix*, which includes C's bit-wise `NOT` operator '`~`' as well as its pre-increment and pre-decrement operators, `++j` and `--m`. They associate from right to left.

GalaxC adopts another versatile mathematical syntax: the *absolute value* notation, e.g., `|x|`. Along with its normal use, the absolute value syntax provides a familiar notation for vector length, complex number magnitude, string length, type size, as well as other metrics. Absolute values have a lower precedence than prefix or suffix operators. Here are some examples:

```
-|x|      |x-y|      |x| - |y|      | |x| - |y| |      |x|*|y|
```

Since GalaxC also has the C '`|`' and '`||`' operators, you have to be careful when combining absolute value with other operators. For example, you must use "`| |x| - |y| |`" instead of "`||x| - |y||`" because GalaxC considers '`||`' (with no space) to be a single token instead of two '`|`' characters. Here are some other potential errors:

`|n|!`	`|n|` has lower precedence operator than '`!`': use "`(|n|)!`".
`sin |x|`	GalaxC parses "`sin |x`" as a bitor expression: use "`sin (|x|)`".
`|x| |y|`	GalaxC parses "`|x| |y`" as a bitor expression: use "`(|x|)(|y|)`".

In general, `|x|` conflicts with the concatenation operator (next section) and combining them usually requires parentheses.

Absolute value is an example of a *unary* operator, which is lower priority than a *prefix* operator. Unary operators also include C prefix operators that look like binary operators, such as unary `+` and `-`, as well as C prefix operators `*` and `&`. GalaxC makes this distinction because of concatenation. Since C does not have `|x|` or concatenation, it can treat all prefix and unary operators the same syntactically.

3.1.3 Concatenation: Function Calls and Implied Multiplication

The next construction of interest is the concatenation (*concat*) expression, which consists simply of one expression followed by another, e.g.,

```
a b        sin x        a[2] x^^2
```

The precedence of the concat operator is approximately the same as unary. Specifically, a concatenation may have a unary expression as its first term, but the remaining terms must be *prefix* expressions, i.e., operators that do not look like binary operators. For example, "`-a p++ n!`" is a legal concatenation parsed as "`((-a) (p++)) (n!)`" because `++` and `!` are never binary operators. On the other hand, GalaxC parses "`a *b`" as product "`a*b`": it doesn't care about the space. Here are some more examples:

`+a b`	is parsed as `(+a) b`.
a_2 x^2	is parsed as (a_2) (x^2).
`sin -x`	is parsed as `sin-x` (subtraction).
`sin (-x)`	is parsed correctly.

The primary uses of the concat are function call patterns and implied multiplication. Here are some sample function calls:

```
sin² x      log₂ N      cos (ω*t + a)      f(x,y,z)
```

Note that parentheses around arguments are only required when there are multiple arguments or the expression is sufficiently complex. Parentheses can always be added to an argument: `sin x` can be written `sin (x)`. This contrasts with other programming languages like C where parentheses are always required. On the other hand, `sin(x)` without the space is syntactically different in GalaxC: it is a *call* expression rather than the concat of `sin` and `(x)`. The function call `f(x,y,z)` is also a call: writing it as "`f (x,y,z)`" would change it to the concat of `f` and `(x,y,z)`, which is different syntax.

GalaxC function names can be quite general and can include subscripts and superscripts. In fact, any legal syntax can denote a function call, e.g., "`x+y`", "e^x", and even a phrase like "`draw line from z1 to z2`", where `z1` and `z2` are Points. Here is a sample call to that function:

```
draw line from [100,200] to (z + [1,1])
```

This example illustrates why spaces are not optional before '`[`' and '`(`'. GalaxC parses this expression as five concat expressions, with subexpressions grouped as:

```
((((draw line) from) [100,200]) to) (z + [1,1])
```

If spaces were optional, GalaxC would parse "`from [100,200]`" as the subscript expression "`from[100,200]`" instead of concatenated subexpressions. Similarly, GalaxC would parse "`to (z + [1,1])`" as the call "`to(z + [1,1])`". Galaxy '91 used the conventional approach of allowing separators between all tokens. We found that this required lots of non-intuitive parentheses in situations like the one above. GalaxC's approach improves readability, though it does require programmers to be consistent about spaces before '`[`' and '`(`'.

The concatenation notation may also be used to denote implied multiplication as in mathematics. Here are some examples:

`a b`	Same as `a*b`.
`(a+b)(c+d)`	Same as `(a+b)*(c+d)`.
$a_2x^2 + a_1x + a_0$	Standard polynomial notation.

There are some minor syntactic differences between the concat and the multiplication operator. Concat has a higher precedence than multiplication and division, so GalaxC parses "`a b / c d`" as "`(a b)/(c d)`" and parses "`a*b / c*d`" as "`((a*b)/c)*d`" since '`*`' and '`/`' have the same precedence. On the other hand, GalaxC parses "`-a b`" as "`(-a) b`" and parses "`-a*b`" as "`(-a)*b`", which is essentially the same. Like multiplication, concat associates from left to right so GalaxC parses "`a b c`" as "`(a b) c`" and parses "`a*b*c`" as "`(a*b)*c`". This means you should write "`sin cos x`" as "`sin (cos x)`" so GalaxC doesn't parse it as "`(sin cos) x`".

A question now presents itself: how does GalaxC tell the difference between a function call and an implied multiplication? Is "`sin x`" a function call or the product of varibles `sin` and `x`? As far as parsing is concerned, it makes no difference at all. The expression syntax here is concerned only with collecting tokens into well formed expressions, and "`sin x`" is parsed the same way whether it is a function call or multiplication. It is only when GalaxC analyzes the parsed expression that it is necessary to determine which interpretation is valid. This is easily done by checking which symbols have been defined. If `sin` is defined to be a variable, then multiplication is understood. If `sin x` is instead defined as a function pattern, then a function call is performed. This will be described further in Chapter 5.

[footnote: The current version of GalaxC does not define any functions or macros for implied multiplications, even though they are syntactically correct. Concatenation is mostly used for descriptive function names.]

GalaxC treats the C logical `NOT` operator '`!`' as having a lower priority than concat. This turns out to be very useful for inverting GalaxC descriptive macros and functions. For example, you may define a macro "`x is even`", where `x` is an integer. You can then write "`!y is even`", which GalaxC parses as "`!(y is even)`". Giving '`!`' a low priority eliminates many parentheses. The GalaxC '`?`' operator has the same syntax.

3.1.4 Relational and Logical Expressions

Like most programming languages, GalaxC has logical expressions with relational and logical operators. The relational operators include the C operators:

```
==      !=      <      >      <=      >=
```

which correspond to equal, not equal, less than, greater than, less than or equal to, and greater than or equal to. GalaxC adds Symbol font characters ≠, ≡, ≤, and ≥ as single-character equivalents to `!=`, `==`, `<=`, and `>=`. Relational operators have lower precedence than any of the operators encountered so far. Relational operators associate from left to right, allowing compound relations to be specified syntactically. They may or may not be defined semantically. Here are some examples of relational expressions:

```
a < b
a+b < c+d          Equivalent to: (a+b) < (c+d).
a > b > c          Grouped as: (a > b) > c.
```

The value of a relational expression is normally defined to be `Boolean`, which is a subtype of `int` with values `TRUE` (1) and `FALSE` (0).

Like C, GalaxC has bit-wise operators '`~`' (`NOT`), '`&`' (`AND`), '`|`' (`OR`), and '`^`' (exclusive `OR`). GalaxC adds a bit-wise *bit clear* (`BIC`) operator '`#`', which is the same as '`&`' except that it complements its second operand: "`a # b`" is the same as "`a & ~b`". *[footnote: We are describing the usual semantics for these operators. Since GalaxC separates syntax from semantics, they could be defined as something else entirely.]* In C, '`#`' is a pre-processor operator. Since GalaxC does not have a C-like pre-processor, we are free to re-use '`#`' for a very useful operation.

Bit-wise `AND` is often used to test the presence of a status or option bit, such as:

```
if (x & 0x100) != 0 then ...
```

C permits an *if-then* or *while-do* statements to have an `int` condition, which allows the more concise:

```
if (x & 0x100) ...
```

Since this is a common operation, GalaxC adds a *bit test* (`BIT`) operator '`&?`' so that you can write:

```
if x &? 0x100 then ...
```

The expression "`a &? b`" is the equivalent to "`(a & b) != 0`". Similarly, GalaxC has a `NOT AND` (`NAND`) operator '`!&`' where "`a !& b`" is the equivalent to "`(a & b) == 0`". It is the same as C's "`!(a & b)`".

Bit-wise operators combine integer values, which they treat as bit vectors. In addition, GalaxC has logical operators that combine `Boolean` values such as relational expressions. These include the logical `NOT` operator '`!`' which is the same as C except that the GalaxC result is `Boolean` instead of `int`. "`!x`" is equivalent to "`x == 0`". If `x` is `Boolean`, `!TRUE` is `FALSE` and `!FALSE` is `TRUE`.

GalaxC has a logical `TEST` operator '`?`' which tests whether its operand is non-zero. "`?x`" is equivalent to "`x != 0`". It's used in *if-then* statements like "`if ?x then ...`" which is equivalent to C's "`if (x) ...`".

GalaxC also has the C logical `AND` '`&&`' and logical `OR` '`||`' operators, which have `Boolean` operands. Like C, the second argument is only evaluated if necessary, for example "`p != NULL && f(p)`" only evaluates `f(p)` if `p` is not `NULL`. GalaxC provides reserved words "`and`" and "`or`" as more readable equivalents to '`&&`' and '`||`'. For example, the last expression can be written: "`p != NULL and f(p)`". The author prefers to use "`and`" and "`or`", reserving '`&`' and '`|`' for bit-wise operations. However, this is only a matter of taste.

As shown in Table 3-2, `Boolean` `AND` (*conj*unction) and `OR` (*disj*unction) have lower precedence than bit-wise `AND` and `OR`. This is the same as C. Note that there is no `Boolean` version of exclusive `OR`: it can be accomplished using an inequality operator '`!=`' or '≠'. Similarly, Boolean equivalence can be accomplished using '`==`' or '≡'. Note that the equality operators have lower precedence than other relational operators. Logical binary operators associate from left to right.

Here are some more examples and how GalaxC parses them:

```
a < b and c > d          (a < b) && (c > d).
x and !y                 x && (!y).
~a & ~b                  (~a) & (~b).
x & y & z                (x & y) & z.
a & b | c and d          ((a & b) | c) && d.
```

GalaxC avoids confusion between the absolute value '`|`' and the logical operator '`|`' by preventing the use of absolute values with the concatenation operator and requiring the expression within an absolute value to be of higher precedence than *bitor*.

GalaxC has two shift operators, '`<<`' (shift left) and '`>>`' (shift right). A shift expression has the form *x* `<<` *n* or *x* `>>` *n*, where *x* is the value to shift and *n* is the number of bits to shift left or right. A left shifts (logical shift left) is equivalent to multiplication by a power of two. A right shift (logical shift right if *x* is unsigned, arithmetic shift right if *x* is signed) is approximately equivalent to division by a power of two, but you have to be careful of rounding especially if *x* is negative. Here are some examples of expressions using shifts:

```
x << 2                   x shifted left 2 bits.
y >> 3                   y shifted right 3 bits.
x << 2 & y >> 3          Bitwise AND of the above expressions.
```

3.2. Non-Mathematical Expressions

To produce a complete GalaxC program it is necessary to augment the mathematical formulas with non-mathematical expressions such as data access, data transformation, declarations, definitions, control statements, and assignment expressions. These expressions are included in the GalaxC syntax and co-exist with mathematical expressions. Table 3-2 lists all types of GalaxC expressions with operator precedence

from top to bottom, along with operator associativity.

Table 3-2: GalaxC Expressions.

Expression name	*Examples*	*Operators*	*Associativity*
prim	`42, a, "s", 'c', (x)`	`(), [], {}`	
sub	`f(x), a[2], z.Re,` a_2	`f(), a[], '.', ->`	left to right
suffix	`n!, x^^2,` x^2	`!, ++, --, ^^`	left to right
prefix	$\sim x$	`~, ++, --, @`	right to left
unary	$-x$, $\lvert x \rvert$	`+, -,` $\lvert x \rvert$`, *, &, $, #`	right to left
concat	`sin x`		left to right
factor	`!x, ?x`	`!, ?`	right to left
term	`x / y`	`*, /, %`	left to right
arith	`x + y`	`+, -`	left to right
shift	`x << 3`	`<<, >>`	left to right
relation	`x > 3`	`<, >, >=, <=,` ≥, ≤	left to right
equality	`x == y`	`==, !=,` ≡, ≠	left to right
bitand	`x & y, x # 3`	`&, #, &?, !&`	left to right
bitxor	`x ^ y`	`^`	left to right
bitor	`x | y`	`|`	left to right
conj	`x && y, x and y`	`&&, and`	left to right
disj	`x || y, x or y`	`||, or`	left to right
cond	`a? b: c`	`?:`	
expr	`x = y, i += 3, p #= b`	=, +=, -=, *=, etc.	right to left
defin	`var n = 0`		
stmt	`if a then b, while a do b`		
colon	`i:j, ch + 4: char`	`:`	right to left
comma	`1, 2, 3`	`,`	right to left
semi	`x = 3; f(x)`	`;`	right to left

3.2.1 Data Access

GalaxC supports record-oriented data structures with named fields (*structs*), as will be discussed in detail Chapter 8. The fields of these structs can be accessed in many different ways using the flexible GalaxC syntax and may look quite different from conventional field access constructs. For example, suppose there is a type `complex` which has fields `Re` and `Im`. Some possible field access notations include:

- `Re z, Im z`
- `Re{x}, Im{z}`
- z_{Re}, z_{Im}

You can also use the C-like *dot* notation, i.e., `z.Re` and `z.Im`. The dot operator has the same precedence as other sub operators and associates from left to right. Here are some examples and how GalaxC parses them:

```
z1.Re + z2.Re              (z1.Re) + (z2.Re)
b.z0.Re                    (b.z0).Re
a.z0₂                      (a.z0)₂
```

Typically, only one form of field access notation is defined for a given struct. The variety of notations available in GalaxC for data extraction helps to make GalaxC a very expressive language.

3.2.2 Primitive Expressions

The expressions with the highest precedence are called **primitives**. These include the non-special tokens defined in Chapter 2: identifiers, numbers, character literals, string literals, and floating-point numbers. Here are some examples:

```
a     123     0xA4F3     'c'     "cat"     6.023E23
```

Parenthesized expressions are also primitives: an arbitrarily complex expression can be enclosed in parentheses and then used as a primitive. Three kinds of parenthesized expressions are available which can be defined for different purposes:

- **parentheses**: `(` *expression* `)`
- **brackets**: `[` *expression* `]`
- **braces**: `{` *expression* `}`

In Section 2.12, we saw that GalaxC converts XXICC Object (XO) tags like `TABLE` and `ENDOBJ` into XO tokens. When parsing, GalaxC treats an XOST token as an opening parenthesis and `ENDOBJ` as a closing parenthesis:

- *XOST expression* `ENDOBJ`

As with other parenthesized expressions, *expression* can be arbitrarily complex and may contain nested XOSTs. Since XO tags already define the tree structure of complex XOSTs, parsing them is quite simple.

In Section 2.11, we saw that GalaxC may insert a *style prefix* (SPF) token before a *styled identifier* with non-default character formatting. GalaxC parses an SPF followed by an identifier as a primitive. An identifier has at most one SPF.

3.2.3 Assignment Expressions

Probably the most frequent activity in a program is assigning values to variables. Many applications seem to do nothing but assign values. The GalaxC assignment expression (called an *expr*) is essentially the same as C. Here are some examples:

```
a = 3       a_i = b_i + c_i       Re z = Re v + Re w
```

Like C, GalaxC can have compound assignments that assign the same value to multiple variables, e.g., "`x = y = 3`". The value of an assignment expression is the same as the value assigned, so it is possible to include assignments as side-effects of more complex expressions, such as: "`if (p = q) != NULL then...`" which assigns the value of `q` to `p` and compares it to `NULL`. Assignment operators associate from right to left.

GalaxC also has C "*op*=" assignments where "`x` *op*`= y`" is equivalent to "`x = x` *op* `y`". For example, "`x *= 3`" is the same as "`x = x*3`". *[footnote: They are not quite the same:* "`x *= 3`" *only evaluates* `x` *once, whereas* "`x = x*3`" *may evaluate* `x` *twice, which makes a difference if evaluating* `x` *has a side-effect.]*

Note on terminology: while every syntactically-correct construct in GalaxC is an expression, we will usually

refer to a construct with type `void` as a *statement* and reserve *expression* for those with a non-`void` types or when we don't known. Syntactically, an *expr* is an assignment expression or simpler, and may include type `void` statements like `return` and `break`. A *stmt* is a control statement, a definition, or an *expr*, which may be `void` or non-`void`. See Section 3.3 for details.

3.2.4 Control Statements

This section describes the syntax of GalaxC's control statements. We will briefly mention the default GalaxC semantics, but defer details to Chapter 7. Note that these notations can be used for other purposes. All control statements are more complex than *expr*.

GalaxC has *if-then* and *if-then-else* expressions similar to C `if` statements:

```
if a then b
if a then b else c
```

Expression *a* is expr or higher. Expressions *b* and *c* are stmt or higher. The two forms correspond to C statements "`if (a) b`" and "`if (a) b; else c`", but there are some important differences described in Section **7.1**.

In nested *if* expressions, `else` is associated with the nearest `then` preceding it. For example, the expression:

```
if a then if b then c else d
```

is parsed as:

```
if a then {if b then c else d}
```

Use braces to specify the nesting explicitly:

```
if a then {if b then c} else d
```

GalaxC has a *switch* statement similar to C, for example:

```
switch ch
{
  case {'a', 'b'}: printf("Found a or b\n");
  case {'c', 'd'}: printf("Found c or d\n");
  default: printf("Did not find a, b, c, or d.\n");
}
```

GalaxC's switch does not require parentheses around the selector expression as long as it's unary or higher. Also, GalaxC does not require an annoying `break` at the end of most cases. See Section **7.7** for a full description.

GalaxC has the following **iterative** statements:

```
while a do b
repeat b until a
for (i; a; u) b
for a do b
```

Condition a is *expr* or higher (*comma* or higher for the first *for*). *Body b* is *stmt* or higher (*semi* or higher for *repeat-until*). The standard GalaxC behavior for *while-do* is to test the value of *a* and repeat evaluation of *b* as long as *a* is `TRUE`. This is essentially the same as C's "`while (`*a*`)` *b*", except that GalaxC has the Pascal `do` instead of C's parentheses and *a* must be `Boolean`. GalaxC's *repeat-until* evaluates *b* first and then tests the value of *a*, repeating evaluation of *b* until *a* becomes `TRUE`. This is similar to C's "`do` *b* `while (`*a*`)`".

GalaxC has two *for* statements. The first has the same syntax as C, except that *a* must be `Boolean`. Expressions *i* (initialize) and *u* (update) are *comma* expressions. The second form is a general-purpose template for defining Pascal-style *for* loops like:

```
for i = 1 thru 10 do ...
for i = 10 downto 0 do ...
for i = 2 thru 10 by 2 do ...
```

In these cases *a* is an *expr* that contains assignment and concatenation. These forms will be described further in Section **7.5**.

3.2.5 Definitions

The remaining statements are *defin* statements. These include declarations of variables and arguments, e.g.,

```
Boolean var {found match, OK = FALSE}
string arg s
var #found = 5
```

Variable names in GalaxC can have any legal syntax, e.g., `found match` and `#found` as in the above example. They are not limited to identifiers like C. Variable declarations may or may not include initialization. If they do, they may infer type from the initialization expression. See Section **4.4** for details.

Here are some *defin* statements for functions, macros, and types:

```
int fn n! = n == 0? 1: n * (n-1)!
def NUL = '\0'
typedef Point = struct(z) {int z.x, int z.y}
fn |z| = sqrt(z.x * z.x + z.y * z.y)
```

As with variables, functions and macros can have any legal syntax, e.g., `n!` and `|z|`. See Section 3.3 for precise *defin* syntax and Chapters 5 and 8 for how they are used.

3.2.6 Colon Expressions

The **colon** operator '`:`' has a number of uses, including specifying subranges and **typecasts**, where an expression of one type is considered to be a different type. It is also available as a general purpose notation for defining functions and macros. The precedence of the colon operator is between statements and the comma operator. Colon associates from right to left. Here are some uses of the colon operator:

`s[i:j]`	Specifies characters `i` through `j` of string `s`.
`a: long`	Interpret the value of variable `a` as a `long` value.
`0x10430010: float`	Interpret the hexadecimal number as a floating-point value.

In contrast, C typecasts use the notation "`(`*type*`)` *expr*" and have very high precedence. Typecasts are discussed in detail in Chapter 4.

To reduce the need for parentheses in GalaxC expressions, colon expressions can sometimes be included in higher-precedence expressions where they would not be ambiguous. For example, an assignment *a* = *b* allows *b* to be a colon expression, e.g., "`s = NULL: string`". Without this trick, you would have to write: "`s = (NULL: string)`". Colons are also used in declarations like "`var pc = NULL: PSIptr`". See the syntax summary for details.

3.2.7 Comma and Semi Expressions

Expressions are often combined using the *comma* and *semicolon* operators. Both operators have lower precedence than any others, with semicolon having the lowest precedence. Both operators associate from right to left.

Comma expressions are most frequently used to form argument lists for functions defined using C-like syntax. For example, here is a such a function call:

`f(x,y,z)` Function call with three arguments.

In this case, comma is used as part of the syntax and not as an operator.

In other contexts, comma can be used as an operator. For example, with the standard GalaxC semantics the expression "`(a,b)`" causes the values `a` and `b` to be pushed onto the evaluation stack in reverse order so that they appear in memory with `a` first. This is often used to create instances of struct types. The comma operator will be discussed in detail in Section **4.7**.

Semicolons can also be used as a general syntactic construct. For example, "`f(a;b;c)`" is perfectly valid syntax for a function call. However, the primary use of a semicolon is as an operator. By default, the semicolon operator evaluates a pair of expressions, discarding the result of each expression after it is evaluated. This has the same effect as a sequence of statements in other programming languages. For example:

```
temp = x; x = y; y = temp;
```

causes the expressions `temp = x`, `x = y`, and `y = temp` to be evaluated in that order. Even though this is the same behavior as Pascal or C, we do like to make the following distinctions: In some languages, like C, semicolon is a *statement terminator*. In others, like Pascal, it is a *statement separator*. In GalaxC, semicolon is an *operator* which evaluates its two sub-expressions, throwing each result away. In GalaxC, a statement is just another kind of expression.

Note: GalaxC syntax allows the last statement of a sequence to have a semicolon even if it is not followed by another statement. This is equivalent to following the last semicolon with `nop` ("no operation"), the GalaxC equivalent of C's *null* statement.

Comma and compound expressions can be enclosed in parentheses and inserted inside any other expression. This recursion allows all GalaxC programs to be written using the syntax just described. Finally, a GalaxC program in a text file is defined to be a single semi expression followed by end of file. A GalaxC program expressed as a formatted document is a single PARABOX XO.

3.3. Syntax Summary

This section specifies the current GalaxC syntax. While this syntax is currently fixed, future releases may allow users to extend or modify this syntax.

XOleaf = XO leaf node, see Chapter *[xref XXICCobjects]*.
XOprefix = XXICC Object prefix text, see Chapter *[xref XXICCobjects]*.

prim = *identifier* | *number* | *character* | *string* | *float_num* | *SPF identifier* | *XOleaf* |
'(' *semi* ')' | '[' *semi* ']' | '{' *semi* '}' | *XOST* [*XOprefix*] [*semi*] ENDOBJ

nospace = there is no space or other separator between two tokens.
beginsub = begin subscript SCH token (§2.11).
endsub = end subscript SCH token.
sub = *sub* . *prim* | *sub* -> *prim* | *sub nospace* '(' *semi* ')' | *sub nospace* '[' *semi* ']' |
sub beginsub semi endsub | *prim*

suffop = ++ | -- | !
beginsup = begin superscript SCH token.
endsup = end superscript SCH token.
suffix = *suffix suffop* | *suffix* ^^ *sub* | *suffix beginsup semi endsup* | *sub*

prefop = @ | ~ | ++ | --
prefix = *prefop prefix* | *suffix*

[|*bitxor*| is absolute value.]
unop = + | - | * | & | # | $
unary = *unop unary* | '|' *bitxor* '|' | *prefix*

[Only the first item in a concat can be *unary*.]
concat = *unary* | [*concat*] *prefix*

factop = ! | ?
factor = *factop factor* | *concat*

mulop = * | / | %
term = [*term mulop*] *factor*

addop = + | -
arith = [*arith addop*] *term*

shiftop = << | >>
shift = [*shift shiftop*] *arith*

relop = < | <= | > | >= | ≤ | ≥
relation = [*relation relop*] *shift*

eqop = '==' | != | ≡ | ≠
equality = [*equality eqop*] *relation*

bandop = & | # | &? | !&

bitand = [*bitand bandop*] *equality*

bitxor = [*bitxor* ^] *bitand*

bitor = [*bitor* '|'] *bitxor*

andop = && | and
conj = [*conj andop*] *bitor*

orop = '||' | or
disj = [*disj orop*] *conj*

cond = *disj* [? *cond* : *cond*]

assop = '=' | += | -= | *= | /= | %= | ^^= | <<= | >>= | &= | #= | ^= | '|=' | := | ::=
expr = *cond* [*assop colon*]

if_expr = if *expr* then *stmt* [else *stmt*]
switch_stmt = switch *unary* '{' *semi* '}'
while_stmt = while *expr* do *stmt*
repeat_stmt = repeat *semi* until *expr*

[*for_stmt* and *defin* may require the parser to undo work.]
for_stmt = for '(' *comma* ; *comma* ; *comma* ')' *stmt* | for *expr* do *stmt*

defop = var | arg | def | fn | inline | spcl | spclarg | typedef
defin = [*factor*] *defop disj* ['=' *colon*]

stmt = *if_expr* | *switch_stmt* | *while_stmt* | *repeat_stmt* | *for_stmt* | *defin* | *expr*

colon = *stmt* [: *colon*]
comma = *colon* [, *comma*]
semi = *comma* [; [*semi*]]
program = PARABOX *semi* ENDOBJ | *semi EOF*
EOF = end of file

3.4. Summary

GalaxC syntax is both simple and powerful. Many useful mathematical and other domain-specific expressions can be coded directly in GalaxC, allowing GalaxC programs to be as close to the user's problem domain as possible. Remember that the syntax merely defines how tokens are grouped into syntactic constructions under control of operator precedence -- semantic meaning has not yet been attached to these expressions. In the following chapters, we will see what syntactic constructs mean.

Chapter 4
Types and Variables

> All problems in Computer Science can be solved by another level of indirection... Except for the problem of too many layers of indirection.
>
> *David Wheeler*

GalaxC is a strongly typed language. This means that every data object in a GalaxC program must have a well-defined **type** to indicate how that object should be interpreted. Types let us categorize information and ensure that expressions have valid semantic interpretations. A common criticism of strongly typed languages is that they are overly restrictive and do not provide adequate means to convert data between types when the need arises. GalaxC circumvents this criticism by providing several mechanisms for well-defined conversions between types. These include a powerful **type inheritance** mechanism, a **typecast** operator, and the concept of a **quasi-type** for intermediate values that do not have explicitly defined types.

GalaxC built-in types, such as `int` and `Boolean`, are defined using the same type definition mechanisms that programmers use to define their own types. This contrasts sharply with many languages such as Pascal and C where built-in types may have special properties not available to user-defined types. For example, overloading of operators in arithmetic expressions may only be allowed for types `integer` and `real`. In GalaxC, the same mechanisms used to define built-in types and operations are available to application programmers, giving them the same power as the compiler writer.

This chapter describes the type inheritance and manipulation capabilities of GalaxC, using the built-in arithmetic types as illustration. These types are automatically available to all GalaxC programmers. Programmer-defined types are built using the same mechanisms, and are discussed in Chapter 8.

4.1. Type Inheritance

Type inheritance is a powerful and useful concept. The best known early example of a language supporting type inheritance is Smalltalk [Smalltalk 1983], a highly dynamic object-oriented language used in artificial intelligence and other applications. The central concept of type inheritance is that types have related **subtypes** and **supertypes** which share operations and properties. For example, `Real` might be declared to be a subtype of `Complex` and `Integer` a subtype of `Real`. A subtype of a given type τ can be used in any context that requires τ. For example, if a complex number is required in an expression, you may use a real or integer instead. Type inheritance provides a convenient way to specify automatic type conversion in expressions.

Since languages such as Pascal and C do not support type inheritance, automatic type conversion must be built into the compiler and is consequently only available for certain types and certain expressions in a predefined manner. By supporting type inheritance, GalaxC allows such conversions to be defined by programmers and subsequently invoked transparently as required.

One possible use of subtypes in GalaxC is for numbers with units. For example, you can define new types `meter` and `sec` as subtypes of `float`. Then addition can be defined for two `meter` numbers and for two `sec` numbers, so that two `meter` numbers produce a `meter` value and two `sec` numbers produce a `sec`. This allows arithmetic with physically dimensioned numbers to maintain the correct dimension. It is still possible to add a `meter` number to a `sec` number; however, the result will be of type `float`, since both operands will then need to be converted to their common supertype. Thus it is not possible to add `meter` to `sec` and produce a physically meaningful result. Attempting to assign such an expression of type `float` to a type `sec` or `meter` variable will result in a type error.

An entire algebra of physical arithmetic can be constructed in this manner to ensure physically correct GalaxC programs. Dimension checking occurs at compile time and involves zero run-time overhead. Note that while Pascal allows the programmer to create such types, it does not check for consistent physical dimensions in arithmetic expressions, nor are user-defined versions of arithmetic operators allowed. ADA is one of the few other languages with dimension checking capability.

4.2. GalaxC Basic Types

4.2.1 Integer Types

We can illustrate GalaxC type inheritance at the same time we present the integer types currently defined in GalaxC. There are seven such types, based on C integer types:

`ulong`	32-bit unsigned
`long`	32-bit signed, subtype of `ulong`.
`int`	Alternate name for `long` assuming a 32-bit target architecture.
`short`	16-bit signed, subtype of `long`.
`sbyte`	8-bit signed, subtype of `short`.
`ushort`	16-bit unsigned, subtype of `long`.
`ubyte`	8-bit unsigned, subtype of `ushort`.

In all current GalaxC implementations signed integers are represented as 2's complement numbers.

The type inheritance structure of the integer types is shown in Fig. 4-1.

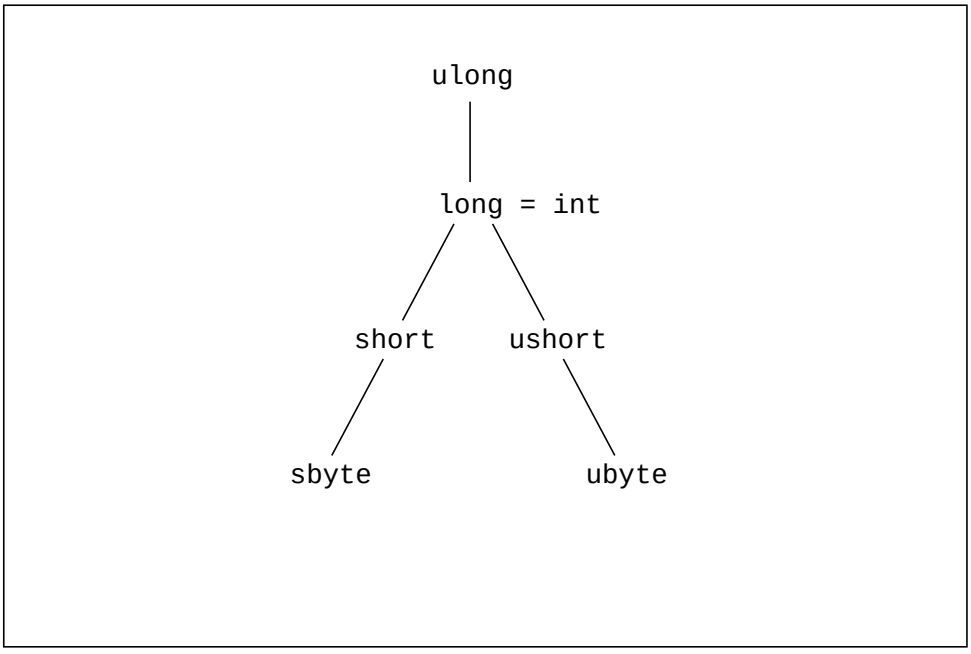


Figure 4-1: Integer Type Hierarchy.

This diagram shows which types can be used in place of another. For example, an `sbyte` can be used anywhere a `short`, `long`, `int`, or `ulong` is required. On the other hand, while a `short` can be used in place of a `long`, `int`, or `ulong`, it cannot be substituted for a `ubyte` or `ushort`. Similarly, a `ushort` cannot be used as an `sbyte` or `short`. A `ulong` cannot be used in place of any of the others.

Type `ulong` is called a **root type**. Its supertype is a special type called `void` with size equal to zero bytes. In GalaxC, a type can have at most one supertype, but can have any number of subtypes. The type hierarchy specifies which *automatic* or *implicit* type conversions are allowed. You can also use *explicit* type conversions. For example, you can convert a `ulong` value `x` to `ubyte` using the explicit conversion expression `ubyte(x)`. You may use explicit conversions in place of always relying on the automatic conversions. For example, you may convert an `sbyte` value `y` to `long` using the expression `long(y)`even though this explicit conversion is not necessary to use `y` as a `long`.

