LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

A Dataflow Compiler Substrate
Revised Version 002

Computation Structures Group Memo 261-1
March 27, 1991

Ken Traub
Revised by: James Hicks and Shail Aditya

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

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

A Dataflow Compiler Substrate
Revised Version 002

Ken Traub
Revised by: James Hicks and Shail Aditya

March 27, 1991

CSG261 : Revised Version 001

This document is a revised version of the CSG Memo 261, written by Ken Traub, last revised on March 24, 1986. This is called Version 001 because it is now kept online under Version Control Conventions provided by Motorola Inc.

Even though few changes have occurred in the Dataflow Compiler Substrate, the original document needed a major revision since it had some important sections missing. This document has new chapters on defining compilers and syntax directed attribute management to fill that gap.

Also, due to a recent wave of enthusiasm and concern within the Computation Structures Group regarding more complete documentation of the software systems, the organization of this document has also undergone some change. Now, this document not only serves as a reference guide for what external LISP functions are provided in the DFCS package, but also describes in words, the structure and the mechanisms employed in providing those facilities. We also attempt to document any useful debugging tools and internal check points that may help an advanced compiler hacker.

James Hicks
Shail Aditya
January 15, 1991

CSG261 : Revised Version 002

This revised version incorporates suggestions and comments furnished by Christine Flood, R Paul Johnson, and Boon Seong Ang, as well some other minor changes.

Chapter 2 has several changes. Discussion on compiler families, modules, and compilers has been revised. Some examples and figures have been included. A brief description of cycling the compiler has been included. A small walk through example illustrates the dynamic scheduling imposed by the substrate.

Chapter 4 now includes several examples drawn from a simple expression grammar. The discussion of unimplemented graph attributes in section 4.3 has been concretized.

Chapter 6 incorporates some minor bug fixes.

Shail Aditya
March 27, 1991

Contents

1 Introduction **7**
- 1.1 Overview . . . 7
- 1.2 Common Lisp . . . 10
- 1.3 Packages . . . 10

2 Defining Compilers **11**
- 2.1 Static Structure of the Compiler Substrate . . . 11
  - 2.1.1 Compiler Family . . . 11
  - 2.1.2 Compiler Options . . . 13
  - 2.1.3 Compiler Modules . . . 14
  - 2.1.4 Compilers . . . 18
- 2.2 Dynamic Control Structure of the Compiler Substrate . . . 19
  - 2.2.1 The Dynamic Environment of a Compiler . . . 20
  - 2.2.2 Cycling the Compiler . . . 20

3 Data Structures **25**
- 3.1 Lexical Tokens and Parse Trees . . . 25
  - 3.1.1 Lexical Tokens . . . 25
  - 3.1.2 Parse Tree Nodes . . . 26
  - 3.1.3 Places . . . 30
  - 3.1.4 Parse Trees . . . 31
- 3.2 Dataflow Graphs . . . 32
  - 3.2.1 Instructions . . . 32
  - 3.2.2 Instruction Sources and Sinks . . . 36
  - 3.2.3 Arcs . . . 37
  - 3.2.4 Frames . . . 39
  - 3.2.5 Dataflow Graphs . . . 44

4 Syntax Directed Operations **45**
- 4.1 Parse Trees and Grammars . . . 45
  - 4.1.1 Grammars . . . 45
  - 4.1.2 Productions . . . 46
  - 4.1.3 A Grammar for the Expression Language . . . 46
  - 4.1.4 Grammar Abbreviations . . . 47
  - 4.1.5 Some abbreviations for the Expression Language . . . 47
  - 4.1.6 Traversing Parse Trees . . . 48
- 4.2 Parse Tree Attributes . . . 50

4.2.1 Attribute Declarations 50
4.2.2 Attribute Example 51
4.3 Graph Attributes (Unimplemented) 52

5 Exsym Tables: Separate Compilation Support 55
5.1 Creating Exsym Tables 55
5.2 Exsyms 55
5.3 Exsym Properties 56
5.4 Exsym Assumptions 56
5.5 Consistency Checking 57

