# A Dataflow Compiler Substrate

Computation Structures Group Memo 261

Version of March 24, 1986

Ken Traub

This research was supported in part by Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract no. N00014-75-C-0661. Ken Traub is supported by a fellowship from the National Science Foundation.

# Table of Contents

# List of Figures

# 1. Introduction

**WARNING: This document is preliminary and subject to change.**

This document describes the data structures and abstractions that underlie the ID Compiler, Version 2, a compiler from the programming language ID to machine code for the MIT Tagged-Token Dataflow Architecture. The most important attribute of Version 2 is flexibility, as it must be adaptable to changes in the language, changes in compilation strategies, and changes in the target machine architecture. Furthermore, it must be capable of being used in a tinker-toy fashion: the parser output may feed a back end for a non-dataflow implementation, the intermediate graphs may be processed by a user application and then fed back into the compiler for machine code generation, and so forth.

This need for flexibility has led to a design that is as independent from the language and the dataflow machine as possible, so that the substrate will be immune to all but the most radical changes to these. Beyond this adaptability to the changing needs of the TTDA project, this independence results in the additional benefit that the compiler substrate can be used not only for an ID to TTDA compiler, but also for a VIMVAL to static dataflow architecture compiler, a SISAL to Manchester architecture compiler, *etc.* It is even conceivable that given an ID to TTDA compiler and a VIMVAL to static dataflow architecture compiler, both built upon the abstractions described herein, it would be possible to construct ID to static and VIMVAL to TTDA compilers with a relatively small amount of additional code.

The substrate described here is designed for compilers whose overall structure is as shown in Figure 1-1. The compiler is a collection of modules, each of which operates on an intermediate representation of the program being compiled. The intermediate representations, which are fully described here, serve as the only channel of communication between the modules. A simple top-level procedure supervises the passing of control from module to module.

---

**Design Note:** At this point, it is not clear whether the entire source code will move from one module to the next or whether smaller units, such as procedure definitions, will successively move through the compiler. While no stand is being taken at present, certain conventions may be introduced in the future.

---

Referring to the figure, in the first phase of compilation the source code is parsed, resulting in a parse tree. Initially, the parse tree is just a hierarchical representation of the source code, with no other information or annotations beyond some indications of where in the input file (line number, character position, *etc.*) each construct appeared. The parse tree then undergoes a series of transformations, each of which may alter the original parse tree in several ways. A module may add, delete, or replace nodes of the tree, the effect being equivalent to a source-to-source transformation. A prime example of this kind of parse tree modification is "desugaring", in which a program construct is replaced by a semantically equivalent construct. A module may also *annotate* the parse tree by adding information to nodes of the tree. These annotations do not affect the structure of the program, but may affect how later phases interpret constructs found in the tree. An annotation might be employed by a type checking module, for example, to indicate that a construct results in a particular type of data. Finally, a module may introduce new nodes into the parse tree which could have not been produced by the parser. This might be done, for example, to replace an overloaded construct by one of two non-overloaded constructs which later phases can deal with separately.

While any kind of parser may be used to produce the parse tree, it is expected that the parser used for Id/83s will be a DFA lexical analyzer followed by a LALR parser, with a stream of lexical tokens connecting the two. As this model is applicable to a wide variety of languages, an abstraction for lexical tokens is described here. If some other type of parser is used, the lexical token stream may be absent entirely.

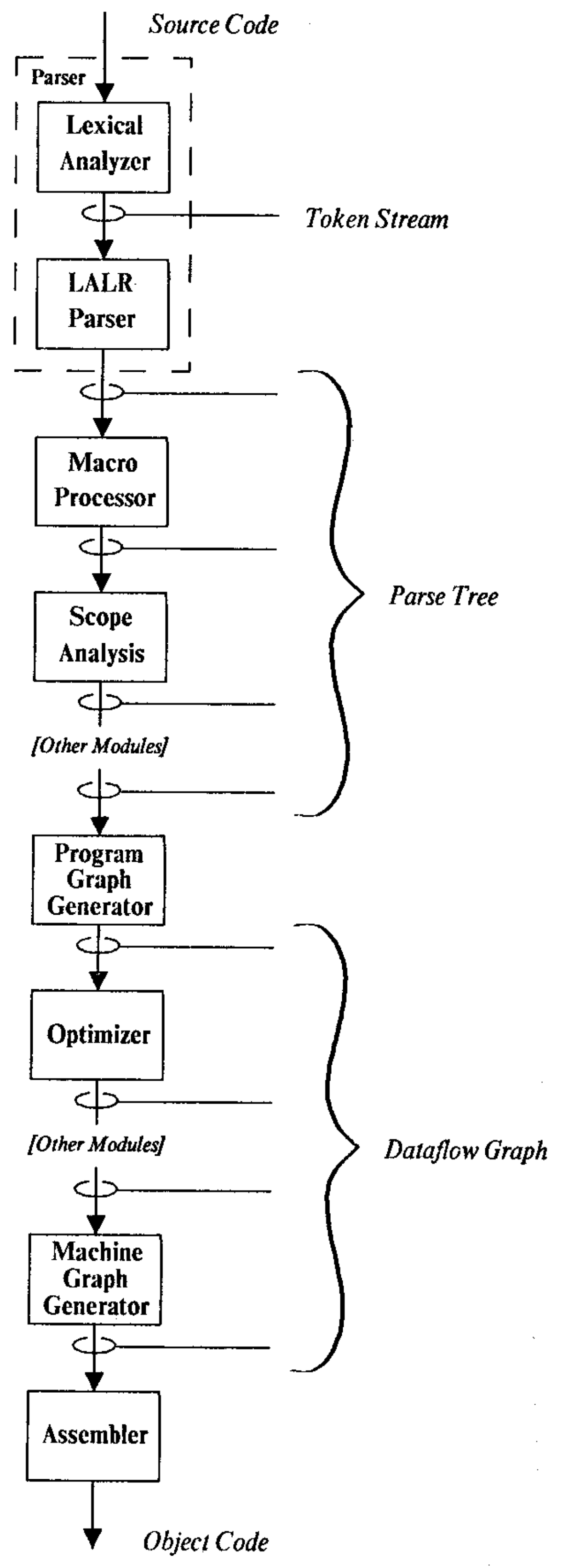


**Figure 1-1:** Block Diagram of a Dataflow Compiler

When all transformations to the parse tree have been completed, the program is converted into a dataflow graph. The initial form that this dataflow graph takes is called the *program graph*, because the level of detail present in this graph is roughly the same as that found in the source program. For example, a procedure application might appear as a single APPLY instruction in the program graph, even though this may later be expanded into a whole collection of dataflow instructions depending on the implementation of procedure linkage.[1] Following the initial transformation to program graph there may be several modules which transform the program graph, such as optimizers, *etc.* At some point, the a transformation takes place which replaces large, machine-independent instructions such as APPLY and LOOP with the machine instructions actually necessary to implement these schemata. The resulting graph, called the *machine graph*, contains only instructions executable by the dataflow machine. Additional optimization phases may follow machine graph generation, and the result is finally fed to an assembly phase, which assigns addresses to the machine graph and produces output in a form understandable by the various dataflow implementations.

As described above, there are really two kinds of dataflow graphs used in the compiler: the program graph and the machine graph. The advantage of using these two forms is that many if not all of the optimizations performed at the graph level can be performed upon the program graph, which is fairly independent of the details of the target dataflow machine. As a result, changing the dataflow machine (altering the instruction set, changing restrictions on the number of destinations, redefining the procedure linkage mechanism, *etc.*) will require few if any modifications to the bulk of the graph manipulation phases. The program graph is also likely to have fewer instructions in it than the corresponding machine graph, and so optimizations may be faster. It is important to realize, however, that program graphs are not entirely independent of implementation details; for example, data-driven and demand-driven (a la Pingali) program graphs for the same program will be quite different.

Although there are conceptually two kinds of graphs, both program graphs and machine graphs are built on the same abstractions. Hence, this document only describes one kind of graph data structure, called a dataflow graph. The distinction between program graph and machine graph, then, is not one built into the compiler substrate, but is enforced as a convention by the compiler modules that manipulate graphs.

Again, it is emphasized that this document only describes the substrate of the Id Compiler, Version 2, and that this substrate is applicable to a wide variety of compilers and compiler related programs. Details of the programming language Id, what constitute legal parse trees and legal program graphs for Id, what constitutes legal machine graphs for the Tagged-Token Dataflow Architecture, and how Tagged-Token machine code is represented are described in another document.

[1]In the past, the program graph has also been referred to as the "abstract graph".

# 2. General Issues

## 2.1. Common Lisp

The compiler substrate and all compilers built on top of it are written in Common Lisp. Every attempt has been made to make the compiler conform to the specifications and conventions defined in Steele's *Common LISP: The Language*[2], hereafter referred to as "The Common Lisp Manual". The compiler substrate and any compilers built on top of it are to depend only on language features found in this manual, as far as possible. Furthermore, programs should conform as much as possible to coding conventions and practices described in the manual. These include, but are not limited to, conventions for naming symbols (pages 24-25), for indicating comments (page 348), for naming predicates (page 71), and for indicating `nil` (page 4). The compiler writer should strive for consistency with Common Lisp and with the compiler substrate in naming and choosing arguments for functions.

It is assumed that the reader of this document is intimately familiar with all the material in the Common Lisp Manual.

## 2.2. Packages

The code that makes up the compiler substrate is found in the `id-compiler-v2` package, and the symbols described in this document are all in that package (descriptions in this document do not include the `id-compiler` package qualifier as it is understood that all symbols described here that are not a part of the Common Lisp system are in the `id-compiler` package). This package has nicknames `id`, `id-compiler`, and `idv2`.

---

Deficiency: In the very short term. the nicknames `id` and `id-compiler` may not exist to avoid conflicts with the existing Id Compiler (Version 1). This will no longer apply when the old compiler's package has been renamed to `id-compiler-v1`.

---

In addition, the symbols described in this document are exactly the external symbols of the `id-compiler` package. It is intended that compiler modules will exist in other packages which use the `id-compiler` package[3], allowing modules to refer to compiler substrate functions without package qualifiers, yet preventing modules from conflicting with each other and with internal functions of the substrate. For the benefit of users developing code, a `id-compiler-user` package is provided which is just a package that uses the `id-compiler` and the `lisp` package, with nothing in it initially.

## 2.3. Defining Compilers

[This section will describe a facility for the automatic composition of compilers from component modules. The facility will take care of managing which modules operate on an entire program and which operate on a piece (*e.g.*, procedure definition) at a time, compiler version numbers, modules interdependencies, *etc.* The goal is to make it easy to splice experimental modules into the compiler, handle

[2]Steele, Guy, *Common LISP: The Language.* Digital Press, Burlington, Massachusetts, 1984.

[3]The word "use" in this context is defined in Chapter 11 of the Common Lisp Manual.

compiler options, and the like.]

# 3. Data Structures

## 3.1. Lexical Tokens and Parse Trees

### 3.1.1. Lexical Tokens

As discussed in Chapter 1, lexical tokens may or may not be used in a compiler implementation depending on the parser chosen. An implementation for tokens is given here, however, because it is expected that most compilers will use a regular-expression lexical analyzer/LALR parser pair as their parser.

The job of the lexical analyzer is to break the source text into small, contiguous pieces called *tokens*[4], some of which are passed on to the parser. Each token is a conceptually indivisible syntactic unit, such as an identifier, a number, a keyword, a mark of punctuation, etc. A token has at least two slots: a *class*, which indicates what kind of syntactic unit the token represents, and the *value*, which gives the actual fragment of source text corresponding to the token. For example, the fragment **Var1** might result in a token whose class is **:INTEGER** and whose value is **"Var1"**. The parser only examines the class slot when making parsing decisions, but may include the data from the value slot in the parse tree it produces. It is worth pointing out that the lexical analyzer may suppress the generation of tokens for some pieces of the program text such as whitespace and comments; the parser never sees these.