Some automatic conversions -- e.g., conversion from `long` to `ulong` -- do not require data transformation: they have the same internal bit representation. The automatic conversion merely tells GalaxC to consider the value to be an unsigned instead of a signed value. Other implicit type conversions may require data transformations. Converting from `short` to `long` requires sign extension of a 16-bit number to a 32-bit number. Converting from `ushort` to `long` requires zero extension of a 16-bit number to a 32-bit number. Converting from `ubyte` to `ushort` requires zero extension of an 8-bit number to a 16-bit number. Converting from `sbyte` to `short` requires sign extension of an 8-bit number to a 16-bit number. To account for these cases, each subtype must have an operation defined for it to convert to its supertype. This conversion operation is part of the type definition.

Let us now see how the integer types are defined in GalaxC. Each type has three attributes defined for it: (1) its supertype (`void` if a root type), (2) its size in bytes, and (3) a conversion operator to be invoked automatically to convert from the type to its supertype. Most conversion operators are *intrinsics*, described in Sections 5.6.1 and 6.3.1. The five integer types are defined as:

`typedef ulong = roottype(4)`
: Defines root type `ulong` with size equal to 4 bytes. Since `ulong` is a root type, its supertype is `void` and GalaxC converts from `ulong` to `void` by popping the `ulong` value off the stack. An equivalent definition is: "`typedef ulong = subtype(void, 4, pop)`".

`typedef long = subtype(ulong, 4, intrinsic L2UL)`
: Defines type `long` which is a subtype of `ulong`. Type `long` is the same size as `ulong` (4 bytes) and uses the intrinsic `L2UL` operator to convert from `long` to `ulong`. In practice, `long` and `ulong` have the same bit representation so `L2UL` is equivalent to `NOP`.

`def int = long`
: Defines `int` to mean the same as `long`: they are *equivalent types*. This is actually a macro definition (§5.3) rather than a type definition.

`typedef short = subtype(long, 2, intrinsic S2L)`
: Defines type `short` which is a subtype of `long`. Type `short` is 2 bytes long and uses the intrinsic `S2L` operator to convert from `short` to `long` via sign extension.

`typedef sbyte = subtype(short, 1, intrinsic SB2S)`
: Defines type `sbyte` which is a subtype of `short`. Type `sbyte` is 1 byte long and uses the intrinsic `SB2S` operator to convert from `sbyte` to `short` via sign extension.

`typedef ushort = subtype(long, 2, intrinsic US2L)`
: Defines type `ushort` which is a subtype of `long`. Type `ushort` is 2 bytes long and uses the intrinsic `US2L` operator to convert from `ushort` to `long` via zero extension.

`typedef ubyte = subtype(ushort, 1, intrinsic UB2US)`

Defines type `ubyte` which is a subtype of `ushort`. Type `ubyte` is 1 byte long and uses the intrinsic `UB2US` operator to convert from `ubyte` to `ushort` via zero extension.

The above type definition constructs, along with others, can be used by programmers to create their own types and subtypes. The complete set of type definition constructs will be described in Chapter 8.

4.2.2 `Boolean`, `char`, and `string`

In addition to integer data types for representating numbers, GalaxC has two symbolic types: `Boolean` and `char`, both defined as subtypes of `ubyte`. Boolean is 1 byte long and has two defined values: `TRUE` (1) and `FALSE` (0). The size of a `Boolean` is 1 byte. Here is the definition of `Boolean` and its two constants:

```
typedef Boolean = subtype(ubyte);
def TRUE  = 1: Boolean;
def FALSE = 0: Boolean
```

Since `Boolean` and `ubyte` have the same bit representation, we use a simplified form of `typedef`. It is equivalent to "`typedef Boolean = subtype(ubyte, 1, intrinsic NOP)`". Since `Boolean` is defined as a subtype of `ubyte`, `Boolean` values can be included directly in integer expressions. However, the converse is not true: to use an integer value where a `Boolean` is required, you must explicitly typecast it to `Boolean` as in the definitions of `TRUE` and `FALSE`, making sure the integer is indeed just 0 or 1, or else use it in an expression that has a Boolean value such as "`x != 0`".

Type `char` represents a single 8-bit character and is also defined as a subtype of `ubyte`:

```
typedef char = subtype(ubyte)
```

Characters can be included directly in integer expressions, but not vice-versa. For example, the expression "`ch = x`" where `ch` is `char` and `x` is `int` is not valid: you should write "`ch = char(x)`". GalaxC does have macros to add or subtract a `ulong` (or a subtype) to or from a `char`:

```
char arg ch, ulong arg y,
def {ch + y} = ch + y: char,
def {ch - y} = ch - y: char
```

Type `string` is defined as a pointer to `char`, which is usually in an array of `chars`. Here are the definitions of read-only `string` and its read/write counterpart `wstring`. They are macros rather than subtypes:

```
def string  = @(const char),     // Pointer to read-only char array.
def wstring = @char              // Pointer to read/write char array.
```

Strings are described in detail in Section 8.4.1.

4.2.3 `float` and `double`

There are two floating-point types, `float` (32 bit) and `double` (64 bit). Type `double` is a root type and `float` is its subtype:

```
typedef double = roottype(8);
```

```
typedef float  = subtype(double, 4, intrinsic F2D)
```

Since `float` is a subtype of `double`, it is automatically converted using the intrinsic `F2D` operator. On the other hand, you must convert `double` to `float` explicitly using `float(d)`. Also, since integers and floating point are separate type trees, you must convert between them explicitly using `float(i)`, `double(i)`, and `long(f)`, where `i` is `long` or `ulong`, and `f` is `float` or `double`.

Floating-point arithmetic is defined for both `float` and `double`, with the former used whenever possible. Floating-point may not be present in some GalaxC implementations, or may be limited to `float`, or may be implemented in software only.

4.2.4 Types of Literal Tokens

Each kind of literal token -- i.e., integer numbers, floating-point numbers, characters, and strings -- has a corresponding built-in type, as shown in the following table:

Kind of token	*Examples*	*Type(s)*
integer	`123`, `0x42`, `0b1010`	`int`, `ulong`
floating-point	`6.023E23`, `1.602d-19`	`float`, `double`
character	`'k'`, `'cat'`	`char`, `ulong`
string	`"mouse"`	`string = @(const char)`

An integer has type `int` provided that it fits, i.e., is less than `0x8000_0000` in a 32-bit implementation. Otherwise it is `ulong`. (Number tokens are always non-negative: signs are handled at the expression level.) A floating-point literal is `float` unless it has an exponent of ‘`d`’ or ‘D’, or more than six significant digits. A character literal with a single character -- e.g., `'k'` -- has type `char`. Those with multiple characters have type `ulong`.

4.3. Named Constants

In many programs it is useful -- and good programming style -- to define **named constants** (also known as *symbolic constants*), such as π, `TRUE`, `MAX_PATH`, and `MAX_ULONG`. The name can then be used anywhere the constant’s value would be used in a program. Named constants are defined as GalaxC macros using the syntax:

```
def name = value
```

where *name* and *value* can be any valid GalaxC expression (see Section 3.3). Like any other data object, constants must have well-defined types. The type of a constant is implicitly determined by the type of the *value* expression. Note that *value* must be an expression containing only constants or literals, since it must be evaluated at compile time. The GalaxC definitions for the above constants are:

```
def π = 3.14159265,
def TRUE = 1: Boolean,
def MAX_PATH = 260,
def MAX_ULONG = 0xFFFF_FFFF: ulong
```

The *name* of a constant can be an arbitrarily complex *disj* expression. This allows highly descriptive names for constants, such as:

```
kT/q          V_CES          buffer size
```

These constants can then be used in arithmetic expressions, e.g.,

```
e^(kT/q)      V_CES/2      if n > buffer size then ...
```

Currently all constant definitions of this form must be at the top level like other macros. At some point you will be able to put them inside functions according to the usual scoping rules (§4.4.3).

4.4. Variables

Like most programming languages, GalaxC includes the concept of **variables**. Variables are named data values which can change at run time, and can be used in place of literals or named constants in expressions. The value of a variable is normally changed using an assignment expression, e.g., "`x = 3`". Here is a short example:

```
int var {x, y};       // Declare two int variables.
x = 3;                // Set x to the value 3.
y = -x;               // Set y to the value -x.
```

Variables are declared with one of the following syntax forms:

```
τ var name
τ var {list of variables}
```

This declares a single variable or a list of variables separated by commas to have the type τ. A variable must be declared prior to its use and its type cannot be changed.

As with named constants, variable names can have arbitrarily complex syntax (§3.3) allowing names such as:

```
z_0      V_th      #iterations      a*b      a long variable name
```

You must be careful when using complex variable names in a program, since GalaxC parses expressions before it identifies variables. If a variable name is included in an expression with operators of higher precedence than those in the variable name, surprises may occur. For example:

- `p.z[0]` parses as `(p.z)[0]`, which splits variable `z[0]`.
- `-a*b` parses as `(-a)*b`, which splits variable `a*b`.

In both of these cases, GalaxC will not recognize the split variable as the desired variable. Such cases are generally rare since variables are usually simple identifiers or simple subscripted or superscripted expressions. You can always use parentheses to ensure that a variable name is parsed correctly, i.e., `p.(z[0])` and `-(a*b)`in the above examples. Named constants have an analogous situation and means of correction.

By allowing named constants and variables to have any syntax, GalaxC gives far more expressive power to programmers than other languages which require such symbols to be identifiers, sometimes limited to a few characters in length. This allows GalaxC programs to be written using symbols from programmers' problem

domains.

4.4.1 Local and Global Variables

GalaxC adopts block structured scoping rules for variables, a concept first popularized by Algol. **Global variables** are declared at the top level of the program, i.e., they are not declared within a function. For example, in the program:

```
int var {x, y};

int arg {a, b},
fn f(a, b) =
{
    int var {p, q};
    ...
}
```

variables `x` and `y` are global, while variables `p` and `q` are **local** to function `f(a,b)`. GalaxC allocates global variables in static storage; they exist throughout the execution of a program. GalaxC allocates local variables on the evaluation stack. They are automatically deallocated when control leaves the block in which they are declared. In the above example, `p` and `q` are deallocated when function `f(a,b)` returns.

4.4.2 Initialized Variables

By default, variables are not initialized. However, you can initialize a variable by declaring it with the syntax:

[τ] `var` *name* `=` *value*
[τ] `var {`*list of variables*`} =` *value*
[τ] `var {`*name* `=` *value*`,` *name* `=` *value*`,` *etc.*`}`

The first two forms declare a single variable or a list of variables separated by commas to have a common initialization value. The last form declares a list of variables to each have its own value. In all cases, *value* can be any legal expression.

If a variable is initialized, you an omit type τ and let *value* determine the type of the variable. For example, "`var k = 1`" declares variable `k` to be `int` because an integer literal has type `int` (§4.2.4). If you want a different type, then:

1. Specify τ explicitly.
2. Use an explicit type conversion, e.g., `var k = ubyte(1)`.
3. Use the typecast operator, e.g., `var k = 'a': int`.

Currently, only local variables can be initialized. Global variables are in fact initialized to 0 when you first compile or load a program, but if you re-run a loaded program, globals retain their previous values. It's generally best to initialize globals explicitly using initialization code.

4.4.3 Blocks and Scoping Rules

Like C, you can nest local variables to any depth using braces. An expression contained within braces is called a **block**. Any variables declared inside a block are only defined while the block is being executed. A variable or constant declared in an inner block hides a variable with the same name in the outer block. Here

is an example:

```
{    // Outer block.
     int var {a, b};          // Declare ints a and b.
     a = b;                   // a and b are both visible.
     {    // Inner block.
          ubyte var {b, c};   // Declare ubytes b and c.  There are now two declared
                              // variables named b .  Inner ubyte b hides outer int b.
          a = b;              // a is still visible.  Assign ubyte b to int a.
          b = c;              // Assign ubyte c to ubyte b.
     };   // Close inner block.      // ubytes b and c no longer exist.
     b = a;                   // ints a and b still exist.  int b is once again visible.
}    // Close outer block.          // ints a and b no longer exist.
```

Local variables and other declarations in a block may appear anywhere in the block and can be used from that point to the end of the block. The type of a block is always `void`.

4.4.4 Variable Attributes

A *variable attribute* (**vattr**) is a property associated with a variable which changes its behavior and/or restricts how it may be used. They can help compilers generate better code and detect some kinds of programming errors. Variable attributes are based on C, though the syntax is somewhat different. For example, C vattrs are reserved words while GalaxC vattrs are identifiers.

Here are the variable attributes that are currently defined:

`const`

A `const` variable is defined with an initial value and cannot be assigned a new value. An assignment "`x = y`" where `x` is `const` causes a compiler error.

While a `const` variable is similar to a named constant defined using `def`, the difference is that the variable occupies a memory location while the named constant is just a temporary data value. It is possible to point to a `const` variable, but not to a named constant.

`volatile`

A `volatile` variable may change value at any time because the variable is a peripheral device register or it may be changed by another task such as an interrupt service routine. A `volatile` variable must be loaded from or stored to memory on each access instead of being copied to a register by an optimizing compiler. Furthermore, the compiler must access it using its defined type size. For example, if the `volatile` variable is `ubyte`, it must always be loaded as `ubyte` rather than loaded as a `ulong` followed by masking.

`stable`

By default, all variables are `stable` (non-volatile): declaring a variable "`τ var x`" is the same as declaring it "`stable τ var x`". The compiler is allowed to optimize access to a `stable` variable by eliminating loads and stores that it considers unnecessary. However, this only works correctly if the compiler can assume the variable does not change value spontaneously, as might occur if it is a device register or updated by another task.

It is always possible to treat a `stable` variable as if it were `volatile`: all it costs is extra loads and

stores. On the other hand, treating a `volatile` variable as `stable` is incorrect because the resulting code does not know that the variable may spontaneously change value, resulting in different operation depending on how the code is optimized. In many cases it is best to assign the `volatile` variable to a `stable` temporary variable so there is a stable value for processing.

[The current GalaxC compiler actually treats all variables as `volatile`*: it does not optimize loads and stores.]*

`private`

A GalaxC `private` variable is the same as a C `static` variable. A `private` variables behaves like a global variable -- i.e., its value persists as long as the program is running -- but it is private to the source code file in which it is defined. If it is defined in a block, it is private to that block, but still persists like a global variable. *[As with other globals, the current implementation does not allow initialization of* `private` *variables. This restriction is redundant, since it doesn't implement* `privates` *in any case.]*

GalaxC does not implement the C `register` or `restrict` vattr.

Variable attributes `const`, `stable`, and `volatile` may also be associated with types. In most cases, it does not matter whether you consider an attribute to be part of the variable or part of its type.

For details on variable attributes and their implementation, see Appendix A.

4.5. Reference Types

We now take a closer look at what GalaxC means by the type of a variable. Objects may be data or variables. A data object has a value and can be an operand for an expression. However, it cannot be assigned to: the data object is just a value. A variable object contains another object, which may be data or a pointer to another variable. A variable has an address and a value. It is always possible to obtain the value contained in a variable, and you can change the value by assignment provided that the variable is not `const`.

The type of a data object is a **base type**. It may be a predefined type such as `int`, `long`, `short`, `ulong`, or `Boolean`. It may also be a user-defined type created using `typedef`.

A variable has *two* associated types: its value has **value type** τ and its address has **reference type** `&`τ. We say the address *references* or *points to* the value, and "dereferencing an address" means getting the value referenced by the address. Programmers normally think of the variable's value type τ, e.g., "an `int` variable `x`". The compiler normally works with a variable's reference type `&`τ, since the compiler needs the address of a variable to be able to assign to it. For example, the declaration:

```
int var {x, y}
```

really defines variables `x` and `y` to be of type `&int`. If they were just `int`, it would not be possible to assign values to them.

GalaxC uses type inheritance to handle reference types: a reference type `&`τ is a subtype of τ and converts `&`τ to τ by loading the value pointed to by the variable. This is called *automatic dereferencing*, and is normally transparent to the programmer. By the usual type inheritance rules, it is always possible to substitute a subtype for its supertype in an expression. This allows variables to be used in data expressions, such as:

```
x+1        -x*2
```

The variable `x`, which has reference type `&int`, is automatically converted to type `int` by dereferencing, i.e., loading the value pointed to by `x`. In the assignment expression:

```
x = y
```

the expression on the left side must be a reference type (assigning to a value is meaningless) while the right side must be a data value of type `int`. GalaxC object code automatically converts `y` from `&int` to `int` by loading the data pointed to by `y`. This value is then stored at address `x`. By considering all variables as automatically-dereferenced pointers to their base types, GalaxC handles expressions and assignments in a clean, general fashion. *[Footnote: the concept of a reference type is equivalent to a C "lvalue", which means an expression that can be assigned to because it can be on the left side of an assignment.]*

Fig. 4-2 shows the integer type hierarchy from Fig. 4-1 augmented with reference types. Note that a variable of any of the basic types can be used in a `ulong` context.

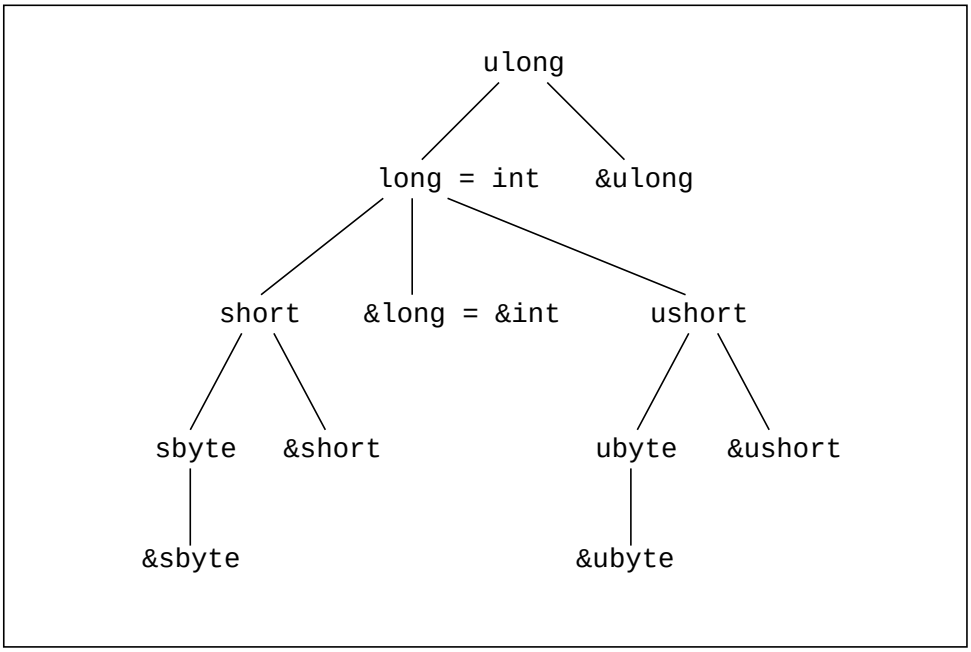


Figure 4-2: Integer Type Hierarchy with Reference Types.

Variable attributes such as `const`, `stable`, and `volatile` are independent of GalaxC's type hierarchy. See Appendix A for details.

4.6. Type Casting: the Colon Operator

GalaxC addresses the most serious criticism of strong typing: that it is impossible or at best difficult to change the type of an expression to some other type even if this is intended and makes sense. For example, there are cases where a `ubyte` integer should be considered to be a `char` or where the address of a variable should be considered `ulong` for performing address arithmetic. In Pascal, this is difficult and requires the cumbersome use of variant types. In GalaxC, this is quite simple using the **typecast** operator '`:`'. For example, to convert a `ubyte` value to a `char` value, you can write:

```
x: Char          // x is a ubyte.
```

To convert a hexadecimal number into a `ulong` variable, e.g., to examine an absolute hexadecimal address, you can write:

```
0x1A4C: &ulong   // Defines the fixed address 0x1A4C to be the address of a ulong value.
```

GalaxC's typecase is equivalent to C's typecast operator "`(`*type*`)` *expr*".

An interesting use of this was with the circa 1990 Apple Macintosh, where certain system variables were defined to be at fixed memory addresses. For example, the global variable `Time` at address `0x20C` gave the number of seconds since January 1, 1904. This can be defined as a named constant:

```
def Time = 0x20C: &ulong
```

It can then be used in an assignment expression:

```
ulong var x;
x = Time;
```

When `Time` is converted to a `ulong` value, the data loaded from address `0x20C` is assigned to `x`.

Typecasts are not the same as explicit conversion expressions such as `sbyte(x)`or `long(y)`, since the latter may change the bit representation of a value. For example, if `y` is an `sbyte` then `long(y)` must sign extend `y`. The typecast operator does not change the bit representation of its argument: it merely tells the compiler to interpret the value as a different type. Since typecast defeats the normal protections of a strongly-typed language, you should use it carefully.

The typecast operator requires that the type of the expression and the new type be the same size in bytes. Here we must consider an implementation detail: Conceptually, GalaxC uses a stack for evaluating expressions. While data types may have any size in bytes, the stack may be constrained to handle only certain sizes. In its default 32-bit implementation, GalaxC requires that all stack entries be multiples of `long`, i.e., multiples of 4 bytes. Therefore, each type in GalaxC has two sizes: a **storage size** (the declared size of the type) and a **stack size**. In the default 32-bit GalaxC implementation, all integers have the same stack size of 4 bytes.

The reason that stack size is important for the typecast operator is that the *stack* sizes of the expression type and the new type must be the same. The *storage* sizes are irrelevant.

4.6.1 Automatic Dereferencing in Typecasts

An interesting case arises when using type casting with variables. Consider the type cast:

```
x: ulong
```

where `x` is an `int` variable. Recall that a variable is really a pointer to its base type, i.e., `x` is of type `&int`. So, does "`x: ulong`" cast the address of `x` into `ulong` or does it cast the value of `x` into `ulong`? The answer is the latter: in the case of a reference type expression, the address is always dereferenced (perhaps multiple times) to yield a base type. In the above typecast, the address of `x` is dereferenced to yield the value of `x` and it is this value that is typecast into `ulong`.

Suppose that, in the above example, we wanted the address of `x` instead of its value. For example, we may

want the address so as to perform address arithmetic. GalaxC handles this case with the **reference** operator '`&`' which converts a reference type `&τ` to a *pointer type* `@τ` without dereferencing it. *[Footnote: We will discuss pointer types in detail in Chapter 8]. For now, a pointer type is the same as a reference type except that it is not automatically dereferenced.]* For example, this code assigns the address of `x` to pointer variable `y`:

```
y = &x
```

You can typecast the pointer produced into any type with the same size. For example,

```
&x: ulong
```

obtains the address of `x` as a `ulong` value. Variable `x` is *not* dereferenced since `&x` is a pointer type rather than a reference type.

4.7. The Comma Operator and Quasi-types

We will now consider the **comma** expression, which has the form:

```
a, b
```

where *a* and *b* are any expressions. The comma operator simply evaluates its operands, leaving the values on the evaluation stack. *[Footnote: Each value on the stack is a multiple of 4 bytes in the default GalaxC implementation.]* The expression:

```
1, 2
```

leaves `int` values `1` and `2` on the stack, with the last value `2` on top. Both operands are dereferenced. For example, if `x` and `y` are `int` variables, the expression:

```
x, y
```

leaves the values of `x` and `y` on the stack, with `y` on top. To push the addresses of `x` and `y`, use the reference operator:

```
&x, &y
```

Normally, comma evaluates its operands from left to right and pushes them in that order, as described above. However, if you enclose a comma expression in parentheses -- e.g., "`(3, 4)`" -- GalaxC evaluates the operands in reverse order and pushes them right to left, so that `3` is the top of stack. This mimics the way C usually evaluates function arguments. By default, the GalaxC stack grows from high addresses to low addresses, so "`(3, 4)`" leaves the values on the stack with left to right operands at increasing memory addresses.

In addition, if the comma operator is in parentheses then the default GalaxC implementation converts each argument that is less than 4 bytes long to its 4-byte supertype if it has one. For example, GalaxC converts an `sbyte` to a `long` by sign-extending it. This is done after dereferencing.

The type of a comma expression is derived by combining the types of its operands. If one of the operands is `void`, then the type of the comma expression is the type of the other operand. If neither type is `void`, then the type of the comma expression must have a size equal to the sum of the stack sizes of the operands. In

the above example, "`x, y`" produces an 8-byte result. We need to find a type that represents the size of this expression.

Every data object in GalaxC must have a type. Usually these are named types such as `int`, `&short`, and `string`. However, sometimes it is desirable to create objects that have a type different from any of the named types. In most strongly typed languages this is not possible -- expressions must always evaluate to a predefined, named type. GalaxC, on the other hand, introduces the concept of a **quasi-type** or **Qtype**. Qtypes are used to give a type to an expression that does not evaluate to a declared type. For example, the above expression "`x, y`" has type `Qtype(8)`, i.e., an 8-byte quasi-type.

Each quasi-type has a size associated with it, indicating the size in bytes of the quasi-typed expression. An expression with no value is of type `Qtype(0)` which is also known as `void`. In GalaxC, all quasi-types have non-negative size. *[footnote: Galaxy '91 had negative quasi-types, which were used for implementing low-level stack operations. GalaxC uses a different approach for this, and as such does not have a use for negative Qtypes.]*

Quasi-types cannot be part of type hierarchies, i.e., they cannot be a subtype or supertype of any other type. (Except for `void`, which is the supertype of all root types.) They cannot be used as types of variables or arguments. However, a Qtype can be recast into another type with the same stack size using the typecast operator. For example, here is a macro that defines construction of `complex` numbers given `float` real and imaginary parts:

```
float arg {Re, Im},
def Re + i Im = (Re, Im): complex
```

The expression "`(Re, Im)`" has type `Qtype(8)` assuming 4-byte `float` numbers. Assuming type `complex` is 8 bytes long, the expression `(Re, Im)` can be recast as `complex` since both typecast operands have the same stack size. Since they are in parentheses, `Re` and `Im` are pushed on the stack in reverse order, so they appear in memory in the order `Re` followed by `Im`.

Quasi-types, combined with the colon and comma operators, provide a well-defined, structured mechanism for circumventing the restrictions of classical strongly-typed languages while maintaining the advantages. Qtypes are an important unifying concept of GalaxC and are essential for easily implementing the compiler in itself.

Chapter 5
Functions and Macros

"But 'glory' doesn't mean 'a nice knock-down argument'," Alice objected.

"When *I* use a word," Humpty Dumpty said, in rather a scornful tone, "it means just what I choose it to mean -- neither more nor less."
Lewis Carroll, Through the Looking-Glass

One of the most powerful and useful features of GalaxC is the way functions and macros are defined. Significantly different from other languages, programmers can define any syntactic patterns for function and macro calls, allowing GalaxC programs to model closely the natural language and expression forms of a given problem domain. Essentially, by defining the patterns for function and macro calls, the programmer extends GalaxC into a special-purpose language for a particular problem domain. Coding, comprehension, debugging, and maintenance are all improved because the notation is in the expected natural form. Other languages require the programmer to *encrypt* the problem into the rigid notation of the programming language, a tedious and error-prone process. In GalaxC, if you need a new notation or are unhappy with an existing one, it is easy to create your own. Here are some examples of the syntactic possibilities for GalaxC functions and macros:

```
e^x      x + i y      sin x      sin² x      log₂ N      a[i]
      draw line from z1 to z2        open file "xyz"
```

Functions and macros have very similar structure. A function or macro definition consists of a **pattern** for defining the syntax of a function or macro call and a **body** for defining the meaning of the function or macro. A function body is a block of code which is compiled once and can be called from many places in the program. A macro body defines a block of source code which is substituted in-line and compiled for each macro call. You typically use functions where there is a large body which should be shared. You use macros when there is a small body and you want to avoid the run-time overhead of function calls.

Function and macro definitions follow this general form:

[*argument declarations*]
`fn` *function call pattern* `=` *function body*

[*argument declarations*]
`def` *macro call pattern* `=` *macro body*

Any number of argument declarations may precede a function or macro definition -- not necessarily immediately, as described below. The definition's pattern and body can have any legal GalaxC syntax, so they are very flexible. The various aspects of these definitions will be described in more detail in the remainder of this chapter.

5.1. Arguments

Both functions and macros can have **arguments**, which are symbols used to pass values to or from a function or macro. Arguments are declared in precisely the same way as variables and precede the function or macro definitions that use them. This is obviously different from most function-oriented languages such as Pascal and C which define argument types as part of the function definition. GalaxC's approach prevents pattern definitions from becoming cluttered with type declarations so as to improve readability and prevent confusion. GalaxC also allows multiple definitions to share argument declarations.

Arguments are declared with one of the following syntax forms:

```
τ arg name
τ arg {list of arguments}
```

This declares a single argument or a list of arguments separated by commas to have the type τ. Here are some examples:

```
int arg {x, y},          // Declares two int arguments.
ubyte arg p,             // Declares a ubyte argument.
complex arg {z1, z2}     // Declares two complex arguments.
```

Like variables, arguments may have any syntax, though usually they are simple identifiers. Arguments, functions, and macros must currently be declared at the global program level; they cannot be nested inside a block.

Argument declarations, and function or macro definitions are delimited by either commas or semicolons. The difference is that the semicolon operator *hides* all preceding argument declarations so that they are not visible to the following function and macro definitions. The comma operator *retains* all arguments declared since the last global-level semicolon so that they can be shared.

For example, in the following program fragments the arguments `x` and `y` are used by both functions `f` and `g`:

```
int arg {x, y},
fn f(x, y) = body of f,

complex arg z,
fn g(x, y, z) = body of g
```

Note that `f(x,y)` is followed by '`,`' to allow `g(x,y,z)` to share arguments `x` and `y`.

On the other hand, the '`;`' after the first "`def x+y = ...`" in the following example hides `ulong` arguments `x` and `y` from the second "`def x+y = ...`":

```
ulong arg {x, y},        // Declares ulong arguments x and y.
def x + y = body;        // Pattern x+y uses ulong arguments.

// x and y are now hidden.

int arg {x, y},          // Declares int arguments x and y.
def x + y = body         // Pattern x+y uses int arguments.
```

5.1.1 Variable Argument Lists

Like C, GalaxC allows the last argument in function call to be a *variable argument list*. C uses this for functions like `printf` that have one or more fixed arguments and zero or more optional arguments. Here is an ANSI C definition of `printf`:

```
int printf(char* format, ...)
```

The fixed argument `format` encodes how to interpret any optional arguments, e.g.,

```
printf("There are %d cats named %s\n", n, name);
```

expects argument `n` to be an `int` and argument `name` to be a string. It is the programmer's responsibility to ensure that the argument list makes sense for the given `format`.

In GalaxC you define a variable argument list as a single argument of type `varglist`, e.g.:

```
string arg fmt, varglist arg va,
fn printf(fmt, va) = ...
```

In a function call, a `varglist` matches a comma expression or a single expression. You may also omit the `varglist` entirely along with the comma before it, e.g., `printf("Hello\n")`. If you use a `varglist`, it must be the last item in a comma expression.

GalaxC treats a `varglist` argument exactly as if it were a comma expression in parentheses, i.e., it dereferences and pushes the arguments from right to left and converts subtypes of `long` to `long`, perhaps sign-extending them.

Currently, you cannot define GalaxC functions with a `varglist` argument. They are only used for calling external C functions library functions, as described in Section *5.6.2*.

5.2. Functions

As with most programming languages, GalaxC functions are used to reduce the overall complexity of a program by partitioning it into small, easily understood segments. Functions also reduce the size of a program by replacing repeated blocks of code with a single function called from many locations. GalaxC extends this concept by allowing function calls to match the notations of the problem domain.

To define a function, you must specify its arguments, its pattern, its body, and its type. The general form is:

[*optional argument declarations*]
τ `fn` *pattern* = *body*

This defines a function of type τ. *Pattern* is any valid syntax and may include any arguments currently defined. It is not necessary to use all visible arguments. If *pattern* is more complex than a *disj*, place braces around it. GalaxC ignores the outer braces of a *pattern*. If the pattern requires braces, put in two sets.

Body is an expression which defines the behavior of the function given its arguments. *Body* can have any valid syntax, and does not require braces or parentheses unless *body* is more complex than *colon*. If you put braces around a function body, it becomes a *block*.

You may omit type τ if it can be inferred from body, either because the body is a simple expression or has a `return` statement, described later. If the body is a block, the inferred type is `void`. You must specify τ explicitly for forward and recursive definitions, described later. *Body* can only see *pattern* if τ is specified.

Here is a simple example:

```
int arg x,
fn x² = x*x
```

The function argument is "x", the pattern is "x²", and the body is "x*x". Functions can have any number of arguments, including none. For example, you can define a function `Time` which has no arguments and returns the time of day. An argument can occur at most once in the pattern. It is not possible to have the pattern `x+x` if `x` is an argument.

Any syntax in the pattern which is not an argument is **literal syntax**, i.e., it must be matched exactly. In the previous example, the superscript `2` is literal syntax. If you forget an argument, or misspell it, or accidentally declare it as a `var` instead of an `arg`, the pattern may end up with unexpected literal syntax.