6 External Representation 59
6.1 CIOBL Objects 60
6.2 CIOBL Tokens 61
6.3 CIOBL Streams 62
6.3.1 Creating CIOBL Streams 62
6.3.2 Reading and Writing CIOBL Streams 63
6.3.3 User Defined Objects 64
6.4 Encodings 64
6.4.1 Standard Encoding 64
6.4.2 Compressed Encoding 65
6.4.3 Binary Encoding 66
6.4.4 Character Codes 69

7 Miscellaneous 71
7.1 Errors 71
7.1.1 Message Hooks 72
7.2 Performance Metering 72
7.2.1 Space Meters 73
7.2.2 Time Meters 73
7.3 Sxhash Tables 74
7.4 Miscellaneous Functions 74

A Files of the Dataflow Compiler Substrate 75

B Acknowledgments 77

List of Figures

1.1 Block diagram of a Dataflow Compiler 8

2.1 A `parse-tree` Representation of a program at various Levels. 12
2.2 Representations, Levels, and Equivalences in `my-compiler-family`. 13
2.3 Structure of the Compiler `compiler1`. 22

3.1 Parse Tree for `Var1 + 6.847` 28
3.2 (a) A Dataflow Graph Fragment; (b) Its Internal Representation 33
3.3 A Typical Frame 40

A.1 List of files for Dataflow Compiler Substrate System 76

Chapter 1

Introduction

WARNING: This document is subject to change.

This document describes the data structures and abstractions that underlie the ID Compiler, Version 2 [3], 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.