In addition to the class and value slots, a token has a slot for a *place*, which indicates the token's position within the source file. This information is transferred to the parse tree by the parser, and is used by later compiler phases to construct messages to the user that refer to specific places within his/her program. Places are described in Section 3.1.3.

The relationship between the lexical analyzer and the parser is somewhat unusual in that the lexical analyzer supplies tokens to the parser only upon demand. This is in contrast to all other modules of the compiler, which are invoked by a top-level procedure that passes data from one module to another. As a result, lexical tokens are quite short-lived in that the parser removes the information from a token and discards it shortly after receiving the token. To help prevent needless consing and garbage collection, a list of unused tokens is maintained, to which tokens should be explicitly returned when they are no longer needed.

#### 3.1.1.1. Selectors

**token-class** *token* *[Function]*

Returns the contents of the **class** slot of *token*, which is a keyword symbol indicating to what lexical class the token belongs. May be used with **setf**.

**token-value** *token* *[Function]*

Returns the contents of the **value** slot of *token*, which is a string giving the fragment of source text corresponding to the token. May be used with **setf**.

[4]Lexical tokens, of course, are not to be confused with tokens that carry data in a dataflow machine!

**token-place** *token* *[Function]*

Returns the contents of the **place** slot of *token*, which indicates where in the source text the token occurred. May be used with **setf**.

### 3.1.1.2. Constructors

**make-token** *class value* **&optional** *place* *[Function]*

Returns a token whose **class**, **value**, and **place** slots have been initialized from the corresponding arguments. The **place** slot will be **nil** if no *place* argument is given. **make-token** uses a token on the free token list if one exists, otherwise it creates ***token-allocation-quantum*** new tokens, puts them on the free list, and then uses one of them.

**return-token** *token* *[Function]*

Puts *token* onto the list of free tokens, where it can be reused.

***token-allocation-quantum*** *[Variable]*

Controls how many new tokens are created when the list of free tokens becomes empty.

## 3.1.2. Parse Tree Nodes

The parse tree is the data structure that is produced by the compiler's parser, and represents the source program in a form that reflects its syntactic structure. Following parsing, the parse tree may be subjected to several transformation phases such as macro expansion or type checking. These phases may annotate the parse tree (add information to nodes already existing in the tree) or alter the tree itself (add or delete nodes). Finally, the transformed parse tree is passed to code generation phases of the compiler. A fuller description of parse tree manipulations can be found in Chapter 1.

As the name suggests, the parse tree is a tree structure, where each node of the tree is called a *Parse Tree Node*, or ptnode for short. The representation of a parse tree in the Id Compiler differs somewhat from the usual theoretician's conception of a parse tree. Consider the following (admittedly ambiguous) grammar:

(1) *Expression* ← *Expression* + *Expression*

(2) *Expression* ← *Expression* * *Expression*

(3) *Expression* ← - *Expression*

(4) *Expression* ← *Identifier*

(5) *Expression* ← *Number*

Note that the grammar contains three types of symbols: *Non-Terminals*, which always appear on the left hand side of productions as well as on the right; *Keyword Terminals* like +, which appear in the source exactly as they do in the grammar; and *Pseudo-Terminals* like *Identifier* and *Number*, which actually represent classes of terminals that are treated exactly the same by the parser. The distinction between keyword terminals and pseudo-terminals is one not normally drawn in the literature, but as will be seen is quite important here.

Now consider the following fragment of source code: **Var1 + 6.847**. A theoretician might draw the parse tree for this expression as a node with three descendants: a node with the single descendant **Var1**, the character **+**, and a node with the single descendant **6.847**. This kind of representation is awkward in a compiler for two reasons. First, the compiler must search the descendants for keywords to determine whether the expression is an addition, multiplication, or negation. Second, the parse tree contains nodes for productions (4) and (5) which are pretty much information-free.

A more useful representation is used here. To address the first problem, each node of the parse tree contains not only a list of descendants, but also a tag indicating which production of the grammar was responsible for that node. The presence of this tag means that the only descendants included in the parse tree are non-terminals (other parse tree nodes) and pseudo-terminals; keyword terminals are not found anywhere in the parse tree. To address the second problem, the writer of the grammar may indicate that certain productions are not to produce parse tree nodes. Note that it is reasonable to suppress a particular production only if its right hand side is a single non-terminal or pseudo-terminal.

A parse tree node, therefore, has at least two slots: a tag identifying a production, and a list of descendants (children). Several other slots are also included: a pointer to the node's parent, for ease in traversing the parse tree, a *place* which indicates the position within the source file of the text that produced the node, and an **other-slots** slot which holds any additional information or annotations modules of the compiler wish to attach.

Pseudo-terminals are also represented as ptnodes; they can be distinguished from ptnodes representing internal parse tree nodes by the value of their production tag slot. While pseudo-terminals have no children, they do have a *value*. The value of a pseudo-terminal immediately after parsing is just the string corresponding to that pseudo-terminal as taken from the source text. Processing phases immediately following parsing may change the values of pseudo-terminals to more convenient representations; for example, the value of a pseudo-terminal representing a constant may be changed from a string to an actual integer or flonum. When a ptnode is used for a pseudo-terminal, the **children** slot holds the value.

Production tags are keyword symbols. For pseudo-terminals, the symbol is pseudo-terminal name as it appears in the grammar, for example, **:IDENTIFIER** for the pseudo-terminal *Identifier*. For productions, the tag is assigned by the parser generator, and will consist of the left-hand side's non-terminal followed by a slash and a unique number, for example, the parser generator might assign the tags **:EXPRESSION/1**, **:EXPRESSION/2**, and **:EXPRESSION/3** to the first three productions of the grammar above. Later phases of compilation may add ptnodes to the graph, and care must be taken to either choose an appropriate pseudo-terminal or non-terminal tag, or invent a new tag, depending on how the new node is to be treated by the succeeding phases of compilation. A facility is provided for dispatching on the tag slot without knowing the precise tag; see the **grammarcase** macro.

Figure 3-1 shows the parse tree for the expression **Var1 + 6.847** with respect to the grammar given above.

#### 3.1.2.1. Selectors

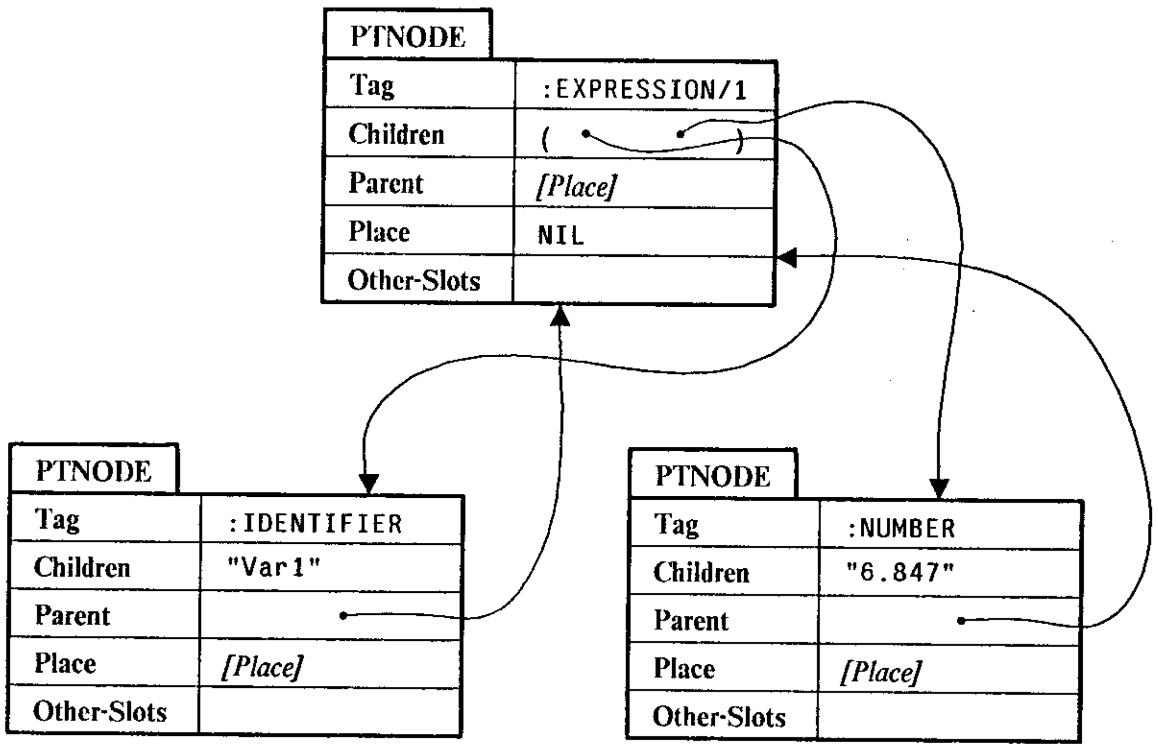


Figure 3-1: Parse Tree for **Var1 + 6.847**

**ptnode-tag** *ptnode* *[Function]*

Returns the production or pseudo-terminal tag for the node. May be used with **setf**.

**ptnode-children** *ptnode* *[Function]*

If *ptnode* represents a pseudo-terminal, as indicated by the **tag** slot of *ptnode*, **ptnode-children** returns the value of the terminal. Immediately after parsing, the value of a terminal is a string giving the exact appearance of the terminal in the source program, but a later analysis phase might change this to some other type such as a symbol or fixnum.

If *ptnode* is not a pseudo-terminal, **ptnode-children** returns a list of ptnodes which are the children of *ptnode*, where the first element of the list is the leftmost child.

May be used with **setf**.

**ptnode-value** *ptnode* *[Function]*

A synonym for **ptnode-children**, which can be used with pseudo-terminal ptnodes for clarity. May be used with **setf**.

**ptnode-parent** *ptnode* *[Function]*

Returns the *ptnodes* parent ptnode, or **nil** if *ptnode* has no parent. May be used with **setf**.

**ptnode-place** *ptnode* *[Function]*

Returns a *place* identifying where in the source file the construct represented by *ptnode* occurred. Exactly what place is indicated depends on the production and the parser: for example, if the production is something like *Let-Expression* ← **let** *Binding-list* **in** *Expression*, the place might indicate the first character of the keyword **let**, while for a production like *Expression* ← *Expression* + *Expression* it might indicate the +. May be used with **setf**.

Additional selectors may be defined by **define-ptnode-slot** (*q.v.*).

### 3.1.2.2. Constructors

**make-ptnode** *tag children* **&key :parent :place** *[Function]*

Makes and returns a new ptnode, initializing its **tag**, **children**, **parent**, and **place** slots from the corresponding arguments. Both non-terminal and pseudo-terminal ptnodes can be created with **make-ptnode**.

**make-parent-ptnode** *tag children* **&key :parent :place** *[Function]*

Makes a new non-terminal ptnode, initializing its **tag**, **children**, **parent**, and **place** slots from the corresponding arguments; the argument *children* must be a list of ptnodes. The new ptnode is returned, and is also stored as the parent of each of the elements of *children*, regardless of whether those ptnodes already have a value in their parent slot. This will often be more useful than **make-ptnode**.

### 3.1.2.3. Mutators

In addition to the functions below, all of the selectors in Section 3.1.2.1 and any selectors defined by **define-ptnode-slot** may be used with **setf**.

**replace-ptnode-child** *ptnode n new-child* *[Function]*

Replaces the *n*th element of the **children** slot of *ptnode* with *new-child*, and replaces the **parent** slot of *new-child* with *ptnode*. *Ptnode* must be a non-terminal ptnode and have at least $n+1$ children. Children are numbered from the left beginning with zero (just as are elements of lists for Common Lisp's **nth** function).