5.2.1 Function Calls

A function call is any expression that matches a function pattern. For example, the expressions:

```
a²      4²      (-5)²      (-b)²
```

are all function calls that match pattern `x²`, assuming that `a` and `b` are of type `int` or a subtype of `int`. The expressions `a`, `4`, `(-5)`, and `(-b)` in the example are called **actual arguments** -- their values are passed as **function argument** `x` to function `x²`.

To match a function pattern, an expression must have exactly the same literal syntax and each actual argument must be an expression that is the same type or a subtype of the corresponding function argument. This pattern matching concept allows function calls to take on a tremendous variety of forms. Multiple functions can share the same syntax, as is discussed in Section 5.4.

Calling a function usually consists of computing the value of each argument of the function from right to left, leaving the results on the evaluation stack, and then calling the subroutine which is the compiled form of the function body. When execution of the function body is complete, it pops the arguments and returns control to the calling routine. The result of the function is returned on the stack. An actual argument that is a subtype of a function argument's type is automatically converted to the supertype.

[Footnote: While conceptually arguments are evaluated from right to left, in practice the compiler has the option to evaluate them in any order or even in parallel, so you should not write code that depends on the evaluation in any specific order. Also, values may be passed in registers rather than on an in-memory stack. By default, GalaxC object code pops arguments off the stack before returning (also known as `stdcall`). However, if the function has a variable number of arguments then the caller pops them instead (also known as `cdecl`).*]*

5.2.2 Passing by Value and Reference

Arguments can have any defined type and use *pass by value* to transfer arguments from the caller to the function, i.e., copies of the argument values are pushed on the stack. Within a function body, an argument is treated as if it were a local variable that was initialized by the function's caller. You can get the address of an argument `x` using `&x`. If an argument is not declared `const`, you can assign a value to it without affecting the actual argument in the caller. For example, here is a function which prints a string `n` times:

```
string arg s, int arg n,
fn print s (n times) = while n-- > 0 do printf("%s", s)
```

The argument `n` is used as a local variable and can be modified by the function body. Since `n` is passed by value, the value of argument `n` was copied onto the stack when the function was called and the copy is discarded when the function completes.

Modifying `n` in the function body does not affect the function's caller. This is seen clearly with the following incorrect function which is intended to swap arguments `x` and `y`:

```
int arg {x, y},
fn BadSwap(x, y) = {var t = x; x = y; y = t}
```

This only swaps the stack copies of `x` and `y`. It has no effect on the caller's variables. For example, if you call "`BadSwap(i, j)`" where `i` and `j` are `int`, there is no effect on `i` and `j`.

In C you can use pointers to swap the values of variables outside the function:

```
void Cswap(int *x, int *y)
{int temp = *x; *x = *y; *y = temp;}
```

You then call `Cswap(&i, &j)`, where `&i` and `&j` are the addresses of the variables `i` and `j` that we want to swap. This approach requires dereferencing each instance of `x` and `y` explicitly using the '*' operator, and requires using the C address operator '`&`'. Since C is a system programming language, programmers may see this as an advantage since it exposes the programmer's intent explicitly.

While GalaxC permits this approach using GalaxC pointer types (§8.3), it's often nicer to get the effect of *pass by reference* by using reference types (§4.5). Instead of declaring arguments `x` and `y` as "`int arg {x, y}`", declare them as:

```
&int arg {x, y}
```

or its equivalent form:

```
int arg {&x, &y}
```

The value of `x` and `y` is now `&int` instead of `int`, and its reference type is now `&&int` -- a *double reference*. This means that argument `x` -- which is treated as a local variable `x` -- is a memory location that contains the address of the memory location containing the `int` data to be swapped. While this is almost the same as a C pointer to an `int`, there is an important difference: GalaxC *automatically dereferences* reference types, while C pointers must be explicitly dereferenced using `*x`.

As described in Section 4.5, a reference type `&τ` is a subtype of `τ`. Similarly, `&&τ` is in turn a subtype of `&τ`. When `x` is used in a context that requires an `int`, such as "`x+1`", GalaxC converts `x` to `int` by first loading the `&int` address *p* stored in variable `x`, and then loading the `int` value from *p*. To use `x` in an `&int` context, such as "`x = y`", GalaxC just loads the `&int` value stored in variable `x`.

We can now rewrite the `Cswap` function using a GalaxC reference type:
[footnote: This is similar to C++ pass-by-reference notation.*]*

```
int arg {&x, &y},
fn GoodSwap(x, y) = {var t = x; x = y; y = t}
```

While this looks almost exactly like `BadSwap`, the fact that we are passing `&int` arguments instead of `ints` makes all the difference. Functionally, it is identical to `Cswap`: the only difference is that GalaxC's automatic dereferencing eliminates the need to dereference explicitly each use of `x` and `y` in the function body.

Calling `GoodSwap` is likewise simple: if `i` and `j` are `int` variables, we simply call "`GoodSwap(i, j)`" -- there is no need for `&i` or `&j`. This works because arguments `x` and `y` expect `&int` actual arguments. Since `i` and `j` are variables, their type is really `&int`, so they already match the function arguments and GalaxC just needs to push the addresses of `i` and `j` onto the stack.

Note that while the effect is pass by reference, we are actually still using pass by value. It's just that the value being passed is a reference type. However, for convenience we will refer to this as "pass by reference", and refer to arguments with reference types as *reference arguments.*

One of the most useful applications of reference arguments is to provide additional outputs to a function besides its return value. In some cases, it's nice to have these outputs be optional by passing a `NULL` address for the argument. This is easy to do in C:

```
void f(int x, int *out1, int *out2)
{
    ...
    if (out1) *out1 = ...;      // Return out1 if defined.
    if (out2) *out2 = ...;      // Return out2 if defined.
}
```

With GalaxC we have a problem. The comparison "`out1 != NULL`" doesn't work, because `out1` gets dereferenced twice to an `int` value which does not match GalaxC's `NULL`, which must be a pointer. The comparison "`&out1 != NULL`" doesn't work either, because `&out1` is the address of argument `out1` on the stack, not the address contained in `out1`.

To handle this case, GalaxC provides a macro `defref(out1)` which indicates whether `out1` is defined by looking at the once-dereferenced value of `out1`. Here is the above C example rewritten in GalaxC:

```
int arg {x, &out1, &out2},
void f(x, out1, out2)
{
    ...
    if defref(out1) then out1 = ...;     // Return out1 if defined.
    if defref(out2) then out2 = ...;     // Return out2 if defined.
}
```

For the curious, here are the definitions of generic macro `defref` and its complement `nullref`, which use a number of concepts to be explained in later chapters. First, we define generic macro `refadx(x)`, which gets the address pointed to by reference argument `x`:

```
type arg T, &T arg x,
def refadx(x) = &x: &address
```

Argument `x` is any reference type `&`τ, including double references like `&&int`. "`&x`" converts `x` to a pointer type, which is not automatically referenced. "`&x: &address`" converts the `&&int` pointer to an `&address`, which will get dereferenced once when used in a context that requires type `address`. Type `address` is a subtype of `ulong` which treats addresses as unsigned integers.

Now we can define `defref(x)` and `nullref(x)`:

```
type arg T, &T arg x,
def defref(x)  = refadx(x) != 0,
def nullref(x) = refadx(x) == 0
```

These macros simply see if the address pointed to by `x` is non-`NULL` or `NULL`. We compare to 0, since `address` is a subtype of `ulong`.

5.2.3 The Function Body

The body of a function is simply an expression. This expression is evaluated each time the function is called. The body can refer to the function's arguments any number of times. Whenever an argument is required, its value is obtained from the stack. *[footnote:* Parts of the stack may be implemented as registers.*]* Arguments are evaluated once -- when the function is called.

The value returned by a function is normally the value of the body expression. For example, the function `x^^2` defined earlier returns value `x*x`. The `GoodSwap` function has no value because blocks always have type `void`. *[footnote:* Strictly speaking, it does return a value of type `void`. However, since `void` is zero bytes long it's the same as saying it does not return a value.*]*

Another way to specify the value of a function is using a `return` statement, which terminates execution of the function and optionally returns a value. These function definitions are equivalent:

```
int arg x,
fn x² = x*x,
fn x² = return (x*x)
```

Use `return` when there is no convenient way to write the function body to return the desired value implicitly, or if a function should terminate early and possibly return a result, e.g.:

```
float arg {a, b, c},
fn x1(a, b, c) =
{
    var r = b² - 4*a*c;
    if r < 0 then return 0.0;
    return ((-b + sqrt r)/(2.0*a))
};
```

Any function body that is a block returns `void` by default and requires a `return` statement to return anything else.

A `return` statement can have an argument expression (any type) or can be used by itself, in which case its returns `void`. The value of the `return` statement itself is `void`. Note that `return` is *not* a reserved word, so the expression "`return x`" is syntactically a *concat*. If `x` is more complex than *prefix* you must put it in parentheses.

As with comma and colon expressions, GalaxC dereferences the `return` statement's argument to its base type unless the function has an explicit type τ. In this case, all `return` expressions must return τ or a subtype of τ -- the latter are automatically converted to τ. If the function has no explicit type τ, all `return`

statements must return the same type. [*fn: Exception*: A function that returns a pointer may also have `return NULL` statements. The non-`NULL` return values must have the same pointer types.] If the body is a block that has a `return` statement anywhere that returns a non-`void` type, the body usually needs to end with a `return` statement with the same type.

A function can return any type, including a quasi-type, though this is not very useful except for `void`.

5.2.4 Recursive Functions and Forward Definitions

Most functions and macros do not need to declare their return types. Usually, type can be inferred from the body, making explicit declaration unnecessary. However, in some cases it is necessary to specify the type of a function before its body can be compiled. This is particularly true of **recursive** functions, which are defined using themselves. Consider the following definition of the factorial function:

```
int arg n,
fn n! = if n == 0 then 1 else n*(n-1)!
```

While this is a very simple and elegant definition of factorial, GalaxC cannot compile the body since it requires the type of "`n!`" to be defined before "`(n-1)!`" can be compiled. In fact, the body cannot even see the pattern `n!` unless we declare the type of the function explicitly, like this:

```
int arg n,
int fn n! = if n == 0 then 1 else n*(n-1)!
```

GalaxC functions can also be declared before they are defined, using a **forward definition**. This is useful in many applications, and required for **mutually recursive** functions where two or more functions call each other. For example, here are two mutually recursive functions `f` and `g`:

```
// Common arguments for functions f and g.
int arg {x, y},

// Forward reference of f.
int fn f(x, y) = forward,          // Note: forward is not a reserved word.

// Definition of g.
fn g(x, y) = if x == 0 then y else f(x-1, y),

// Actual definition of f.
fn f(x, y) = g(x, y-1)
```

Forward definition functions, like recursive functions, must have their result types declared explicitly in the forward definition. The forward and actual definitions of a function must be in the same source code file. Macros cannot have forward definitions.

Forward function definitions are not implemented in the current version of GalaxC. For now, use function pointers instead.

5.3. Macros

The purpose of a GalaxC **macro** is to define a convenient, possibly abstract notation for a program construct. Once the notation has been defined, it can be used anywhere in the rest of the program. Good macro-defined

notations make programs easier to write, understand, and debug by matching the familiar notations of a problem domain, and avoiding the tedious and error-prone encryption process.

A macro is defined exactly the same way as a function except that it uses reserved word `def` instead of `fn`:

[*optional argument declarations*]
`τ def` *pattern* = *body*

For example, here is a macro definition for creating a complex number:

```
int arg {x, y},
def x + i y = (x, y): complex
```

This macro defines the notation "`x + i y`" to mean construction of a `complex` number, implemented by pushing the real and imaginary parts onto the stack (in reverse order) and calling the result `complex`.

As with function calls, a **macro call** is simply an expression that matches a macro pattern. To match a macro pattern, all arguments must be either the same type or a subtype of the macro argument types and all literal syntax must match exactly. Here are some macro calls to the "`x + i y`" macro defined above:

```
3 + i 4     1+i 0     a + i b     (3+5) + i(6+7)
```

To compile one of these macro calls, e.g., "`3 + i 4`", GalaxC binds the actual arguments `3` and `4` to the macro arguments `x` and `y`, and then re-compiles the macro body with the actual arguments substituted for the macro arguments. GalaxC compiles the expression "`(x, y): complex`" with `3` substituted for `x` and `4` substituted for `y`. This results in compilation of the expression "`(3, 4): complex`". The macro provides notational convenience and clarity: it is easier to understand and use the notation "`x + i y`" than the notation "`(x, y): complex`".

An actual argument that is a subtype of a macro argument is automatically converted to the macro argument type. For example, in the call "`a + i b`" where `a` and `b` are `int` variables, GalaxC automatically converts each reference type `&int` to `int` by loading the value of `a` or `b`. This is done *each time* the argument occurs in the body.

GalaxC macros are quite different from C macros (`#defines`). C macros are instantiated by the C preprocessor and use string substitution. They have no knowledge of C syntax or semantics, and as such can easily be the source of tricky bugs which are hard to track down. For example, C macros have no knowledge of operator precedence so it is generally necessary to put lots of extra parentheses. C macro calls are limited to identifiers and C function call notation. Even so, C macros are quite powerful and are useful for getting around limitiations of the C language. GalaxC macros are powerful in a different way, particularly for their general syntax.

While GalaxC macro bodies are usually very short, they can be arbitrarily complex if desired. They can include local variables and labels for `goto` statements, which we use in Chapter 7 to define iteration statements as macros. A macro body can have `return` statements, which have the same behavior as with function bodies except that in macros `return x` does not automatically dereference `x` to its base type. This allows a macro to return a reference type `&τ` which can then have a value assigned to it.

A macro's pattern is not visible in the macro's body, even if the macro's type is defined explicitly. Macros cannot be recursive.

5.3.1 Functions versus Macros

While functions and macros are defined in the same manner, their behavior is markedly different. A function body is compiled once into compiled code; each call to the function involves pushing arguments onto the stack, calling the compiled code, and popping arguments leaving the result on the stack. A macro body is not compiled when the macro is defined. Instead, it is merely analyzed to determine if it contains any errors and to derive its type from the macro body. Each time the macro is called, the body is re-compiled *in place* with actual arguments substituted for the macro arguments. Each call results in the generation of a new copy of the macro body with customized argument substitution.

Within each call, each reference to a macro argument generates new code to evaluate that argument. If a frequently referenced actual argument is a complicated expression, considerable extra code will be generated unless an optimizing compiler detects these common subexpressions. Worse, if the actual argument has a side-effect -- e.g., `i++` which increments the value of variable `i` -- multiple evaluations of the argument in the macro body can result in surprising behavior. In general, it's best to make sure each macro body only uses its argument once, perhaps by creating local variables.

Functions and macros each have their purposes in GalaxC. Functions are most useful when the body is large and makes many references to function arguments. Function calls have a fair amount of overhead: pushing arguments, calling the function itself, accessing arguments, returning results, and popping the arguments. Macros are most useful when the body is small and the overhead of function calls is prohibitive. Macros have unacceptable overhead if the body is large (since each call creates an additional copy) or if arguments are referenced many times (each reference creates a copy of the code to evaluate that argument).

In most cases, there is no logical difference between macros and functions. The above complex construction macro would be logically equivalent if defined as a function:

```
int arg {x, y},
fn x + i y = (x, y): complex
```

The difference is in performance. The macro version of "`x + i y`" merely requires the evaluation of "`(x, y): complex`" which consists of evaluating `x` and `y`, leaving both on the stack, and calling the `Qtype(8)` result `complex` (which does not generate any code). The function version requires creating space for the return result, evaluating `x` and `y`, performing the function call, accessing argument `x` from the stack, accessing argument `y` from the stack, calling the result `complex`, copying the result to the space reserved for the return value, popping arguments `x` and `y`, and returning from the function. The function version of "`x + i y`" wastes considerable time and space with additional stack manipulation. In this case the macro version is obviously better.

In other instances functions are preferred. The quadratic equation example shown earlier is better suited to a function definition, since it is large and makes multiple references to arguments. Recursive macros are not possible, so the factorial example is only possible as a function. In addition, factorial refers to its argument many times.

Later in this chapter we will look at `inline` functions, which are somewhere between functions and macros.

5.4. Search Order and Overloading

When GalaxC analyzes an expression, it must search the defined patterns to find which program object -- e.g., variable, argument, function, or macro -- matches the expression. To predict which definition is in fact

matched, we need to understand GalaxC's *search order*. Given an expression to match, GalaxC searches in the reverse order of definition, i.e., the objects that have been defined last are matched first. For example, it is possible to *hide* an existing function by creating a new function with the same syntax and argument types. When two or more patterns have the same syntax, we say that pattern syntax is **overloaded**.

Overloading can be very useful to produce functions and macros that have the same pattern syntax, but different argument types. For example, here are functions for printing different types of data:

```
string arg x,
fn print x = code to print string;

int arg x,
fn print x = code to print int;

char arg x,
fn print x = code to print char
```

The programmer only needs to remember one syntax for printing data, and lets the compiler worry about which one to use in a given context. GalaxC has no problem dealing with overloading since it considers the *types* of arguments along with literal syntax when matching patterns.

When using overloading, one must be careful that later definitions do not unintentionally hide previous ones because of GalaxC's search order. For example, the expression "`print 'a'`" correctly matches the "`print char`" function. However, if the "`print int`" and "`print char`" functions were reversed, "`print 'a'`" would instead match "`print int`" since that would be as defined after "`print char`" and `char` is a subtype of `int`.

5.5. Function Pointers

GalaxC allows functions to be called indirectly using the type `fnptr` (**function pointer**). Like any other pointer, a `fnptr` can be assigned to a variable, passed as an argument, or returned as a function or macro value. You can then call a function indirectly using the `call` expression. Function addresses behave the same way as function pointers in C but require a distinct notation because, unlike C, GalaxC overloading allows many functions to have the same name. To create a `fnptr` value, you must specify a *dummy function call* that matches the literal syntax and arguments of the function pattern.

The general form of an expression that creates a `fnptr` is:

```
fnptr  function call pattern
```

where *function call pattern* is any pattern that matches the function's pattern. For example, the following code creates a `fnptr` variable `fact` and initializes it with a pointer to the factorial function:

```
var fact = fnptr 0!
```

The argument `0` is a dummy argument used to satisfy the need for an `int` expression in the pattern "`n!`". Any `int` expression could be used. The `fnptr` expression does not generate code to call the dummy function -- it is present only to match the function's pattern.

The token `fnptr` is *not* a reserved word, so the expression "`fnptr` *pattern*" is syntactically a *concat*. If *pattern* is more complex than *prefix* you must put it in braces: GalaxC always ignores the outer braces for a

`fnptr` pattern.

To call a `fnptr`, use one of the following forms:

```
call f(args; τ)
call f(args)
call f: τ
call f
```

where *f* is a `fnptr` expression (usually a variable), *args* is a list of arguments separated by commas, and τ is the return type. As with direct function calls, *args* (if present) are evaluated and pushed right to left. They are equivalent to a comma expression in parentheses (§4.7): each argument is automatically dereferenced to its base types and converted from a subtype of `long` to `long`. Note that there is no separator between *f* and the left parenthesis.

If τ is absent the indirect call returns `void`. If both *args* and τ are absent, *f* is called with no arguments and returns `void`. As with `fnptr`, `call` is not a reserved word so *f* must be a *sub* or *prim*: put it in parentheses if necessary.

Here is an example of calling the above factorial function:

```
var x = call fact(5; int)
```

This is equivalent to "`var x = 5!`".

In the current version of GalaxC, `call` does not check the types of the arguments, nor does it check the type of the result. All `fnptrs` have the same type. You must be very careful to ensure that each `call` matches the function you assign to it. Since `call` arguments are automatically dereferenced, you must convert reference arguments to pointers using '`&`' to prevent automatic dereferencing. For example, to call `GoodSwap(x,y)` indirectly, use the following code:

```
int var {a, b};
var swap = fnptr GoodSwap(a, b);
call swap(&a, &b)
```

Make sure that if the function to be called returns a result, the correct result type is specified in the `call`. Otherwise `call` will write the result over other stack data causing undefined behavior. We plan to remedy this some day when GalaxC has parameterized types.

Provided that the above warnings are heeded, function pointers can be a very powerful programming construct. They allow the programmer to defer bindings to run time and make program structure dynamic. They are an indispensible part of the GalaxC Simplified Window Manager, described in [JFB 11: G-SWIM]. In the current version of GalaxC, function pointers are the only way to implement forward function definitions.

5.6. Inline Functions

GalaxC `inline` functions are somewhere between called function and macros. Like called functions, they use the stack model where arguments are evaluated once and the result is returned on the stack. Like macros, they make an in-line copy of *body*. They are often used for functions that have small bodies but

reference arguments multiple times. Special forms of `inline` functions are also used for low-level code generation.

An `inline` function is defined exactly the same way as a called function except that it uses reserved word `inline` instead of `fn`. For example, here is a definition of the "`&=`" operator with `int` arguments:

```
int arg {&x, y},
inline {x &= y} = (x = x & y)
```

While we could have defined "`x &= y`" as a macro, `inline` is safer since it only evaluates argument `x` once. This way we can write "`*p++ &= 0xFF`" without remembering that `int` pointer `p` might get incremented twice.

This example has some interesting syntactic and semantic considerations. First, the braces around "`x &= y`" are needed because an `inline`'s pattern must be *disj* or simpler. GalaxC ignores the outermost set of braces in a pattern definition. Second, the parentheses around "`x = x & y`" are optional because this expression is simpler than the required *colon* expression. However, we put them in to avoid visual confusion with the `inline` definition's equal sign.

It's important that we used parentheses rather than braces for the body. Like C, a GalaxC assignment expression treats the assigned value as the value of the expression, allowing compound assignments. "*op*=" assignments have this property as well. By putting "`x = x & y`" in parentheses the value (and type) of the assignment is the value (and type) of the `inline`'s body. If we were to use braces instead, the body would be a block with `void` type and no value.

The actual GalaxC definition of "`&=`" is quite similar to the above example, except that it is defined generically for all types τ. See Chapter 9 for details.

Like macros, an `inline` function's pattern is not visible from its body. Inline functions cannot be recursive.

5.6.1 Intrinsic Operators

By default, GalaxC generates code for a simulated stack machine called the Postfix Stack Interpreter (**PSI**), described in detail in the next chapter. PSI code is in turn be optimized and/or translated into a native instruciton set. A special form of `inline` links low-level GalaxC expressions to PSI operators. For example, here is an `inline` that defines addition of two `ulongs`:

```
ulong arg {x, y},
ulong inline {x + y} = intrinsic ADD
```

This specifies that an expression of the form "`ulong + ulong`" should push the arguments on the stack and then generate a PSI `ADD` instruction, which will add the top two values on the stack leaving a `ulong` result. `ADD` is simply an integer constant with the correct PSI opcode. Unlink most function calls, `inline` arguments are pushed left to right unless the opcode is negative, in which case they are pushed right to left. *[footnote*: The negative opcode only changes the order in which arguments are pushed. The PSI opcode actually generated is the absolute value of the integer constant.*]*

The general form of an `intrinsic` definition is:

[*argument declarations*]

```
τ inline pattern = intrinsic opcode
```

where *opcode* is a constant integer expression. Since `intrinsic` is not a reserved word, *opcode* must be prefix or simpler. In particular, if *opcode* is negative it probably needs to be in parentheses.

One must be very careful defining `intrinsic` operators because they must match they types of arguments and result value exactly or the stack will get totally messed up. Most GalaxC programmers won't ever need to define `intrinsic` operators.

5.6.2 Calling C Functions

Inline functions are also used to define calls to library functions outside GalaxC, so that it can use the standard C library as well as graphics libraries like X11 and Win32. Here are the general forms:

```
[argument declarations]
τ inline pattern = cdecl
τ inline pattern = stdcall
```

In both cases, *pattern* must be a C-style function call "*f*(*argument list*)", where *f* is the name of a function (an identifier), or just *f* by itself. Type τ is the return type of the function. Here are some `cdecl` examples:

```
// Standard C strcpy(dst, src) and strcat(dst, src) functions.
string arg {dst, src},
string inline strcpy(dst, src) = cdecl,
string inline strcat(dst, src) = cdecl;

// Standard C sprintf(buf, fmt, ...) with variable argument list.
string arg {buf, fmt}, varglist arg va,
int inline sprintf(buf, fmt, va) = cdecl
```

To call a C function, write an expression that matches *pattern*, e.g., `sprintf(buf, "Hello\n")`. The first time *pattern* is used, GalaxC searches the shared object (`.so`) or dynamically linked (`.dll`) libraries to find a library symbol that matches *f*. Each call pushes the arguments on the stack in right-to-left order and calls the C function. The two identifiers `cdecl` and `stdcall` (not reserved words) define how to handle function returns. A `cdecl` function requires the caller to pop arguments off the stack, so GalaxC does so. Most C libraries use `cdecl` since it allows functions with variable argument lists. A `stdcall` function pops its own arguments off the stack, so GalaxC only needs to know how much was popped. This approach saves a small amount of code for each function call in exchange for not being able to have variable argument lists. It's mostly used by Microsoft Windows APIs. Currently each form is limited to 10 arguments.

5.7. Summary

This chapter has described the syntax and semantics of functions and macros. Their simplicity belies their power and elegance. Programmers who can free their minds from the patterns engrained in them by previous languages can use them to carry the state of program expression into new realms. The possibilities are limited primarily by the imagination. As a simple example, but demonstrating how old dogs can use new tricks, in the next chapter we will see how GalaxC basic arithmetic is defined using the functions and macros described in this chapter. Recall that typically such things must be fixed in the internal functioning of the compiler. In GalaxC, this implies that if we wish we can easily redefine basic arithmetic without reimplementing the compiler.

Chapter 6
The Postfix Stack Interpreter (PSI)

By default, GalaxC generates code for a simple stack machine called the **Postfix Stack Interpreter** (PSI or Ψ). PSI is a 32-bit stack machine with 16-bit instructions. It is designed to be easy to interpret, easy to optimize, and easy to translate into native instruction sets such as Intel Architecture, ARM, and PowerPC. By providing a simple intermediate form, the *Analysis and Code Generation* (ACG) section of the GalaxC compiler (§10.1.3) can be written without worrying about the details of any particular target architecture.

Object code for PSI is called *PSI-code*. The generic code optimizer is called *PSI-CO*. Each code generator for a specific target architecture is called *PSI-xCG* where *x* is the target.

Given the high performance of modern computers, PSI by itself is often fast enough for many applications. Indeed, the current version of GalaxC does not have any native code generators, though these were certainly planned from the beginning. Code optimization and translating to a specific target have known solutions.

Using an interpretive machine is nothing new: most Java implementations use a bytecode interpreter, and the history of programming languages has many notable examples including SmallTalk bytecode and Pascal p-code. Stack-based programming languages have been around a long time as well, most notably Forth and Minim, the latter used to implement the first version of Galaxy '91.

One unusual feature of PSI is that it is also the internal representation of XXICC objects such as tables, dialog boxes, and even entire XOE documents. Part of the PSI instruction set is reserved for representing each kind of XO. For example, the `TEXT` instruction forms a `TEXT` XO using the string operand at the top of the stack as string data. This operand may be a string literal, or it may be the value of a string expression, e.g., the current date or time. An XOST begins with an XOST opcode, e.g., `HSTACK`, and ends with the `ENDOBJ` instruction. An entire XOE document is a paragraph box beginning with `PARABOX` and ending with `ENDOBJ`.

To process a XOE document, PSI simply executes the PSI code for that document. XO opcodes are interpreted using a vector of function pointers, with a different vector for each kind of processing. PSI also has instructions to tag incremental changes to the memory copy of a XOE document so they can be reversed using the `UNDO` command. These tags are cleaned out when storing a XOE document to a file. XOE documents in files are "data only", i.e., they only contain PSI data instructions and XO opcodes. Otherwise it would be easy to transfer a virus as part of a XOE document.

Using a single representation for code and data is an old idea, mostly explored using LISP. The Galaxy CAD System [JFB 92] used the executable data concept extensively for processing CAD data.

6.1. PSI Instruction Formats

PSI instructions contain opcodes and/or data, as indicated by the MSbs. In this chapter we will use "word" to mean 16-bits, i.e., the word size of PSI instructions, and "long" to refer to 32-bit PSI stack operands, which may be signed (`long`) or unsigned (`ulong`). Like most modern computers, the PSI stack grows towards lower memory addresses. It is always aligned to a long address.

0	0	*14-bit number*
1	1	*14-bit number*

A **small integer** is a 16-bit 2's complement signed integer from –16,384 to +16,383. PSI sign-extends

the value to 32 bits and pushes it onto the stack.

1	0	*14-bit opcode*

An **opcode** operates on the top long values on the stack, leaving the result on the stack. For example, `ADD` adds the top two longs leaving a single long result. It has the net effect of popping one long. The low 256 opcode values are reserved for XOs.

[Improve table formatting some day.]

0	1	*1 ≤ N ≤ 8191 (13 bits)*	1	*N-word string constant (padded)*

A **string constant** includes an *N*-word string terminated with `NUL` and padded to an even count if needed. PSI pushes a long *pointer* to the constant onto the stack. The string data itself resides in the PSI instruction space. Strings are limited to 16,382 bytes including `NUL`. Note that *N* is a word count: you can convert to a byte count by `AND`ing with `0x3FFE`.

A string constant can actual contain any kind of data: *N* determines its length, not the `NUL`.

0	1	*1 ≤ M ≤ 4095 (12 bits)*	0	0	*M-long stack constant (2*M words)*

A **stack constant** includes *M* longs of stack data, e.g., the contents of a multi-word data structure. For example, `0x4004` followed by *X* is the single-long stack constant *X*. PSI pushes the constant onto the stack with the stack copy appearing exactly the same as in the PSI code. Since stacks grow towards low memory, the first long of the constant becomes the new top of stack. Note that *M* is a long count: you can convert to a byte count by `AND`ing with `0x3FFE` just like string constants.

0	1	*12-bit opcodex*	1	0	*Long operand X (two words)*

An **opcodex** is similar to an opcode except it has a long operand *X*. *X* might represent a PSI code space address, a pointer to an external C library, data to be pushed onto the stack, or housekeeping information.

6.2. Compiling GalaxC Literals

GalaxC translates literals into code that pushes constants. Integer literals -- including characters -- generate small constant instructions when possible, otherwise stack constants. String literals generate PSI string constants, and concatenated string are combined into a single PSI string constant with maximum length 16,382 including `NUL`. Floating-point literals generate 4-byte or 8-byte stack constants.

6.3. PSI Opcodes

This section lists the PSI opcodes generated by GalaxC code. Since each opcode requires implementation in the interpreter, the code optimizer, and code generators we have tried to keep the number of instructions limited. However, some non-primitive operations such as `i++`, `max`, and `abs` are common enough that it's worthwhile including them even if it adds more opcodes.

In many cases is doesn't matter whether we are dealing with signed or unsigned integers, so we will use "long" to mean "`long` or `ulong`", "short" to mean "`short` or `ushort`", and "byte" to mean "`ubyte` or `ubyte`". If it matters, we will use the `monospaced` type names.

Here is some general PSI terminology, which follows the example of many computer architectures:

PC The *program counter* points to the next instruction to be executed.
SP The *stack pointer* points to the top of the stack, which is the lowest address of the stack that contains a value.
FP The *frame pointer* is a base register for accessing function arguments and local variables on the stack.
TOS is the long at the top of the stack, i.e., the value pointed to by SP.
NTOS is the long next to the top of the stack, i.e., the value pointed to by SP + 1 long.

6.3.1 Basic Arithmetic

`NOP` Perform no operation: leave the stack unchanged.

`ADD`, `SUB`, `MUL`, `DIV`, `REM`, `DIVU`, `REMU`
Compute NTOS + TOS, NTOS – TOS, NTOS * TOS, NTOS / TOS, or NTOS % TOS, replacing the two operands with the result (pops one long). `DIV` and `REM` are signed operations; `DIVU` and `REMU` are unsigned. `MUL` produces the same 32-bit product for signed and unsigned 32-bit TOS and NTOS.

`SHL`, `SHR`, `ASR`
Compute NTOS << TOS or NTOS >> TOS, replacing the two operands with the result. `SHR` is a logical (unsigned) right shift. ASR is an arithmetic (signed) right shift. TOS is only defined for the values 0-31.

`AND`, `OR`, `XOR`
Compute NTOS & TOS, NTOS | TOS, or NTOS ^ TOS, replacing the two operands with the result.

`EQ`, `NE`, `LT`, `GT`, `LE`, `GE`, `LO`, `HI`, `LOS`, `HIS`
Compute NTOS == TOS, NTOS ≠ TOS, NTOS < TOS, NTOS > TOS, NTOS ≤ TOS, or NTOS ≥ TOS, replacing the two operands with the `Boolean` result `TRUE` (1) or `FALSE` (0). LT, GT, LE, and GE, are signed. LO, HI, LOS, and HIS are unsigned.

`NEG`, `COM`, `NOT`, `NEZ`
Compute -TOS, ~TOS, !TOS, or ?TOS, replacing the operand with the result.

`SB2S`, `S2L`, `UB2US`, `US2L`, `L2UL`
Convert integer at TOS to its supertype: `sbyte` to `short`, `short` to `long`, `ubyte` to `ushort`, `ushort` to `long`, or `long` to `ulong`. The run-time stack already represents all integers as long and assumes all bytes and shorts are zero-extended, so `UB2US` and `US2L` perform no operation. Similarly, `long` and `ulong` have the same bit representation so `L2UL` is a `NOP`. They are left in place so that higher level code does not have to know this, which allows future versions of the low-level code to behave differently. On the other hand, `SB2S` and `S2L` must perform sign extension. `SB2S` only sign extends `sbyte` to 16 bits because all shorts must have zero extension. `S2L` sign extends `short` to 32 bits.