1.1 Overview

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

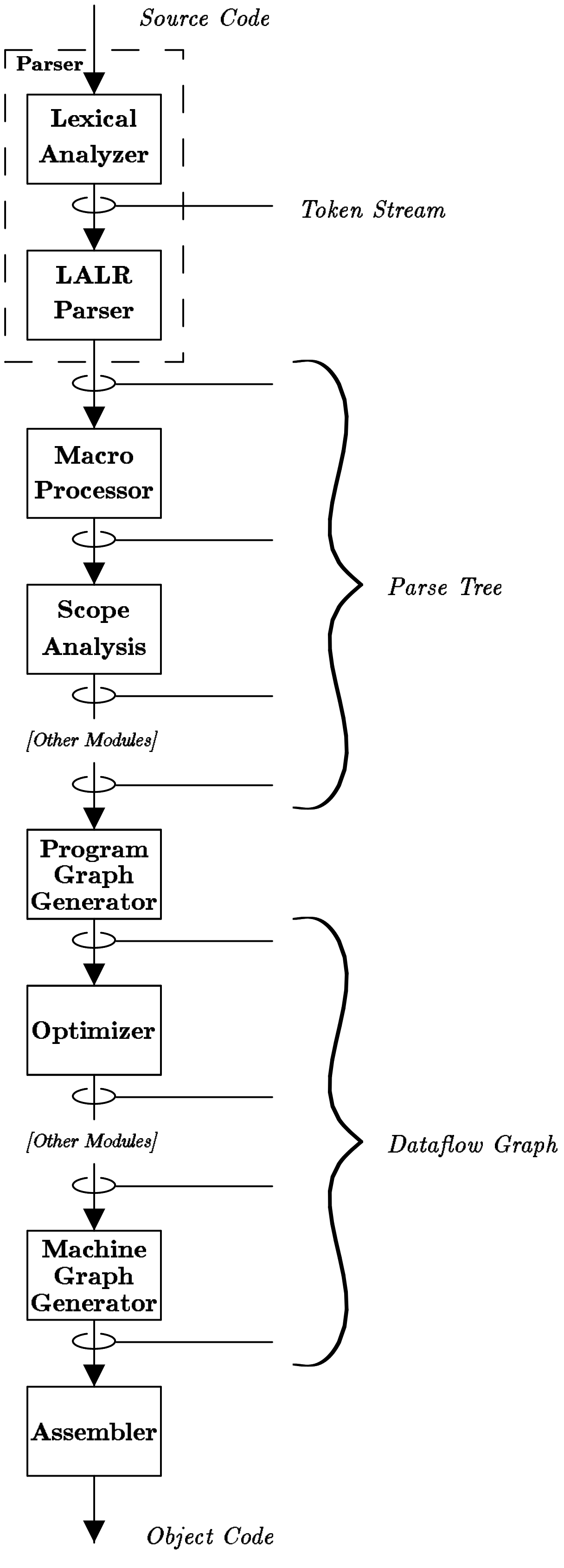


Figure 1.1: Block diagram of a Dataflow Compiler

(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 not have 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 will be a Deterministic Finite Automaton (DFA) lexical analyzer followed by a Look Ahead Left-to-Right (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.

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

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

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.

1.2 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* [1][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.

1.3 Packages

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

In addition, the symbols described in this document are exactly the external symbols of the `dfcs` package. It is intended that compiler modules will exist in other packages which use the `dfcs` 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, an `dfcs-user` package is provided which is just a package that uses the `dfcs` and the `lisp` package, with nothing in it initially.

[2]The latest version is [2].

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

Chapter 2

Defining Compilers

This chapter 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, module interdependencies *etc.* The goal is to make it easy to splice experimental modules into the compiler, handle compiler options, and the like.

We will first describe the static structure of the compiler and its components and then describe how they are glued together and exercised dynamically.

2.1 Static Structure of the Compiler Substrate

2.1.1 Compiler Family

There may be many compilers that share modules, data structures or utilities of each other. In order to facilitate such sharing in a consistent fashion, the substrate requires that compilers belong to a *Compiler Family.* A compiler family is a pool of resources that a group of compilers may share. The resources are one of the following:

- A Data Structure.
- A Compiler Option.
- A Compiler Module.

It is important to understand the significance of defining a compiler family. With a centralized place to look for compiler resources, namely the compiler family, we can independently define resources that belong to that family, and then build as many compilers as we like that use one or more of those resources. For example, in the Id Compiler Family (called `id-compiler`), we have declared various modules and options, and we use parse trees and dataflow graphs (actually program graph and machine graph abstractions build on top of dataflow graphs) as data structures. After this has been done, we can group a string of modules together and form a Id compiler that compiles from the editor into the TTDA target architecture, another compiler may compile from files, yet another one from streams. Similarly, a slightly different set of modules may be grouped together to form a compiler for the Monsoon machine.

Representation : `parse-tree`, **Levels** : `program`, `def`, `procedure`

$$
\begin{array}{c}
\texttt{program} \\
\Updownarrow \\
(\ \texttt{def}_0\ \texttt{def}_1\ \cdots\ ) \\
\Updownarrow \\
(\ (\ \texttt{proc}_{00}\ \texttt{proc}_{01}\ \cdots\ )\ (\ \texttt{proc}_{10}\ \cdots\ )\ \cdots\ )
\end{array}
$$

Figure 2.1: A `parse-tree` Representation of a program at various Levels.

Data Structure Representations and Levels (Units)

The data structures of a family are the most important resources around which the modules and the compilers are defined. The job of the modules is to generate, process, and pass on the data structures of the family.

In order to obtain a useful handle on the various kinds of data structures being passed around among the modules of a compiler family, and also to ensure consistent glueing of modules that produce and consume that data structure, a compiler family must be provided with a name of the data structure representation. A *Representation* is the name of such a data structure. This can be viewed as the "type" of the data structures flowing across a module boundary. A representation is simply a class of data structure objects and is used to ensure consistent glueing of modules. Several representation classes may, in fact, be implemented using the same Lisp data structure objects. A convenient example is the program graph and machine graph representations in the Id Compiler family, both of which are abstractions built on top of the Dataflow Graph data structure described later in this document.