**replace-ptnode-children** *ptnode new-children* *[Function]*

Replaces the **children** slot of *ptnode* with *new-children*, which must be a list of ptnodes, and replaces the **parent** slot of each of the elements of *new-children* with *ptnode*. (Note: the list is not copied.)

### 3.1.2.4. Miscellaneous

**define-ptnode-slot** *selector-name* *[Macro]*

Defines a new "slot" for ptnodes which may be used as if it were one of the slots already provided by the compiler substrate. For example, the form

```
(define-ptnode-slot ptnode-desired-type)
```

defines a new ptnode slot called **desired-type**, which may be accessed by **(ptnode-desired-type** *ptnode***)** and written by **(setf (ptnode-desired-type** *ptnode***)** *value***)**. *Selector-name* must be a symbol whose print name begins with **ptnode-**, and may not be any of the symbols **ptnode-tag**, **ptnode-children**, **ptnode-value**, **ptnode-parent**, **ptnode-place**, or **ptnode-other-slots**. The slot is actually implemented as a property stored on the property list contained in the **other-slots** slot of ptnodes. The indicator used is a keyword symbol giving the name of the slot; for the example given, the indicator would be **:desired-type**. Note that initial values for slots defined by **define-ptnode-slot** may not given in calls to **make-ptnode** or **make-parent-ptnode**; they are always initialized to **nil**.

---

Rationale: The "other slots" mechanism was designed to meet three goals: to provide a convenient way to annotate ptnodes, to keep different annotations separated, and to provide an easy way of making commonly used annotations a permanent part of the compiler substrate. The latter might be desirable because built-in slots are both faster and take up less space. Changing an annotation from a **define-ptnode-slot** slot to a built-in slot will require no changes in programs that make use of the slot, since both kinds of slots are manipulated in the same way.

---

**let-ptnode-children** ({*var*}*) *ptnode* {*form*}* *[Macro]*

**let-ptnode-children** is used to bind variables to the children of a ptnode. Specifically, the form

```
(let-ptnode-children (var-1 var-2 ...) ptnode
  form-1
  form-2
  ...)
```

is equivalent to the form

```
(let ((var-1 (first (ptnode-children ptnode)))
      (var-2 (second (ptnode-children ptnode)))
      ...)
  form-1
  form-2
  ...)
```

except that **let-ptnode-children** takes special care that *ptnode* is evaluated only once. It is an error for *ptnode* to be a pseudo-terminal or to have fewer children than there are *vars*. While is it is legal for *ptnode* to have more children then there are *vars*, this usage is discouraged.

**grammarcase** *keyform* {(({({*key*}*) | *key*} {*form*}*)}* *[Macro]*

This form, which is similar to Common Lisp's **case** macro, is provided for dispatching on the **tag** slot of ptnodes. Its general form is

```
(grammarcase keyform
  (keylist-1 consequent-1-1 consequent-1-2 ...)
  (keylist-2 consequent-2-1 ...)
  (keylist-3 consequent-3-1 ...)
  ...)
```

**grammarcase** first evaluates *keyform*, which must evaluate to a ptnode, and then takes the **ptnode-tag** of the result, yielding the *key object.* **grammarcase** then considers each clause in turn. If the key object matches any item in a clause's keylist, then the consequents of that clause are executed as an implicit **progn**, with the value of the last consequent being returned as the value of the **grammarcase.** If the satisfied clause has no consequents, or if no clause is satisfied, **grammarcase** returns **nil.**

Each item in a keylist must be either a keyword symbol or a list of the form (*symbol* **<-** *symbol-or-string symbol-or-string* ..). The latter form is provided so that a production may be referred to by its appearance, rather than by the keyword symbol assigned to it by the parser generator. This feature may only be used when the grammar is known at compile time and has been loaded into the Lisp world [more detail later].

If there is only one item in a keylist, then the item itself may be used as the keylist. In addition, the symbols **t** and **otherwise** may be used in place of a keylist; if used, they must appear in the last clause, which will be executed if all of the other clauses fail. Here is an example of **grammarcase**:

```
(grammarcase x
  (:identifier
   (handle-identifier x))
  ((expression <- "(" expression ")")
   (handle-parenthesized-expression x))
  ((:constant
    (expression <- "[" constant "]"))
   (print-warning "Constants used")
   (handle-constants x))
  (otherwise
   (print-error "Unknown ptnode tag")))
```

This is equivalent to the following:

```
(case (ptnode-tag x)
  (:identifier
   (handle-identifier x))
  (:expression/1
   (handle-parenthesized-expression x))
  ((:constant :expression/2)
   (print-warning "Constants used")
   (handle-constants x))
  (otherwise
   (print-error "Unknown ptnode tag")))
```

Note that the actual symbols used in place of **:expression/1** and **:expression/2** would depend on the grammar being used at the time.

### 3.1.3. Places

A *place* is a small data structure that indicates a particular place within the source file. It has three slots: `line`, which gives the line number of the place, `column`, which gives the horizontal position within that line, and `character`, which gives the position within the text when viewed as a sequence of characters. All tree fields are zero-based. The idea is that `line` and `column` are most useful when printing messages, while `character` is most useful for use with text editors and other programs that actually manipulate the source file. [The definition of places is subject to change.]

#### 3.1.3.1. Selectors and Constructors

`place-line` *place* *[Function]*

Returns the contents of the `line` slot of *place.* May be used with `setf`.

`place-column` *place* *[Function]*

Returns the contents of the `column` slot of *place.* May be used with `setf`.

`place-character` *place* *[Function]*

Returns the contents of the `character` slot of *place.* May be used with `setf`.

`make-place` *line column character* *[Function]*

Returns a new place whose slots are initialized according to the arguments.

### 3.1.4. Parse Trees

When passing a complete parse tree from module to module, it is often necessary to pass along some additional information, such as the compiler version, *etc.* The `parse-tree` abstraction is provided for this purpose: it contains the root node of the parse tree along with a property list that can give any additional information needed.

`make-parse-tree` *root-ptnode* `&optional` *plist* *[Function]*

Creates and returns a new parse tree, whose root node is *root-ptnode,* and with property list *plist.* The default for *plist* is an empty property list.

`parse-tree-root` *parse-tree* *[Function]*

Returns the root node of parse tree *parse-tree.* May be used with `setf`.

`parse-tree-plist` *parse-tree* *[Function]*

Returns the property list of parse tree *parse-tree.* May be used with `setf`. The following function is probably more useful.

**parse-tree-get** *parse-tree indicator* **&optional** *default* *[Function]*

Returns the *indicator* property of *parse-tree*'s plist, or *default* if that property does not exist. In other words, **parse-tree-get** combines **parse-tree-plist** with **getf**. May be used with **setf**.

## 3.2. Dataflow Graphs

When all parse tree manipulations are complete, the parse tree is converted to a dataflow graph. Initially, the dataflow graph has a level of detail comparable to that of the original source program; this graph is called the program graph. At some point toward the end of compilation, the program graph, which may have undergone various transformations, is converted to a graph containing more detail and which is more specific to the particular target dataflow architecture; this graph is called the machine graph. After possibly further manipulations, this graph is assembled into object code for the target machine.

A single abstraction suffices to represent both program graphs and machine graphs; the distinction is merely one of restrictions imposed by the various compiler modules. As these restrictions depend on the language being compiled and the target architecture, they will not be discussed in this section. Instead, the abstractions for manipulating dataflow graphs, be they program graphs or machine graphs, are described here.

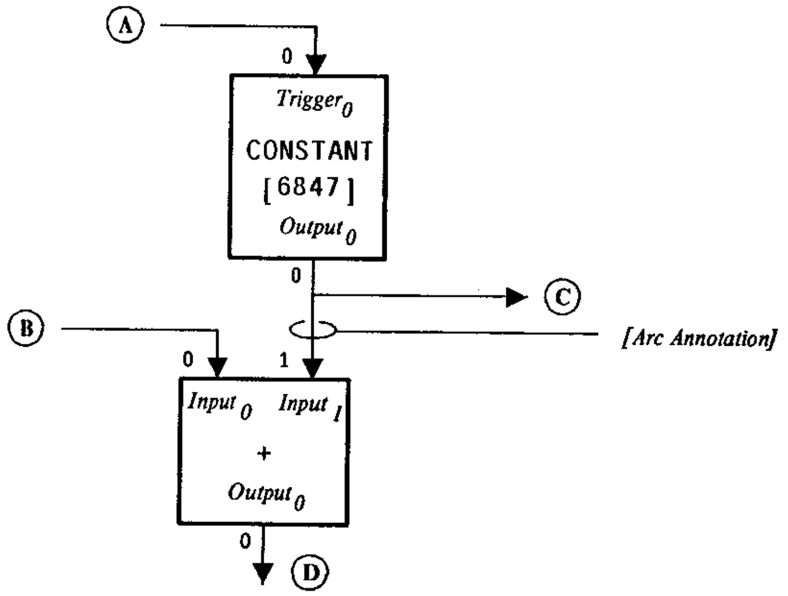
Figure 3-2 shows a fragment of a dataflow graph and its internal representation within the compiler. The components of this figure are explained in the sections that follow.

### 3.2.1. Instructions

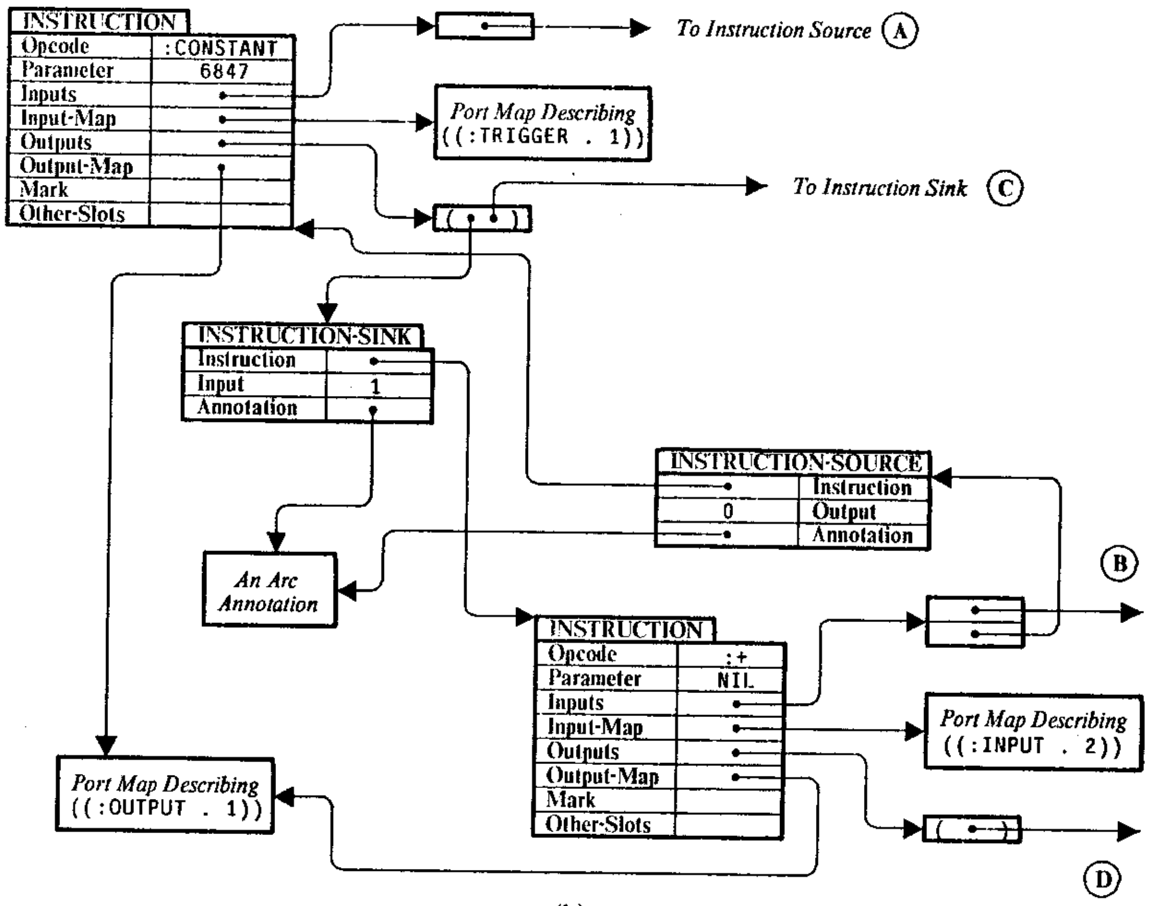
A dataflow graph is simply a collection of *instructions*, which are connected together by *arcs*. Each instruction contains an *opcode*, which identifies what operation the instruction performs, a certain number of *inputs*, and a certain number of *outputs*. Although the compiler substrate assigns no semantics to instructions in dataflow graphs, instruction inputs should be thought of as receiving data from other instructions, and instruction outputs should be thought of as sending data to other instructions. Collectively, the inputs and outputs of an instruction are referred to as its *ports*.