`L2US`, `L2UB`
Convert long at TOS to `ushort` or `ubyte`, zero-extending the short or byte so that the MSBs of the long TOS value are set to zero. There is no `L2S` nor `L2SB` because PSI zero-extends all bytes and shorts.

`MIN`, `MAX`, `MINU`, `MAXU`

Find the signed or unsigned minimum or maximum of the TOS and NTOS, replacing the two operands with the result. While min and max can be computed using conditional branches, we feel it is common enough to have special instructions for them. This allows architectures such as ARM and PowerPC to take advantage of conditional execution and improve pipelined execution.

`ABS` Compute the absolute value of `long` TOS, replacing the operand with the result. Like min and max, this allows conditional execution for improved pipelined execution.

`PRINC`, `PRINCS`, `PRINCB`

Pre-increment or pre-decrement the long, short, or byte variable pointed to by TOS by adding NTOS, replacing the two operands with the *updated* value of the variable. This implements expressions like `++i`, `--n`, and `x += 32`. Decrements use a negated value of NTOS.

`POINC`, `POINCS`, `POINCB`

Post-increment or post-decrement the long, short, or byte variable pointed to by TOS by adding NTOS, replacing the two operands with the value of the variable *before it was updated*. This implements expressions like `i++` and `n--`. Decrements use a negated value of NTOS.

`PRINV`, `PRINVS`, `PRINVB`, `POINV`, `POINVS`, `POINVB`

Same as `PRINC`, `PRINCS`, `PRINCB`, `POINC`, `POINCS`, `POINCB` except treat the variable as `volatile` when optimizing code.

6.3.2 Loads, Stores, and Stack Manipulation

We will use "double" to mean any value that requires two longs. This includes `double` floating-point numbers, but also includes (for example) a complex number with `long` or `float` components.

`LOAD`, `LOADS`, `LOADB`, `LOADD`

Given an address at TOS, load a long, short, byte, or double replacing the address with the value. Loading a double has the effect of pushing one long. Zero-extend bytes and shorts to long. Signed `short` and `sbyte` values are sign-extended by `S2L` and `SB2S`, so there is no need for signed loads.

`LOADN`

Given an address at NTOS and a count at TOS, load TOS longs onto the stack, replacing the two operands. The stack copy has the same byte order as memory.

`LOADV`, `LOADVS`, `LOADVB`, `LOADVD`, `LOADVN`

Same as `LOAD`, `LOADS`, `LOADB`, `LOADD`, `LOADN` except treat memory as `volatile` when optimizing code.

`STORE`, `STORS`, `STORB`, `STORD`

Given an address at TOS, store NTOS as a long, short, byte, or double. Pop TOS but not NTOS.

`STORN`

Given an address at NTOS and a count at TOS, store the TOS longs on the stack below NTOS to memory. Pop both TOS and NTOS, but not the data. The memory copy has the same byte order as the stack.

`STORV`, `STORVS`, `STORVB`, `STORVD`, `STORVN`
Same as `STOR`, `STORS`, `STORB`, `STORD`, `STORN` except treat memory as `volatile` when optimizing code.

`PUSH`, `POP`
Push or pop one long onto or off the stack. Pushed `longs` are not initialized.

`POPD` Pop one double off the stack.

`POPN` Pop TOS longs off the stack by setting SP = SP+TOS. Also pop TOS, which is included in the count. If TOS ≤ 0, leave TOS on the stack and push |TOS| longs, which are not initialized. Do not count on TOS retaining its value.

`DUP`, `DUPD`
Duplicate the long or double at the top of the stack, pushing one or two longs.

`DUPN`
Duplicate TOS longs starting at NTOS, replacing TOS with the copy of NTOS, pushing TOS–1 longs.

`LDPOP`, `LDPOPS`, `LDPOPB`
Extract a long, short, or byte field from a multi-long data structure that is on the stack. The stack contains TOS = structure length in bytes (must be a multiple of 4), NTOS = field offset in bytes, and the data structure itself. PSI replaces all arguments including the structure with the contents of the field, zero-extending bytes and shorts to long.

`LDPOPN`
Same as `LDPOP`, except that TOS = field length in longs as an additional argument. Replace all arguments (including the structure) with the contents of the multi-long field.

`LONG` *X*
Push the long value *X* of a global variable onto the stack. *X* is the operand of an opcodex instruction.

6.3.3 Addresses of Variables, Functions, and Types

PSI currently supports two kinds of variables: global variables allocated in a common global space, and stack variables which include local variables and arguments.

`STAKVAR`
Convert the TOS byte offset to a stack variable byte address relative to frame pointer FP which is set by the `FUN` instruction at the beginning of a function. If TOS > 0, FP + TOS is the address of the stack copy of an argument which was passed by value. If TOS < 0, FP + TOS is the address of a local variable. Use `LOAD`, `LOADS`, etc. to load the value pointed to by the address.

`GLOBVAR` *X*
Push the byte address *X* of a global variable onto the stack. *X* is the long operand of an opcodex instruction. While this is mostly equivalent to `LONG` *X* or pushing *X* as a stack constant, the `GLOBVAR` tags the instruction so that GalaxC's linker can tell the long constant is a global variable and should be linked accordingly. Use `LOAD`, `LOADS`, etc. to load the value pointed to by the global address.

`FNPTR` *X*, `DLLADX` *X*, `TYPEPTR` *X*
Push the address *X* of a PSI function, an external C function in a DLL or .so library, or GalaxC type

onto the stack. This is the same as `GLOBVAR`, but tags the instruction for GalaxC's linker.

6.3.4 Program Control: Gotos, Calls, and Returns

We next look at PSI instructions that affect which instruction is executed next. As with most computers, PSI has a *program counter* (PC) which points to the next instruction to be executed. In the following text, PC normally refers to its value *after* fetching the instruction.

`GOTO` *X*

Jump to address PC + *X*, where PC is the instruction *after* `GOTO` *X*. `GOTO` does not affect the stack.

`GOTRUE` *X*, `GOFALSE` *X*

Pop TOS and see if it equals zero. `GOTRUE` *X* jumps to address PC + *X* if TOS ≠ 0; otherwise PSI continues at PC. `GOFALSE` *X* jumps to address PC + *X* if TOS = 0; otherwise PSI continues at PC. Note that PSI considers any non-zero TOS value to be `TRUE`: the value does not have to be `1`.

`POPGOTO` *X*

Pop TOS longs and then jump to address PC + *X*. This is equivalent to `POPN` followed by `GOTO` *X*.

`FUN`

Begin execution of a function: push the current frame pointer FP onto the stack and set FP to point to the new TOS. The caller has already pushed a return address. PSI allows simplified functions that do not begin with `FUN`: they neither save nor modify FP.

`RET`

Return from a function call that began with `FUN`. Assume TOS is the caller's frame pointer and NTOS is the return address to the caller. Set FP = TOS and PC = return address, popping both off the stack.

`RETN`

Same as `RET`, but also pop *N* longs containing function arguments. Assume the stack contains *N* followed by the caller's FP, the return address, and *N* longs containng arguments. `RETN` implements `stdcall`-style functions where the called function is responsible for popping arguments.

`RETNF`

Return from a function that did not begin with `FUN`. Assume TOS is the return address to the caller. Set PC = return address and pop it off the stack. `RETNF` does not affect FP.

`RETV`, `RETVN`

These return values from called functions. Save the long return value NTOS for a function by copying it to address SP+TOS. Pop both TOS and NTOS. Stack offset TOS assumes the stack offset has been popped but not the data. `RETVN` is the same except that the return value is *N* longs. The additional operand *N* is TOS, followed by stack offset, and *N* longs of data, all popped. `RETV` and `RETVN` are normally followed by `RET` or `RETN`, or else a `GOTO` or `POPGOTO` to `RET` or `RETN`.

`IRES`, `IRESN`

These are similar to `RETV` and `RETVN` but are used to return results from inline functions and macros. Save the long result NTOS by copying it to address SP+TOS, and pop the stack to SP+TOS. Stack offset TOS assumes the stack offset has been popped but not the data. `IRESN` is the same except that the result is *N* longs. The additional operand *N* is TOS, followed by stack offset, and *N* longs of data, all popped to SP + stack offset. `IRES` and `IRESN` are normally followed by `GOTO` or end the inline or

macro instantiation.

`CALL` *X*
: Call the function at address *X*. Push PC (the instruction after `CALL` *X*) as the return address and set PC = *X*. `CALL` does not affect FP: that is the called function's responsibility. By default, PSI functions pop their own arguments using `RETN`. Otherwise the caller must pop arguments using `POPN` or `LDPOP`.

`ICALL`
: Indirectly call the function pointed to by TOS. Push PC as the return address and set PC = TOS. The value in TOS is usually set by `FNPTR` *X*. Otherwise `ICALL` is the same as `CALL`.

`CDECL`, `STDCALL`
: Call the C library function at address TOS (popped) with NTOS arguments. TOS is normally set with `DLLADX` *X*. Currently, PSI assumes the C library function returns a long value. *There are a number of other details which are subject to change as GalaxC evolves. See the source code if you're interested.*

`HALT`
: Mark the end of a program. If PSI reaches this instruction the interpreter stops and returns to its caller, either somewhere in XXICC or to the operating system. `HALT` is mostly used to identify the end of a program for system management.

`SWITCH`
: Execute a GalaxC `switch` statement. TOS is an index into a jump offset table, usually stored in program memory as a string constant. NTOS points to the jump table, which contains longs. `SWITCH` sets PC to PC + NTOS[TOS], popping both TOS and NTOS. The code generator must make sure TOS does not access outside the jump table by using `MINU` and a default entry.

`EXCEPT` *X*
: Create an exception value and push it onto the stack. A GalaxC exception consists of 3 longs: a saved FP, a saved PC, and a saved SP. `EXCEPT` *X* has two operands: stack offset TOS and PC offset *X*. `EXCEPT` first replaces TOS with SP + TOS as the saved SP, then pushes PC + *X* as the saved PC, and finally pushes the current value of FP. The 3-long exception can then be stored in a global variable to be raised as needed.

`RAISE`
: Raise an exception given a GalaxC exception value at TOS. Raising an exception consists of setting the current FP, PC, and SP to the values in the exception. The exception must set SP to a higher value than the current SP: the exception and all stack data up to the new SP are popped.

6.3.5 Floating-Point Operators

`FADD`, `FSUB`, `FMUL`, `FDIV`, `DADD`, `DSUB`, `DMUL`, `DDIV`
: Compute NTOS + TOS, NTOS – TOS, NTOS * TOS, or NTOS / TOS replacing the two operands with the result (pops one or two longs). `FADD`, `FSUB`, `FMUL`, and `FDIV` are `float` operations with long operands. `DADD`, `DSUB`, `DMUL`, and `DDIV` are `double` operations with double operands.

`FNEG`, `DNEG`
: Compute `-`TOS, replacing the `float` or `double` operand with the result.

`FEQ`, `FNE`, `FLT`, `FGT`, `FLE`, `FGE`, `DEQ`, `DNE`, `DLT`, `DGT`, `DLE`, `DGE`

Compute NTOS == TOS, NTOS ≠ TOS, NTOS < TOS, NTOS > TOS, NTOS ≤ TOS, or NTOS ≥ TOS, replacing the two operands with the `Boolean` result `TRUE` (1) or `FALSE` (0). `FEQ` -- `FGE` are `float` operations with long operands. `DEQ` -- `DGE` are `double` operations with double operands.

`F2D`, `D2F`

Convert `float` at TOS to `double`, or vice-versa.

`L2F`, `UL2F`, `L2D`, `UL2D`

Convert signed or unsigned long integer at TOS to `float` or `double`.

`F2L`, `D2L`

Convert `float` or `double` at TOS to a signed long integer, truncating any fractional part.

`FFLOR`, `FCEIL`, `DFLOR`, `DCEIL`

Compute *floor*(TOS) or *ceiling*(TOS), replacing the `float` or `double` at TOS with a floating-point value equal to the next integer ≤ TOS or ≥ TOS.

`FMIN`, `FMAX`, `DMIN`, `DMAX`

Find the minimum or maximum of the `float` or `double` operands at TOS and NTOS, replacing them with the result.

`FABS`, `DABS`

Compute the absolute value of the `float` or `double` at TOS, replacing the operand with the result. While this can be done using comparisons, absolute value is often available as a (co)processor instruction.

`FSQRT`, `DSQRT`

Compute the positive square root of the `float` or `double` at TOS, replacing the operand with the result. Square root is often available as a floating-point instruction.

6.3.6 XXICC Objects and Tags

The remaining PSI opcodes are mostly for manipulating XOs in XOE and elsewhere. See [JFB 11: XXICC Objects].

6.4. Defining GalaxC Operators

In most languages, operations on built-in types are buried somewhere deep in the compiler. In GalaxC, they are fully exposed as extensions to the language. This makes the GalaxC compiler easier to understand, maintain, and extend.

Most GalaxC operators are defined as `intrinsic` operators, described earlier in Section 5.6.1. The actual code is in `gxclib.gal`, which defines most of GalaxC's operators and constants, some of its data types, and `cdecl inlines` from the standard C library. In this text we will usually refer to `gxclib.gal` as "`gxclib`" and copy examples from it liberally.

6.4.1 Integer Unary Operators

Let's begin by defining the integer unary operators `+x`, `-x`, `!x`, and `|x|`:

```
ulong arg x,
ulong inline {+x}   = intrinsic NOP,
ulong inline {-x}   = intrinsic NEG,
Boolean inline {!x} = intrinsic NOT,
Boolean inline {?x} = intrinsic NEZ;

int arg x,
int inline {+x} = intrinsic NOP,
int inline {-x} = intrinsic NEG,
int inline |x|  = intrinsic ABS;
```

These specify that each unary expresion should be compiled by generating PSI code to evaluate argument `x` on the stack, automatically converting subtypes to `ulong` or `int`, and then generate PSI opcode `NOP`, `NEG`, `NOT`, or `ABS`. Even though there is a PSI `NOP` instruction, the `NOP` generated for `+x` is actually discarded by the code generator.

Note that there are two sets of definitions, one for unsigned `ulongs` and one for signed `ints` (equivalent to `longs`). This is because GalaxC wants an expression containing integers to have type `int` whenever possible: it should only becomes `ulong` if a subexpression is `ulong`. Since `int` is a subtype of `ulong`, an `int` value of `x` matches the `ulong` version of the intrinsics. Thus the `int` versions must be defined *after* the `long` versions, since GalaxC searches for patterns in reverse order (§5.4). In all cases, the type of the result appears before `inline`.

The `NOT` operator `!y` has a `Boolean` result so there is no need for separate signed and unsigned versions. Similarly, `ABS` is only defined for a signed operand, so there is no need for a `long` version.

There are no short or byte versions of these operators: GalaxC and PSI perform all integer arithmetic as longs (with a few exceptions).

Except for `|x|`, we have shown each pattern in braces. This is optional for all these patterns since they are simpler than *disj*. Including braces may help prevent visual confusion between the pattern and the rest of the definition.

An intrinsic definition normally specifies its *opcode* using a PSI mnemonic, as in the above example. However, the current version `gxclib` uses hard-coded numbers since `gxclib` is currently the only GalaxC file that references most of the PSI opcodes. In the C files that make up the lowest levels of the current GalaxC compiler, each PSI opcode is prefixed with "`PSI_`" -- e.g., `PSI_NOT` -- to prevent collision with other symbolic constants. Our plan is to introduce a "private enumeration" type that allows symbolic constants like `NOT` and `NEG` to be visible only in contexts where they are expected. This will be used when we will replace the current C files with a GalaxC-only version of the GalaxC compiler.

6.4.2 Integer Arithmetic

Next we have the integer arithmetic operators. As with the unary operators, we have separate intrinsic definitions for `ulong` and `int`:

```
ulong arg {x, y},
ulong inline {x + y}    = intrinsic ADD,
ulong inline {x - y}    = intrinsic SUB,
ulong inline {x * y}    = intrinsic MUL,
```

```
ulong inline {x / y}    = intrinsic DIVU,       // unsigned
ulong inline {x % y}    = intrinsic REMU;       // unsigned

int arg {x, y},
int inline {x + y}      = intrinsic ADD,
int inline {x - y}      = intrinsic SUB,
int inline {x * y}      = intrinsic MUL,
int inline {x / y}      = intrinsic DIV,        // signed
int inline {x % y}      = intrinsic REM;        // signed
```

GalaxC compiles each binary expresion by generating PSI code to evaluate and push arguments `x` and `y` onto the stack, in that order, and then generate PSI opcode `ADD`, `SUB`, `MUL`, etc. `ADD`, `SUB`, and `MUL` use the same PSI opcode for `ulong` and `int` operands; `DIV` and `REM` have different opcodes.

6.4.3 Integer Relational Operators

Here are the definitions of the relational operators:

```
ulong arg {x, y},
Boolean inline {x == y} = intrinsic EQ,
Boolean inline {x != y} = intrinsic NE,
Boolean inline {x < y}  = intrinsic LO,         // unsigned
Boolean inline {x > y}  = intrinsic HI,         // unsigned
Boolean inline {x <= y} = intrinsic LOS,        // unsigned
Boolean inline {x >= y} = intrinsic HIS;        // unsigned

int arg {x, y},
Boolean inline {x < y}  = intrinsic LT,         // signed
Boolean inline {x > y}  = intrinsic GT,         // signed
Boolean inline {x <= y} = intrinsic LE,         // signed
Boolean inline {x >= y} = intrinsic GE;         // signed
```

The relational operators leave a `Boolean` `TRUE` or `FALSE` value on the stack. We can define a single intrinsic for `EQ` and `NE` because the result will be the same `Boolean` whether the arguments are `ulong` or `int`. On the other hand, we do need separate instrinsics for `LT`, `LO`, and others because unsigned relational operators are different from signed ones.

6.4.4 Shift Operators

Shift operators are a little different:

```
ulong arg {x, y},
ulong inline {x << y}   = intrinsic SHL,
ulong inline {x >> y}   = intrinsic SHR,        // unsigned

int arg x, ulong arg y,
int inline {x << y}     = intrinsic SHL,
int inline {x >> y}     = intrinsic ASR;        // signed
```

The type of a shift expression matches the shifted data `x` and does not depend on the shift amount `y`, so we declare `y` to be `ulong` for maximum flexibility. Both signed and unsigned left shift use PSI's `SHL`. Right

shifts use SHR for ulong and ASR for int.

6.4.5 Bitwise Logical Operators

Here are the bitwise logical operators:

```
ulong arg {x, y},
ulong inline {~x}    = intrinsic COM,    // One's complement.
ulong inline {x & y} = intrinsic AND,
ulong inline {x | y} = intrinsic OR,
ulong inline {x ^ y} = intrinsic XOR,
def {x # y}  = x & ~y,       // Bit Clear (BIC)
def {x &? y} = ?(x & y),     // Bit Test (BIT): x intersects y
def {x !& y} = !(x & y);     // NAND: x does not intersect y

int arg {x, y},
int inline {~x}    = intrinsic COM,      // One's complement.
int inline {x & y} = intrinsic AND,
int inline {x | y} = intrinsic OR,
int inline {x ^ y} = intrinsic XOR,
def {x # y} = x & ~y;    // Bit clear
```

As with other integer operators, we have separate operators for ulong and int to preserve whether an expression is signed or unsigned. We have defined the bit clear operator # as a macro since it does not have (or need) a PSI operator.

6.4.6 Boolean Operators

The Boolean and and or operators use lazy evaluation so that the second argument is only evaluated if the first argument does not determine the expression's value. GalaxC implements them as macros containing *if-then-else* expressions:

```
Boolean arg {a, b},
def a && b = if a then b else FALSE,
def a || b = if a then TRUE else b;

Boolean arg {a, b},
def a and b = a && b,
def a or b  = a || b;
```

6.4.7 Miscellaneous Operators

Here are a few miscellaneous operators to complete integer arithmetic:

```
ulong arg {x, y},
ulong inline min(x, y)  = intrinsic MINU,        // unsigned
ulong inline max(x, y)  = intrinsic MAXU,        // unsigned
def x is even = (x & 1) == 0,
def x is odd  = (x & 1) != 0;

int arg {x, y},
int inline min(x, y)    = intrinsic MIN,         // signed
```

```
int inline max(x, y)    = intrinsic MAX;        // signed
```

The `x is even` and `x is odd` macros are simple examples of descriptive macro names. You can use them to write expressions like: "`if x is odd then ...`"

Finally, here is GalaxC's `nop` expression, which is often used like a C null statement:

```
void inline nop = intrinsic NOP;
```

The `nop` expression simply generates a PSI `NOP` which is then discarded, producing no code. It has type `void`.

6.4.8 Integer Type Conversions

Conversions between integer types may be automatic or explicit. Automatic type conversions are included in type definitions and are described at the GalaxC level in Section 4.2.1. Let's review those conversions now that we know more about PSI operators:

```
typedef ulong  = roottype(4);
typedef long   = subtype(ulong,  4, intrinsic L2UL);
typedef short  = subtype(long,   2, intrinsic S2L);
typedef sbyte  = subtype(short,  1, intrinsic SB2S);
typedef ushort = subtype(long,   2, intrinsic US2L);
typedef ubyte  = subtype(ushort, 1, intrinsic UB2US);
```

Each `intrinsic` is one of the PSI type conversion operators defined in Section 6.3.1. They define which PSI operator to generate for each subtype-to-supertype conversion. For example, if `b` is an `sbyte`, the expression `b+1` requires `b` to be converted to `long` (same as `int`). Converting `b` to `long` consists of evaluating `b` on the stack, followed by `SB2S` and `S2L` operations.

Since shorts and bytes are already zero-extended on the PSI stack, the `UB2US` and `US2L` operators are treated as `NOP`s and do not generate any code.

Explicit type conversions are of the form `ushort(x)`. They are needed to go from a wider type to a narrower type -- e.g., if `x` is `ulong` -- but can be used even if not needed -- e.g., if `x` is `ubyte`.

Here is how GalaxC defines explicit type conversions in `gxclib`. As is always the case, we must define patterns for supertypes before their subtypes so that overloaded patterns are matched in the correct reverse order. First we define conversions for `ulong`, which also apply to `long`:

```
ulong  arg x,    // x is ulong, long, ushort, short, ubyte, or sbyte.
ulong  inline ulong(x)  = intrinsic NOP,     // No change except automatic.
long   inline long(x)   = intrinsic NOP,     // No change except automatic.
int    inline int(x)    = intrinsic NOP,     // No change except automatic.
ushort inline ushort(x) = intrinsic L2US,    // Zero-extend short.
short  inline short(x)  = intrinsic L2US,    // Zero-extend short.
ubyte  inline ubyte(x)  = intrinsic L2UB,    // Zero-extend byte.
sbyte  inline sbyte(x)  = intrinsic L2UB,    // Zero-extend byte.
```

If `x` is a subtype of `ulong`, first GalaxC converts it to `ulong` using automatic type conversions. `ulong(x)`,

`long(x)`, and `int(x)` do nothing because `x` is already a 32-bit number. However, `ushort(x)` and `short(x)` must clear the two MSBs of TOS since PSI shorts must be zero-extended on the stack. Similarly, `ubyte(x)` and `sbyte(x)` must clear the three MSBs of TOS.

These are actually enough to define explicit conversions completely, but they can generate lots of unnecessary operations. For example, `ubyte(b)` where `b` is an `sbyte` requires converting `b` to `ulong` (`SB2S` followed by `S2L`) and then converting the `ulong` down to `ubyte` with `L2UB`, a total of 3 operations. In fact, in this case we don't need any operation because PSI bytes on the stack are zero-extended whether they are signed or unsigned.

While these redundant operators could be removed by a code optimizer, it's quite simple to define some additional conversions to prevent them from occuring in the first place. They must appear after the `ulong` versions so they have priority.

```
ushort arg x,    // x is ushort or ubyte.
ushort inline ushort(x) = intrinsic NOP,
short  inline short(x)  = intrinsic NOP,

short  arg x,    // X is short or sbyte.
ushort inline ushort(x) = intrinsic NOP,
short  inline short(x)  = intrinsic NOP;
```

Converting a short to another short does not change the PSI stack representation, so all of these are `NOPs`. If `x` is an `sbyte` there is still the `SB2S` conversion. We didn't bother to define any short to byte conversions such as `ubyte(short)` since they are less common and the `ulong` versions do not add much overhead.

Similarly, converting a byte to another byte does not change the PSI stack representation, so all of these are `NOPs` as well:

```
ubyte  arg x,    // x is ubyte.
ubyte  inline ubyte(x)  = intrinsic NOP,
sbyte  inline sbyte(x)  = intrinsic NOP,

sbyte  arg x,    // x is sbyte.
ubyte  inline ubyte(x)  = intrinsic NOP,
sbyte  inline sbyte(x)  = intrinsic NOP;
```

`gxclib` also defines two operators `signed x` and `unsigned x` which treat their integer argument `x` as the signed or unsigned integer of the same size. Note that `signed x` is a concat expression which does not require parentheses. If `x` is a complex expression, you may put it in parentheses but you must put in a space, e.g., `signed (x + y)` rather than `signed(x + y)`. Here are the `inlines` that define `signed` and `unsigned`:

```
ulong  arg x,    // x is ulong, long, ushort, short, ubyte, or sbyte.
ulong  inline unsigned x = intrinsic NOP,    // Treat x as ulong.
long   inline signed x   = intrinsic NOP;    // Treat x as long.

ushort arg x,    // x is ushort or ubyte.
ushort inline unsigned x = intrinsic NOP,
short  inline signed x   = intrinsic NOP;
```

```
short  arg x,    // X is short or sbyte.
ushort inline unsigned x = intrinsic NOP,
short  inline signed x   = intrinsic NOP;

ubyte  arg x,    // x is ubyte.
ubyte  inline unsigned x = intrinsic NOP,
sbyte  inline signed x   = intrinsic NOP;

sbyte  arg x,    // x is sbyte.
ubyte  inline unsigned x = intrinsic NOP,
sbyte  inline signed x   = intrinsic NOP;
```

6.4.9 Loading Variables

Loading the value of a variable is actually handled as part of automatic type conversion deep in the GalaxC compiler. Basically, each type definition includes the type size in bytes. Whenever it is necessary to dereference a variable with address type &τ, GalaxC calls internal function `GenLoad(n, vol)` where `n` is the type's storage size in bytes. `GenLoad` then generates the appropriate PSI opcode: `LOAD`, `LOADS`, `LOADB`, `LOADD`, or `LOADN`. If `vol` is `TRUE`, generate a `volatile` load: `LOADV`, `LOADVS`, `LOADVB`, `LOADVD`, or `LOADVN`.

6.4.10 Assignment Expressions

An assignment "`x = y`" evaluates expression `y`, converts it to the value type of `x`, and stores TOS in variable `x`. The converted value remains on TOS as the value of the assignment expression.

Most assignments are performed by a *generic macro*, a concept which we will examine in detail in a later chapter. However, the assignment macro is simple enough that we can explain most of it here.

```
type arg T, &T arg x, T arg y,
T def {x = y} = Assign(x:T, y);
```

The pattern "`x = y`" is matched by any expression where `x` has reference type `&T` and `y` is of the same type `T` or a subtype. `T` can be any type, which is also the type of the expression. The body simply calls internal GalaxC special function `Assign` which first pushes `y`, converting it to `T`, then pushes the address of variable `x`, and finally generates `STORE`, `STORS`, `STORB`, `STORD`, or `STORN` depending on the storage size of `T`. The `STORE` instruction pops `x` and leaves the converted `y` on the stack as the assignment expression's value.

If `T` is `volatile`, `Assign` generates `STORV`, `STORVS`, `STORVB`, `STORVD`, or `STORVN`.

The generic assignment macro handles assigning a subtype to a supertype, e.g., assigning `x = 'a'` where `x` is a `long`. However, it does not handle assigning a supertype to a subtype, e.g., `b = 123456` where `b` is a `ubyte`. To handle these, we define some additional assignment macros that assign a `ulong` (or subtype) to each kind of integer:

```
&long arg x, ulong arg y,
def {x = y} = Assign(x: long, long(y)),
&ushort arg x,
def {x = y} = Assign(x: ushort, ushort(y)),
&short arg x,
def {x = y} = Assign(x: short, short(y)),
```

```
&ubyte arg x,
def {x = y} = Assign(x: ubyte, ubyte(y)),
&sbyte arg x,
def {x = y} = Assign(x: sbyte, sbyte(y));
```

As with the generic macro, each body calls `Assign` to generate the PSI code for the assignment. In each case, we use an explicit type conversion to convert `ulong y` to the value type of `x`. This may generate some redundant `L2US` or `L2UB`, but in this case we will let the code optimizer take care of them. The type of the `Assign` function is the type specified after the colon.

6.4.11 Integer Variable Increment and Decrement

Incrementing and decrementing integer variables using `i++`, `m--`, `--j`, `--n`, `p += 5`, `q -= 7`, etc., is so common in GalaxC programs that PSI has special operators for them -- `PRINC`, `POINC`, etc. -- so that it doesn't have to generate long sequences of stack operations. Here are the generic macros that generate those operators:

```
type arg T, &T arg x, ulong arg y,
def {x += y} = Increment(x, T, +y, preinc),
def {x -= y} = Increment(x, T, -y, predec),
def {++x}    = Increment(x, T, +1, preinc),
def {--x}    = Increment(x, T, -1, predec),
def {x++}    = Increment(x, T, +1, postinc),
def {x--}    = Increment(x, T, -1, postdec);
```

`Increment(x, T, y, dir)` is an internal GalaxC special function which generates PSI code to increment an integer `x` of type `T` (long, short, or byte) by `long` offset `y`. "`dir`" is an identifier which specifies whether PSI should increment or decrement, and whether the value of the expression is the variable's value before or after it is updated. `Increment` evaluates and pushes `y`, then pushes the address of `x`, and then generates one of the `PRINC`-style opcodes. If `T` is `volatile`, `Increment` uses one of the `PRINV` opcodes. For example, "`x += 3`" where `x` is a global `ushort` variable calls `Increment(x, ushort, 3, preinc)` which generates the code: "`3 GLOBVAR x PRINCS`". The generated code leaves the perhaps updated value of `x` on the stack, and the expression has type `T`.

`gxclib` handles decrement by negating argument `y` before calling `Increment`. The `predec` and `postdec` indentifiers are used for error messages.

`T` can also be a pointer type, described in Chapter 8. It behaves the same as an integer, except that `Increment` multiples all offsets by the storage size of the referenced type.

Chapter 7
Control Statements

This chapter describes the semantics of the default GalaxC control statements, which include the usual structured language control constructs -- *if-then-else*, *while-do*, *repeat-until* -- along with *switch*, various *for* statements, *goto*, and exceptions. GalaxC statements are based on Algol-W, Pascal, and C. Their syntax was described in detail earlier in Section 3.2.4.

As with everything else in GalaxC, control statements are actually language extensions. They may be superceded and/or augmented by alternate programmer-defined extensions. In practice, these control expressions have become a natural means of expressing algorithms and as such tend to be treated as an integral part of GalaxC.

7.1. *If* Expressions and Statements

If expressions and statements provide GalaxC with conditional execution capability. They have two forms:

```
if a then b
if a then b else c
```

Expression *a* is called the *condition*, expression *b* is the *consequent*, and expression *c* is the *alternative*. To evaluate an *if* expression, first GalaxC evaluates *a*, which must be a `Boolean` expression. If the result is `TRUE` then GalaxC evaluates *b*, which becomes the value of the entire *if* expression. If the result is `FALSE` then GalaxC evaluates *c*, which becomes the value of the *if* expression. Note that *a* is always evaluated, and either *b* or *c* is evaluated.

These correspond to C statements:

```
if (a) b
if (a) b else c
```

but there are some important differences:

1. Like Algol and Pascal, GalaxC requires the reserved word `then`. This means that GalaxC does not need parentheses around *a* like C.

2. C uses '`;`' to convert an expression into a statement so it is perfectly fine to have it appear before `else`, e.g., "`if (x == 2) y = 12; else z = 13;`". GalaxC treats '`;`' as an *operator* that appears between two statements, terminating the first statement. In GalaxC (as in Pascal), it is always incorrect to put '`;`' before `else` since `else` never begins a statement.

3. C *if* statements do not have values. GalaxC *if* expressions do have values, so you can write: "`x = (if a then b else c)`". Expressions *b* and *c* must have compatible types, perhaps `void`, which is the type of the *if* expression. This is described in detail below.

   C has a *conditional expression* of the form "*a*`? b: c`" which does produce a value. GalaxC also has *cond* expressions: they have exactly the same meaning as "`if` *a* `then` *b* `else` *c*" except that the type of the expression must not be `void`.

4. In C, *a* is an integer or pointer and C checks whether it is 0 (false) or non-zero (true). This permits

some compact notations, such as "`if (*s++ = *t++) ...`", but it's easy to mistake assignment '=' for equality "==", one of the most common C errors for both beginners and experienced programmers. GalaxC requires *a* to be `Boolean`, so you must write "`if (*s++ = *t++) != '\0' then ...`" While it's wordier, it's safer and can generate exactly the same object code.