A complete program may not be the best unit for independent property maintenance and execution on a given execution vehicle. A complete program would typically consist of several definitions, which may have embedded procedures that may need to be lifted out, each procedure, in turn, may need to be further subdivided into independently executable codeblocks. A data structure representation can be used to represent a complete program at one or more of these levels. A *level*, therefore, is the "type" of hierarchical clustering of the independent pieces of a complete program. Each data structure representation also specifies the names of all the levels it supports. The complete program can be specified using the representation data structure at the highest level (usually program level) supported by the representation, or equivalently, as lists of representation objects at lower levels. As an example, consider figure 2.1. The representation `parse-tree` has three levels as shown. A complete program is represented as either a single parse tree at the `program` level, or a list of parse trees each at the `def` level, or a list of list of parse trees each being at the `procedure` level.

When there are multiple data structure representations in a family, as is often the case, we also need to specify which levels of a representation correspond to which levels of another representation, so that it may be reasonable for a module to take input of one representation level and produce output of an equivalent representation level. An equivalence class of such representation and level pairs is also called a *unit*. In the following discussion and examples, we will use the term "unit" to refer to a particular representation and level combination from its equivalent class, or the class itself, interchangeably.

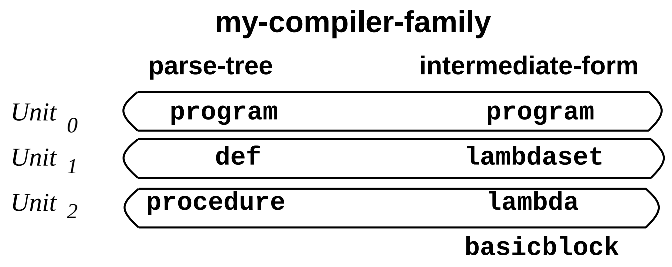


Figure 2.2: Representations, Levels, and Equivalences in `my-compiler-family`.

Declarations

`defcompiler-family` *family* `&clauses` [*Macro*]
{(`:representation` *representation* {*level*}+)}+
{(`:equivalence` {(*representation level*)}+)}*

Defines a compiler family *family* along with its data structure clauses. Each `:representation` clause defines a representation named *representation* with the given level names. The levels appear in ascending order of size. There must be at least one representation clause having at least one level. The `:equivalence` clauses defines certain pairs of representation and level to be equivalent. Errors are signalled if the system of equivalences is not reasonable.

Example of a Compiler Family

As an example, consider the following compiler family declaration:

```
(defcompiler-family my-compiler-family
   (:representation parse-tree procedure def program)
   (:representation intermediate-form basicblock lambda lambdaset program)
   (:equivalence (parse-tree procedure) (intermediate-form lambda))
   (:equivalence (parse-tree def) (intermediate-form lambdaset))
   (:equivalence (parse-tree program) (intermediate-form program)))
```

This family has two representations, `parse-tree` and `intermediate-form`, along with their various levels. The equivalences of levels into units is depicted in figure 2.2.

2.1.2 Compiler Options

An *Option* is a piece of external data that a module may use during the course of its computation. This may be viewed as an external parameter passed to the module from the environment under which the module is invoked. Typically, an option may be used by one or more modules, or by the particular compiler being constructed, and therefore, is an independent resource of the compiler family.