A dataflow graph is constructed by wiring the outputs of some instructions to the inputs of other instructions; it is not possible to wire outputs to outputs or inputs to inputs. An additional restriction is that while outputs may have any number of arcs leading away from them, each input may have only one arc leading to it. The implications of this restriction are discussed below. While arcs are thought of as being unidirectional, leading from outputs to inputs, they are actually implemented as bidirectional links for easy traversing of graph structure.

An instruction actually has eight slots: **opcode**, **parameter**, **inputs**, **input-map**, **outputs**, **output-map**, **mark**, and **other-slots**. **opcode** is a keyword symbol identifying the instruction, as described above. The **parameter** slot is intended for use when one opcode stands for a whole family of instructions. For example, there might be an instruction with opcode CONSTANT which emits a certain value upon the receipt of any input; the **parameter** slot could be used to indicate which constant is to be emitted for particular CONSTANT instruction. The slots **inputs** and **outputs** contain arrays which hold the inputs and outputs of the instruction. The sizes of these arrays are fixed at the time the instruction is created, and their exact contents is discussed below. The slots **input-map** and **output-map** each contain a data structure called a *port map*, which allows the inputs and outputs of an instruction to be referred to by



(a)



(b)

Figure 3-2: (a) A Dataflow Graph Fragment; (b) Its Internal Representation

symbolic names. Certain graph manipulation algorithms require the ability to mark nodes of the graph as they are encountered (so that each node is processed only once, for instance), and so a `mark` slot is provided for this purpose. Finally, `other-slots` holds a property list for additional slots defined by compiler modules, analogous to the `other-slots` slot of ptnodes (see Section 3.1.2.

An instruction's inputs and outputs are each numbered consecutively from zero, and can be referred to by number. Usually, however, it is more convenient to refer to a port by a symbolic name that suggests its function. A symbolic name for a port may be a simple name (a keyword symbol), or it may be a subscripted name consisting of a keyword symbol and a non-negative integer. Subscripted names are particularly useful for complicated instructions like IF and LOOP used for Id/83s, whose ports can be logically grouped into sets of varying size. The IF instruction, for example, has three sets of outputs: a set of outputs feeding the graph for the "then" side, a set of outputs feeding the "else" side, and a set of outputs that is the result of the conditional itself. Using symbolic port names, a compiler module can easily refer to the second "else" output, for example, even though the actual port number for that output may depend on how many "then" outputs the instruction has.

The rules for naming inputs or outputs are as follows:

- An input or output may have at most one name. (Arvind's principle)
- No two inputs or two outputs of the same instruction may have the same name, although an input and an output may have the same name (it is always possible to tell whether an input or output is meant).
- A name is either a keyword symbol (a simple name) or a cons whose car is a keyword symbol and whose cdr is a non-negative integer (a subscripted name).
- A simple name is an abbreviation for a subscripted name with a subscript of zero.
- If (*symbol* . *n*) is a name for an instruction's port, then so is (*symbol* . *i*), for all *i* from zero through *n*.
- Subscripted names with consecutive subscripts always map to ports with consecutive numbers.

In other words, port names define a partition of an instruction's ports, with the first port in a partition always having subscript zero.

There must be some way of translating symbolic names for ports to the corresponding numbers, and so each instruction contains an *input map* and an *output map* which give the appropriate translations. Maps can be created with the function `make-port-map`, which takes a description of the port names as input. Since the configuration of ports for an instruction cannot change, the maps for an instruction cannot change, and so a map may be shared by several instructions that have the same configuration of inputs or outputs. In fact, such sharing is encouraged because it saves space. One way to do this is to create maps for commonly used configurations and save them in some variables or in a table. The only function provided for manipulating maps is `make-port-map`; the user should not attempt to play with the internal structure of maps.

**`make-instruction`** *opcode input-map output-map* **`&key :parameter`** *[Function]*

Creates and returns a new instruction, whose **`opcode`**, **`input-map`**, **`output-map`**, and **`parameter`** slots have been initialized from the corresponding arguments. The number of inputs and outputs is inferred from the information in *input-map* and *output-map*, which must be port maps as created by **`make-port-map`**. *Opcode* must be a keyword symbol.

**`make-port-map`** *map-description* *[Function]*

Creates and returns a port map, based on a description of its contents. The description is a list of descriptors, each of which is either a keyword symbol or a cons of a keyword symbol and a positive integer. Each descriptor specifies a given number of ports with the same symbol and subscripts running consecutively from zero. If a descriptor is just a symbol, it is the same as the cons of that symbol and the number one. The map created assigns port numbers to the names in the descriptor from left to right. Naturally, no symbol may appear in the description twice. Here's an example:

```
(make-port-map '((:structure . 1)
                 (:subscript . 3)
                 :value
                 (:trigger . 2)))
```

returns a port map that describes the following mapping of port names to port numbers:

| Number | Name | Number | Name |
|---|---|---|---|
| 0 | `:STRUCTURE` | 4 | `:VALUE` |
| 1 | `(:SUBSCRIPT . 0)` | 5 | `(:TRIGGER . 0)` |
| 2 | `(:SUBSCRIPT . 1)` | 6 | `(:TRIGGER . 1)` |
| 3 | `(:SUBSCRIPT . 2)` | | |

**`instruction-opcode`** *instruction* *[Function]*

Returns the contents of the **`opcode`** slot of *instruction*. May be used with **`setf`**, although this fact is of limited utility since the number of inputs and outputs of an instruction cannot be altered after creation.

**`instruction-parameter`** *instruction* *[Function]*

Returns the contents of the **`parameter`** slot of *instruction*. May be used with **`setf`**.

**`define-instruction-slot`** *selector-name* *[Macro]*

Defines a new "slot" for instructions, analogous to **`define-ptnode-slot`**. *Selector-name* must be a symbol whose print name begins with **`instruction-`**, and may not be any of the symbols **`instruction-opcode`**, **`instruction-parameter`**, **`instruction-input-map`**, **`instruction-output-map`**, **`instruction-inputs`**, **`instruction-outputs`**, **`instruction-mark`**, or **`instruction-other-slots`**.

**`instruction-n-inputs`** *instruction* *[Function]*

Returns the number of input ports that *instruction* has.

**`instruction-n-outputs`** *instruction* *[Function]*

Returns the number of output ports that *instruction* has.

**`instruction-input-name-to-number`** *instruction input-name* *[Function]*

Returns the input number corresponding to the input of *instruction* named *input-name*, or **`nil`** if that input name does not exist.

**`instruction-output-name-to-number`** *instruction output-name* *[Function]*

Returns the output number corresponding to the output of *instruction* named *output-name*, or **`nil`** if that output name does not exist.

**`instruction-input-number-to-name`** *instruction input-number* *[Function]*

Returns the input name corresponding to the input of *instruction* with number *input-number*, or **`nil`** if there is no input with that number. Always returns a subscripted name.

**`instruction-output-number-to-name`** *instruction output-number* *[Function]*

Returns the output name corresponding to the output of *instruction* with number *output-number*, or **`nil`** if there is no output with that number. Always returns a subscripted name.

**`wire-instruction-to-instruction`** *instruction-1 output instruction-2 input* **`&optional`** *annotation* *[Function]*

Wires *instruction-1*'s output *output* to *instruction-2*'s input *input*. *Output* and *input* may be either port numbers or port names. The argument *annotation* is discussed in Section 3.2.3.

## 3.2.2. Instruction Sources and Sinks

The only primitive provided thus far for wiring instructions together is **`wire-instruction-to-instruction`**, which takes the instructions and port names/numbers as separate arguments. Often, however, it is convenient to pass around an object that refers to a particular input or output of a particular instruction. Two additional primitive types are provided for this: *instruction sinks*, and *instruction sources*. The terminology is that a sink is anything to which an instruction's output may be wired, and a source is anything to which an instruction's input may be wired. Putting it another way, a sink is something that can absorb a dataflow token, while a source is something that can emit a dataflow token. An instruction sink, therefore, describes a particular input of a particular instruction, while an instruction source describes a particular output of a particular instruction. As will be seen in a later section, there are other kinds of sinks and sources as well.

**`make-instruction-sink`** *instruction input* **`&optional`** *annotation* *[Function]*

Creates and returns an instruction sink referring to input *input* of instruction *instruction*. *Input* may be either an input name or an input number. The *annotation* argument is discussed in Section 3.2.3.

**make-instruction-source** *instruction output* **&optional** *annotation* *[Function]*

Creates and returns an instruction source referring to output *output* of instruction *instruction*. *Output* may be either an output name or an output number. The *annotation* argument is discussed in Section 3.2.3.

**wire-source-to-sink** *source sink* **&optional** *annotation* *[Function]*

Wires the source *source* to the source *sink*. If *source* and *sink* are instruction sources and sinks, respectively, then this is equivalent to **wire-instruction-to-instruction**. Other kinds of sources and sinks are discussed in Section 3.2.4. The *annotation* argument is discussed in Section 3.2.3.

**wire-source-to-instruction** *source instruction input* **&optional** *annotation* *[Function]*

Equivalent to calling **wire-source-to-sink** with a call to **make-instruction-sink** as the second argument.

**wire-instruction-to-sink** *instruction output sink* **&optional** *annotation* *[Function]*

Equivalent to calling **wire-source-to-sink** with a call to **make-instruction-source** as the first argument.

Sources and sinks make it easy to write procedures such as the following:

```
(defun compile-binding (lhs rhs)
  (let ((lhs-sink (compile-lhs lhs))
        (rhs-source (compile-expression rhs)))
     (wire-source-to-sink rhs-source lhs-sink)))
```

where **compile-lhs** and **compile-rhs** return a sink and a source, respectively.

## 3.2.3. Arcs

Instruction sources and sinks serve an additional role beyond being passed around by modules of a compiler: they are actually stored in the **inputs** and **outputs** slots of instructions themselves. The **inputs** slot of an instruction contains an array with as many elements as there are inputs, each element containing either an instruction source, if the input is wired, or **nil**, if it is not. The instruction source points to the instruction output to which the input is wired. Similarly, the **outputs** slot of an instruction contains an array with as many elements as there are outputs, and the elements of the array contain instruction sinks. Because an output may feed several inputs, however, each element of the array contains not a single instruction sink but a list of instruction sinks. The order in which sinks appear in these lists is unimportant.

**instruction-inputs** *instruction* *[Function]*

Returns the array containing the inputs of *instruction*, as described above. May *not* be used with **setf**.

**instruction-outputs** *instruction* *[Function]*

Returns the array containing the outputs of *instruction*, as described above. May *not* be used with **setf**.