The *if-then* statement is the same, except that if *a* is `FALSE` then the alternative behavior is "no operation". Here is how *if-then* is defined in `gxclib` as a macro:

```
spclarg {a, b},
def {if a then b} = if a then (b; nop) else nop,
```

Arguments `a` and `b` are declared as *special arguments*, which means that they can take on any type. This will be described in detail in Chapter 10. The expression "`(; nop)`" is needed to for conditional compilation.

Expressions *b* and *c* must have compatible types, i.e., they must be subtypes of the same supertype. When it compiles an *if-then-else*, GalaxC determines the lowest common supertype τ of *b* and *c* and generates code to convert *b* and *c* to that τ, which is the type of the *if* expression. Here are some examples:

- `if` *a* `then 3 else '@'`: since `char` is a subtype of `int`, the common supertype is `int`.

- `if` *a* `then x else 5`, where `x` is an `int` variable: `x` is really of type `&int` so GalaxC converts it to common supertype `int` by dereferencing it.

- `if` *a* `then x else y`, where `x` and `y` are both `int` variables: `x` and `y` are really of type `&int` so the common supertype is `&int`. GalaxC does not dereference *b* and *c* so is is possible to write:

  ```
  (if p == q then x else y) = z
  ```

  We can also write it as a *cond* expression: `(p == q? x: y) = z`. Since a *cond* is simpler than *expr*, it is possible to write this without parentheses if desired.

- `if` *a* `then x else ch`, where `x` is `&int` and `ch` is `&char`: in this case the lowest common supertype is `int`, so GalaxC needs to dereference both `x` and `ch`. It is *not* possible to write the assignment as in the last example because the common supertype is not a reference type.

In GalaxC, all types are subtypes of `void`, so *b* and *c* always have a common supertype even if it's just `void`. To convert an expression to `void` we simply pop it from the stack, so if the only common supertype is `void`, GalaxC pops both *b* and *c* and the *if* expression has type `void`, making it a statement like C.

Except for the non-`void` restriction, choosing *if* or *cond* expressions is purely a matter of taste, for example, here are two equivalent versions of the factorial function:

```
int arg n,
int fn n! = if n == 0 then 1 else n*(n-1)!

int arg n,
int fn n! = n == 0? 1: n*(n-1)!
```

The top one looks more like a math text, the bottom one more like C. Generally, as conditional expressions get more complicated the *if-then-else* version starts to get cumbersome and *a?b:c* is preferred, as in these

equivalent versions of the *signum* (sign) function:

```
int arg x,
fn sgn(x) = if x < 0 then -1 else if x == 0 then 0 else 1

int arg x,
fn sgn(x) = x < 0? -1: x == 0? 0: 1
```

However, consider this alternate version:

```
int arg x,
fn sgn(x) = x < 0? -1: x != 0
```

In this case we use "`x != 0`" to simplify "`x == 0? 0: 1`". It has a `Boolean` value, which gets converted to the common `int` supertype. We can even go one step further and eliminate the *cond* entirely:

```
int arg x,
fn sgn(x) = (x > 0) - (x < 0)
```

Here we use the the fact that `Boolean` is a subtype of `int` so we can use `int` subtraction.

An *if-then* statement is always has a `void` value. Here is an example of its use as part of inorder tree traversal:

```
Tree arg p,
void fn Inorder(p) =
{
    if p == NULL then return;
    Inorder(p.Left);        // Traverse left subtree.
    Visit(p);               // Do something to the root of the tree.
    Inorder(p.Right);       // Traverse right subtree.
}
```

7.1.1 Conditional Compilation

If the condition *a* in an expression "`if` *a* `then` *b* `else` *c*" is a constant at compile time, GalaxC just generates the code for *b* (if *a* is `TRUE`) or *c* (if *a* is `FALSE`) . This gives the same effect as *conditional compilation* in C (`#if`) , except that C performs conditional compilation in the pre-processor and requires that conditionally compiled code begin and end at source code line boundaries. GalaxC conditional compilation can occur anywhere in a GalaxC program, both within function/macro bodies and at the global level.

The conditionally compiled code in *b* and *c* must be syntactically correct, but only the expression that is selected needs to be semantically correct. The other one is not analyzed at all, so it can have undefined variables and other errors. This is important because some variables may only be defined if compile-time constant α is `TRUE`, so we should not try to generate code if α is `FALSE`.

[At the present time we have not used conditional compilation much, so there may be unexpected bugs.]

7.2. *While-Do* Statements

The *while-do* statement is the first iterative control statement in GalaxC. The statement reëvaluates an expression as long as a condition remains TRUE. It has the form:

```
while a do b
```

Expression *a* is called the *condition* and is a Boolean expression. Expression *b* is called the *body* and can be any type. To evaluate a *while-do* statement, first GalaxC evaluates *a*. If its value is FALSE, then evaluation of the *while-do* is complete and its value is void. If *a* is TRUE, then GalaxC evaluates body *b*, popping and discarding any result. This sequence repeats until the condition becomes FALSE.

If *a* is FALSE upon initial evaluation, then *b* is not evaluated at all. If *a* is always TRUE, then the *while-do* statement may be an infinite loop. Expression *b* should, in most cases, include an expression that changes the value of *a*.

The *while-do* expression has many applications in programming. For example, here is a function which searches an integer array A for a value x, returning the index if found, or -1 if not found. GalaxC uses the notation @A to declare A as a pointer to an array of int. This is equivalent to C's "int *A".

```
int arg {@A, x, n},          // A = pointer to array of int.
fn find x in A(n) =          // n = Number of defined elements in A.
{                            // x = find this value.
    var {i = 0, found = False};
    while i < n and !found do if A[i] == x then found = TRUE else i++;
    return (found? i: -1);
}
```

We have used Boolean variable found to terminate iteration of the loop prior to scanning all elements of the array. This function can be simplified by using the return statement (§*5.2.3*) to terminate the loop, eliminating variable found:

```
int arg {@A, x, n},          // A = pointer to array of int.
fn find x in A(n) =          // n = Number of defined elements in A.
{                            // x = find this value.
    var i = 0;
    while i < n do if A[i] == x then return i else i++;
    return (-1);
}
```

7.3. *Repeat-Until* Statements

The second GalaxC iterative statement is *repeat-until*, which reëvaluates an expression until a condition becomes TRUE. It has the form:

```
repeat b until a
```

Like *while-do*, *a* is the Boolean condition and *b* is the body, which can be a *semi* expression since it is terminated by reserved word until. The behavior of *repeat-until* is slightly different from *while-do*: first, body *b* is evaluated and discarded, then condition *a* is evaluated. If *a* is TRUE, then evaluation of *repeat-until* is complete and it value is void. If *a* is FALSE, then the repeat statement is executed again.

The body is always evaluated at least once. If *a* is always FALSE, then the repeat expression may be an infinite loop. The body should, in most cases, include an expression that changes the value of the condition.

The above search function can be rewritten using *repeat-until*. Note that this requires that n ≥ 1:

```
int arg {@A, x, n},          // A = pointer to array of int.
fn find x in A(n) =          // n = Number of defined elements in A, ≥ 1.
{                            // x = find this value.
    var i = 0;
    repeat if A[i] == x then return i else i++ until i >= n;
    return (-1);
}
```

We can simplify the *repeat-until* slightly by moving i++ into the condition:

```
repeat if A[i] == x then return i until ++i >= n;
```

Also, since the body is a *semi* we can put a semicolon after it:

```
repeat if A[i] == x then return i; until ++i >= n;
```

7.4. Break and Continue

In many cases it is advantageous to terminate a loop prior to normal completion. We did this in the previous examples using return statements. Unfortunately, return is somewhat drastic as it not only terminates the loop; it also terminates the function. For those situations where it is desirable to exit a loop without returning from the function, GalaxC provides the C break statement, which causes the innermost loop containing it to terminate. All GalaxC interative statements can include break. As an example, here is yet another version of search:

```
int arg {@A, x, n},          // A = pointer to array of int.
fn find x in A(n) =          // n = Number of defined elements in A, ≥ 1.
{                            // x = find this value.
    var i = 0;
    repeat if A[i] == x then break until ++i >= n;
    return (i < n? i: -1);
}
```

Another application of break is a loop where we want the termination condition in the middle of the loop instead of at its beginning or end. Such loops have the form:

```
repeat                       while TRUE do
    b;                       {   b;
    if a then break;             if a then break;
    c;                           c;
until FALSE                  }
```

These two loops are equivalent. Expression *b* is always executed at least once. The condition *a* determines when the loop terminates. Note that the expressions "repeat *b* until FALSE" and "while TRUE do *b*" would both be infinite loops except for the break expressions in the middle of the loops.

Another useful loop control capability is termination of an individual loop iteration without termination of the entire loop. In GalaxC, this is done using the C `continue` statement. Although `continue` is used less frequently than `break`, it is very useful in those situations when it applies. A loop with a `continue` statement has one of the following forms:

```
repeat                          while a do
    b;                          {   b;
    if d then continue;             if d then continue;
    c;                              c;
until a                         }
```

The `continue` statement -- if executed -- skips the rest of the current evaluation of the loop body, avoiding the evaluation of *c*, and proceeds directly to testing the loop termination condition. The condition is tested to see whether additional iterations of the loop are required.

A loop can always be written without using `break`, `continue`, or `return`. However, the program may be more efficient and readable using these early termination expressions.

Syntax note: `break`, `continue`, and `return` are *not* reserved words.

7.5. *For* Statements

GalaxC's third kind of iteration is the *for* statement, which has two forms:

```
for (i; a; u) b
for a do b
```

The first form is like C, except that *a* must be `Boolean`. To evaluate a C-style *for* statement, GalaxC first evaluates *init* expression *i* (discarding any result) which initializes the loop. Then it evaluates condition *a*. If *a* is `FALSE`, evaluation of the *for* is complete and its value is `void`. If *a* is `TRUE`, then GalaxC evaluates body *b*, discarding any result. Then it evalute *update* expression *u*, also discarding the result. Then it goes back to evaluating *a* and the sequence repeats until *a* becomes `FALSE`.

A C-style *for* is essentially the same as evaluating *i*, followed by a *while-do*. In fact, it's tempting to write it as the following macro:

```
spclarg {a, b, i, u},
def {for (i; a; u) b} = {i; while a do {b; u}}
```

This would be correct except for one thing: a `continue` statement in a *for* loop must perform the update *u*. We'll see the actual definition of this *for* later in the chapter.

The *for* statement gives us a nice way to write our search function:

```
int arg {@A, x, n},          // A = pointer to array of int.
fn find x in A(n) =          // n = Number of defined elements in A.
{                            // x = find this value.
    int var i;
    for (i = 0; i < n; i++) if A[i] == x then break;
    return (i < n? i: -1);
```

```
}
```

The init and update expressions can be arbitrarily complex, for example a series of initializations or updates separated by commas:

```
int var {i, j};
for (i = 1, j = 2; i <= 10; i += 1, j += 2) printf("%d, %d\n", i, j)
```

Each of these comma expression leaves multiple values on the stack, but they all get popped and discarded so it doesn't matter. Since these comma expressions are not in parentheses, they are evaluated left to right. Init and update can also be *semi* expressions, but they must be in braces.

The init, update, and body must all be be some kind of expression even if it's just `nop`. This is different from C, which allows null statements. Here is an example:

```
for (var i = 1; i <= 10; nop) printf("%d\n", i++)
```

This example also declares a local variable `i` which is only defined inside the *for* statement.

GalaxC's second *for* statement has the syntax: "`for` *a* `do` *b*". This serves as a template for a collection of Pascal-style *for* loops, e.g,:

```
for i = 1 thru 10 do ...
for i = 10 downto 0 do ...
for i = 2 thru 10 by 2 do ...
```

In all of these cases *a* is an *expr* that contains assignment and concatenation. All of the Pascal-style *for* loops are defined as macros:

```
spclarg {i, a, b, c, body},
def {for i = a upto b do body}         = for (i = a; i <  b; i++) body,
def {for i = a thru b do body}         = for (i = a; i <= b; i++) body,
def {for i = a through b do body}      = for (i = a; i <= b; i++) body,

def {for i = a upto b by c do body}    = for (i = a; i <  b; i += c) body,
def {for i = a thru b by c do body}    = for (i = a; i <= b; i += c) body,
def {for i = a through b by c do body} = for (i = a; i <= b; i += c) body,

def {for i = a downto b do body}       = for (i = a; i >  b; i--) body,
def {for i = a downthru b do body}     = for (i = a; i >= b; i--) body,
def {for i = a downthrough b do body}  = for (i = a; i >= b; i--) body,

def {for i = a downto b by c do body}  = for (i = a; i > b; i -= c) body,
def {for i = a downthru b by c do body} =
    for (i = a; i >= b; i -= c) body,
def {for i = a downthrough b by c do body} =
    for (i = a; i >= b; i -= c) body;
```

These macros simply provide an alternative syntax for the same function. For example, now we can re-write the *for* statement in our search function as:

```
    for i = 0 upto n do if A[i] == x then break;
```

One advantage of this *for* statement is that the word `upto` explicitly indicates the intent that values of `i` are to go from `0` up to but not including `n`. It's very easy in C to write '`<=`' in place of '`<`' accidently and get a "fencepost" error.

In the above macros, `upto`, `thru`, and `through` (for people who like to type) indicate increasing values of index variable `i`. Decreasing values are indicated by `downto`, `downthru`, and `downthrough`. The macros increment or decrement `i` by `1` unless a "`by` *c*" clause indicates a different amount, which should always be positive.

Note that the expression *a* is always of the form "`i =` *concat*". Since `upto`, `downto`, and the other identifiers are not reserved words like `for` and `do`, you must be careful that `a` is *unary* or simpler and that `b` and `c` are *prefix* or simpler. Put them in parentheses if necessary. Also, since `i` is used multiple times in the definition of a Pascal-style *for*, it should be a simple variable and not an expression (and certainly never an expression with side-effects).

We have not said anything about the type of `i`. This is because since it is declared as a `spclarg`, `i` can be any type. However, when GalaxC expands the macro, it must find a match for each expression in the corresponding C-style *for*. For example, "`for i = a upto b do body`" requires that "`i++`" be defined for whatever type `i` is. This usually limits `i` to an integer or pointer type.

On the other hand, "`for i = a upto b by c do body`" only requires that the comparison and "`i += c`" be defined. This allows `i` to be far more types, such as floating point: "`for x = 1.0 upto 10.0 by 0.25 do body`".

The great thing about "`for` *a* `do` *b*" is that it can be customized to all sorts of iterations and types. The Pascal-style *for* statements are just a few examples of what is possible.

7.6. `Goto` Statements

Like C, GalaxC has labels and `goto` statements. Structured programming purists claim that `gotos` are always bad and always result in hard-do-debug spaghetti code. While they can be abused -- like any programming construct -- there are occasions where a well-named `goto` simplifies structure, e.g., handling error conditions. The purpose of structured programming is to make software easier to understand and maintain, so complexity added to solely to avoid a `goto` may be worse than using a `goto`.

Besides, all major computer architectures have conditional and unconditional jump instructions. When these are replaced by *while-do* and other structured control constructs, then it may be time to rethink having `gotos` in high-level languages.

A `goto` statement must have a label, which may be defined before or after the `goto`. To define a label, use the statement:

`label` *name*

The label's *name* can have any syntax, but it is usually an identifier since `label` and `goto` are not reserved words. The label is only defined within a block, and cannot be jumped to from outside the block. It's fine to jump out of an inner block to a label defined in an outer block, but not to cross function boundaries.

To jump to a label, use the statement:

`goto` *name*

If *name* is already defined, GalaxC can immediately generate code to jump to it. If *name* is not yet defined, GalaxC assumes that it will become defined at some point and prepares a list of forward references. When name becomes defined using "`label` *name*", GalaxC updates the forward `goto` chain with the label's address. If GalaxC reaches the end of a function before *name* is defined, GalaxC reports an error.

For example, here is yet another way to write our search function:

```
int arg {@A, x, n},          // A = pointer to array of int.
fn find x in A(n) =          // n = Number of defined elements in A.
{                            // x = find this value.
    int var i;
    for (i = 0; i < n; i++) if A[i] == x then goto found;
    return (-1);             // Did not find x.
label found:
    return i;                // Did find x.
}
```

In C, a label is an identifier followed by '`:`'. In contrast, GalaxC treats "`label` *name*" as just another kind of statement and treats '`:`' as an operator that behaves in this context the same as '`;`'. In fact, you can use either '`:`' or '`;`' to separate "`label` *name*" from the following expression.

Internally, a label has two attributes: its address (where to jump to) and its stack level for local variables. GalaxC implements `goto` using the PSI `POPGOTO` instruction which first pops the stack to the proper level and then jumps to the label's address.

7.6.1 Implementation of Iterative Statements

In most languages all the basic control statements are built into the compiler. In contrast, GalaxC uses `label` and `goto` to implement *while-do*, *repeat-until*, and *for* as macros. Here are their definitions from `gxclib`:

```
spclarg {a, b, i, u},
def {while a do b} =
{label looplabel:
 if a then {b; label contlabel: goto looplabel};
 label breaklabel
},

def {repeat b until a} =
{label looplabel:
 {b; label contlabel};
 if !a then goto looplabel;
 label breaklabel
},

def {for (i; a; u) b} =
{i; label looplabel:
 if a then {b; label contlabel: u; goto looplabel};
 label breaklabel
};
```

Each macro defines three *local labels*:

- `looplabel` is the beginning of the loop after any initialization `i`.
- `contlabel` is where `continue` should go, i.e., the end of loop body `b`. *While-do* and *repeat-until* immediately go to the condition test; *for* evaluates update expresssion `u`.
- `breaklabel` is where `break` should go, i.e., the end of the entire loop expression.

Since labels are defined only within blocks, `contlabel` is only defined in the block containing `b` and the other two are only defined within the iteration statement.

Given these local labels, we can now define `break` and `continue` as `gotos`:

```
def break = goto breaklabel;
def continue = goto contlabel;
```

In short, all the behavior of iterative statements is defined using macros instead of being buried in the compiler.

7.6.2 Exceptions

By default, labels and `gotos` are local to the function in in which they are defined. However, there are times when it is useful to jump across function boundaries. A good example of this is when a fatal error occurs in a deeply-nested series of function calls. Without special exception handling, each intermediate function must return a status code that is checked after each function call. It may be preferable to jump directly to exception handling code.

The C library provides `longjmp` for to handle exceptions. GalaxC provides a simular mechanism using the built-in `exception` data type. A variable of type `exception` contains all the information needed to set the proper execution state so as to resume execution correctly at the exeception handling code. For PSI, this consists of the program counter (PC), the stack pointer (SP), and the frame pointer (FP).

GalaxC exceptions are a generalization of labels and `gotos`. Here is how to define and use an exception:

1. Create a global variable of type `exception`:

   ```
   exception var Error;
   ```

2. In an outer function that "catches" the exception, make the global `exception` variable point to a local label in that function by performing the assignment:

   ```
   Error = exception error_label
   ```

   where `error_label` is a local label at the beginning of the exception handling code. The expression "`exception error_label`" generates a PSI `EXCEPT` instruction which calculates and pushes PC, SP, and FP onto the stack. The assignment copies the three addresses to variable `Error`. Like a `goto`, the label may be defined before or after "`exception error_label`".

3. When a function discovers a problem that requires raising exception `Error`, it executes a `raise`

statement:

```
raise Error
```

This pushes the three addresses in variable `Error` onto the stack and generates a PSI `RAISE` instruction which assigns these three values to PC, SP, and FP. This causes the computer to jump immediately to the "catching" function's exception handling code.

Here is a more complete example:

```
exception var Error;                        // Global exception variable.

int arg {a, b},
fn g(a, b) =                                // Inner function.
{   ...                                     // Execute this code unconditionally.
    if a == 0 then raise Error;             // Raise exception Error.
    ...                                     // Execute this code if a ≠ 0.
};

int arg {x,y},
fn f(x, y) =                                // Outer function.
{
    Error = exception error_label;          // Set global exception to local label.
    g(x, y);            // Call function g(x,y).
                        // If error occurs in g(x,y), then go to error_label.
    ...                 // Execute this code if no error in g(x,y).
label error_label:
    ...                 // Execute this code if (but not only if) error in g(x,y).
};
```

In `f`, we set global `Error` to the value of local label `error_label` prior to calling `g`. If an error occurs in `g`, then "`raise Error`" jumps directly to `error_label`, popping the execution stack automatically.

GalaxC `exception` values can be treated like any other data type: they can be assigned from variable to variable, passed as function arguments, returned as function values, etc. It's very important to treat the contents of an `exception` variable carefully, because raising a corrupted `exception` can jump to almost anywhere, with disasterous consequences. Still, with care they are a powerful way to handle exceptional conditions and are an important part of XXICC software.

7.7. `Switch` Statements

GalaxC has a `switch` statement similar to C for selecting between multiple alternatives using an integer selector. For example, the sequence of *if* expressions:

```
// Assume x is an int variable.
if x == 1 then printf("one") else
if x == 2 then printf("two") else
if x == 3 then printf("three") else
if x == 4 then printf("four") else printf("out of range")
```

can be replaced with the single `switch` statement:

```
switch x
{ case 1:  printf("one");
  case 2:  printf("two");
  case 3:  printf("three");
  case 4:  printf("four");
  default: printf("out of range");
}
```

The `switch` statement is generally more efficient than a sequence of *if* statements since it uses a jump table in place of a sequence of compares.

Here is the C equivalent of the above `switch`:

```
switch (x)
{ case 1:  printf("one");   break;
  case 2:  printf("two");   break;
  case 3:  printf("three"); break;
  case 4:  printf("four");  break;
  default: printf("out of range");
}
```

C requires that the selector be in parentheses. In GalaxC, the parentheses are optional if `x` is `unary` or simpler. C also requires that each statement ends with `break`, otherwise C falls through to the next case. In practice, falling through to the next statement is quite rare so most `break` statements just add visual clutter and forgetting one can result in a bug that can be quite hard to track down. In C, it's good programming style to put in a comment like `/* fall through */` to indicate when a `break` is absent so a "helpful" programmer maintaining the code doesn't add one by mistake.

In contrast, GalaxC assumes a C `break` at the end of each alternative and provides a `fall thru` statement to fall through to the next alternative.

A GalaxC `switch` statement has the form:

`switch` *a* `{` *body* `}`

The *selector a* is an `int` (or subtype) expression. *Body* is a list of *alternative* statements separated by semicolons. Each alternative is one of:

`case` *c*`:` *x*
`case {`*c*`:` *d*`}:` *x*
`case {`c_1`,` c_2`, ...}:` *x*
`default:` *x* // `default` must be the last alternative.

where *c* is an `int` constant, *c*`:`*d* is a range of `int` constants, and c_1, c_2, etc, are `int` constants or ranges. They can all be subtypes of `int` as well. Each `case` constant can appear only once.

If selector *a* matches any of `case`'s constants, we execute *action x*, which is one or more statements separated by semicolons. Any variables or labels declared in an action are local to that action, i.e., each action behaves like a block whether or not it is in braces.

Each alternative has only one `case` expression: multiple constants must be contained in a single set of braces. In C, each `case` is a single constant so you have to write:

```
case 1: case 3: case 5: printf("x is odd");  break;
case 2: case 4: case 6: printf("x is even"); break;
```

There is an implied "fall through" between adjacent `case` expressions. Here is the same code in GalaxC:

```
case {1, 3, 5}: printf("x is odd");
case {2, 4, 6}: printf("x is even");
```

Like `label`, `case` can be followed either by '`:`' or '`;`'. This has syntax ramifications, since "`case` *c*`:` *x*" is a *colon* while "`case` *c*`;` *x*" is a *semi*. This does not make any difference in practice.

In GalaxC, it is possible to write:

```
case 1: case 3: case 5: printf("x is odd");
```

but it means:

```
case 1: nop; case 3: nop; case 5: printf("x is odd");
```

so cases `1` and `3` don't do anything. This is the opposite of C.

As mentioned earlier, each alternative automatically jumps to the end of the `switch` unless you put in a `fall thru` statement or its equivalent `fall through`. A `fall thru` can be anywhere in an action and simply jumps to the beginning of the next alternative's action.

Occasionally, you may want to leave a `switch` statement from the middle of an action. GalaxC does this using a `leave` statement, which is like a C `break` statement. GalaxC defines the new statement `leave` so that if a `switch` is in an iterative loop you can choose whether you wish to terminate the `switch` (with `leave`) or terminate the loop (with `break`).

Syntax note: `case`, `default`, and `leave` are not reserved words, though `switch` is.

GalaxC compiles a `switch` in two passes. First it examines all the `case` expressions and finds the minimum and maximum `int` constants. The difference is limited to 4000. On the second pass, GalaxC allocates a jump table with up to 4001 entries, the last one for `default`. GalaxC generates code to compile selector *a* and index the jump table using the PSI `SWITCH` instruction. It generates a PSI `MAXU` operation to make sure all out-of-range values of *a* go to the `default` statement. Next GalaxC compiles *body* as a series of *semi* expressions.

Each time GalaxC encounters a `case` expression, it completes the previous action, generating a forward `goto` to the end of the `switch`. Then it sets jump table entries for all the `case`'s constants to the current program counter (PC), and checks for conflicting `case` constants. If the last action had a `fall thru`, GalaxC makes its forward `goto` jump to the current PC as well. GalaxC also discards any local variables created by the previous action.

The statements in an action are compiled as ordinary statements. Variable declarations create local variables; `fall thru` and `leave` generate forward `goto`s.

The `default` alternative is handled just like a `case`, except that it sets all empty entries in the jump table to the `default`'s action.

When GalaxC is done compiling *body*, it generates a null `default` if necessary. Then it resolves all the forward `gotos` to the end of the `switch`, i.e., ends of alternatives and `leave` statements.

Chapter 8
Programmer-Defined Types

In Chapter 4, we discussed the built-in GalaxC types. In this chapter, we will see how to add new, programmer-defined types to GalaxC, including scalar, structure, and pointer types. An unusual feature of the GalaxC implementation is that these type creation constructs were used to create the GalaxC "built-in" types such as `long`, `short`, and `address`.

8.1. Scalar Types

Scalar types have no internal structure, i.e., they represent a single number rather than a record or array. They have two properties, a size and a supertype. *Root types* have type `void` as their common supertype (§4.7). To create a root scalar type, use the construct:

```
typedef τ = roottype(n)
```

where τ is the name of the type being created and *n* is the storage size of the type in bytes (§4.6). τ can have any syntax, but is usually an identifier for convenience. In the current implementation, *n* is limited to 16,384. Note that there is no space between `roottype` and '`(`'.

To obtain the storage size of type τ, use the notation `|τ|`, which returns *n*. This is the same as "`sizeof(τ)`" in C. *[footnote:* In C one can also use `sizeof` to get the size of any expression. This does not work in GalaxC, where `|x|` matches whatever function or macro is defined for the type of `x`.*]* Since types are defined at compile time, `|τ|` is a compile-time constant.

8.1.1 Subtypes

Most programmer-defined types are defined to be subtypes of existing types. The simplest way to create a type τ that is a subtype of `S` is to write:

```
typedef τ = subtype(S)
```

Type τ is declared to be the same size as `S` and has the same binary representation. Any expression of type τ can be used in a context that requires type `S`. Currently `S` must be a base type. There is no space between `subtype` and '`(`'.

Let's look at an example of using subtypes to define scientific arithmetic with units. First, we create two types representing meters and seconds:

```
typedef meter = subtype(float);
typedef sec   = subtype(float);
```

Both `meter` and `sec` are defined to be subtypes of `float`. They can use all `float` operations, e.g., addition, subtraction, multiplication, and division. For example, if `x` and `y` are of type `meter`, you can write the expression "`x+y`" which yields a `float` result. You can also write "`r = x`", where `r` is a `float` variable, since it is possible to assign a `meter` value to a `float` variable: a subtype value can always be assigned to a supertype variable.

However, the converse is not true: you cannot assign of a `float` value to a `meter` or `sec` variable. For example, the expression:

```
meter var x;
x = 1.0;
```

is not allowed by the compiler. This makes sense, because `1.0` does not have any dimensions. One can force the value `1.0` to have a `meter` value using a typecast:

```
x = 1.0: meter;
```

While this will work, it is cumbersome and a little dangerous since the typecast accepts any type that is the same size as `meter`.

A better way is to write macros that convert `float` values into `meter` and `sec` values:

```
float arg x,
def x meter = x: meter,
def x sec   = x: sec
```

You can then write the assignment:

```
x = 1.0 meter;
```

Next we look at arithmetic. Unless defined otherwise, the expression "`x+y`", where `x` and `y` are `meter`, yields a `float` result. As shown above, we cannot assign the expression directly to a `meter` variable. Using the above `meter` macro, we could write:

```
meter var {x, y, z};
z = (x + y) meter;
```

However, this is ugly since we must write `meter` in each such expression. Worse, there is no protection from mistakenly adding a distance `x` to a time `t` -- this code compiles fine:

```
meter var {x, y}; sec var t;
y = (x + t) meter;
```

The expression "`x+t`" yields a `float` expression -- it has lost its dimensions.

A safe way to implement dimensioned arithmetic is to write dimensioned macros. Here are the macros for addition and subtraction:

```
meter arg {x, y},
def x + y = x + y: meter,
def x - y = x - y: meter;

sec arg {x, y},
def x + y = x + y: sec,
def x - y = x - y: sec;
```

Macros cannot be recursive, so a macro's pattern is invisible when compiling its body. The expression "`x + y`" in the above macro bodies match "`float + float`".

These macros define addition and subtraction of two `meter` or two `sec` values to produce a `meter` or `sec`

result. Adding a `meter` to a `sec` still produces a `float` result which cannot be directly assigned to a `meter` or `sec` variable. Multiplication of `meter` or `sec` values should result in other definable types such as `sec`^{2}, `meter`^{2}, and `meter sec`. Division can also be defined, using macros similar to the above.

8.1.2 Automatic Type Conversion

We now look at the general form of subtype definition, which explicitly defines how to convert subtype τ to supertype `S`:

```
typedef τ = subtype(S, n, conversion)
```

This defines τ to be a subtype of `S` with `|τ|` = *n* bytes. Currently `S` must be a base type. There is no space between `subtype` and '`(`'. *Conversion* is an expression which converts a value of type τ into type `S`. For example,

```
typedef ushort = subtype(long, 2, intrinsic US2L)
```

defines `ushort` to be a 2-byte subtype of `long` using intrinsic operator `US2L` to convert from `ushort` to `long`. *Conversion* has one of three possible forms:

`pop`

Pop the *n*-byte value off the stack. This converts type τ to `void` by discarding the value. It is the usual conversion for root types.

`intrinsic` *opcode*

Apply intrinsic operator *opcode* to convert from τ to `S`. It assumes a value of type τ is already on the stack. This is used to convert integer types to their supertypes as in the above definition of `ushort`.

expression

Evaluate *expression* as if it were the body of an inline function, and assume there is a hidden self-reference argument `self` of type τ. If *expression* does not include self, assume there is already a τ value on the stack and treat the value as the first member of a comma expression. Since *expression* is arbitrary, this is a very powerful feature.

You can change `self` to a different name if desired using the statement: "`selfref` *name*".

[This feature has not been implemented yet. It may change significantly when it comes time to do so.]

The `roottype` definition described earlier is actually a macro:

```
spclarg name, long arg n,
def {typedef name = roottype(n)} = (typedef name = subtype(void, n, pop))
```

This defines root type `name` to be subtype of `void` using `pop` conversion. The `name` argument is declared as a `spclarg`, which is an unevaluated parse tree that is passed to a special function at compile time. Special functions and `spclargs` are described in Chapter 10.

Similarly, we can in principle define the simple `subtype` definition as the macro:

```
spclarg name, type arg super,
def {typedef name = subtype(super)} =
    (typedef name = subtype(super, |super|, intrinsic NOP))
```

This defines `name` to be subtype of `super` with the same bit representation. The conversion operation is `NOP` which leaves the top-of-stack value unchanged. *[At the present time the simple* `subtype` *type definition is implemented as a special function and the above macro does not work.]*

8.2. Structures

Structures are multi-byte values composed of a number of named **fields**. For example, the structure `complex` has two fields, `Re` and `Im`, which contain `float` values. GalaxC structures are similar to C structures and Pascal records, but allow any syntax to access the fields.

The type for a structure is defined using a `struct` definition, which has the form:

`typedef` τ `= struct(`*x*`) {`*ftype*$_1$ *name*$_1$`,` *ftype*$_2$ `{`*name*$_2$`,` *name*$_3$`}, ...}`

This defines new root type τ to be `struct` with multiple fields. Each field is a variable with a *field type* and a *field name*, both of which can have any legal syntax. You can declare multiple field names with a single field type using braces. Each field name is an expression which includes argument *x* denoting an object of type τ. For example,

```
typedef complex = struct(z) {float {Re z, Im z}}
```

defines our `complex` type with `float` real and imaginary parts named "`Re z`" and "`Im z`", where `z` is a `complex` value. Here is the equivalent C `typedef`:

```
typedef struct {float Re; float Im} complex;
```

C requires fields to be accessed using the notation `z.Re` and `z.Im`, while GalaxC allows any syntax. Also, C separates fields using semicolon instead of comma.

When GalaxC defines a `struct`, it defines *field-access macros* that access the fields of the `struct` using the field name expressions. By default, GalaxC defines two forms of field access for a given type τ. The first extracts a field from a *value* of type τ. This requires that all the other bytes of the value be discarded, leaving the desired field on the stack. For example, the expression "`Re (3.0 + i 4.0)`" pushes the `complex` value "`3.0 + i 4.0`" onto the stack and then discards the imaginary part, leaving `3.0` on the stack. It uses the PSI `LDPOP` instructions defined in Section 6.3.2.