Declarations

`defcompiler-option` *name family* `&clauses (:type` *type*`)` [*Macro*]
[`(:how-defined` *how-defined*`)`] [`(:default` *default*`)`]
[`(:mentioned-default` *mentioned-default*`)`]
[`(:documentation` *documentation-string*`)`]
[`(:importance` *importance-number*`)`]

`*option-types*` [*Constant*]

`defcompiler-option` adds an option named *name* to the family named *family*. The option must be one of the legal types. `*option-types*` denotes the set of recognized legal option types. These are `:symbol`, `:string`, `:number`, `:integer`, `:date`, `:pathname`, `:boolean`, `:enumeration`, and `:stream`.

how-defined, if present, specifies how this option will be defined in the argument list of the compiler in which it is included. The possible values are `:keyword` or `:positional`. It defaults to `:keyword`. Keyword options must have a `:default` clause which is specified as *default*. The *documentation-string* describes the option. The *importance-number* is a number between 0.0 and 1.0, and is used to sort the options into an argument list of the compiler, from the highest to the lowest. It defaults to 0.5.

`option` *option-name* [*Macro*]

Returns the current value of the option named *option-name*. This can be used within the modules of the compiler to access the various options.

`option-exists-p` *option-name* [*Macro*]

Returns a boolean value reflecting whether the option *option-name* is a valid option for the current compiler or not.

Example of a Compiler Option

Here is an example of an option that may be used by a parser module of the compiler.

```
(defcompiler-option input-file my-compiler-family
  (:type :string)
  (:how-defined :positional)
  (:default *default-input-filename*)
  (:importance 1.0)
  (:documentation "Pathname of Source Program"))
```

This defines a positionally specified string option with maximum importance (will probably become the first argument of the compiler in which it is used).

2.1.3 Compiler Modules

A *Module* is an abstraction that packages up a particular phase of compilation. It specifies the data structures that flow across its input and output boundaries, their representations and levels, compiler options that can be used to conditionalize the code inside the module, and points to appropriate toplevel functions that initialize, trigger, and cleanup that phase of compilation.

A module operates on instances of the data structures specified within a compiler family. When a module is provided with an object in a particular representation at a particular level at its input boundary, it produces another object at the same level in the same or equivalent representation at its output boundary. A module may only have an output data structure boundary, in which case, a fresh object is generated by the module and placed at its output boundary each time the module is triggered. Such a module is called a *generator*. Conversely, a module may only have an input data structure boundary, in which case, when an object is placed at its input boundary, the module consumes it and does not produce any output object. Such a module is called a *collector*. A module with both input and output data structure boundaries specified is called an *intermediary*.

Operating Levels of a Module and Level Marking

Different modules may operate on different levels of the same or equivalent representations. Operating at lower levels than the highest possible level (*e.g.*, `program` level) permits incremental compilation of the program. The program may be compiled definition by definition or procedure by procedure, if the compiler modules have the capability to operate at those levels. This reduces resource requirements of the compiler and improves throughput, because the entire program need not be available during the various phases of compilation. Examples of such modules and their operation is described in section 2.2.2.

When there are modules that work on levels other than the highest level of a representation, it is necessary to figure out how to group their output together to form higher levels. For example, a module producing parse trees procedures must some how "mark" the end of a def and a program in the sequence of procedures it produces. Usually, this grouping is obtained by carrying through the same grouping that existed on input to the module. For example, suppose the input to a module working at the `basicblock` level is as follows:

```
 ( ( ( A B ) ( C ) ) ( ( D ) ) )
pr d p     p p   p d d p   p d pr
```

Here, A, B, C, and D are basic blocks. The grouping of basic blocks into procedures is logically indicated by the "p" annotated parentheses; the grouping of procedures into defs is indicated by the "d" annotated parentheses, and the grouping of the whole program is indicated by the "pr" annotated parentheses. Note that we invoke the module on a basic block at a time, so the output will be A', B', C', and D' respectively for each of four calls we make to the module's function. The substrate logic automatically inserts the "p", "d", and "pr" boundaries in a manner corresponding to their positions in the input to the module, so that the output will look like:

```
 ( ( ( A' B' ) ( C' ) ) ( ( D' ) ) )
pr d p       p p    p d d p    p d pr
```

But there are two situations where one cannot obtain this grouping information just from looking at the module's input. These are,

1) If the very first module in the compiler (the generator) works at a lower level than the whole program, then boundaries of its higher levels cannot be automatically inferred.