**instruction-input** *instruction input* *[Function]*

Returns the contents of input *input* of instruction *instruction*. *Input* may be either an input number or an input name. The value returned is either an instruction source, if the input is wired, or **nil**, if it is not. May be used with **setf**, although the wiring functions are the preferred way of altering instruction inputs.

**instruction-output** *instruction output* *[Function]*

Returns the contents of output *output* of instruction *instruction*. *Output* may be either an output number or an output name. The value returned is a (possibly empty) list of instruction sinks. The order of sinks in this list is unimportant. May be used with **setf**, although the wiring functions are the preferred way of altering instruction outputs.

**instruction-sink-instruction** *instruction-sink* *[Function]*

Returns the instruction referred to by *instruction-sink*. May *not* be used with **setf**.

**instruction-sink-input** *instruction-sink* *[Function]*

Returns the input number referred to by *instruction-sink*. Note that the value returned is always a number, as **make-instruction-sink** converts input names to input numbers. May *not* be used with **setf**.

**instruction-sink-annotation** *instruction-sink* *[Function]*

Returns the **annotation** slot of *instruction-sink*. May *not* be used with **setf**.

**instruction-source-instruction** *instruction-source* *[Function]*

Returns the instruction referred to by *instruction-source*. May *not* be used with **setf**.

**instruction-source-output** *instruction-source* *[Function]*

Returns the output number referred to by *instruction-source*. Note that the value returned is always a number, as **make-instruction-source** converts output names to output numbers. May *not* be used with **setf**.

**instruction-source-annotation** *instruction-source* *[Function]*

Returns the **annotation** slot of *instruction-source*. May *not* be used with **setf**.

As can be seen from the above definitions, sources and sinks are immutable objects; the way an instruction's inputs and outputs are changed is by storing new sources and sinks, not by modifying the existing ones. Hence, there is no danger in sharing sources and sinks. Note too that instruction sources and sinks always carry port numbers instead of port names, as this minimizes the number of name-to-number translations that have to be performed.

When an instruction output is wired to an instruction input, the resulting connection is called an arc. In the graph, the arc is represented by a source and a sink: a sink pointing to the second instruction is stored in the appropriate output if the first instruction, and a source pointing to the first instruction is stored in the appropriate input of the second instruction. There is a certain amount of redundancy here, but having both the source and the sink available makes it easy to traverse a graph in any direction.

`remove-arc` *instruction-1 output instruction-2 input* *[Function]*

If there is an arc from output *output* of instruction *instruction-1* to input *input* of instruction *instruction-2*, `remove-arc` removes it by removing the sink from *instruction-1*'s output and removing the source from *instruction-2*'s input. Returns either `t` or `nil`, depending on whether the arc actually existed or not, respectively.

Associated with every arc is an annotation, which may be used to describe the data flowing on the arc. This annotation might be, for example, an indication of a variable name from the source program. Once an arc is put in place, both the source and the sink comprising that arc carry a pointer to the annotation, for convenience (only one of them really needs to carry it). Sources and sinks can also carry annotations even when they are not part of an arc, and these annotations are used in determining what the final annotation for an arc will be. When two instructions are wired together, the following rules for determining the annotation are applied in the order given:

1) If the call to the wiring function includes the optional *annotation* argument, then the value of that argument becomes the annotation.

2) If the wiring function takes a sink as an argument, and that sink carries an annotation, then that annotation becomes the annotation for the arc.

3) If the wiring function takes a source as an argument, and that source carries an annotation, then that annotation becomes the annotation for the arc.

4) Otherwise, the arc has no annotation.

---

Design Note: It was decided that sinks should take priority over sources in determining arc annotations because each source may be wired to many sinks, but not *vice versa*. Thus, in some sense, sinks carry more specific information. More complicated schemes, such as retaining *both* the sink's and source's annotation if they exist, were rejected as needlessly complex. The annotation policy is subject to revision.

---

A sink or source has no annotation if the `annotation` slot contains `nil`, and so `nil` can never be used as an annotation. Unlike the `other-slots` slots of ptnodes and instructions, the `annotation` slot of sources and sinks are not constrained to be property lists, and there is no "define slot" feature for arc annotations.

`arc-annotation` *instruction-1 output instruction-2 input* *[Function]*

Returns the annotation of the arc connecting output *output* of instruction *instruction-1* with input *input* of instruction *instruction-2*. May be used with `setf`.

## 3.2.4. Frames

A situation that commonly arises during compilation to dataflow graphs is that you want to wire something to an instruction, but you don't know what that something is. For example, consider the expression **a + b**. To compile this, you must create a + instruction, and then wire **a** and **b** to its inputs. Depending on when you encountered **a + b**, however, you may or may not yet have generated the graphs that produce **a** and **b**. To deal with this situation, the compiler substrate includes a facility called *frames*.

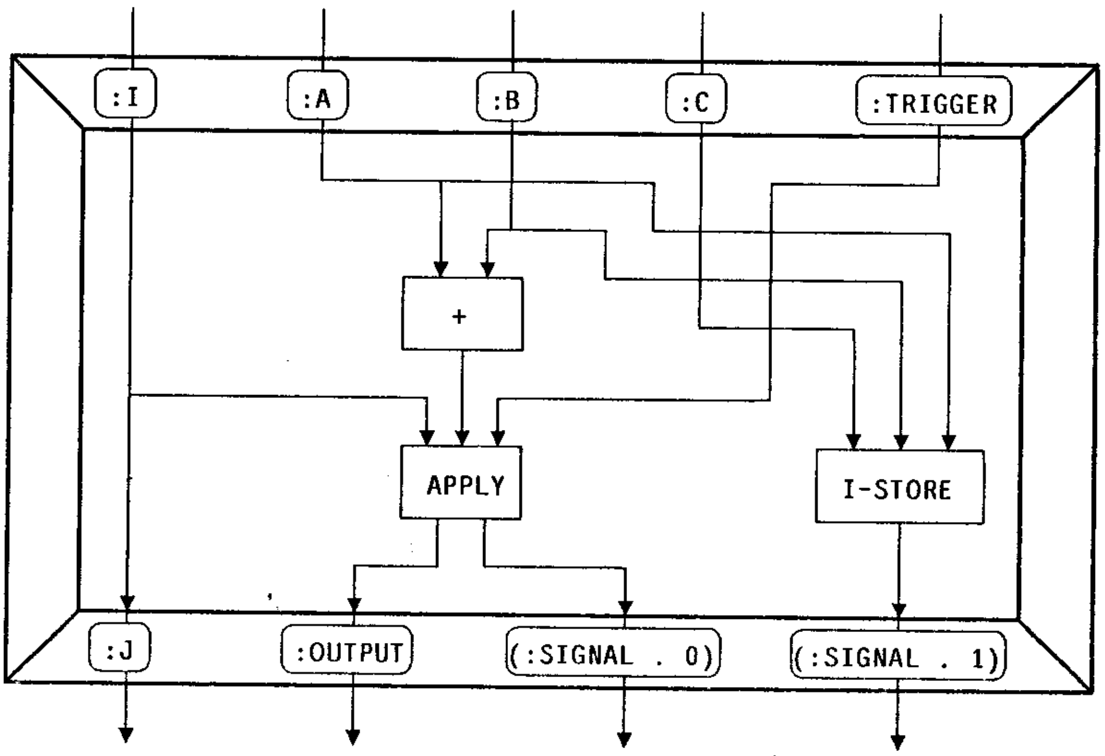


**Figure 3-3:** A Typical Frame

Figure 3-3 depicts a typical frame. As the figure shows, the name "frame" was chosen by analogy to a picture frame — in this case, the frame encloses a dataflow graph. A frame consists of a number of *frame inputs* and a number of *frame outputs*; collectively, a frame's inputs and outputs are known as its *points*. Each point serves as a wiring point, to which one source and any number of sinks may be connected. If the source and sinks are an instruction source and instruction sinks, the net effect is to cause the source to be wired to each of the sinks. For example, in Figure 3-3, if the output of an instruction were to be wired to frame input **:A**, then that output would be automatically wired to the inputs of the + and **APPLY** instructions shown.

As the figure shows, each frame input and frame output has a name associated with it. Frame point names are like instruction port names, and the rules for naming frame inputs and outputs are exactly the same. Unlike instruction ports, however, frame points can only be referred to by name; there is no number corresponding to a frame input or output. Another difference is that while the number of ports of an instruction is fixed when the instruction is created, frames are always created with no points at all, and additional inputs and outputs may be added to a frame at any time, although they can never be removed.

It should be noted that there is really no difference between a frame input and a frame output, since each can have one source and any number of sinks wired to it. But frames are most commonly used to "enclose" a subgraph, and so it is useful to separate the frame's points into those that lead to the enclosed subgraph (the frame inputs) and those that lead away from the enclosed subgraph (the frame outputs). Having separate frame inputs and outputs allows a frame to have both an input and an output with the same name.

Here is how frame points work. A frame point is created by adding an input or output to some frame. Any number of sinks may then be wired to the point, and they will be recorded in the frame. When a source is wired to that point, the system automatically wires the source to each of the sinks recorded for that point, and then records the source in the frame. Each time a sink is wired to the point thereafter, the sink is automatically wired to the point's source. Any attempt to wire another source to the point is an error.

Any kind of sink or source may be wired to a frame point, including sinks or sources referring to other frame points. This allows the creation of chains of interconnected frame points, with the system taking care of proper propagation of information. Since a frame point may have only one source but several sinks, interconnected frame points actually form a tree structure. When an instruction source is wired to the root of such a tree, that instruction output is wired to all instruction sinks at the leaves of the tree, and to any additional sinks that are later wired to the leaves. The system will detect any attempt to create a circular interconnection of frame points, and will signal an error.

There is an important difference between wiring instructions to instructions and wiring instructions to frame points. When an instruction is wired to another instruction, an arc is created consisting of a source/sink pair recorded in the output and input of the instructions involved. Thus, if instruction A is wired to instruction B, a pointer to instruction B is stored within instruction A and a pointer to instruction A is stored within instruction B. When an instruction is wired to a frame point, however, that fact is recorded only in the frame, and not in the instruction. For example, if an output of instruction C is wired to frame point D, an instruction source is recorded within D's frame, but no sink is recorded in instruction C's output. Similarly, if an input of instruction E is wired to frame point F, an instruction sink is recorded in F's frame, but no source is stored in E's input. The arcs between instructions and frame points and between pairs of frame points are called *virtual arcs*, to contrast them with true arcs between instructions.

This fact is significant because it means you cannot tell if an instruction is wired to a frame point by examining the instruction. Consider the following situation: instruction input A is wired to frame point B, and then that same input is wired to instruction output C. One might expect the latter wiring to cause an error, since it will be the second time instruction input A is wired. Because wiring the input A to the frame point did not affect A's instruction, however, the wiring of A to instruction output C does not cause an error, and an arc between A and C is placed normally. An error *does* occur if an instruction output is later wired to frame point B, since the system will then attempt to wire that output to instruction input A, and A is already wired.

---

**Design Note:** The behavior of frame points as described in the preceding paragraph was chosen to make the implementation of frames more efficient, even though it results in anomalous situations as described. The alternative was to record virtual arcs in the same manner as real arcs, which means that virtual arcs would have to be replaced whenever a chain of interconnected frame points was extended, or finally connected to instructions. The scheme adopted allows frames to be used heavily with little overhead, but the behavior of frame points may be revised if the anomalous situations described become a problem.

---

Frames are an intermediate data structure only; they cannot be passed from compiler module to compiler module.

**make-frame** *[Function]*

Creates and returns a new frame with no points.