The second form accesses a field given a *variable* (or function argument) of type τ. In this case, reference type `&`τ is converted to reference type `&`*ftype*$_i$ by adding the offset to field *i*. The resulting reference treats the field as a variable, the value of which can be read or written. For example, given a variable `z` of type `complex`, the expression "`Re z`" addresses the real part of `z` and is of type `&float`. You can read its value (`x = Re z`) or write it (`Re z = 6.0`). You cannot write to a field of a `complex` *value*: "`Re (3.0 + i 4.0) = 5.0`" is illegal. The second form also works for pointers (§*8.3.4*).

The fields of a structure appear sequentially in memory, with additional bytes inserted when necessary to achieve the proper byte alignment. In the `complex` case, the real field is at offset `0` and the imaginary field is at offset `4`. GalaxC automatically defines field-access macros for a `complex` variable to be the equivalent

of:

```
complex arg &z,
def Re z = address(&z): &float,
def Im z = address(&z) + 4: &float
```

As we will discuss in detail later in the chapter, `address(&z)` obtains the address of variable `z` and treats it as a `ulong` number. The expression "`address(&z): &float`" converts that address into a reference to a `float` value so you can read or write `Re z`. The definition of `Im z` is the same, except that we add 4 bytes to get to the `Im` field.

Pushing entire `struct` values onto the stack is only useful for very small structures such as `complex`. Larger structures should only be accessed as variables. To enforce this, define the `struct` with '`&`' before the argument *x*, i.e.,

`typedef τ = struct(&`*x*`) {`*ftype*$_1$ *name*$_1$`,` *ftype*$_2$ `{`*name*$_2$`,` *name*$_3$`}, ...}`

This tells GalaxC only to generate the field-access macros for variables or pointers of type τ.

8.2.1 Constructing `struct` *Values*

GalaxC provides an easy way to construct a `struct` value for a small structure like `complex`. Basically, we simply push the fields onto the stack in reverse order (since stacks grow down) and define the result to be the `struct`. For the type `complex`, this has the form: *[footnote:* Recall that the comma operator pushes its arguments onto the stack in reverse order if in parentheses.*]*

```
float arg {x, y},
def x + i y = (x, y): complex
```

This defines the notation "`x + i y`", where `x` and `y` are `float` and `i` is literal syntax, to mean creation of a `complex` value on the stack. We can also define the alternate notation:

```
float arg {x, y},
def x - i y = (x, -y): complex
```

to allow writing "`3.0 - i 4.0`" instead of "`3.0 + i (-4.0)`".

Once a structure has been defined, we can define operators. Here are function definitions for `complex` add and subtract:

```
complex arg {z1, z2},
fn z1 + z2 = (Re z1 + Re z2) + i (Im z1 + Im z2),
fn z1 - z2 = (Re z1 - Re z2) + i (Im z1 - Im z2)
```

These definitions are the same as those in most mathematics textbooks. Note that we do need a space between `i` and '`(`' so it is not parsed as a *call* expression.

Complex add and subtract can be defined as macros instead of functions, but this would probably be a bad idea. Since they are functions, `z1` and `z2` are treated as variables during the evaluation of the bodies, allowing the more efficient variable form of field access. If they were defined as macros, then the value form of field extraction would be needed, requiring complete `complex` values to be pushed onto the stack.

Furthermore, if the arguments are complicated expressions, then these must be reëvaluated each time they are used in the macro. Function arguments are only evaluated once. Implementing them as `inline` is better than macro, and may be better than `fn` depending on how efficient function calls are on a given machine architecture.

8.2.2 Subtypes of Structures

In GalaxC's type hierarchy, it is always possible to convert a subtype to its supertype and the type definition specifies how to do this. While this does not have to be the case, the integer and floating-point subtypes are always smaller than their supertypes and converting from the subtype to the supertype never loses information. A structure can have a subtype, but in this case the subtype is larger than the supertype: specificically, it has additional fields appended to the supertype. We convert from sub-structure to super-structure by discarding the additional fields.

This is somewhat confusing terminology. You would think that a sub-structure would mean a subset of the fields of a larger structure. However, what sub-structure means in the type hierarchy sense is that the subtype is more specific than the supertype, and having additional fields (and thus additional properties) makes the sub-structure more specific.

In GalaxC we use sub-structures for a specific version of a supertype. For example, the GalaxC Simplified Window Manager (G-SWIM) defines structure `GwinStruct` which stores the generic properties of a window, such as its size and title. XOE defines a subtype of `GwinStruct` called `XOEstruct`, which supplements `GwinStruct` with additional fields for document editing, such as the total size of the document, its scroll position, and the current cursor location.

A sub-structure is defined using a `substruct` definition, which has the form:

`typedef τ = substruct(S,` *x*`) {`*ftype*$_1$ *name*$_1$`,` *ftype*$_2$ `{`*name*$_2$`,` *name*$_3$`}, ...}`

This is the same as a `struct` definition except for supertype `S`. In fact, the `struct` definition can be defined as a macro with `S` = `void`:

```
spclarg {name, x, fields},
def {typedef name = struct(x) {fields}} =                [This macro has
    (typedef name = substruct(void, x) {fields})          not been tested.]
```

As a simple example, here are `structs` for two- and three-dimensional points with coordinates named `p.x`, `p.y`, and `p.z`. `Point3D` is defined as a substruct of `Point2D`:

```
typedef Point2D = struct(p) {int p.x, int p.y};
typedef Point3D = substruct(Point2D, p) {int p.z};
```

`Point3D` can also be defined as:

```
typedef Point3D = struct(p) {int p.x, int p.y, int p.z};
```

This has the same bit representation as the `substruct` version. The only difference is that you cannot directly assign a `Point3D` value to a `Point2D` variable, which you can do if one is a subtype (substruct) of the other.

8.3. Pointer Types

GalaxC pointers are essentially the same as C pointers. A pointer *p* is simply the memory address of a datum of any type τ. The type of *p* is **pointer type** `@`τ. Pointers have a number of important uses in GalaxC, including:

1. Linked data structures such as trees and linked lists.
2. Accessing elements of arrays.
3. Strings and files.
4. Function pointers, described earlier in Section 5.5.

Internally, pointer types are basically the same as reference types (Section 4.5) with an important difference: you must explicitly dereference a pointer whereas reference types automatically dereference themselves as part of subtype-to-supertype conversion. Pointer types give the programmer full control over when pointers are dereferenced. Reference types, on the other hand, are a mechanism for implementing the behavior of variables and arguments using GalaxC's type hierarchy. As we will see in this section, pointers and references work together to provide a clean way to implement both pointers and variables.

There is no need to define a pointer type: like reference types, any type τ can become a pointer type using the notation `@`τ. For example, to declare two pointer variables `p` and `q` with type `@int` write:

```
@int var {p, q}
```

or its equivalent notation:

```
int var {@p, @q}
```

The value types of `p` and `q` is `@int`. Their reference types are `&@int`. All pointer and reference types have the same storage size, 4 bytes in the current implementation.

GalaxC considers pointer types to be root types: there is no automatic conversion between a pointer type `@`τ and its base type τ, and the supertype of `@`τ is `void`. However, you can convert between pointers and references but in most cases you need an explicit operator as described below.

Here is a pointer version of the `GoodSwap` function from Section 5.2.2:

```
int arg {@x, @y},
fn Pswap(x, y) = {var temp = *x; *x = *y; *y = temp}
```

Each reference to the values of `x` and `y` requires an explicit dereference, just as with the `Cswap` function in Section 5.2.2.

Here are some examples to clarify normal variables, reference variables, and pointer variables:

- A normal variable has reference type `&`τ so it can be automatically dereferenced once to obtain its type τ value. The following code assigns the value of variable `y` to variable `x`:

```
int var {x, y};      // x and y have reference type &int.
x = y;               // Dereference y to get its int value. Do not dereference x.
```

- A reference variable has reference type `&&τ` so it can be automatically dereferenced once or twice to obtain `&τ` or `τ`, as needed. The following code assigns the value of the `int` referenced by variable `y` to the `int` referenced by variable `x`:

```
int arg {&x, &y},     // x and y have reference types &&int.
x = y;                // Dereference y twice to get int value.
                      // Dereference x once to get &int variable to assign to.
```

- A pointer variable has reference type `&@τ` so it can be automatically dereferenced once to obtain `@τ`. A second explicit dereference gets the `τ` value:

```
int var {@x, @y};     // x and y have reference types &@int.
x = y;                // Dereference y to get its @int value. Do not dereference x.
*x = *y;              // Dereference y twice to get int value.
                      // Dereference x once to get &int variable to assign to.
```

  The first assignment "`x = y`" assigns the `@int` pointer in variable `y` to variable `x`. The second assignment "`*x = *y`" assigns the `int` pointed to by variable `y` to the `int` pointed to by variable `x`. This is equivalent to "`x = y`" with reference variables: the only difference is that the pointer variables must be dereferenced explicitly while the reference variables dereference themselves automatically. Pointers let you choose whether you want to manipulate the `@int` pointers in `x` and `y` or the `int` values they point to.

Pointers are a lot more powerful than references. Pointers can be defined with arbitrary levels of indirection, whereas references generally have just one or two. A pointer's value can be changed to point to a different location, whereas references always point to the same location. GalaxC also has pointer arithmetic, which is not available for references. The flexibility of pointers requires more explicit notations, since sometimes one wants the pointer itself and other times the value pointed to. References are much more limited, so it is possible to simplify notations.

8.3.1 Converting between Pointers and References

GalaxC provides the dereference `*p` and reference `&v` operators to convert between pointers and references. The GalaxC '`*`' and '`&`' operators are very similar to C, but not exactly the same. In most cases they have the same effect, but not necessarily for the same reason.

`*p` The dereference operator '`*`' first performs any automatic dereferencing of `p` to convert it to an `@τ` pointer. Then it converts the `@τ` pointer to reference type `&τ` so that it can be automatically dereferenced once more to load or store a `τ` value. In essence, `*p` changes pointer `p` into a variable of type `τ` so that its value can be obtained or updated. If `p` is a variable with reference type `&@τ`, GalaxC loads the `@τ` pointer value and changes it to reference type `&τ`. If `p` is a pointer-valued expression rather than a variable, then there is no extra load since the value is already of type `@τ`. In other words, `*p` does not by itself load a value of type `τ`, but a load or store usually occurs due to automatic dereferencing of `&τ`.

`&v` The reference operator '`&`' converts a variable `v` with reference type `&τ` to a pointer type `@τ`, which prevents any further dereferencing. For example, "`p = &v`" assigns the address of variable `v` (of reference type `&τ`) to pointer `p` (with value type `@τ`). Even though `&v` produces a pointer rather than a reference, GalaxC uses the notation because it is consistent with C and is generally used in exactly the

same way.

GalaxC does not need an operator when passing a pointer to a function's reference argument or assigning an initial value to a reference variable. These are handled implicitly in the compiler functions that match patterns, pass arguments, and declare variables. For example, in the variable declaration:

```
var &x = p
```

expression `p` can have type `@τ`, and in a call to function `GoodSwap(x, y)` the values passed to `&int` arguments `x` and `y` can be `&int` (variables of type `int`) or `@int` (pointer values or expressions). Macros are handled a little differently: see Section 8.3.5.

For the curious, let's see how GalaxC implements `*p` and `&v`. First, here is the '`*`' operator, defined as a *generic macro*:

```
type arg T, @T arg p,
def *p = p: &T
```

A generic macro (Chapter 9) is defined for an arbitrary type `T`, which may or may not appear in the macro pattern. The `*p` macro matches any argument `p` provided that it is a pointer type `@T` or a subtype of a pointer type, e.g., a variable with reference type `&@T`. The body of the macro is trivial: it simply leaves `p` on the top of the stack and gives it the type `&T`, so that now `p` is a reference instead of a pointer and can be treated as a variable of type `T`. In the body, `T` is whatever type was matched by `@T`, with `@` stripped off.

When calling `*p`, GalaxC automatically converts the actual argument to type `@T`. Consider this example:

```
int var {@x, @y};
*x = *y;
```

In the assignment "`*x = *y`", GalaxC calls macro `*p` twice. Variables `x` and `y` have reference types `&@int`, so when calling `*p`, GalaxC automatically loads the `@int` values stored in `x` and `y`. The dereference operator simply tells GalaxC to treat the `@int` pointers as `&int` references, and the assignment loads the value pointed to by `y` into the variable pointed to by `x`.

It would be tempting to define the reference operator `&v` the same way:

```
typearg T, &T arg v,
def {&v} = v: @T
```

However, this does not work because '`:`' automatically dereferences `v` which is exactly what we don't want. The reference operator is instead defined as a special function, described in Section 10.2.6.

GalaxC has a special construct for obtaining a pointer to a reference argument. As we saw in Section 5.2.2, `&v` for a reference argument `v` gives us the address of argument `v` on the stack, not the variable pointed to by `v`. Instead, use `refptr(v)` which converts a reference argument of type `&&τ` to pointer type `@τ`. It is defined as a generic macro:

```
& &T arg v,
def refptr(v)  = refadx(v): @T,
```

The body of the macro extracts the address pointed to by `v` (which may be `NULL`) and makes it a pointer.

`refadx` is defined in Section 5.2.2.

8.3.2 Special Types

GalaxC provides some special types needed by pointers. They have special properties and are built into the compiler.

`nulltype`

All pointers may have the value `NULL`, implemented as the `ulong` value `0`. `NULL` is a special case since it can be assigned to any type of pointer or reference variable or argument. `NULL` has special type `nulltype`, which is treated as a subtype of all pointer types `@τ` and reference types `&τ`. Here is the definition of `NULL` from `gxclib`:

```
def NULL = 0: nulltype
```

While `nulltype` is mostly used for implementing `NULL`, it can be used for other values as well. For example, `gxclib` defines `MAX_ADDRESS` to be the largest possible 32-bit address:

```
def MAX_ADDRESS = 0xFFFF_FFFF: nulltype
```

Like `NULL`, `MAX_ADDRESS` can be used with any pointer or reference.

In Section 5.2.3, we stated that if a function returns a value using a `return` statement, then all `return` statements must return the same type, which is determined by the first `return` statement found in the body. However, if the function returns a pointer type, the body's type is determined by the first `return` statement that does not return `nulltype`. You may have `return` statements before it that return `nulltype` values, e.g., returning `NULL` if certain conditions are met.

`unknown`

GalaxC uses type `unknown` to declare objects whose types are determined later. For example, if you omit the type in a function or macro definition, this is equivalent to declaring the object as type `unknown` which is replaced by the body's type when the latter is compiled. GalaxC defines untyped definitions as macros:

```
spclarg pattern, spclarg body,
def {fn pattern     = body} = (unknown fn  pattern = body),
def {def pattern    = body} = (unknown def pattern = body),
def {inline pattern = body} = (unknown def pattern = body)
```

Type `unknown` is mostly used within GalaxC and is not used by most programmers. We will see some interesting uses of `unknown` in Section 9.3 and Chapter 10.

`@unknown`

Type `@unknown` is the universal pointer type, used by functions that need to pass a value by reference but do not know what type it is until run time. `@unknown` is equivalent to `void*` in C, but has a more intuitive name. All references and pointers are treated as subtypes of `@unknown`.

A number of C library functions have `@unknown` arguments and return values. For example, the C `memcopy` and `memmove` functions copy bytes without knowing what type those bytes represent. `gxclib` defines them as :

```
@unknown arg {src, dst}, int arg n,
@unknown inline memcpy(dst, src, n)  = cdecl,
@unknown inline memmove(dst, src, n) = cdecl
```

They both return `dst` as an `@unknown`.

Similarly, `malloc` allocates `size` bytes of memory in dynamic storage and returns an `@unknown` pointer to it:

```
int arg size,
@unknown inline malloc(size) = cdecl
```

The returned pointer is usually typecast into a defined pointer type. Use `@unknown` with extreme caution as it bypasses GalaxC's protection mechanisms.

Currently you cannot define a type `T` to be a subtype of a pointer type `@S`. One place this would be very useful is defining a subtype of `@unknown` to represent a pointer to a structure you know nothing about, such pointer to a file descriptor returned by C library function `fopen`. However, since this is not available at the present time, we need to use a trick. Here is how type `file` is currently defined in `gxclib`:

```
typedef amorphFile = roottype(0);
def file = @amorphFile;
```

Type `amorphFile` (short for amorphous file) is defined as a root type with size `0`. We really don't care what size it is because we will never dereference a pointer to it. Then `file` is defined to be equivalent to a pointer to `@amorphFile`, i.e., it is an *amorphous pointer*. Once `file` is defined, we can use it to define C library file access `cdecls`:

```
string arg {name, mode},
file inline fopen(name, mode) = cdecl,

unknown arg @buf, int arg {size, n}, file arg f,
int inline fread(buf, size, n, f)  = cdecl,
const unknown arg @buf,     // fwrite doesn't change contents of buf.
int inline fwrite(buf, size, n, f) = cdecl;
int inline fclose(f) = cdecl
```

We could have just made `file` equivalent to `@unknown`, but then every pointer could be passed to `fread`, `fwrite`, `fclose`, etc. instead of just pointers of type `file`.

At some point we will allow `file` and other types to be defined as subtypes of `@unknown`.

8.3.3 Pointer Arithmetic

Unlike reference types, pointers have a rich set of pointer arithmetic. For the most part, GalaxC pointer arithmetic is the same as C and is defined using a small collection of generic macros in `gxclib`.

When performing pointer arithmetic, GalaxC treats pointers as `ulong` values. Specifically, it converts them to a subtype of `ulong` called `address`:

```
typedef address = subtype(ulong)
```

To convert a pointer to `address`, we use the generic macro:

```
type arg T, @T arg p,
def address(p) = p: address,
```

The pattern `address(p)` matches any `p` that is of type `@T` or a subtype of `@T`. Calling `address(q)` converts `q` to its pointer (super)type `@T`, dereferencing `q` if it is a variable. The macro body simply typecasts the `@T` value into `address` so it can be treated as `ulong`.

We can test whether a pointer is even or odd using the following generic macros that test whether `address(p)` is an even or odd integer:

```
type arg T, @T arg p,
def p is even = address(p) is even,
def p is odd  = address(p) is odd,
```

The macro bodies use the "`x is even`" and "`x is odd`" macros that are defined for `ulong` values of `x` in Section 6.4.7.

Adding and subtracting pointers is more interesting. As with C, adding `1` to a GalaxC `@τ` pointer adds `|τ|` to the pointer's address, i.e., it points to the following `τ` in memory. Adding *n* adds *n*`*|τ|`. This allows C and GalaxC to treat a block of memory pointed to by an `@τ` pointer as an array of type `τ` values. (Arrays will be treated in detail later in the chapter.) GalaxC implements adding an integer to a pointer using the following generic macros:

```
type arg T, @T arg p, ulong arg n,
def p + n  = address(p) + n*|T|: @T,
def p - n  = address(p) - n*|T|: @T
```

Given a pointer `p` of any pointer type `@T`, the expression "`p + n`" is the `@T` pointer you get by skipping forward `n*|T|` bytes of memory (backward if `int(n) < 0`). Similarly, "`p - n`" is the `@T` pointer you get by skipping backward `n*|T|` bytes of memory (forward if `int(n) < 0`). The product `n*|T|` is the same whether `n` is signed or unsigned: if you multiply two 32-bit integers and only keep the low 32 bits of the product, the result is the same for both unsigned and 2's complement integers.

As described in Section 6.4.11, you can use the C-style increment and decrement operators for pointers: `p++`, `p--`, `++p`, `--p`, `p += n`, and `p -= n`. They automatically multiply the increment/decrement offset by `|T|`.

As with C, the difference "`p - q`" between two pointers `p` and `q` is the number of type `T` values between them. It's only defined if they point into the same array. Calculating "`p - q`" requires dividing the difference of the addresses by `|T|`:

```
type arg T, @T arg {p, q},
def p - q  = ((p: long) - (q: long))/|T|      // Signed arithmetic.
```

Pointers `p` and `q` must be the same pointer type `@T` and we must use signed division to calculate the correct result if `p < q`.

It does not make sense to add two pointers, so "p + q" is not defined.

GalaxC provides min and max operators for pointers, defined as:

```
type arg T, @T arg {p, q},
def min(p, q) = min(address(p), address(q)): @T,
def max(p, q) = max(address(p), address(q)): @T
```

As with "p - q", pointers p and q must be the same pointer type @T. The bodies match the ulong versions of min and max.

GalaxC also has a set of relational operators for pointers. Pointers p and q must be the same pointer type @T:

```
type arg T, @T arg {p, q},
def p == q = address(p) == address(q),
def p != q = address(p) != address(q),
def p <  q = address(p) <  address(q),
def p >  q = address(p) >  address(q),
def p <= q = address(p) <= address(q),
def p >= q = address(p) >= address(q)
```

The bodies all match ulong relational operators.

You can also compare pointers for equality to NULL or any other nulltype value:

```
type arg T, @T arg p, nulltype arg q,
def p == q = address(p) == (q: address),
def p != q = address(p) != (q: address)
```

The nulltype value must be on the right hand side of '==' or '!='.

GalaxC also allows the C language notation "!p" to compare p to NULL, as well as GalaxC's "?p". They are defined by the macros:

```
type arg T, @T arg p,
def !p = !address(p),       // Equivalent to "p == NULL".
def ?p = ?address(p)        // Equivalent to "p != NULL".
```

8.3.4 Pointers to Structures

A particularly important use of pointers is dynamically-linked data structures, such as linked lists and binary trees. For example, here is the definition of a binary tree node that has an int data value and two pointers to other tree nodes:

```
typedef TreeNode = struct(&p)
{
    int         p.data,         // Data stored in the node.
    @TreeNode   p.left,         // Pointer to left subtree.
    @TreeNode   p.right         // Pointer to right subtree.
};
def Tree = @TreeNode;
```

We have used Pascal/C notation for the three fields `p.data`, `p.left`, and `p.right`; we could have used any legal notation. The field argument `p` is defined using `&p` which means that fields-access macros are only defined for `&TreeNode` variables and `@TreeNode` pointers, not `TreeNode` values. We will not have occasion to push an entire `TreeNode` onto the stack.

Fields `p.left` and `p.right` are `@TreeNode` pointers. This recursion is fine as long as the fields are pointers or references. It would be incorrect to have a field "`TreeNode p.bad`" since it would imply an infinitely nested data structure. The `@TreeNode` pointers have the usual address size = 4 bytes each.

After defining `TreeNode`, we use a macro to define `Tree` to be equivalent to `@TreeNode` since we expect to be using pointers to `TreeNode` much more often than nodes. The size of type `TreeNode` is 12 bytes. The size of type `Tree` is 4 bytes, like any other pointer type.

`TreeNode` is a simple recursion, where a `struct` has fields which are pointers to the same type of `struct`. Complex data structures may have several `struct` types which are mutually recursive. In this case, it may be necessary to declare a forward type, e.g.,

```
typedef Node = @forward
```

This has not been implemented yet and the notation may change when we do.

Defining `TreeNode` creates field-access macros equivalent to:

```
Tree arg p,
def p.data  = address(p): &int,
def p.left  = address(p) + 4: &Tree,
def p.right = address(p) + 8: &Tree
```

Each macro takes a `Tree` pointer `p`, adds a fixed offset to the named field, and then treats it as a variable by casting it into a reference type so that the field can be read or written.

We can use the above definition of `Tree` to implement a sorted binary tree search algorithm. The tree is created such that for each subtree, all values less than or equal to the root's `data` are in the `left` subtree and all values greater than the root data are in the `right` subtree. Here is the search algorithm:

```
// Find the value x in tree p.
// If present, return a pointer to the node that matches x.
// If not present, return NULL.
int arg x, Tree arg p,
Tree fn find x in p =
    // If tree p is NULL, then x is not found.
    if p == NULL then return NULL else
    // If root data equals x, then we have found a matching node.
    if x == p.data then return p else
    // If x is less than root, then search left subtree.
    if x < p.data then return (find x in p.left)
    // Otherwise search right subtree.
    else return (find x in p.right)
```

Like all recursive functions, we must explicitly declare "`find x in p`" to return type `Tree`. We did not need to cast `NULL` to type `Tree` explicitly since `nulltype` is a subtype of every pointer type.

C programmers will note that we use the notation `p.left` rather than `p->left`. In GalaxC, you only define one notation to access a given field access and it is used whether `p` is a pointer, a variable, or a `struct` value on the stack. The author finds it annoying that you have to use different notations for `struct` pointers and variables in C, as if the compiler couldn't figure it out for itself.

The "`find x in p`" function can be rewritten in a much more compact form using conditional expressions:

```
int arg x, Tree arg p,
Tree fn find x in p =
    p == NULL or x == p.data? p:
    find x in (x < p.data? p.left: p.right)
```

Next, we consider the tree insertion function. In this case, `x` points to a new `Tree` node that we want to insert into sorted `Tree p`. Note that we pass `p` by reference so that it can be modified if it's `NULL`:

```
Tree arg {x, &p},
void fn insert x into p =
    if p == NULL then
    {    // Replace p with x.
        p = x; x.left = x.right = NULL;
    } else
    if x.data <= p.data then insert x into p.left
    else insert x into p.right
```

Once we have created a sorted tree, we can print it in increasing order by performing an inorder traversal:

```
Tree arg p,
void fn print p =
{
    if p == NULL then return;
    print p.left;
    printf("%d\n", p.data);      // Print each node on a new line.
    print p.right;
}
```

Indenting the printout to show tree structure is quite simple:

```
Tree arg p, int arg indent,
void fn print p (indent) =
{
    if p == NULL then return;
    // Print 0 to 20 spaces by starting at offset 20-indent into a string of 20 spaces.
    printf("                    " + max(20-indent, 0));
    print p.left (indent+2);
    printf("%d\n", p.data);      // Print each node on a new line.
    print p.right (indent+2);
}
```

Call this function with the statement: "`print root (0)`" where `root` points to the root node of the tree.

8.3.5 Variant Fields

In some cases, it is useful to use the same `struct` for multiple purposes without having to redefine it from scratch. For example, we might want to reuse `Tree` nodes for `float` or `string` data. This can be done by defining **variant fields**, such as:

```
Tree arg p,
def p.fdata = &p.data: &float,          // Use p.data to store a float value.
def p.sdata = &p.data: &string          // Use p.data to store a string pointer.
```

This allows notations `p.fdata` and `p.sdata` to treat `p.data` as a `float` or `string` variable. Note the use of '`&`' in the example: "`&p.data`" prevents reference `p.data` from being automatically dereferenced by the colon operator. `&float` and `&string` treat the reference `p.data` as a `float` or `string` variable so you can read or assign to it.

The `p.fdata` and `p.sdata` macros require a `Tree` argument, which is equivalent to `@TreeNode`. They do not allow you to read or write the fields of a `TreeNode` variable. For example, it is incorrect to write:

```
TreeNode var tn;
tn.fdata = 1.4;
```

Now we could rewrite the above macros as:

```
TreeNode arg &p,
def p.fdata = &p.data: &float,          // Use p.data to store a float value.
def p.sdata = &p.data: &string          // Use p.data to store a string pointer.
```

This would allow us to access the fields of a `TreeNode` variable, but not a node pointed to by a `Tree`. To get both, define the macros as:

```
TreeNode arg *p,                        // p can be &TreeNode or @TreeNode.
def p.fdata = &p.data: &float,          // Use p.data to store a float value.
def p.sdata = &p.data: &string          // Use p.data to store a string pointer.
```

This allows `p` to be either a `TreeNode` variable (`&TreeNode`) or a `TreeNode` value (`@TreeNode`). The field-access macros defined automatically by `typedef struct` and `typedef substruct` allow both forms of access.

Variant fields are similar to C unions and Pascal variant records. As with both of those, you must be very careful defining and using variant fields since they evade the protections of strong typing. At some point, we would like to add a *generic structure* capability to GalaxC so that you can define structures that have all fields the same except for some generic fields that can be any type τ, and also add *generic functions* where `find x in p`, `insert x into p`, and `print p` could be defined for a generic type τ instead of only for type `int`. Currently GalaxC limits generic definitions to macros and inline functions (Chapter 9) and special functions (Chapter 10), which have been more than adequate for our needs so far.

8.4. Arrays

GalaxC arrays are based on C. Currently they can only be defined with a single dimension, though it always possible to treat a one-dimensional array as two-dimensional or more by defining suitable macros. In fact, it's

hard to come up with the "right" definition of a two-dimensional array since its efficient implementation may depend on a lot of factors, such as symmetry and sparseness. One-dimensional arrays have worked out fine so far.

GalaxC arrays are normally defined in a `var` declaration. For example:

```
float var v[200]
```

defines an array of 200 `float` values. The array size in a `var` declaration must be a compile-time constant and has an implementation-defined maximum. At the present time you cannot initialize an array as part of its declaration.

To access an element of an array, use the notation `v[i]`, where *index* `i` is an `int` value from `0` up to but not including the size of the array. Like C, GalaxC does not check array limits, so it is possible for incorrect code to go past the beginning or end of an array.

Like C, GalaxC arrays are closely coupled to pointers. In fact, a pointer to type τ can always be thought of as pointing to element 0 of a τ array, and the type of an array variable is `@`τ. For example, variable `v` defined above has type `@float`. The concepts of value type and reference type do not apply to array variables since they are not automatically dereferenced.

`gxclib` defines the standard array access notation as a generic macro:

```
type arg T, @T arg p, ulong arg n,
def p[n] = address(p) + n*|T|: &T
```

This is the same as the definition of "`p + n`" (§8.3.3) except that the calculated address of the array element is cast into reference type `&T` so that `p[n]` is treated as a type `T` variable. You can easily define alternate notations for accessing array elements, for example using formatted subscripts.

You can also declare arrays as `struct` fields, e.g.,

```
typedef Node = struct(&p)
{
    @Node   p.next,          // Pointer to next node in linked list.
    int     p.data[20]       // Data stored in the node.
};
```

At the present time you cannot have an array argument: you must always pass arrays as pointers, which amounts to the same thing.

8.4.1 Strings

One of the most common use of arrays in GalaxC is `char` arrays, also known as `strings`. A `string` is equivalent to a `char` pointer and is defined as:

```
def string  = @(const char),
def wstring = @char
```

A `string` is read-only: you can read the characters, but not write to them. A `wstring` is read/write. A literal character string like `"cat"` is a read-only `string`, not a writable `wstring`.

You can define global or local storage for a `string` or `wstring` by declaring a `char` array:

```
char var buf[200]           // buf is a wstring = @char.
```

and then use `buf[i]` to access the `i`th character of the string (counting from 0). You can also initialize a `string` or `wstring` variable `s` to point to `buf`, e.g.,

```
var s = buf                 // s is the same type as buf.
```

Since `buf` is a `char` array, its type is `@char` so the type of `s` is also `@char` or its equivalent `wstring`. We can now fill `buf` with the letters '`a`' through '`z`':

```
char var ch;
for ch = 'a' thru 'z' do *s++ = ch
```

Note that we cannot write:

```
for ch = 'a' thru 'z' do *buf++ = ch
```

This is because `buf` is a `wstring`, but it's not a `wstring` variable like `s`. You cannot assign a value to `buf`, only to its elements. On the other hand, you can write this:

```
for ch = 'a' thru 'z' do buf[ch - 'a'] = ch
```

You can create a local `string` variable that is initialized with a pointer to a string literal, e.g.,

```
var s = "Peter Rice eats fish"
```

In this case, GalaxC allocates a character array containing "`Peter Rice eats fish`" somewhere and provides a `string` pointer to it, which becomes the initial value of `s`. GalaxC treats the character array pointed to by `s` as a constant, so you cannot write:

```
s[6] = 'M'
```

However, a fiendish programmer could cast `s` into a `wstring` and write:

```
(s: wstring)[6] = 'M'
```

which changes the string to "`Peter Mice eats fish`". This is called "clobbering the constant" and can produce difficult-to-unravel bugs, or may cause an access violation if string constant storage is protected by the operating system.