2) If the output of a module is in a different representation than the input, and there are levels in the output representation that have no corresponding equivalent level in the input.

In both of these cases, the module must explicitly mark the boundaries of levels that the system will not be able to figure out from the input automatically. In the first situation, the generator must mark all higher levels than the level of its output; in the second situation, the module must mark all levels that are both higher than its output and for which there is no corresponding level in its input representation.

Module *Before* and *After* Functions

Another consequence of having modules operate at lower levels than the highest program level is that global data structures used in a module operating at lower levels need to be setup prior to invoking the module on an object at that level. For this purpose, a module may specify functions that will be invoked once at the appropriate higher level to setup and cleanup such global data structures. These functions are called the *before-functions* and *after-functions*.

The before and after functions may be used to setup and cleanup data structures that need to be maintained at a level higher than the module's operating level. Typically, such data structures dynamically accumulate properties over multiple objects at the module's operating level. Using representation example from the last section, for a module operating at the `def` level, we may setup a global data structure using a before-function at the `program` level. The module will accumulate/use global properties in this data structure pertaining to each definition it processes. An after-function at the `program` level can then cleanup the data structure when all the definitions have been processed.

Declarations

`defcompiler-module` *name family* `&clauses` [`(:input` *representation level*`)`] [*Macro*]
[`(:output` *representation level*`)`] `(:function` *function-name*`)`
{`(:before-function` *unit* [*function-name*]`)`}*
{`(:after-function` *unit* [*function-name*]`)`}*
[`(:levels-marked` {*level*}+`)`] [`(:options` {*option*}+`)`]
[`(:wrapper-macro` *macro-name*`)`]

Defines a module *name* in the compiler family *family* along with various clauses. At least one of `:input` or `:output` clauses must be specified. `:function` clause must always be specified. All other clauses are optional.

`:input` and `:output` clauses specify the representation and the level of the data structures that the module consumes and produces respectively; if either is omitted, the module is assumed to be a generator or collector, as appropriate. The input and output representation and levels must be equivalent, which means that they must have been declared as such in the module's family. In addition, they must satisfy glueing rules as described later in section 2.1.4.

`:function` clause specifies *function-name* to be the toplevel function which implements the module. This is used to invoke the given module with a data structure object of the input representation at the specified input level, and must produce a data structure object of the output representation at the output level.

`:before-function` and `:after-function` clauses may be used to make the module aware of boundaries of levels higher than its input level. The *function-name* specified in a `:before-function` clause is invoked before an object belonging to the given unit is passed on to the module for processing. Similarly, the `:after-function` clause specifies the function to be invoked at the end of processing an object belonging to the given unit. Unit may be of the form (*representation level*) where *representation* is the input representation and *level* is a higher level than the module's input level. It could also be specified as simply *level*, where the input representation is assumed.

Even though several `:before-function` and `:after-function` clauses may be supplied, there should be atmost one `:before-function` or `:after-function` clause per unit. For generator modules, atmost one `:before-function` and `:after-function` clause is permitted, and *unit* is omitted; those clauses refer to functions invoked before and after the entire compilation.

`:levels-marked` clause gives levels of the output representation higher than the modules output level which the module function (main or before or after) "marks" by calling `mark-level` when a boundary for that level is to be recorded.

The `:options` clause names compiler options required by the module; they must have already been defined by `defcompiler-option`. Finally the `:wrapper-macro` clause, if present, sets up *macro-name* as a wrapper macro for the module when it is used in a compiler. Such a wrapper can be used to set up a dynamically scoped environment inside which the invocation of the module is placed. We will describe these wrappers in more detail in section 2.2.1.