**add-input-to-frame** *frame name* *[Function]*
**add-output-to-frame** *frame name* *[Function]*

Adds a frame input (output) named *name* to frame *frame*. It is an error if an input (output) with that name already exists. *Name* may be either a keyword symbol or the cons of a keyword symbol and a non-negative integer; specifying just a symbol is equivalent to specifying the cons of that symbol and zero. If *name* is a subscripted name, then inputs (outputs) are created for all names with the same symbol and subscripts running from zero to the given subscript, if any of those names do not already exist.

**add-next-input-to-frame** *frame name* *[Function]*
**add-next-output-to-frame** *frame name* *[Function]*

*Name* must be a keyword symbol. If *frame* has no input (output) named *name*, an input (output) named `(`*name* `. 0)` is added to *frame*. Otherwise, an input (output) named `(`*name* `.` $n + 1$`)` is added to *frame*, where *n* is the subscript of the input (output) of *frame* that has symbol *name* and the largest subscript. Returns the name of the input (output) that was actually created.

**frame-input-names** *frame* **&optional** *symbols-only?* *[Function]*
**frame-output-names** *frame* **&optional** *symbols-only?* *[Function]*

If *symbols-only?* is `nil` or unspecified, returns a list of frame input (output) names for *frame*, where each element of the list is the (subscripted) name of the input (output) with the highest subscript of all names bearing that name's symbol. Otherwise, just returns a list of symbols, one for each frame input (output) with a different symbol.

**frame-input-exists?** *frame name* *[Function]*
**frame-output-exists?** *frame name* *[Function]*

Returns `t` if *frame* has a frame input (output) with name *name*, `nil` otherwise.

**frame-number-of-inputs-with-symbol** *frame symbol* *[Function]*
**frame-number-of-outputs-with-symbol** *frame symbol* *[Function]*

Returns the number of frame inputs (outputs) of frame *frame* whose symbol is *symbol*. This number is one greater than the subscript of the name with the highest subscript, or zero if there are no inputs (outputs) with that symbol.

**make-frame-input-source** *frame name* **&optional** *annotation* *[Function]*
**make-frame-output-source** *frame name* **&optional** *annotation* *[Function]*

Creates and returns a source referring to the frame input (output) named *name* of frame *frame*. This source may then be wired to an instruction input, or to another frame input or output. *Annotation* is discussed below.

**make-frame-input-sink** *frame name* **&optional** *annotation* *[Function]*
**make-frame-output-sink** *frame name* **&optional** *annotation* *[Function]*

Creates and returns a sink referring to the frame input (output) named *name* of frame *frame*. This sink may then be wired to an instruction output, or to another frame input or output. *Annotation* is discussed below.

The function **wire-source-to-sink** works for any combination of instruction, frame input, or frame output sources and sinks. In addition, the following functions are provided in case one of the arguments to **wire-source-to-sink** would be a call to one of the **make-?-sink** or **make-?-source** functions.

**wire-frame-output-to-frame-output** *frame-1 output-1 frame-2 output-2* **&optional** *annotation* *[Function]*
**wire-frame-output-to-frame-input** *frame-1 output frame-2 input* **&optional** *annotation* *[Function]*
**wire-frame-output-to-instruction-input** *frame output instruction input* **&optional** *annotation* *[Function]*
**wire-frame-output-to-sink** *frame output sink* **&optional** *annotation* *[Function]*
**wire-frame-input-to-frame-output** *frame-1 input frame-2 output* **&optional** *annotation* *[Function]*
**wire-frame-input-to-frame-input** *frame-1 input-1 frame-2 input-2* **&optional** *annotation* *[Function]*
**wire-frame-input-to-instruction-input** *frame input instruction input* **&optional** *annotation* *[Function]*
**wire-frame-input-to-sink** *frame input sink* **&optional** *annotation* *[Function]*
**wire-instruction-output-to-frame-output** *instruction output-1 frame output-2* **&optional** *annotation* *[Function]*
**wire-instruction-output-to-frame-input** *instruction output frame input* **&optional** *annotation* *[Function]*
**wire-source-to-frame-output** *source frame output* **&optional** *annotation* *[Function]*
**wire-source-to-frame-input** *source frame input* **&optional** *annotation* *[Function]*

The rules for annotating an arc are necessarily complicated by the presence of frame points. Here are the rules when frame points are involved:

- When a frame point is wired to an instruction or another frame point, a virtual arc is created and recorded in the frame (see above). The annotation for this virtual arc is derived from the rules on page 22: annotations given in call to the wiring function take priority, followed by annotations on the sink involved, followed by annotations on the source.

- When a chain is completed, the annotation that will be included in the actual arc from instruction output to instruction input is determined by starting at the virtual arc leading to the instruction input and tracing up the chain until an annotation is found. If no annotation is found, the actual arc will have no annotation.

Although the rules for arc annotation may seem horribly complex, they are designed to the following simple principle: the system should automatically choose the annotation that best describes the arc. By using **setf** with **arc-annotation**, the compiler writer can always override an annotation chosen by the system.

### 3.2.5. Dataflow Graphs

When passing a complete dataflow graph from module to module, it is often necessary to pass along some additional information, such as the compiler version, *etc.* The **`dataflow-graph`** abstraction is provided for this purpose. Like the **`parse-tree`** abstraction, the **`dataflow-graph`** abstraction packages up a graph along with a property list carrying additional information. Unlike a parse tree, however, a dataflow graph is not necessarily connected, and so it does not suffice to include a single instruction in the **`dataflow-graph`** structure. Instead, it carries a *root set*, which is a list containing one or more instructions of the graph. Any instruction reachable from one of the instructions in the root set is considered part of the graph. The root set must therefore contain at least one instruction from each connected component of the graph; it does not matter if more than one instruction from each component is included.

**`make-dataflow-graph`** *root-set* **`&optional`** *plist* *[Function]*

Creates and returns a new dataflow graph, whose root set is *root-set*, and with property list *plist*. The default for *plist* is an empty property list.

**`dataflow-graph-root-set`** *dataflow-graph* *[Function]*

Returns the root set of dataflow graph *dataflow-graph*. May be used with **`setf`**.

**`dataflow-graph-plist`** *dataflow-graph* *[Function]*

Returns the property list of dataflow graph *dataflow-graph*. May be used with **`setf`**. The following function is probably more useful.

**`dataflow-graph-get`** *dataflow-graph indicator* **`&optional`** *default* *[Function]*

Returns the *indicator* property of *dataflow-graph*'s plist, or *default* if that property does not exist. In other words, **`dataflow-graph-get`** combines **`dataflow-graph-plist`** with **`getf`**. May be used with **`setf`**.

# 4. External Representation

It should be possible to extract the intermediate form of a program between any two compiler modules, process it off-line or with another piece of software, and then feed it back in for the completion of processing. To this end, the external representation of parse trees and dataflow graphs is given here. The external representation of machine code for the tagged-token dataflow architecture, while based on material in this Section, is described in another document.

For any type of compiler output, there are three possible formats in which it can appear:

1) Standard format, which can both be written by the compiler and read back in for further processing. A file in standard format is a sequence of standard characters, and so can easily be transferred between machines through a variety of media, including electronic mail.

2) Binary format, which also can be written and read by the compiler. A file in binary format is a sequence of eight-bit bytes, and so may contain unprintable and control characters if viewed as ASCII or some other character set. Its advantage is that binary format files occupy less space, and can be read and written more rapidly by compiler software.

3) Verbose format, which can be written by the compiler but not read in. Verbose format is easily read by humans, and exists for hand inspection of compiler output.

Standard and binary format are meant to be interchangeable, in that both carry the same information, and both are machine readable. Parse trees and dataflow graphs have a fixed, documented representation in these formats. Verbose format, on the other hand, is not machine readable, and is not standardized. It is mentioned here only for completeness.

The Standard and Binary formats for parse trees, dataflow graphs, and other types of compiler output are not defined directly, but are defined in terms of an intermediate form called the *Compiler Input/Output Base Language*, or CIOBL. The role of CIOBL is depicted schematically in Figure 4-1. Expressing external representations in terms of CIOBL has two advantages: programs for converting new types of compiler input/output can deal with I/O at a very abstract level, and both standard and binary format for the new type become immediately available. The compiler substrate does not just provide a general external representation for *any* internal data structure because particular data structures may have properties (*e.g.*, lack of sharing) that lead to more efficient representations than the most general. Furthermore, requiring an explicit translation from internal to external representation means that file formats can remain the same even when internal data structures are revised.

CIOBL is a fairly simple language consisting of objects of the following five types: Integers, Floats, Symbols, Strings, and Lists. Strings must contain only Common Lisp standard characters. Lists may contain any number of objects of any of the five types, and as in Lisp the empty list is indistinguishable from the symbol `nil`. This base language was chosen to be adequate to represent complex data structures but yet quite portable between different ID Compiler implementations and other programs that interact with the compiler. A file simply consists of a sequence of zero or more objects from the base language; the detailed structure of a file depends on the type of data structure being represented.

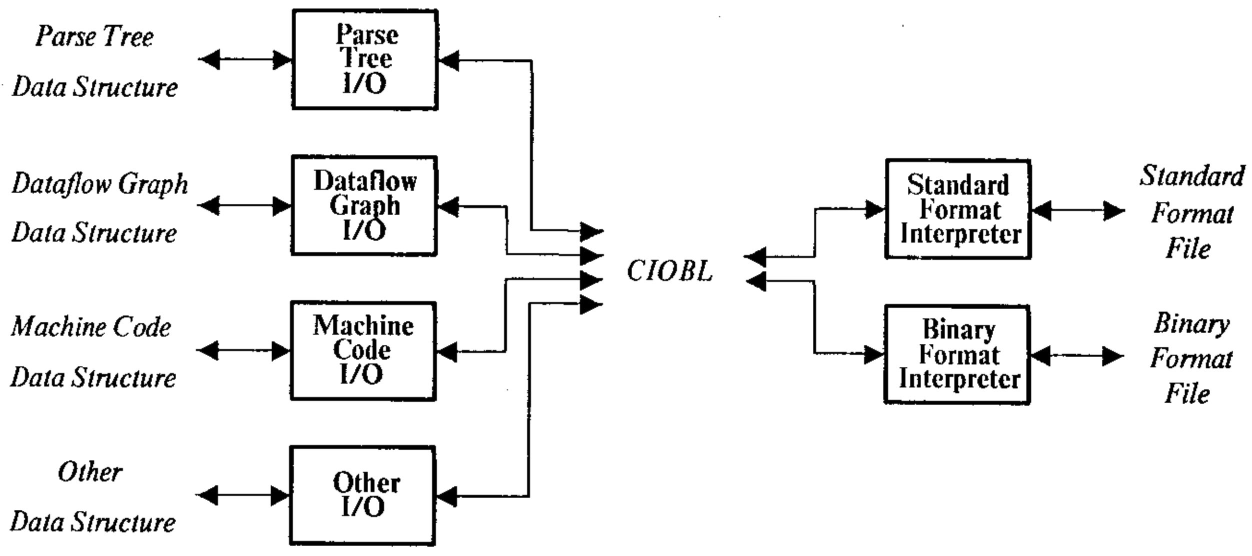

Parse Tree Data Structure
Parse Tree I/O
Dataflow Graph Data Structure
Dataflow Graph I/O
Machine Code Data Structure
Machine Code I/O
Other Data Structure
Other I/O
CIOBL
Standard Format Interpreter
Binary Format Interpreter
Standard Format File
Binary Format File

**Figure 4-1:** The Role of the Compiler Input/Output Base Language

## 4.1. Parse Tree Files

A parse tree file contains zero or more parse trees. Each parse tree, as described in Section 3.1.4, consists of a tree of ptnodes along with a property list which can be used to give information about the whole tree. Furthermore, another property list is included at the beginning of the file which gives information about the whole file. Values included in this property list might include compiler version, date compiled, *etc.*

The format of parse tree files is as follows:

*global-plist* {*parse-tree-plist root-ptnode*}*

where *global-plist* is the property list for the whole file, and each *parse-tree-plist* is the property list for the parse tree immediately following. Each plist is of the form:

({*indicator value*}*)

where each *indicator* is a symbol, and each *value* is any object in the compiler input/output base language.