You can also allocate large strings using `malloc`, e.g.:

```
var s = malloc(n): wstring;     // malloc returns @unknown: typecast it into wstring.
if s == NULL then ...           // Make sure malloc succeeded.
```

8.4.2 String Operations

Since GalaxC strings are implemented the same as C strings, GalaxC can use string functions from the C library. For example, `gxclib` defines `cdecl` calls to the C library functions `strcpy` (string copy), `strcat` (string concatenation), `strlen` (string length), `strcmp` (string compare), and `strcasecmp` (string compare ignoring upper/lower case).

```
wstring arg dst, string arg src,
wstring inline strcpy(dst, src) = cdecl,
wstring inline strcat(dst, src) = cdecl;

string arg {s, t},
int inline strlen(s) = cdecl
int inline strcmp(s, t) = cdecl,
int inline strcasecmp(s, t) = cdecl
```

GalaxC programs can use these as is, but since GalaxC has flexible syntax we can do clever things like use `|s|` for string length:

```
string arg s,
def |s| = strlen(s)
```

We can also define alternate notations for `strcpy` and `strcat` that make it clear which argument is `dst` and which is `src`:

```
wstring arg dst, string arg src,
def {dst = copy src} = strcpy(dst, src),
def {dst = append src} = strcat(dst, src)
```

Comparing strings is a pain in C. The intuitive notations "`s == t`", "`s < t`", "`s >= t`", etc., compare the *pointers* to the strings, not their contents. For comparing the contents, you must call function `strcmp(s, t)` which compares the contents lexicographically and returns `-1` if `s` < `t`, `0` if `s` = `t`, and `+1` if `s` > `t`. This results in C code like:

```
if (!strcmp(s, t)) ...          // Do ... if s and t have the same contents.
```

"`!strcmp(s, t)`" is equivalent to "`strcmp(s, t) == 0`" which is true if `s` and `t` have the same contents, which seems backwards. `gxclib` hides this from the programmer by providing intuitive macros for comparing string contents:

```
string arg {s, t},
def {s == t (check)} = strcmp(s, t) == 0,
def {s != t (check)} = strcmp(s, t) != 0,
def {s <  t (check)} = strcmp(s, t) <  0,
def {s <= t (check)} = strcmp(s, t) <= 0,
def {s >  t (check)} = strcmp(s, t) >  0,
def {s >= t (check)} = strcmp(s, t) >= 0
```

In GalaxC you can write:

```
if s == t (check) then ...      // Do ... if s and t have the same contents.
```

"`(check)`" is literal syntax that in this case means to use `strcmp`, which distinguishes upper and lower case letters. `gxclib` also defines versions that ignore case:

```
string arg {s, t},
def {s == t (ignore)} = strcasecmp(s, t) == 0,
def {s != t (ignore)} = strcasecmp(s, t) != 0,
def {s <  t (ignore)} = strcasecmp(s, t) <  0,
def {s <= t (ignore)} = strcasecmp(s, t) <= 0,
def {s >  t (ignore)} = strcasecmp(s, t) >  0,
def {s >= t (ignore)} = strcasecmp(s, t) >= 0
```

These call `strcasecmp`, which does not distinguish between upper and lower case.

Chapter 9
Generic Macros and Inline Functions

In previous chapters we have shown examples of generic macros without describing them in detail. In this chapter we will look closely at how GalaxC generic macros and inline functions work.

9.1. Generic Macros

Most GalaxC operations, whether defined as functions or macros, are defined for arguments of specific types. Indeed, one of the strongest features of GalaxC is that it allows the reusing the same syntax with different types of arguments, a technique called *overloading* (§5.4). However, there are cases where we would like to define an operation in essentially the same way for many types. For example, the assignment expression "`x = y`" always does the same thing no matter what type `x` is: GalaxC evaluates expression `y` on the stack, converts it to the value type `τ` of variable `x`, and then stores the top of stack value in `x`. It would be silly to have to repeat this for each defined type, when the only difference between them is `|τ|`.

GalaxC therefore implements assignment using a **generic macro**, which adapts itself to the types of the arguments. Here is its definition from `gxclib`:

```
type arg T, &T arg x, T arg y,
T def {x = y} = Assign(x:T, y);
```

`T` is a *type argument*, i.e., an argument of built-in type `type`. Arguments `x` and `y` are *generic arguments*: their types are based on type argument `T`. Note that `x` is declared to be a reference to `T` (`&T`) while `y` is a value of the same type `T`.

The generic macro pattern "`x = y`" matches *any* assignment expression where `x` is a reference type `&T` (or a subtype such as `&&T`) and `y` is the same type `T` or a subtype (e.g., `&T` or `&&T`). Note that `T` is not explicitly shown in the pattern: it is a **hidden argument** which is set to the type associated with argument `x`. Consider the code:

```
int var i;
i = 3;
```

The expression "`x = y`" matches the assignment pattern with `T` bound to `int`. When it comes time to call the macro, GalaxC calls `Assign(x:T, y)` with `T = int`, `x = i`, and `y = 3`, i.e., `Assign(i:int, 3)`. `Assign` is a special function that creates the code to push `3`, convert it to `int` if it were not already `int`, and store the top of stack value to `i` as a 4-byte `long`. This behavior was described in detail in Section *6.4.10*.

We have explicitly declared the type of the macro to be `T`. This is not strictly necessary, since `Assign` has type `T`. However, it allows the compiler to see that the macro has type `T` without having to call `Assign`, which improves performance.

We have already looked at most of the generic macros defined in `gxclib` in previous chapters. Because generic macros are so powerful, we do not need many of them to define the standard GalaxC operators.

9.1.1 Pattern Matching

Let's take a closer look at how GalaxC matches patterns. All patterns, whether they are variables, arguments, macros, or functions are matched the same way: indeed, GalaxC does not know what an

expression means until it has found a matching pattern. To match an expression *E*, GalaxC looks searches through all the patterns that could possibly match in reverse order (so that later patterns have priority over earlier ones) until it finds a matching pattern.

For each pattern *P*, GalaxC first sees if the syntax of *P* matches the syntax of *E*. This is easy to do, since both are in the form of **parse trees**. For the syntax to match, all the syntax rules must match (e.g., if *P* expects a *bitand* then *E* must also be a *bitand*) and all literal syntax must match. If *P* has arguments, they match whatever parse subtrees are in *E* at those points. The subtrees can be arbitrarily complex.

If the entire pattern syntax matches *E*, then GalaxC sees if the arguments match by analyzing the subtree *S* in *E* associated with each argument and seeing if the type of *S* matches the argument's type. Normal arguments have a fixed type τ, so all we need to do is see if the type of *S* is τ or a subtype of τ. If τ is a reference `&T` and *S* is a pointer `@T` then they match if the pattern is a function or the argument was declared with `*T` (§8.3.5). If τ is `@unknown`, then every pointer is considered to be a subtype. If τ is a pointer or reference, then every *S* of type `nulltype` is a subtype.

Generic arguments are more fun: If `x` is the first generic argument in the pattern, GalaxC tries to match the type of *S* to the form of `x`'s type, taking into consideration whether `x`'s type is a pointer and/or reference. For the assignment macro, the type of *S* must be of the form `&T` or a subtype such as `&&T`. The pattern matcher automatically dereferences the type of *S* until it matches the generic type. If the match fails, the entire pattern match fails and GalaxC goes on to the next pattern. If the match succeeds, GalaxC temporarily binds `T` to the possibly-dereferenced type of *S*.

If an argument `y` is not the first generic argument of type argument `T`, then the type of *S* must match the argument's type using the value of `T` set by the first generic argument. Once the entire pattern has matched, `T`'s binding is restored to its previous value.

Analyzing each subexpression requires recursion. In some cases we need to analyze subexpressions multiple times, which is why we check that the syntax matches before analyzing any subexpressions needlessly.

Once we have found a pattern that successfully matches, we can tell whether *E* is a variable, an argument, a macro call, a function call, or something else. We can then generate code for it as described in Section 10.1.3.

9.1.2 Macros with Unknown Arguments

The "`x = y`" assignment macro shown earlier uses a type argument so that `x` and `y` are based on the same type. Sometimes the type relationship between the arguments doesn't matter. For example, here are the `gxclib` macros that define relational operators ≡, ≠, ≤, and ≥:

```
spclarg {a, b},
def a ≡ b = a == b,     // Symbol ≡ is equivalent to '=='.
def a ≠ b = a != b,     // Symbol ≠ is equivalent to '!='.
def a ≤ b = a <= b,     // Symbol ≤ is equivalent to '<='.
def a ≥ b = a >= b;     // Symbol ≥ is equivalent to '>='.
```

Since `a` and `b` are defined as `spclarg`, the macros do not care what types arguments `a` and `b` have: GalaxC defers type matching to when it analyzes "`a ≡ b`" and the others with actual expressions for `a` and `b`. If the arguments are `ulong` or a subtype, GalaxC expands the macro into an integer compare. If the arguments are floating-point, GalaxC expands the macro into a `float` or `double` compare. We will describe `spclarg` in

more detail in the next section and §10.2.

9.2. Generic Inline Functions

Inline functions can also have generic arguments. The most important use of them is assignment expressions of the form "`a` *op*`= b`" such as "`x &= 0xFF`". `gxclib` defines them as:

```
type arg T, &T arg x, spclarg y,
inline {x += y}  = (x = x + y: T),
inline {x -= y}  = (x = x - y: T),
inline {x *= y}  = (x = x * y: T),
inline {x /= y}  = (x = x / y: T),
inline {x %= y}  = (x = x % y: T),
inline {x &= y}  = (x = x & y: T),
inline {x #= y}  = (x = x # y: T),
inline {x |= y}  = (x = x | y: T),
inline {x ^= y}  = (x = x ^ y: T),
inline {x <<= y} = (x = x << y: T),
inline {x >>= y} = (x = x >> y: T),
inline {x ^^= y} = (x = x ^^ y: T)
```

Let's take a closer look at one of them, e.g., "`x &= y`". The pattern matches any expression "`a &= b`" where `a` is a reference type `&T` -- or a subtype. Since `y` is a special argument (`spclarg`), `b` can have any type and is treated like a macro argument. We will take a closer look at `spclargs` in Section 10.2.

We next consider the body "`x = x & y: T`". Since `x` and `y` are generic, GalaxC cannot properly match the expression "`x & y`" when it test-compiles the body of the macro. Instead, it must re-match it each time the macro is called. This is the point at which `spclarg y` is actually analyzed and we see if there is in fact a pattern that matches "`x & y`". The body includes an assignment, so the type of "`x & y`" must be `T` or a subtype of `T` or else the macro call fails with a compiler error. The successful behavior is to compute "`x & y`", leaving the result on the stack, automatically convert it to `T` if necessary, and assign the result to variable `x`. We then explicitly cast the assignment to `T`: Section 9.3 will explain why.

While we could have declared these as macros, `inlines` are safer because `x` is used multiple times in the body. Note that `x` is a reference argument of each `inline` since it is declared as `&T`.

Now let's take a closer look at:

```
type arg T, &T arg x, spclarg y,
inline {x += y}  = (x = x + y: T),
inline {x -= y}  = (x = x - y: T)
```

These are generic forms of adding an offset to a variable, and are implemented by generating the normal code for the body. If `x` is an integer or pointer and `y` is an integer, we can generate better code by using the `Increment` versions of these operators which we saw earlier in Section 6.4.11:

```
type arg T, &T arg x, ulong arg y,
def {x += y} = Increment(x, T, y, preinc),
def {x -= y} = Increment(x, T, y, predec),
```

These macros are defined in `gxclib` after the `inline` versions so they have priority. While the macros are

also generic, the `y` argument is limited to `ulong`. If an expression has a non-integer `y`, it matches the `inline` version. For example, here is a `float` increment:

```
float var f;
f += 5.0;
```

This matches `inline` "`x += y`" with `T` = `float` and generates the code: "`f = f + 5.0`".

A more interesting example is incrementing a `complex` number as defined in Section 8.2:

```
complex var z;
z += 3 + i 4;
```

The `inline` expands to "`z = z + (3 + i 4)`" and automatically generates a call to the `complex` add function "`z1 + z2`". The marvelous thing about the generic `inline` is that it gives you the *op*= assignments for a new type without having to do any additional work once `x+y`, `x-y`, etc. have been defined. However, GalaxC allows you to implement the *op*= assignments in a different way if you wish.

9.3. Templates

When GalaxC processes a macro or `inline` definition it always compiles the body to make sure it is semantically correct, and then discards any code created by doing so. This is an important feature since it exposes errors when the macro or `inline` is defined, where they are generally much easier to fix than when the macro or `inline` is called. Since GalaxC parses the entire program before performing any semantic analysis, macro and `inline` definitions are always syntactically correct when they are processed, so we are only concerned with semantic errors like an expression that does not match any defined pattern.

In contrast, C macro definitions do not have any semantic (or even syntactic) checking, so errors only show up when the macro is called. These are often challenging to figure out.

Test-compiling a generic macro or `inline` body is a bit of a poser since when GalaxC processes the definition the type argument is unbound and therefore equal to special type `unknown` (§8.3.2). All the generic arguments are `unknown` as well. Still, GalaxC must somehow know that basic expression like "`x+y`" are legal even for `unknown` `x` and `y`.

GalaxC provides a special kind of macro called a **template** to handle this situation. For example, here is a template for "`x+y`":

```
unknown arg {u, v},
def {u + v} = template
```

The template indicates that "`unknown + unknown`" is semantically legal. If no type is specified to the left of `def` then the resulting expression is also `unknown`. Type `unknown` has type size `0` and no code is generated for the template.

Let's see how GalaxC test-compiles the body of the `inline` definition of "`x += y`":

```
type arg T, &T arg x, spclarg y,
inline {x += y}  = (x = x + y: T)
```

When GalaxC test-compiles "`x + y`", `x` and `y` are both of type `unknown`. Therefore they match the template

for "`unknown + unknown`", which produces an `unknown` result. This is then assigned to `x` using the standard assignment macro, which produces an `unknown` result as well. At this point we need to typecast the `unknown` value of "`x = x + y`" to `T` using the colon operator so that the body has a type other than `unknown`. The typecast works because `T` is considered `unknown` while test-compiling the body and thus has the same type size `0` as the assignment expression.

In some cases, we actually want to specify a type for the `unknown` expression, e.g.,

```
unknown arg u,
Boolean def {!u} = template
```

The `NOT` operator should always produce a `Boolean` result, even if the argument is temporarily `unknown`. This turns out to be important for implementing *repeat-until* as a macro (§7.6.1):

```
spclarg {a, b},
def {repeat b until a} =
{label looplabel:
 {b; label contlabel};
 if !a then goto looplabel;
 label breaklabel
}
```

In this case, `a` and `b` are declared as `unknown spclargs`, so when GalaxC test-compiles the body it must deal with `unknown` values of `a` and `b`. Defining a `Boolean` template for `!u` helps GalaxC compile "`if !a then goto looplabel`".

Here are the templates defined in `gxclib`:

```
unknown arg {u, v},
Boolean def {!u} = template,
def {u + v}  = template,
def {u - v}  = template,
def {u * v}  = template,
def {u / v}  = template,
def {u % v}  = template,
def {u & v}  = template,
def {u # v}  = template,
def {u | v}  = template,
def {u ^ v}  = template,
def {u << v} = template,
def {u >> v} = template,
def {u ^^ v} = template,
Boolean def {u == v} = template,
Boolean def {u != v} = template,
Boolean def {u < v}  = template,
Boolean def {u > v}  = template,
Boolean def {u <= v} = template,
Boolean def {u >= v} = template,
def {u = v}  = template,
def {u += v} = template,
def {u -= v} = template,
def {u *= v} = template,
```

```
def {u /= v} = template,
def {u %= v} = template,
def {u &= v} = template,
def {u #= v} = template,
def {u |= v} = template,
def {u ^= v} = template,
def {u <<= v} = template,
def {u >>= v} = template,
def {u ^^= v} = template,
def {++u} = template,
def {--u} = template,
def {u++} = template,
def {u--} = template;
```

Note that relational operators like "`u != v`" are defined as `Boolean`. The templates are (and must be) defined before any generic macros and before any control statements that need them.

Also note that type `unknown` is not at all the same as a `spclarg` that matches any type. The `unknown` arguments `u` and `v` in the above templates only match subtrees whose type is special type `unknown`, usually because the subtree is a generic argument of `unknown` type or because the subtree is an expression that combines `unknown` subexpressions.

Templates were created solely to support generic macros and inlines so that a large part of GalaxC can be defined using `gxclib`, and were not originally intended for general-purpose use. They have worked out well for `gxclib`. They may require extending to be of general use.

Chapter 10
Introduction to Special Functions

A **special function** or `spcl` is a novel programming technique created for Galaxy '91. All the programming constructs we have seen so far, such as functions and macros, only allow you to write programs that combine those constructs. Special functions go one step further: they allow you to define completely new constructs by telling GalaxC how to generate code for a parse tree.

For example, in Section 7.6.1 we saw how the *while-do*, *repeat-until*, and *for* statements are implemented as `gxclib` macros. However, those macros use `label` and `goto` statements, which must be implemented as `spcls` so they can create patterns in GalaxC's *pattern table* and directly generate PSI code.

The principle behind a `spcl` is very simple. An ordinary function call evaluates its arguments, pushing them onto the stack, and then jumps to the function body to process the arguments in some way. All of this is done at run time. In contrast, a `spcl` call pushes its arguments as *unevaluated parse trees* and jumps to the `spcl` body at *compile time*. The `spcl` body has access to all the GalaxC compiler's functions and global variables -- such as the pattern table -- and thus has full control over how code is generated. Since the arguments are unevaluated, they can be used to define patterns and macro or `inline` bodies.

10.1. A Brief Tour of the GalaxC Compiler

Before writing `spcls`, one must have a clear understanding of how the GalaxC compiler is constructed. This section gives a brief overview of each section of the compiler.

10.1.1 Incremental Scan and Parse (ISP)

Since the GalaxC language separates syntax from semantics, it is natural that its compiler separates syntactic analysis from semantic analysis. ISP takes GalaxC source code and converts it into a parse tree by following the token and expression rules in Chapters 2 and 3. ISP has no knowledge of semantic concepts such as variables and functions. In principle, GalaxC's parser is *incremental*, i.e., it only needs to re-parse source code that has changed since the last compilation. This was required for Galaxy '91, since it was developed on computers that were two orders of magnitude slower than current machines. In practice, GalaxC does not use incremental capabilities at this time and does not provide ISP with the source change information it needs to scan and parse incrementally. However, ISP capability is present if we ever need to change GalaxC's compiler from fast to lighting fast.

GalaxC uses a compact, in-line representation for the parse trees that represent expressions and patterns which avoids the need for pointers between parse tree nodes. For details, see file `isp.h` and [B&B 91].

10.1.2 Postfix Stack Interpreter (PSI)

GalaxC initially generates code for a simple stack machine called the Postfix Stack Interpreter, which may be interpreted or optimized and translated into a native instruction set. In Chapter 6 we described the PSI instruction set and how `gxclib` uses PSI to implement GalaxC operations like `x+y` and `x-y`.

PSI also performs a small amount of optimization -- in particular, *constant evaluation*. PSI evaluates any expression that consists entirely of numeric constants at compile time. The expression cannot contain strings, loads, stores, or function calls. While this is useful in general, it is mandatory for GalaxC constructs that require a compile-time constant such as type sizes and arrays declared using `var`. PSI has a *constant evaluation stack* (CES) for performing this function. As is usual with PSI, CES has 32-bit long values which

can be interpreted as `long`, `ulong`, or `float` by PSI instructions.

PSI includes a number of utility functions which the GalaxC compiler uses to generate code. These functions are usually called from `spcls` or functions that are called by `spcls`. Here are some of the most important ones:

`GenWord(x)`
: Generate a 16-bit word of PSI object code. See Section 6.1 for the format of PSI instructions for opcodes and data.

`GenConst(x)`
`GenBigConst(v, n)`
: `GenConst(x)` pushes the long value `x` onto CES and increments PSI variable `CESn` to keep track of how many longs are on CES.

  `GenBigConst(v, n)` pushes `n` longs starting at address `v` (declared as `@unknown`) onto CES, incrementing `CESn` by `n`.

`FlushCES`
: Flush all `CESn` longs off CES and generate PSI code to push the values on the normal stack. If they are small integers, generate 16-bit PSI instructions for them using `GenWord`. Otherwise generate a single PSI *stack constant* that pushes them as 32-bit values.

`GenOpcode(op)`
: Generate a 16-bit instruction with opcode `op`. If `op` is allowed to combine constants and CES has enough longs, PSI executes this instruction immediately using the operands at the top of CES, leaving the result at the top of CES. It may push or pop one or more longs, and always discards the generated `op` instruction.

  If `GenOpcode(op)` cannot combine CES constants it calls `FlushCES` and then generates the 16-bit `op` instruction, which is equivalent to `GenWord(op + 0x8000)`.

`GetIntConst(v)`
`GetConstValue(v, n)`
: Check `CESn` to see if CES contains at least `n` longs (`n` = `1` for `GetIntConst`). If so, pop the top `n` longs from CES storing them in reference variable `v` and return `TRUE`. Otherwise leave CES unchanged and return `FALSE`.

`GenString(pc, n)`
: The GalaxC compiler calls `GenString` twice to generate an `n`-byte PSI *string constant*. First it calls `GenString` with `pc = NULL`, which makes sure there is enough space in PSI object code for an `n`-byte string. If there is, `GenString` returns the current PSI program counter `PSIpc` as the address of the string constant, otherwise it generates a compiler error. The caller then fills in the characters of the string including its terminal `NUL` and pad byte, after which it calls `GenString` a second time with the `pc` returned from the first call. `GenString` fills in the first word of the PSI string constant and updates `PSIpc` to the address after the `n`-byte string constant. The second call is allowed to have a smaller value of `n` than the first call.

`GenPop(n)`
: Generate code to pop `n` longs off the stack, starting with any longs in CES. If `n` is negative, flush CES

and push |n| longs. Generate PSI POP, POPD, POPN, or PUSH instructions. One use of GenPop is converting a base or pointer type to its supertype void.

GenLoad(n, vol)
: Generate code to load n bytes onto the stack given a TOS address (popped). GenLoad generates opcode LOAD, LOADS, LOADB, LOADD, or LOADN. If vol, generate volatile opcode LOADV, LOADVS, LOADVB, LOADVD, or LOADVN. The most common use of GenLoad is dereferencing pointers.

GenStore(n, vol)
: Generate code to store n bytes given a TOS address (popped). Do not pop the n bytes of data. GenStore generates opcode STORE, STORS, STORB, STORD, or STORN. If vol, generate volatile opcode STORV, STORVS, STORVB, STORVD, or STORVN. The main use of GenStore is special function Assign(x:T, y), which is called by the assignment macro.

GenPopGoto(opcode, target, tsp)
: Generate a GOTO, POPGOTO, or EXCEPT instruction given target = PC to jump to, tsp = SP at target, and which opcode to generate. GalaxC uses this for compiling goto statements and exception expressions where the label is already defined.

ForwardGoto(opcode, prev)
: This is similar to GenPopGoto, except in this case the target PC and SP are not yet known. ForwardGoto generates a POPGOTO or EXCEPT instruction with incorrect target PC and SP values and makes it the end of a *forward reference chain* (FRC). Argument prev points to the previous FRC end (NULL if FRC was empty) and ForwardGoto returns the new FRC end.

ResolveForwardReferences(last)
: Resolve all forward references in the FRC that ends with last so that they point to the current PSIpc and PSIsp. GalaxC calls this when a label statement defines a label that has a list of unresolved forward references.

ResolveIfGotos(jmp1, jmp2)
: Resolve forward GOFALSE, GOTRUE, or GOTO when creating an *if-then-else* statement or expression. See Section 10.2.8.

See psi.c for source code and additional PSI utility functions.

10.1.3 Analysis and Code Generation (ACG)

ACG is the heart of the GalaxC compiler. Its job is to take syntactically-correct parse trees generated by ISP, analyze each subexpression to see if it's a variable, function/macro call, etc., and then call PSI to generate code for the kind of construct matched. It is highly recursive, but for the most part simple and modular.

In the present phase of the XXICC project, ISP, PSI, and ACG are currently written in C. This is because the first version of GalaxC had to be written in something that already had a compiler available. *[footnote: Similarly, C was actually first implemented in an interpretive language called B, which was in turn implemented in an assembler called A.]* Now that GalaxC's compiler is running, at some point we will rewrite it in itself. This is quite easy to do, because the basic computing models of GalaxC and C are very similar, and the resulting GalaxC code will probably end up looking quite a bit like the C version.

However, currently we do not have any native code generators for PSI so running a GalaxC version of the GalaxC compiler interpretively would be rather slow. So we might as well wait on translating the GalaxC compiler to GalaxC until we have good native code generators. We know it can be done: the Galaxy '91 compiler was written in itself.

GalaxC analyzes an expression `x` by calling function:

```
Analyze(x, T, caller)
```

where `x` is a of type `formula`, a built-in GalaxC type that points to a parse tree, and `caller` identifies the caller for debugging. On success, `Analyze` returns a pointer to the pattern matched by expression `x` after setting reference argument `T` to the type of expression `x`. `Analyze` returns `NULL` on failure.

`Analyze` first sees if `x` is a literal token such as a number, character, or string. If so, it returns a predefined literal pattern and sets `T` to the literal token's type (§4.2.4).

Otherwise `Analyze` performs the pattern matching process described in Section 9.1.1 where it compares `x` to each pattern that might match to see if literal syntax matches and subexpressions of `x` match argument types. If this succeeds, `Analyze` returns a pointer to the matched pattern `pat` and sets `T` to the *evaluated type* of expression `x`. This is normally the same as `pat`'s type, but if `pat` is a generic macro/`inline` or `spcl` then `T` may depend on the type of subexpression(s) within `x`, in which case `Analyze` returns the specific type of `T` determined by the pattern matching process.

`Analyze` saves the matched pattern with `x` so that if it needs to re-analyze `x` then it already knows which pattern it matches and does not have to repeat pattern matching. In some cases it, such as the body of a generic macro, it must repeat the pattern match on each analysis.

Each pattern has a `PatternStruct` which defines its syntax, type, and value for generating code. Here is the GalaxC version of `PatternStruct`:

```
typedef PatternStruct = struct(p)
{
    @PatternStruct  p.next,     // Next pattern in linked list.
    ubyte           p.kind,     // Kind of pattern.
    ubyte           p.flags,    // Flags.
    short           p.Mnum,     // Module number (used internally by GalaxC).
    PatSyntax       p.syntax,   // Pointer to pattern syntax.
    type            p.type,     // Pattern type.
    ulong           p.value,    // Kind-specific value.
    ulong           p.aux,      // Kind-specific auxiliary value.
    ushort          p.size,     // Type size or kind-specific value.
    ushort          p.depth     // Type depth or kind-specific value.
};
def Pattern = @PatternStruct;
```

The `syntax` field points to a variable-length parse tree that encodes syntax rules, literals, and arguments. Each argument has an offset to another `PatternStruct` that defines the argument's type and other properties.

Each kind of pattern uses the `value`, `aux`, `size`, and `depth` fields differently. Variables use `value` as the

stack offset of a local variable or the address of a global variable. Functions use `value` to store the code address of the function. Intrinsic `inlines` use `value` for the PSI opcode of the operator. Macros and ordinary `inline` use `value` to point to the parse tree for the macro/`inline` body. See `acg.h` for details.

GalaxC generates code by calling:

```
Compile(x, typeOnly)
```

where `x` is a parse tree of type `formula`. `Compile` calls `Analyze(x, T)` to find a pattern `pat` that matches `x`. If it fails, `Compile` produces the compiler error "Cannot find match for formula".

If `x` matches successfully, `Compile` checks `Boolean` argument `typeOnly`: if `TRUE`, just return type `T` instead of generating code. `T` is normally `pat.type`, but if `pat.type` is `unknown` then `Compile` calls `GetUnknownType(pat, x)` to compute `T`.

If `typeOnly` is `FALSE`, `Compile` generates the code that is appropriate for each kind of pattern by calling additional functions:

Local or global `var`:

Call `CompileVar(pat)` to generate PSI `STAKVAR` or `GLOBVAR` instruction to push the address of a local or global `var`. GalaxC normally follows this with `LOAD` or `STORE` instructions. Function bodies treat arguments as local variables and use `CompileVar(pat)`.

Ordinary function, intrinsic `inline`, `cdecl`, or `stdcall`:

Call `CompileFunctionCall(pat, x)` to generate a function call. Use `pat.syntax` to find where the actual argument subexpressions are in `x`, and generate code to evaluate them and push them onto the stack in left-to-right (if intrinsic with positive opcode) or right-to-left order (all others), converting subtypes to supertypes as appropriate. By "evaluate", we mean to call `Compile` recursively for each subexpression.

Once the arguments are all pushed, `CompileFunctionCall` calls PSI function `GenOpcode` to compile an intrinsic `inline`, `GenCall` to generate a normal function call, `GenCDecl` for a `cdecl`, or `GenStdCall` for a `stdcall`. The PSI functions all take care of stack housekeeping. For `cdecl` and `stdcall`, `CompileFunctionCall` calls `DLLfnptr` to get the DLL or shared object address of the target function and assigns it to `pat.value`.

Literals:

Call `CompileLiteral(x)` to convert literal token `x` into a numeric value and call `GenConst` or `GenBigConst` to push it onto CES. If `x` is a string, `CompileLiteral` calls `GenString`.

Macros, and inlines not mentioned above:

Call `CompileMacroCall(pat, x, T)` to instantiate a macro or `inline` body. For macros, `CompileMacroCall` binds subexpressions of `x` as parse subtrees to their corresponding arguments in `pat.syntax`, saving the previous bindings on a stack. For inlines, `CompileMacroCall` treats most subexpressions of `x` as function arguments. It generates code to evaluate and push them from left to right, along with any sub-to-supertype conversions. However, it treats `unknown spclargs` like macros arguments.

Next, `CompileMacroCall` compiles the macro or inline body by calling `Compile(pat.value)` recursively. For each macro argument (or `unknown spclarg`) in the body, call `Compile` recursively

to compile a copy of the subexpression of `x` bound earlier, and then perform any sub-to-supertype conversion. Inline arguments other than `unknown spclarg` are handled like ordinary function arguments, i.e., as stack variables.

When we are done compiling the body, restore any argument bindings saved earlier.

The actual code in `compile.c` is more complicated because it also handles generic arguments and variable argument lists.

Special function:

Call `CompileSpcl(pat, x)` to compile `pat` as a `spcl`. Use `pat.syntax` to find where the actual argument subexpressions are in `x`. For each `type` argument, evaluate the subexpression as a compile-time constant and push it on the stack. All other arguments must be `spclargs`: for each one, push a `formula` that points to the parse subtree for the subexpression. `CompileSpcl` does not perform any sub-to-supertype conversion: it just passes the parse subtree as an unevaluated expression. `CompileSpcl` then calls the `spcl` pointed to by `pat.value` *at compile time*. The called special function generates code and/or creates patterns, and when it is done it returns a new value of `T`.

When `Compile` is finished it returns type `T`. This is usually the type returned by `Analyze`, but may get changed by a macro or `spcl`.

In many cases, if `Compile` returns a reference type we would like to automatically dereference it to a base type. We do this by calling `Dereference(x, typeOnly)`:

```
formula arg x, Boolean arg typeOnly,
fn Dereference(x, typeOnly) =
{
    T = Compile(x, typeOnly);
    // If typeOnly, find the base type of reference type T by finding its supertype.
    if typeOnly then while T is reftype do T = supertype(T)
    // If !typeOnly, perform the sub-to-supertype conversions.
    else while T is reftype do T = ConvertToSuper(T);
    return T;
}
```

As with `Compile(x, typeOnly)`, if `typeOnly` then we just want the base type of expression `x`. Otherwise, we compile `x`, leaving the result on the stack, and then perform sub-to-supertype conversions to convert reference type `T` to its base type.

That pretty well covers how GalaxC compiles a parse tree. The details are embedded in special functions.

10.1.4 `gxclib.gal`

The GalaxC library `gxclib.gal` (normally called `gxclib`) defines defines most of GalaxC's operators and constants, as well as some of its data types. Basically, anything that doesn't require a special function is defined in `gxylib`. This includes basic arithmetic operators (§6.3.1), pointer arithmetic (§8.3.3), assignment expressions (§6.4.10), iterative control expressions (§7.6.1), as well as standard functions from the standard C library (§5.6.2). See `gxclib.gal` itself for details.

10.2. Examples of Special Functions

The easiest way to explain special functions is to look at how the most important ones are defined in GalaxC. These will be illustrated with GalaxC source code. The C versions actually in use in the current version are in `spcls.c` and other source files.

Since special functions are called at compile time, their arguments must be defined at compile time. As mentioned earlier, all arguments to `spcls` must be `types` (which are always defined at compile time, at least currently) or `spclargs`, which are unevaluated parse trees. A `spclarg` may have a type, in which case `Analyze` analyzes the subexpression associated with it to make sure the type matches, or it may be `unknown` in which case `Analyze` does not analyze it at all.

If a `spclarg` is declared without a type, it is considered to have type `unknown`. This is defined in `gxclib` with the macro:

```
unknown spclarg name,
def {spclarg name} = unknown spclarg name
```

Within the body of a `spcl`, all `spclargs` are treated as local variables of type `formula` (§10.1.3). This means that the body treats all `spclargs` as if they are `unknown`. `CompileSpcl` does not perform any dereferencing or other sub-to-supertype conversion when passing a subexpression as a `formula`. That must be done by the `spcl`'s body. *[footnote:* Macros treat all their arguments like `spclargs`: they are passed as parse subtrees and are re-compiled using `Compile(x)` wherever they appear in the macro body.*]*

10.2.1 Colon Expressions

A colon expression "`x: T`" dereferences `x` and typecasts it to type `T`. The stack sizes of dereferenced `x` and `T` must be equal. Here is its definition as a `spcl`:

```
spclarg x, type arg T,
spcl {x: T} =
{
    if typeOnly then return T;
    var Tx = Dereference(x, FALSE);
    if stacksize(Tx) != stacksize(T)
    then ACGerror("Types must be the same size");
    return T;
}
```

Argument `x` is an `unknown spclarg` and can thus be any type. The `spcl` itself is declared as type `unknown`, so `Analyze` must call it to determine the type of a colon expression "`x: T`". In addition to its declared type, each `spcl` also returns as its function value the type of whatever code it has generated (or will generate, if `typeOnly`). This is the same as its declared type τ unless τ is `unknown`, in which case it will most likely be something else.