`mark-level` *level-name* `&optional (`*when* `:before)` [*Function*]

`mark-level` may be called by module functions to record the boundaries of the level *level-name*. This level must be one of the levels of the module's output representation. For main functions of the modules, *when* should be either `:before` or `:after`, indicating that the separator is intended before or after the output produced by the module. For before and after functions, *when* is ignored.

Examples of Compiler Modules

Below we present some examples of module declarations. We will use the family and the option declarations given earlier.

```
(defcompiler-module def-parser my-compiler-family
  (:output parse-tree def)
  (:levels-marked program)
  (:before-function initialize-lexer-parser)
  (:function parse-def)
  (:wrapper-macro parser-wrapper)
  (:options input-file))

(defcompiler-module def-xform my-compiler-family
  (:input parse-tree procedure)
  (:output intermediate-form lambda)
  (:function xform-def)
  (:before-function program xform-before-program)
  (:after-function program xform-after-program))

(defcompiler-module program-printer my-compiler-family
  (:input intermediate-form program)
  (:function print-program))
```

We have defined three modules. `def-parser` is a generator that generates parse trees at the `def` level. In order to inform the rest of the modules when the program level has finished, it will mark the `program` level during its execution when the end of the input file has reached. The before function is run before the entire compilation and it will be used to initialize global data structures used across the whole program. The wrapper macro, in this case, may be used to open the input file accessed via the compiler option `input-file` given in the last section.

The `def-xform` module converts a parse tree procedure into a intermediate-form lambda. It uses before and after function at the program level to setup and cleanup its internal global data structures.

Finally, the `program-printer` module is a collector that prints the entire program at once.

2.1.4 Compilers

With all the resources in hand, it is now possible to start building compilers. A *Compiler* is a sequence of modules, the first of which must be a generator and the last one must be a collector, all the others are intermediaries. Each of the internal boundaries between pairs of modules is provided with a glueing mechanism that takes care of the representations and the levels of data structures that flow through them. The first module must obtain its input specially from a source code stream, which is usually an editor buffer or a file. Similarly, the last module must output object code specially into an object code stream which is typically a file or a special stream into the loader. Therefore, in the compiler substrate, we do not worry about these overall input and output streams, and leave their specification upto the modules that handle them which may use compiler options to specify them.

Each module must follow the following glueing rules.

1) A module's input level must be equivalent[1] to its output level. Its input representation can be different from its output representation, in which case the module acts like a translator for the program from its input data structure representation to its output data structure representation.

2) A module's output representation must match the subsequent module's input representation. But, a module's output level can be different from the input level of the subsequent module. The substrate takes care of such mismatch of levels by appropriately queuing up objects collected from a previous module and grouping them into a level acceptable to the subsequent module.

Apart from the sequence of modules, a compiler also specifies what options are being used by the compiler. These are a union of all the options used by its component modules and any other specifically declared within the compiler declaration.

Declarations

`defcompiler` *name family* `&clauses` [(`:wrapper-macro` *macro-name*)] [*Macro*]
[(`:message-hook` *hook-name*)] [(`:options` {*option*}+)]
[(`:lambda-list` {*argument*}*)] [(`:option-default` *option default*)]
`&body` *modules*

Defines a compiler *name* belonging to the compiler family *family* with the given sequence of modules. The restrictions on modules (generator, intermediary, ..., collector; matching repre-

[1]Either both should have the same representation and level or they must have been declared as equivalent in the corresponding compiler family.

sentations) are enforced. The various optional clauses are used in building the toplevel compiler invocation function.

`:wrapper-macro` clause supplies a macro name for the entire compiler to be wrapped into. `:message-hook` clause allows the hook function *hook-name* to be invoked whenever the compiler generates a message. More will be said about error messages later.

`:options` clause specifies what options are to be included in the compiler apart from those that are used by the modules of the compiler. `:option-default` permits changing the default of the specified option for the purpose of this compiler. `:lambda-list` permits the specification of a LISP-like lambda list for the compiler invocation function. The list is checked for the presence and proper specification of the options of the declared compiler.