Each *root-ptnode* is the root ptnode of a parse tree, where a ptnode is of the form:

(*tag place other-slots children*)

The fields *tag*, *place*, *other-slots*, and *children* give the values for the corresponding slots of the ptnode. *Children* is either a list of ptnodes or some other value, depending on whether the ptnode is an internal node or a terminal node, respectively. Note that the **parent** slot of ptnodes is not written to files, since it can be inferred from the *children* fields.

For example, a parse tree file containing the parse tree of Figure 3-1, with an empty file property list and parse tree property list, would appear as the following (italicized comments are not part of the file):

```
()                              File Property List
()                              Parse Tree Property List
(:EXPRESSION/1 NIL NIL ((:IDENTIFIER NIL NIL "Var1")
                         (:NUMBER NIL NIL "6.847")))
```

## 4.2. Dataflow Graph Files

A dataflow graph file contains zero or more dataflow graphs. Like the parse tree file, the dataflow graph file also contains a property list for the whole file as well as property lists for each dataflow graph in the file. The format is as follows:

*global-plist* {*dataflow-graph-plist root-set* (*instruction**) (*map**)}*

The format of property lists is as for parse tree files. *Root-set* is a list of integers, each indicating an instruction in the graph's root set; instructions are numbered consecutively from zero. (While the root set could be determined by analyzing the connectivity of the graph after reading the file, it is included explicitly for convenience.) Instructions have the following format:

(*opcode parameter other-slots input-map-number output-map-number* {(*destination**)}*)

The *opcode*, *parameter*, and *other-slots* fields give the values for the corresponding slots of the instruction. The *input-map-number* and *output-map-number* fields indicate which of the port maps following the list of instructions in the file are the input map and output map, respectively, of the instruction. The maps are numbered consecutively from zero. There are as many destination lists as there are outputs of the instruction, and each destination list may contain any number of destinations. A destination has the following format:

*instruction-number input-number annotation*

The *instruction-number* field refers to another instruction in the same graph in the same file; the instructions are numbered consecutively from zero. *Input-number* indicates which input of the instruction the output is wired to, while *annotation* gives the annotation for that arc. Note that the pointers from inputs to outputs are not included in the file, since they can be inferred. Note too that the **mark** slot of instructions is not written to files.

Each *map* is a list in a form acceptable as input to **make-port-map**. The reason that maps are not included directly in instructions is because maps will probably be shared among many instructions, and this format allows that sharing to be expressed, reducing the size of dataflow graph files.

As an example, the dataflow graph in Figure 4-2 would be represented as follows:

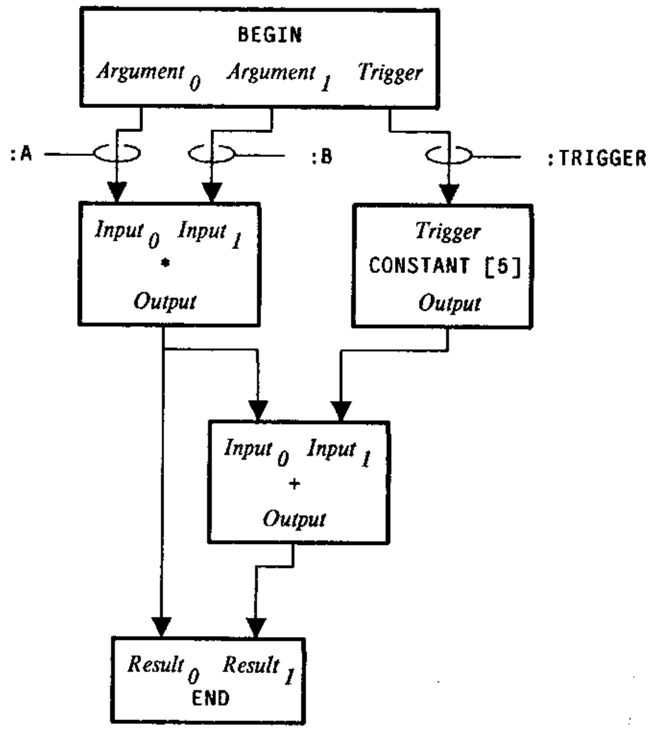


Figure 4-2: A Small Dataflow Graph

```
NIL
NIL
((:BEGIN NIL NIL 0 1 (1 0 :A) (1 1 :B) (2 0 :TRIGGER))
 (:* NIL NIL 2 3 (4 0 NIL 3 0 NIL))
 (:CONSTANT 5 NIL 4 3 (3 1 NIL))
 (:+ NIL NIL 2 3 (4 1 NIL))
 (:END NIL NIL 5 0 ()))
(()
 ((:ARGUMENT . 2) :TRIGGER)
 ((:INPUT . 2))
 (:OUTPUT)
 (:TRIGGER)
 ((:RESULT . 2)))
```

## 4.3. Formats

The preceding sections described file formats in terms of the compiler input/output base language consisting of symbols, strings, integers, floats, and lists. This section gives the standard and binary formats in which this base language can appear.

## 4.3.1. Standard Format

Standard format is a representation for the compiler input/output base language that contains only Common Lisp standard characters. It is specifically designed to be compatible with Lisp's reader. The standard format representation of CIOBL objects is as follows:

| | |
|---|---|
| *Symbols* | A symbol is represented by its print name, including the package qualifier relative to the **id-compiler** package. If necessary, the symbol name and/or package name is enclosed in a pair of vertical bar (**#\\|**) characters. Vertical bars are necessary if the print name otherwise would not be correctly interpreted by Common Lisp's **read** function. The backslash escape is never used in standard format. |
| *Strings* | A string is represented by the value of the string enclosed in quotation marks, with any quotation marks that are part of the string preceded by backslash characters. The string must contain only Common Lisp standard characters. |
| *Integers* | An integer is represented by a sequence of digits, optionally preceded by a hyphen. The digits give the absolute value of the integer in base 10, and the hyphen indicates a negative number. The first digit is never a zero unless the integer itself is zero; zero is represented by the digit zero. |
| *Floats* | A float is represented by a sequence of characters of the form [-]*digit*$^{+}$[.*digit*$^{+}$][E[-]*digit*$^{+}$], with no intervening spaces. The meaning of such a string is according to Common Lisp's rules for representing floating point numbers. See the Common Lisp manual, page 17. |
| *Lists* | Lists are represented by the character **#\\(**, followed by the elements of the list, followed by **#\\)**. Each of the elements of the list must be separated by at least one whitespace character. Whitespace is optional following the **#\\(** and preceding the **#\\)**. |

Successive objects in a file are separated by whitespace. Whitespace means one or more characters from the set {**#\Space**, **#\Newline**, **#\Tab**, **#\Linefeed**, **#\Page**, **#\Return**}. Note that the last four characters in this set are semi-standard, and so the compiler will never include them in output, although they will be accepted as whitespace on input (if the compiler is being run on a system that has these characters).

The above description may seem complex, but it is just a description of the usual Lisp printed form of symbols, strings, integers, floats, and lists.

The actual encodings of characters that appear in standard format files depend on the Lisp implementation that produced those files. For example, a compiler running on a Lisp Machine would probably use the Lisp Machine character set, while a compiler running on an IBM/370 would undoubtedly use EBCDIC. The files are portable because most file transfer software is designed to translate from one character set to another, and limiting standard format files to Common Lisp standard characters virtually guarantees that such translations exist.

### 4.3.2. Binary Format

Binary format is a compact representation for the compiler input/output base language designed to occupy little space and to be quickly read and written. A file in binary format is a sequence of eight-bit bytes, and so may contain characters not in any character set. For this reason, it is less likely to be transferable between machines than standard format.

Binary format is designed so that the length of each object is known, so there is no need for whitespace or other separators. All objects in binary format are represented by a *punctuation byte* that indicates the type of object, followed by zero or more bytes that give the value of the object; the number of bytes that form the object is determined by the punctuation byte, and in some cases, by the first few bytes following the punctuation byte. For convenience in programming, constants have been defined which give the value of punctuation bytes. The values corresponding to these constants are given below:

| *Value* | *Constant Name* | *Value* | *Constant Name* |
|---|---|---|---|
| 0 | `<symbol>` | 24 | `<pos-long-integer>` |
| 1 | `<keyword-symbol>` | 25 | `<neg-integer-8>` |
| 2 | `<long-symbol>` | 26 | `<neg-integer-16>` |
| 3 | `<nil>` | 27 | `<neg-integer-24>` |
| 10 | `<string>` | 28 | `<neg-integer-32>` |
| 11 | `<long-string>` | 29 | `<neg-long-integer>` |
| 20 | `<pos-integer-8>` | 30 | `<float>` |
| 21 | `<pos-integer-16>` | 31 | `<long-float>` |
| 22 | `<pos-integer-24>` | 40 | `<list-begin>` |
| 23 | `<pos-integer-32>` | 41 | `<list-end>` |

Note that the angle brackets are actually part of the names of these constants. When these names appear in the descriptions below, it is to be understood that the values of these constants are actually meant. The programs that manipulate binary format files should, of course, use the names for the constants rather than their values.

*Symbols* The usual format for symbols is

`<symbol>` $n\ char_1\ char_2 \ldots char_n$

where $char_1\ char_2 \ldots$ forms the print name of the symbol, including a package qualifier relative to the `id-compiler` package, if necessary. Since keyword symbols are quite common, a special format is provided for them:

`<keyword-symbol>` $n\ char_1\ char_2 \ldots char_n$

This saves a byte over the usual format since the package qualifier need not be given. If the print name of the symbol is longer than 255 characters, the long format must be used:

`<long-symbol>` $m\ b_{m-1}\ b_{m-2} \ldots b_0\ char_1\ char_2 \ldots char_n$

where $m$ gives the number of bytes in the integer $n$, which is represented by $b_{m-1}\ b_{m-2} \ldots b_0$. Finally, the symbol `nil` may be represented by the single character `<nil>`. There is no need for vertical bars or other escape characters in binary format.

*Strings*

If the string is 255 characters or less, the following format is used:

**<string>** $n\ char_1\ char_2 \ldots char_n$

Otherwise, the long format must be used:

**<long-string>** $m\ b_{m-1}\ b_{m-2} \ldots b_0\ char_1\ char_2 \ldots char_n$

which is similar to the long format for symbols explained above. There is no need for backslashes preceding quotation marks in binary format.

*Integers*

Because small integers are more common than large ones, several formats for integers are used depending on the size of the integer.

| *Size of Integer* | *Punctuation Character* | *Number of Bytes After Punctuation* |
|---|---|---|
| $-2^{255} < n \leq -2^{32}$ | **<neg-long-integer>** | *[See Below]* |
| $-2^{31} < n \leq -2^{24}$ | **<neg-integer-32>** | 4 |
| $-2^{24} < n \leq -2^{16}$ | **<neg-integer-24>** | 3 |
| $-2^{16} < n \leq -256$ | **<neg-integer-16>** | 2 |
| $-256 < n < 0$ | **<neg-integer-8>** | 1 |
| $0 \leq n < 256$ | **<pos-integer-8>** | 1 |
| $256 \leq n < 2^{16}$ | **<pos-integer-16>** | 2 |
| $2^{16} \leq n < 2^{24}$ | **<pos-integer-24>** | 3 |
| $2^{24} \leq n < 2^{32}$ | **<pos-integer-32>** | 4 |
| $2^{32} \leq n < 2^{255}$ | **<pos-long-integer>** | *[See Below]* |

With the exception of **<pos-long-integer>** and **<neg-long-integer>**, the punctuation character is followed by the most significant byte, then the next most significant byte, *etc.*, and finally the least significant byte of the absolute value of the integer represented. **<pos-long-integer>** and **<neg-long-integer>** are followed by a byte giving the number of bytes that follow, and then the bytes of the integer itself.