In some cases, GalaxC sets global variable `typeOnly` before it calls a `spcl`. This means that it just wants to know the type of an expression and doesn't want the `spcl` wasting time generating code that will need to be discarded. `GetUnknownType(pat, x)` uses this feature. Global variable `typeOnly` is not the same as `Compile`'s `typeOnly` argument, though they are used similarly.

In the case of the colon `spcl`, if `typeOnly` we just return `T` as the type of the colon expression. We will

check whether T is correct later. If !typeOnly, we call Dereference(x, FALSE) which calls Compile(x, FALSE) and if necessary generates code to convert x to its base type. Dereference returns the base type of x, which we assign to Tx. Next, we check that the stack sizes of T and Tx are equal and generate a compiler error if they are not. At this point we are done: the colon operator does not change the value of x: it just tells the compiler that its type is now T by returning T as the value of the spcl.

10.2.2 Parentheses

In most languages, parentheses are merely a syntactic construct for changing operator precedence. In GalaxC, they are a little more involved since a comma expression in parentheses is compiled differently, viz., its arguments are evaluated right to left instead of left to right. This requires parentheses to be implemented as a spcl:

```
spclarg x,
spcl (x) =
    x is Cnode(COMMA_RULE)? CompileCommaRev(x): Compile(x, typeOnly)
```

The expression "x is Cnode(COMMA_RULE)" sees if parse tree x has a COMMA rule as its root. If so, we call CompileCommaRev(x) which compiles a comma expression in reverse order and returns the type of the comma expression (see next section). Otherwise, we just call Compile to compile x, with Compile's typeOnly argument set to global variable typeOnly. Compile does *not* dereference x.

10.2.3 Comma Expressions

If it is not in parentheses, a comma expression "a, b" pushes a and b onto the stack in that order, dereferencing any variables. If one of the expressions has type void, then the other determines the type of the comma expression. Otherwise the type is a quasi-type whose size is the sum of the stack sizes of the two expressions. Here is its definition as a spcl:

```
spclarg {a, b},
spcl {a, b} =
{
    if typeOnly then return unknown;
    var Ta = Dereference(a, FALSE);
    var Tb = Dereference(b, FALSE);
    return (Ta == void? Tb: Tb == void? Ta:
            Qtype(stacksize(Ta) + Stacksize(Tb))
}
```

If typeOnly, we always return unknown. This is needed for a subtle reason if a is a definition and b uses that definition. If !typeOnly, we compile expressions a and b in that order, dereferencing each to its base type. We then check the resulting types Ta and Tb and return either Ta, Tb, or a quasi-type as the type of the spcl.

10.2.4 Semicolon Expressions

A semicolon expression "a; b" evaluates multiple statements, discarding any values left on the stack by converting each statement's value to void. This is different from C where ';' is used to convert an expression to a statement, but has the same overall effect. In addition, ';' hides any arguments declared before it. The type of a semicolon expression is always void. Here is how it is defined as a spcl:

```
spclarg {a, b},
void spcl {a; b} =
{
    compile a to void;
    unlink args;
    compile b to void;
    return void;
}
```

"`compile x to T`" compiles `formula x` and then generates code to convert it to supertype `T` unless it is already `T`. This is an easy way to pop the value of expressions `a` and `b` in the above `spcl` since every type is a subtype of `void` and can be converted to `void` by popping the stack.

"`unlink args`" hides all arguments that are currently defined in the pattern table. They are not removed, since they may still be used by earlier patterns, but they will not be recognized by patterns defined after '`;`'.

10.2.5 Blocks

A block is one or more statements contained in a set of braces "`{ }`" and usually separated by semicolons. Patterns defined within a block are local to that block. The `spcl` for a block is very simple:

```
spclarg x,
void spcl {{x}}
{
    create block x;
    return void;
}
```

Note that we need a second set of braces around pattern "`{x}`", which GalaxC automatically discards. All the real work is performed by function "`create block x`" which saves the state of the pattern table, compiles `formula x`, and then restores the pattern table, discarding any locals defined with block `x`. However, if `x` has any unresolved forward labels, GalaxC retains their pattern table entries.

The last semicolon in a block does not have to be followed by a statement. `gxclib` takes care of that with a macro:

```
spclarg b,
def {b;} = (b; nop)
```

10.2.6 The Reference Operator: `&x`

The reference operator `&x` converts variable `x` of type `&τ` to a pointer of type `@τ` without dereferencing it. As we saw in Section 8.3.1, the reference operator `&v` cannot be defined as a macro. However, it is easily defined as a `spcl`:

```
spclarg x,
spcl {&x} =
{
    var Tx = Compile(x, typeOnly);
    if !Tx is reference then ACGerror("Operand must be reference type");
    return Ref2Ptr(Tx);
}
```

The function "Tx is reference" sees if type Tx is a reference type so &x can generate an error message if x is not a variable. The function Ref2Ptr converts reference type Tx to a pointer type by changing a bit in an internal GalaxC structure.

10.2.7 Assign

Here is the "Assign(x:T, y)" spcl used by assignment expressions "x = y" where x is of type &T:

```
type arg T, spclarg {x, y},
spcl Assign(x: T, y) =
{
    if typeOnly then return T;
    compile y to T; compile x to RefTo(T);
    GenStore(|T|, x is volatile); return T;
}
```

Assign has type argument T, which is also the type returned by the spcl if typeOnly. Otherwise Assign compiles y and if necessary converts it to x's value type T, leaving the result on the stack. Then it compiles x to &T, leaving the variable's address on the stack. Next it calls GenStore to create the PSI code to store y at address x, popping x off the stack but leaving y. Assign returns type T.

This is a simplified version of Assign which doesn't consider variable attributes (Appendix A).

10.2.8 If Expressions

As we discussed in Section 7.1, GalaxC has *if* expressions of the form "if a then b else c" and "a? b: c". In both cases, the *if* expression produces as its value either b or c, and the type of the *if* is the lowest common supertype of b and c. For example, in the expression "if a then x else 5" where x is an int variable, the common supertype is int and we must convert x from &int to int by dereferencing it. There is always a common supertype, though it may be type void. The "a? b: c" form cannot have a void common supertype.

GalaxC compiles *if* expressions by calling CompileIf(a, b, c, check), where a, b, and c are formulas for the condition, consequent, and alternative. If Boolean argument check is TRUE, check whether the common supertype of "a? b: c" is void. Except for the two spcls are identical:

```
spclarg {a, b, c},
spcl {if a then b else c} = CompileIf(a, b, c, FALSE),
spcl {a? b: c} = CompileIf(a, b, c, TRUE)
```

Here is a simplified version of CompileIf. For all the details, see spcls.c.

```
// Condition a, consequent b, and alternative c are formulas. a must be Boolean.
// If check, make sure the common supertype is not void. This is used for a? b: c.
formula arg {a, b, c}, Boolean arg check,
fn CompileIf(a, b, c, check) =
{   // Compile condition a, dereferencing to its base type, and check that it is Boolean.
    var Ta = Dereference(a, FALSE);
    if Ta != Boolean then ACGerror("Condition must be Boolean");
    // If a is known at compile time, just compile b or c. This implements conditional compilation.
```

```
        // The other clause is neither analyzed nor compiled.
        long var va;            // Place to store the value of a.
        if GetIntConst(va)
        then return (va? Compile(b, typeOnly): Compile(c, typeOnly));
        // a is not known at compile time: find the common supertype of b and c.
        var Tsup = CommonSupertype(Compile(b, TRUE), Compile(c, TRUE));
        // An expression a? b: c must have a non-void common supertype.
        if check and Tsup == void
        then ACGerror("Alternatives must have a common supertype");
        // If typeOnly, return Tsup.
        if typeOnly then
        {
            Discard any code generated for evaluating a;
            return Tsup;
        };
        // Generate conditional code using GOFALSE and GOTO.  a is already on the stack.
        var jmp1 = GetPC;                       // jmp1 = current PSIpc.
        GenOpcodeX(GOFALSE, 0);                 // If a is FALSE, go to alternative.
        var sp = GetSP;                         // Consequent: get current PSIsp.
        CompileToType(b, Tsup);                 //   Generate code for b.
        jmp2 = ResolveIfGotos(jmp1, 0);         //   Go to end of if; resolve jmp1.
        SetSP(sp);                              // Alternative: restore PSIsp.
        CompileToType(c, Tsup);                 //   Generate code for c.
        ResolveIfGotos(jmp1, jmp2);             // End of if: resolve jmp2.
        return Tsup;
    }
```

`CompileIf` first compiles condition `a`, dereferencing because `a` might be a `Boolean` variable rather than a value. If the value is known at compile time, `CompileIf` performs conditional compilation (§7.1.1) and only compiles `b` or `c`, leaving the other one unanalyzed.

Otherwise `CompileIf` calls `Compile(b, TRUE)` and `Compile(c, TRUE)` to find the types of `formulas` `b` and `c`, and finds their common supertype `Tsup` by calling `CommonSuperType`. If we are compiling "`a? b: c`", the common supertype must not be `void`.

If `typeOnly`, `CompileIf` discards the code generated for `a` and returns `Tsup`. Unlike most `spcls`, `CompileIf` has to do quite a bit of work to figure out what type to return for `typeOnly`. However, this is not a performance issue in practice for various reasons.

The rest of `CompileIf` generates code of the form:

```
            a               // Test condition a, leaving a Boolean result on the stack.
    jmp1:   GOFALSE lab1    // If a is FALSE, go to lab1.
            b               // Evaluate consequent b and convert to Tsup.
    jmp2:   GOTO lab2       // Go to lab2.
    lab1:   c               // Evaluate alternative c and convert to Tsup.
    lab2:                   // End of if expression.
```

`CompileIf` calls PSI function `GenOpcodeX(GOFALSE, 0)` to create a `GOFALSE` at `jmp1` and calls `ResolveIfGotos` to resolve the `GOFALSE` at `jmp1` and the `GOTO` at `jmp2`.

`ResolveIfGotos` optimizes the `GOTO` structure if `b` or `c` is a `nop`. For example, the statement "`if a then b`" is equivalent to "`if a then b else nop`". In this case `c` is empty and `ResolveIfGotos` generates simpler code by removing `GOTO lab2`:

```
        a               // Test condition a, leaving a Boolean result on the stack.
jmp1:   GOFALSE lab1    // If a is FALSE, go to lab1.
        b               // Evaluate consequent b and convert to Tsup.
lab1:                   // Evaluate alternative c = nop.
lab2:                   // End of if expression.
```

If `a` is a complicated logical expression, it is sometimes clearer to write "`if a then nop else c`". In this case `b` is empty and `ResolveIfGotos` generates simpler code by changing `GOFALSE` to `GOTRUE` and removing `GOTO lab2`:

```
        a               // Test condition a, leaving a Boolean result on the stack.
jmp1:   GOTRUE lab2     // If a is TRUE, go to lab2.
lab1:   c               // Evaluate alternative c and convert to Tsup.
lab2:                   // End of if expression.
```

In both of the simplified forms, `Tsup` is `void` because `nop` has type `void`.

10.2.9 Declaring Variables and Arguments

We saw how to declare variables and arguments using various syntax forms in Sections 4.4 and 5.1. Declaring them always requires special functions because we need unevaluated parse trees for the the variables' and arguments' patterns. The `spcls` for the various forms are very similar:

```
type arg T, spclarg {name, init},
void spcl {T var name} = DefineArgVar(LOCAL_KIND, T, name, NULLformula),
void spcl {T var name = init} = DefineArgVar(LOCAL_KIND, T, name, init),

void spcl {T arg name} =
    DefineArgVar(T == type? TYPEARG_KIND: ARG_KIND, T, name, NULLformula),
void spcl {T spclarg name} =
    DefineArgVar(SPCLARG_KIND, T, name, NULLformula)
```

All of these `spcls` call `DefineArgVar(kind, T, name, init)` which creates one or more `kind` patterns from `formula` argument `name` by calling ACG function `CreatePattern`. `T` is the type of the variable or argument, and may be `unknown` for initialized variables or arguments. "`name`" defines one or more variable or argument names, the latter in braces. Each name may be prefixed with '`@`', '`&`', or '`*`' to indicate pointer and reference types, and each variable name may have an array dimension "`[n]`" or initialization "`= value`". Alternatively, `name` may have a common initialization `formula init` unless it is `NULLformula`. Arguments always have `init` = `NULLformula`. `DefineArgVar` generates `init` code using `Compile`.

The `kind` argument is is set to `LOCAL_KIND` for `var` definitions; `DefineArgVar` turns it into `GLOBAL_KIND` if `var` is at the top level of the program. For arguments, `kind` is `SPCLARG_KIND` for special arguments, `TYPEARG_KIND` for type arguments, and `ARG_KIND` for everything else.

10.2.10 Defining Functions and Macros

The `spcls` for defining functions and macros are also very similar to each other:

```
type arg T, spclarg {pattern, body},
void spcl {T fn pattern = body} =
    DefineFnMac(FN_KIND, T, pattern, body),
void spcl {T def pattern = body} =
    DefineFnMac(DEF_KIND, T, pattern, body),
void spcl {T inline pattern = body} =
    DefineFnMac(INLINE_KIND, T, pattern, body),
void spcl {T spcl pattern = body} =
    DefineFnMac(SPCL_KIND, T, pattern, body),

int spclarg opcode,
void spcl {T inline pattern = intrinsic opcode} =
    DefineFnMac(INLINE_KIND, T, pattern, opcode),
void spcl {T inline pattern = cdecl} =
    DefineFnMac(CDECL_KIND, T, pattern, NULLformula),
void spcl {T inline pattern = stdcall} =
    DefineFnMac(STDCALL_KIND, T, pattern, NULLformula)
```

`DefineFnMac(kind, T, pattern, body)` creates a `kind` pattern from `formula` argument `pattern` by calling ACG function `CreatePattern`. `DefineFnMac` then calls `InsertPatternArgs` to insert any visible arguments into the pattern, and calls `Compile(body)` to analyze `formula` argument `body`. The details of this depend on `kind`: for functions, `Compile(body)` generates a block of code that is called by each function call. For macros and `inlines`, `Compile(body)` makes sure that `body` is semantically correct and then saves a copy of `body` as a parse tree to be instantiated by each macro or `inline` call. The `intrinsic`, `cdecl`, and `stdcall` `inlines` do not have bodies. For an `intrinsic`, `DefineFnMac` just checks that `opcode` is a compile-time constant `int`. For `cdecl` and `stdcall`, `DefineFnMac` makes sure that `pattern` is in the form of a C function call `name(args)` so that it can use `name` to find the function in a shared object library or DLL. Note that the `intrinsic`, `cdecl`, and `stdcall` forms of `inline` are defined after the generic form so that they are matched first.

`T` is the type of the function or macro. If `T` is `unknown`, `DefineFnMac` set the pattern type to the type returned by `Compile(body)`. Otherwise, `DefineFnMac` makes sure `T` is consistent with the `Compile(body)`.

10.3. Summary

This chapter is only intended to give a brief overview of special functions and how GalaxC generates code so that the interested reader has good starting point for examining the GalaxC's source code. At the present time `spcls` must be added at the C level and as such are not quite ready for general use.

Appendix A

Implementation of Variable Attributes

This appendix details how GalaxC implements variable attributes (vattrs). Vattrs `const`, `stable`, and `volatile` are *type* variable attributes (TVAs, pronounced *Tee'-vuz*) and are associated with a variable's type. They can also apply to the types of arguments and structure fields. Vattrs `private`, `public`, and `extern` are *storage* variable attributes and are only associated with variables and structure fields.

TVAs constrain how programs can use variables and how GalaxC generates code. Here are the three TVAs, first introduced in §4.4.4:

`const`

> A `const` variable is defined with an initial value and cannot be assigned a new value. An assignment "`x = y`" where `x` is `const` causes a compiler error.

`volatile`

> A `volatile` variable may change value at any time because the variable is a peripheral device register or it may be changed by another task such as an interrupt service routine. A `volatile` variable must be loaded from or stored to memory on each access instead of being copied to a register by an optimizing compiler.

`stable`

> By default, all variables are `stable` (non-volatile). The compiler may optimize access to a `stable` variable by eliminating loads and stores that it considers unnecessary. However, this only works correctly if the compiler can assume the variable does not change value spontaneously, so treating a `volatile` variable as `stable` may cause errors.
>
> On the other hand, it is always possible to treat a `stable` variable as if it were `volatile`: all it costs is extra loads and stores.

In C, vattrs are reserved words. In GalaxC, they are ordinary identifiers.

TVAs are normally defined in variable declarations, e.g.,

```
const var x = 5
```

declares `const int` variable `x` with value 5. The actual type of `x` is `&(const int)`, i.e., `x` is the address of an `int` which cannot be assigned a new value. TVA `const` changes type `int` into `const int`.

Since every type has a TVA -- by default `stable` -- a value by itself has a TVA as well. For example, "`42`" has type `stable int`. However, GalaxC ignores the TVA of a value since TVA only matters when loading or storing a variable.

An array or pointer has a meaningful TVA. For example, a string literal has type `@(const char)` rather than `@char` which is `@(stable char)`. This prevents accidentally overwriting a string literal's value, i.e., "clobbering the constant".

A pointer or reference variable may have several levels of TVA. For example, a variable that is a pointer to an `@int` pointer has type `&@@int`, and each level may have a TVA. In most cases only the deepest TVA is `const` or `volatile`.

A.1. Special Functions and TVAs

Automatic dereferencing and special functions implement some TVA behavior explicitly:

1. GalaxC loads the values of variables using automatic dereferencing in `ConvertToSuper(types.c)` which in turn calls `GenLoad(psi.c)`. GalaxC loads `stable` and `const` variables using `LOADx`. It loads `volatile` variables using `LOADVx`. PSI-CO can optimize them differently.

2. GalaxC implements assignment by calling special function `Assign(x:T, y)` (`spcls.c`). `Assign` sees if its `x` argument is `const` and if so generates an error. `Assign` calls `GenStore(psi.c)` which generates `STORx` for `stable` and `STORVx` for `volatile`.

   If `x` is a pointer or reference variable, `Assign` makes sure `y` has compatible TVAs. That is, `x` must be at least as restrictive as `y` or else `x` could improperly access data pointed to by `y`.

   `Assign` leaves a value of type `T` on the stack, where `&T` is the type of argument `x`. However, `T`'s vattrs are from `y`: this allows `x` to be more restrictive than `y` without restricting the `Assign`'s result.

3. GalaxC implements pre- and post-increment operators like "`x += 3`" and "`--y`" by calling special function `Increment(spcls.c)`. `Increment` sees if its argument is `const` and if so generates an error. `Increment` generates `PRINCx` and `POINCx` for `stable`, and `PRINVx` and `POINVx` for `volatile`.

A.2. Function Arguments and TVAs

A function body treats function arguments as surrogate local variables. Each local has the same TVAs as its associated argument. Function argument TVAs restrict how the arguments are treated within the body. For example, a `const` argument is initialized once (when the function is called) and cannot be assigned a new value in the body. TVAs also restrict what can be passed to the function as reference and pointer arguments.

When matching patterns, GalaxC ignores TVAs. That is, you cannot have one version of a function with a `stable` argument and another version with the same pattern except for a `volatile` argument. However, GalaxC does check that variables and pointers passed as actual arguments are consistent with the function arguments' TVAs. Here are the rules:

1. A value -- not a variable or pointer -- can be passed to any function argument. The value is pushed onto the stack and the function treats that memory location as a variable or optimizes it into a register.

2. A variable or pointer with a given TVA can be passed to a reference or pointer argument with the same TVA.

3. A `const` variable or pointer must not change value, so it can only be passed to a `const` reference or pointer argument. Otherwise the called function could change the `const`.

4. A `volatile` variable or pointer to `volatile` data can only be passed to a `volatile` reference or pointer argument. Otherwise the called function might optimize out loads and stores and miss the fact the variable is changing value.

5. A `stable` variable or pointer to `stable` data can be passed to any function argument. The caller

doesn't care if the function body restricts its use of the variable. For example, you can pass a `stable` variable to a `volatile` argument. The function may make unnecessary loads and stores, but that doesn't affect results.

If there are conflicts, you can always typecast the caller's variable as long as you are careful about the implications of doing so.

We have stated these rules for a single TVA level. They generalize to multiple TVA levels.

A.3. Macro Arguments and TVAs

Function arguments are fussy about TVAs because a function compiles into a single version of its body. In constrast, since GalaxC instantiates a macro body each time it is called, GalaxC has the flexibility to generate code differently for each call according to the actual arguments' TVAs.

Macro arguments may have TVAs, but GalaxC replaces them with the actual arguments' TVAs when instantiating the macro. This allows any TVA to be passed to a macro, though macro instantiations may be different:

- A `volatile` actual argument may generate additional loads and stores.

- GalaxC generates an error if the instantiation tries to do something forbidden, like trying to change the value of a `const` or pass a variable with the wrong TVA to a function called within the macro.

When defining a macro, GalaxC does a "test compile" to ensure the macro body is semantically correct. The test compile uses macro arguments' TVAs. Type arguments cannot have TVAs.

A.4. Inline Function Arguments and TVAs

An `inline` function with a body expression is handled like a macro. Since the body is instantiated on each call, the instantiation can be different depending on TVA combinations.

A `inline` function without a body expression, e.g., `intrinsic`, `cdecl` and `stdcall` requires the actual arguments to match like a function call. This is rarely important for `intrinsics`, since they seldom have reference or pointer arguments. On the other hand, `cdecl` and `stdcall` generally call C library functions and it's important that the TVAs are consistent.

A.5. TVA Declarations

As described in §4.4, the standard form of a variable declaration is:

`τ var` *name*

You can add a TVA prefix to τ:

TVA `τ var` *name*

The TVA modifies type τ into type "TVA τ". Multiple TVAs are not generally useful: GalaxC simply uses the leftmost one, e.g., "`volatile const int`" is the same as "`volatile int`". Early versions of GalaxC may consider multiple TVAs to be an error.

Here are some variable declaration examples:

```
int var x                       x is &int = &(stable int).
const int var y                 y is &(const int).
volatile int var z              z is &(volatile int).
int var @p                      p is &@int = &(stable @(stable int)).
volatile int var @q             q is &@(volatile int).
@(volatile int) var q           q is &@(volatile int).
(volatile @int) var r           r is &(volatile @int).
```

Note that a variable declaration defines the type of *name* to be &τ so that it is automatically dereferenced. The TVA is nested within '&' as shown with variables x, y, and z. The TVA modifies type τ.

Declaring a pointer variable is more interesting. 'p' is @int, plus '&' for automatic dereferencing. Each reference or pointer is stable by default, so "&@int" is really "&(stable @(stable int))". If you declare a variable to be a pointer by prefixing its name with '@' as in the first declaration of q, then const, volatile, or stable is nested within '@' and applies to the τ value rather than the @τ pointer. Alternatively, you can include '@' with the type, which lets you control whether the TVA applies to the value (as in the second declaration of q) or the pointer (as in the declaration of r). Normally you want the TVA associated with the value as with either declaration of q.

You can use a simplified form of var declaration if you assign an initial value:

```
const var name = init
volatile var name = init
stable var name = init
```

In this case the type τ is deduced from the *init* expression but its TVA is set by the prefix.

Argument declarations are similar to var declarations. For function arguments, the TVAs apply to the surrogate local variables within the function. For macros and inlines with bodies, GalaxC uses the arguments' TVAs when test-compiling the macro or inline body.

You can also use TVAs for field definitions within a struct. Top-level field TVAs are only important if you access a struct variable, i.e., through a reference or pointer. Top-level TVAs are ignored for a struct value.

We have seen TVAs used as prefixes to modify types. You can also use them to typecast a variable or pointer, for example if you need to pass a const string to a stable string argument. Given a variable or pointer v, you can change it to a different TVA using the notation:

```
const v
volatile v
stable v
```

This changes the deepest TVA. If you want something different, use a typecast "*expr*: τ". You should be very careful changing TVAs because they are usually present for a good reason. In many cases, you are better off assigning the value to a variable with a different TVA. For example, if you want to treat a volatile variable as stable, you should copy to volatile value to a new variable as a "snapshot".

A.6. Generic TVA Macros

A function always returns the same type. This can be a bother when working with `strings`. For example, here is a function that finds the first occurrence of `char ch` in `string s`:

```
char arg ch, string arg s,
fn find ch in s =
{
    char var x;
    while (x = *s++) != '\0' and x != ch do nop;
    return (s-1);
}
```

Since `s` is a `string` = `@(const char)`, we can pass a writable `wstring` = `@char`. The caller doesn't care if the function restricts its own use of `s`. However, since "`find ch in s`" always returns a `string` the caller cannot write new characters at the returned address unless the caller explicitly changes `string` to `wstring`. For example, here is some code which copies the string "`Pet the cat`" to a `char` buffer and then replaces the word "`cat`" with "`dog`":

```
char var buf[200];
buf = copy "Pet the cat";               // Copy "Pet the cat" to buf.
find 'c' in buf = copy "dog";           // This does not work.
stable (find 'c' in buf) = copy "dog";  // This does work.
```

The third line fails because function "`find ch in s`" returns `string` and we cannot `copy` to a `string` because it is `const`. The fourth line typecasts "`find ch in s`" to `wstring` using the `stable` operator. Now `copy` works.

`gxclib` uses a GalaxC generic macro to work around this problem:

```
char arg ch, string arg s,
def find ch in s = vattr(s) (find ch in s)
```

If you call macro "`find ch in s`" with a `string` value for `s`, the macro returns type `string`. If you call it with a `wstring` value, the macro returns type `wstring`. With this macro, the third line in the example works correctly. Note that we use the same syntax for both the macro and the function. Since macros are not recursive, the body matches the function and there is no ambiguity.

This form of generic macro does not work all the time in version 0.0c. In version 0.0c, if a macro's type is based on a type argument -- e.g., `T` or `@T` -- GalaxC recomputes the type of the expression each time it encounters it. This is used for generic macros like `p + n`, where `p` is a pointer `@T`. However, if the macro's type is anything else -- e.g., `string` -- in some cases it may use the defined type (`string`) instead of recomputing the macro to get a modified type (`wstring`). For example, this expression does not work:

```
find 'c' in buf + 1 = copy "dog"
```

This is because `p + n` binds its hidden type argument `T` to `const char` instead of recomputing the type of "`find ch in s`" to get `char`.

Fixing this is a little tricky so we've decided to fix it in a future version, most likely 0.0d. For now, callers

may sometimes need to specify vattr explicitly.

A.7. Internal TVA Representation

GalaxC represents type τ's TVAs as an *Attribute Bit Vector* (ABV) of the form:

r	1	a_2	c_2	v_2	a_1	c_1	v_1	a_0	c_0	v_0

This allows up to three reference levels, each with its own TVA. The *r* bit is reserved for possible future implementation of `register`. Each a_i is an *automatic dereferencing flag* which indicates whether level *i* is a reference (a_i = 1) or a pointer (a_i = 0). Each c_i and v_i indicate whether level *i* is `const` and/or `volatile`.

Except for *r*, an ABV is variable-length, with *start bit* '`1`' preceding the left-most a_i or cv_i. Here are the other possible ABVs:

r	0	1	c_2	v_2	a_1	c_1	v_1	a_0	c_0	v_0
r	0	0	0	1	a_1	c_1	v_1	a_0	c_0	v_0
r	0	0	0	0	1	c_1	v_1	a_0	c_0	v_0
r	0	0	0	0	0	0	1	a_0	c_0	v_0
r	0	0	0	0	0	0	0	1	c_0	v_0
r	0	0	0	0	0	0	0	0	0	1

ABVs for variables, arguments, and fields must begin with an *a* bit, i.e., they are 0, 3, 6, or 9 bits long, plus a start bit. Type ABVs can be any of the above lengths.

`gxclib` defines `stable`, `volatile`, and `const` as constant values of type `vattr`, which is defined as a subtype of `long`. `vattr` constants include the above ABV representation. Here is how they are defined:

```
typedef vattr = subtype(long);
def stable   = 0x11: vattr,     // vattr = 1.00.01
def volatile = 0x15: vattr,     // vattr = 1.01.01
def const    = 0x19: vattr;     // vattr = 1.10.01
```

A.8. Storage Vattrs

In addition to TVAs, there are *storage vattrs* (SVAs) such as `private`, `public`, and `extern`. Details are TBA, but here are some notes:

1. Like `stable`, `volatile`, and `const`, SVAs are identifiers rather than reserved words.

2. We can represent SVAs by prefixing additional bits to the front of the ABV described in the last section.

3. We need a vattr concatenation operator for expressions like `private const`. The operator must produce a constant expression at compile time, so we need a new PSI `CATVA` opcode like `SIZEOF`, `REFPTR` and `TYPEVA`.

4. The additional SVA bits should only be used for declarations. The colon operator "*x*`:` τ" should not allow τ to have any SVAs.

A.9. `register` Variables

The C programming language has the `register` variable attribute, which encourages the compiler to implement a local variable as a register. A `register` variable is read/write so you can assign a new value with using "`x = y`" where x is a `register`. However, on most computer architectures a register does not have an address so you cannot point to it, and trying to do so -- e.g., by writing "`&x`" or passing `x` to a reference argument -- causes a compiler error. Other than this, a `register` acts like a `stable` variable.

We have decided not to include `register` in GalaxC. The main reason for this is that the author has never actually used `register` in the large amount of C he has written. Modern compilers generally do a fine job of assigning variables to registers, and don't need the `register` attribute to help them. If the author needs to pass a local variable by reference, he generally has a good reason for doing so -- e.g., returning multiple values from a function. Declaring every other local variable `register` would only clutter up the code. The author has rarely found that he accidentally passed a local variable by reference without meaning to, so the compiler check for `register` would have rarely been useful.

We also found that implementing `register` would add a large amount of complexity to GalaxC's compiler for a feature that is seldom used. The Reduced Software Complexity philosophy thus rules it out.

In case we decide to include `register` in a future version of GalaxC, here are some important considerations.

1. Global variables are never `register`.

2. A `register` variable has a reference type `&τ`. This seems counter-intuitive, but is necessary because without a level of referencing there is no way to assign a value to a `register x`. When generating PSI code, GalaxC treats `register` variables as stack variables, so the address of `x` is defined. If a native code generator assigns the `register` variable to a CPU register, it generates code that assigns values directly to registers instead of using `STORE` instructions. While you can assign to `x` using special function `Assign`, you cannot change its type to `@τ` using the reference operator `&x`.

3. PSI implements `registers` on the stack as if they were normal variables. If PSI-CO detects that a variable's address is never used in an expression or passed to a function then it can assign it to a register. GalaxC ensures that `register` variables have this property.

4. GalaxC implements reference operator `&x` as special function `RefOp(spcls.c)`. `RefOp` would need to see if its argument is `register` and if so generate an error.

5. When calling a function, a `register` variable cannot be passed to a reference argument. This requires passing the address of the `register`, which does not exist. We would need to check actual arguments and see if they are `register`.

6. It is OK to pass a `register` variable to a macro argument, but we would need to check for proper `register` use in the macro body.

7. It is OK to pass a `register` variable to an `inline` function. While a register cannot be passed to function's reference argument, it can be passed to an `inline`'s reference argument since a code optimizer can combine the `inline`'s argument with the caller's `register`.

8. `register` variable declarations are a bit different from the examples in A.5. For example,

```
register int var w              w is &(register int).
register int var @s             s is &(register @int).
```

   Since you cannot point to a `register`, it never makes sense to have `&@(register τ)` and you cannot have `&&(register τ)`. Instead, `register` is at the top level of '`&`'.

9. You cannot have a `register` field in a `struct`.

10. The conditional expression "*a*`?` *b*`:` *c*" has some interesting `register` issues. Normally, "*a*`?` *b*`:` *c*" can be on the left side of an assignment or passed as a reference argument. However, if *b* and/or *c* is a `register` variable, then "*a*`?` *b*`:` *c*" is not a valid reference expression because neither *b* nor *c* has an address. Instead, we must dereference *b* and/or *c*, which then means we cannot assign to it or pass it as a reference argument. For assignments, a clever compiler can rewrite "*a*`?` *b*`:` *c* `=` *d* " as "`if` *a* `then` *b* `=` *d* `else` *c* `=` *d* " in which case `register` *b* and/or *c* can be assigned to. However, this creates ACG complexity for a feature that the author has never used. In the author's opinion, it's better to omit the `register` attribute and let the PSI-CO work out these details.

11. Future popular computer architectures may allow registers to have addresses, which would make `register` restrictions counter-productive.

At some point GalaxC may implement *restricted* variables, which support a compiler optimization.

Bibliography

[B&B 89] Anne F. Beetem and John F. Beetem, "Introduction to the Galaxy Language", *IEEE Software*, pp. 55-62, May 1989.

[B&B 91] Anne F. Beetem and John F. Beetem, "Incremental Scanning and Parsing with Galaxy," *IEEE Transactions on Software Engineering*, vol. 17, no. 7, pp. 641-651, 1991.

[BBP 92] Anne F. Beetem, John F. Beetem, and Jong-Min Park, "Orthophrase Extensibility in Galaxy," *Information and Software Technology*, vol. 34, no. 5, pp. 333-340, May 1992.

[DEC 73] Digital Equipment Corporation, *PDP-11 Peripherals Handbook*, 1973.

[JFB 91] John F. Beetem, *The Galaxy Programming Language*, A. B. Creative Consulting, 1991.

[JFB 92] John F. Beetem, *Galaxy CAD User Manual*, A. B. Creative Consulting, 1992.

[JFB 11] John F. Beetem, *The XXICC Anthology*, 2011.

[K&R 78] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall, Inc. 1978.