*Floats*

Floats appear in the following format:

**<float>** $n\ char_1\ char_2 \ldots char_n$

where $char_1\ char_2 \ldots char_n$ is how the float would be represented in standard format. If the standard format representation would take more than 255 characters, the long format is used:

**<long-float>** $m\ b_{m-1}\ b_{m-2} \ldots b_0\ char_1\ char_2 \ldots char_n$

*Lists*

Lists appear as the character **<list-begin>**, followed by the contents of the list, followed by the character **<list-end>**. The empty list is always represented as **<nil>**, never by **<list-begin>** followed by **<list-end>**.

For example, if the following sequence appeared in a standard format file:

```
(:EX 3 "odd\"str" 5.6 "Test" NIL |ODDsYM| -70000 (A 0))
```

it would appear as the following in binary format (where numbers indicate the numeric value of a particular byte):

```
<list-begin> <keyword-symbol> 2 #\E #\X <pos-integer-8> 3
<string> 7 #\o #\d #\d #\" #\s #\t #\r
<float> 3 #\5 #\. #\6 <string> 4 #\T #\e #\s #\t <nil>
<symbol> 6 #\O #\D #\D #\s #\Y #\M
<neg-integer-24> 1 17 112 <list-begin> <symbol> 1 #\A
<pos-integer-8> 0 <list-end> <list-end>
```

The standard form takes 55 characters, while the binary form takes 48 bytes.

Unlike standard format files, binary files are not meant to be transferred with any character set translations; instead, they must be transferred in "image mode". Indeed, the punctuation bytes and bytes giving field lengths might accidently be translated if interpreted as characters. This means that some stand must be taken with regard to the encodings of characters as they appear in the binary format representations of symbols, strings, and floats.

The convention adopted, then, is that bytes representing characters will be the ASCII encodings of these characters. ASCII was chosen since it is the most commonly used character set (note that since only standard characters will appear in binary format files, the Lisp Machine character set encodings are the same as ASCII). This means that compiler implementations on the IBM/370, for example, will have to translate between ASCII and EBCDIC when handling binary format files.

Given the foregoing, a "dump" of the above example would be (in decimal):

```
040 001 002 069 088 020 003 010
007 111 100 100 034 115 116 114
030 003 053 046 054 010 004 084
101 115 116 003 000 006 079 068
068 115 089 077 027 001 017 112
040 000 001 065 020 000 041 041
```

## 4.4. File Manipulation Functions

The compiler substrate defines a new data type, the *CIOBL stream*, which provides a clean interface between the standard and binary format interpreters and the compiler's I/O modules. CIOBL streams resemble Common Lisp streams in that they produce or absorb data. But while Common Lisp streams produce or absorb either characters or integers, CIOBL streams produce or absorb CIOBL objects. None of the Common Lisp stream functions work on CIOBL streams; instead, a set of functions is provided specifically for manipulating CIOBL streams.

A CIOBL stream is created by specifying a Common Lisp stream and an indication of whether standard or binary format is to be used. The CIOBL stream may then be used to read or write CIOBL objects without regard for which format is being used, the CIOBL stream performing the appropriate translations and operations on the underlying Common Lisp stream. In this way, a single program can do I/O in both standard and binary format.

**make-ciobl-stream** *stream binaryp* *[Function]*

This creates and returns a CIOBL stream that performs I/O on Common Lisp stream *stream*. If *binaryp* is true, *stream* must be a binary stream, and binary format is used, otherwise *stream* must be a character stream, and standard format is used. The CIOBL stream may be used for either input or output as long as *stream* supports input or output, respectively.

**close-ciobl** *ciobl-stream* **&key :abort** *[Function]*

This is analogous to Common Lisp's **close**, except that it works on CIOBL streams.

**with-ciobl-stream (** *var stream* **)** {*declaration*}* {*form*}* *[Macro]*

The form *stream* is evaluated and must produce a Common Lisp stream (not a CIOBL stream!). The variable *var* is bound with the result of calling **make-ciobl-stream** on the stream as its value, and then the forms of the body are executed as an implicit **progn**; the results of evaluating the last form are returned as the value of the **with-ciobl-stream** form. The stream is automatically closed on exit from the **with-ciobl-stream** form, no matter whether the exit is normal or abnormal. The stream should be regarded as having dynamic extent.

In other words, **with-ciobl-stream** is like **with-open-stream**, except that it calls **make-ciobl-stream** and that **close-ciobl** is used to close the stream upon exit instead of **close**.

**ciobl-stream-p** *object* *[Function]*

**ciobl-stream-p** is true if its argument is a CIOBL stream, and otherwise is false.

**(ciobl-stream-p x)** ≡ **(typep x 'ciobl-stream)**

**read-ciobl** *ciobl-stream* **&optional** *eof-error-p eof-value return-list-p* *[Function]*

**read-ciobl** attempts to read a CIOBL object from CIOBL stream *ciobl-stream*, and returns two values: the object read, and a symbol indicating status. If the next object in the input stream is a symbol, string, integer, or float, then the first value returned is that object and the second value returned is the symbol **:symbol**, **:string**, **:integer**, or **:float**, respectively.

If the next object in the input stream is a list, then what is returned depends upon the value of *return-list-p*. If *return-list-p* is true, then objects are read from the input stream until the end of the list is reached, and the first value returned is a list of the objects (some of which may themselves be lists), and the second value returned is the symbol **:list**. The next call will read the object following the end of the list. If *return-list-p* is false, however, then the first value returned is **nil** and the second **:list-begin**, and the next call will read the first object in the list.

A call to **read-ciobl** when the next thing in the stream is the delimiter for the end of a list (a close parenthesis or **<list-end>**, respectively), then the values returned are **nil** and **:list-end**, regardless of the value of *return-list-p*. If *eof-error-p* is true, then reading past the end of file signals an error, but if *eof-error-p* is false, and the input stream is at the end of file, then the first value returned is *eof-value* and the second value is **:eof**. If *return-list-p* is true, and end of file is reached within a list (*i.e.*, before reaching the delimiter that closes the list), then an error is signalled regardless of the value of *eof-error-p*. The default values for *return-list-p* and *eof-error-p* are both true.

---

**Rationale:** The *return-list-p* feature is provided so that files may be read without actually creating the intermediate CIOBL form in memory. For example, when a ptnode is to be read, instead of reading the list that describes the node and extracting the fields of the list, the elements can be read one at a time and placed directly into **ptnode**

structures.

The reason that reading an end-of-list delimiter outside a list does not signal an error even when *return-list-p* is true is so that you can read the elements of a list and have lists returned when the elements are themselves lists, but also be able to easily detect when the list whose elements you are reading is finished.

While the status value obviates the need for *eof-value*, it and *eof-error-p* are provided for consistency with other Common Lisp reading functions.

---

**write-ciobl** *object ciobl-stream* *[Function]*

**write-ciobl** writes *object* to *ciobl-stream* in the appropriate format. An error is signalled if *object* is not a valid object in CIOBL.

---

Deficiency: A method is needed for automatically escaping to user-provided code when non-CIOBL data types are encountered by **write-ciobl**.

---

**write-ciobl-list-begin** *ciobl-stream* *[Function]*
**write-ciobl-list-end** *ciobl-stream* *[Function]*

These functions are useful for writing a list to a file without actually building the list in memory and passing it to **write-ciobl**. It is important that every call to **write-ciobl-list-begin** be followed by a matching call to **write-ciobl-list-end**, or an invalid file will result. Remember, too, that an empty list *must* be written as the symbol **nil** and never as a list-begin immediately followed by a list-end.

**finish-ciobl-output** *ciobl-stream* *[Function]*
**force-ciobl-output** *ciobl-stream* *[Function]*
**clear-ciobl-output** *ciobl-stream* *[Function]*

These functions call **finish-output**, **force-output**, or **clear-output** on the Common Lisp stream underlying *ciobl-stream*. It is also legal to call the latter functions directly on the underlying stream, since the CIOBL translators do no buffering themselves.

**read-parse-tree** *ciobl-stream* **&optional** *eof-error-p eof-value* *[Function]*

Reads a parse tree, including its property list, from *ciobl-stream* and returns a **parse-tree** object. If *ciobl-stream* is at the end of file, an error is signaled unless *eof-error-p* is false, in which case no error is signaled and *eof-value* is returned. An error is always signalled if end of file is reached in the middle of a parse tree. The default value for *eof-error-p* is true. An entire parse tree file would be read by first calling **read-ciobl** to read the file's global plist, then calling **read-parse-tree** repeatedly until end of file was reached.

**write-parse-tree** *parse-tree ciobl-stream* *[Function]*

Writes *parse-tree*, which must be a parse tree object, onto *ciobl-stream*. An entire parse tree file would be read by first calling **write-ciobl** to write the file's global plist, then calling **write-parse-tree** for each parse tree to be written to the file.

`read-dataflow-graph` *ciobl-stream* `&optional` *eof-error-p eof-value* *[Function]*
`write-dataflow-graph` *dataflow-graph ciobl-stream* *[Function]*

These functions are analogous to `read-parse-tree` and `write-parse-tree`, except that they handle dataflow graph objects instead of parse trees.

## 4.5. File Naming Conventions

For consistency's sake, a few conventions are introduced for the naming of compiler input/output files. The "type" field of file's pathnames (sometimes called the "extension") indicates what kind of information is found in the file. For files which contain information representable in CIOBL, the type also indicates what format the data is in. If the type of a standard format file is `xxx`, then the binary format version of that file has type `xxxb`, and the verbose format version has type `xxxv`.

Standard format parse tree files have type `pt`. Standard format dataflow graph files have type `dg`. The binary format and verbose format types are constructed as described above. Compiler listings, which contain the source code along with error messages, statistics, line numbers, and the like, have type `listing`.

# 5. Miscellaneous

## 5.1. Errors

Compiler modules may detect errors or other conditions that require some indication to the compiler user. The following function is provided for making such indications. Note that this is completely orthogonal to Common Lisp's error system; this facility only handles the display of messages. If a compiler module invokes the error system, the action taken by the compiler may depend on whether the compiler is running interactively or not, or on other factors.

**message** *class format-string* **&rest** *format-args* *[Function]*

This function displays a message to the compiler user. The message is obtained by applying **format** to the arguments **nil**, *format-string*, and *format-args* (which yields a string). The argument *class* describes the severity of the condition which caused the message, and is used to decide whether or not to actually display the message, and where to display it (*e.g.*, console or listing file). The following classes are defined:

| | |
|---|---|
| **:unrecoverable** | An error from which the compiler cannot recover, compilation of the entire file is immediately terminated. |
| **:error** | A program error which prevents reasonable compilation. The compiler may continue to process the program (so that other errors may be detected), and may even generate code, but any results are almost assuredly incorrect. |
| **:warning** | The compiler has made an assumption about what the user intended, or has detected a situation which, while legal, probably represents a program bug. |
| **:informatory** | Anything which does not represent a program or compiler error, but which might be of interest to the compiler user. For example, a report on how well an optimization phase performed. |
| **:debug** | A message of interest only to the maintainers of the compiler. |

# 6. Acknowledgments

Every aspect of this compiler is based to some degree on the Id Compiler, Version 1, written by Vinod Kathail. The overall organization as depicted in Figure 1-1 evolved through extensive discussion with Steve Heller. The method of handling annotations arose after discussion with Rishiyur S. Nikhil. The binary format for CIOBL files is based on Symbolics' token list stream and Richard Soley's "CMC" format. Finally, the author is grateful for the enthusiasm of those members of CSG who have expressed an interest in the new compiler.

# Index