Example Compiler

We continue with our examples of `my-compiler-family`.

```
(defcompiler-option log-file my-compiler-family
  (:type :stream)
  (:how-defined :keyword)
  (:default *trace-output*)
  (:importance .2))

(defcompiler my-compiler my-compiler-family
             ((:wrapper-macro my-compiler-wrapper)
              (:message-hook my-compiler-message-hook)
              (:options log-file)
              (:option-default log-file *log-file*))
  def-parser
  def-xform
  program-printer)
```

The option `log-file` indicates a stream where the compiler may send log messages. It is defined using a keyword with a low importance.

The compiler definition specifies a wrapper macro which may be used to set up the lexical environment of the whole compiler. It may also be used to open the log file for output. The specified message hook function will use this log file to record messages from the compiler. The `log-file` option and its overriding default are specified with the compiler declaration since they do not pertain to any specific module. Finally, we have the three modules glued together. Note from their earlier definitions that their input and output representations and levels satisfy the glueing rules. The top-level function generated by the above macro will be called `my-compiler` and it will have arguments specified by the various options present in the above compiler declaration and the declarations of its component modules. In this case, `my-compiler` will have two arguments, `input-file` option used by the first module generates a positional argument, and the `log-file` option generates a keyword argument.

2.2 Dynamic Control Structure of the Compiler Substrate

In this section, we will describe how the various pieces of a compiler as defined in the last section are made to operate dynamically in conjunction. Both the dynamic environment of a compiler and the scheduling of its various modules are controlled by the substrate.

2.2.1 The Dynamic Environment of a Compiler

The compiler substrate is responsible for setting up the dynamic environment under which each module of a compiler is invoked. This is achieved using the declared wrapper macros of the compiler and the individual modules, as well as other dynamic environment initializations described below, inside which an invocation to cycle the compiler is placed.

The definition of a compiler using the above macro `defcompiler` generates a LISP function by the given name and its argument list is generated using the options and clause specifications. This function glues the specified modules in the given order. There are several wrappers and variable bindings that are set up during this glueing and it is worthwhile mentioning some of them.

The outermost wrapper is the code that computes and sets the options. Code to generate default values is used for an unspecified option. Next, an error catching wrapper catches unrecoverable user errors as well as compiler bugs that would abort the compilation and may otherwise throw the user into the LISP debugger. Inside this the compiler message hook is set up and the declared compiler wrapper is applied. Errors and messages are described separately in chapter 7. Finally, the individual module wrappers are applied nested from the wrapper of the first module to the last. Inside all these wrappers lies the call to invoke the compiler.

2.2.2 Cycling the Compiler

The substrate also controls to a first degree the scheduling of the individual modules of a compiler. The overall scheduling order of the modules is chosen to be depth-first incremental, which means that the individual units of source code will be cycled through the sequence of modules as far down as possible from the front-end to the back-end before another unit of source code is pushed into the front end. But this depth-first scheduling strategy must obey the operating levels of individual modules, which gives it enough flexibility to simulate any combination of depth-first and breadth-first schedule.

Declarations

`current-module-name` [*Function*]

Returns the name of the currently executing module.

A Complete Example

To make things clearer, we will walk through an example compiler as defined below.

```
(defcompiler-family family1
  (:representation rep1 def program))

(defcompiler-module def-parser family1
  (:output rep1 def)
  (:levels-marked program)
  (:before-function print-before-compile)
  (:after-function print-after-compile)
  (:function parse-def))
```

```
(defvar *asdf* 0)
(defun print-before-compile () (format t "~&Printing before compile"))
(defun print-after-compile () (format t "~&Printing after compile"))

(defun parse-def ()
  (let ((data (incf *asdf*)))
    (when (zerop (mod data 5))
      (mark-level 'program :after))
    (format t "~&Parsing def ~a" data)
    data))

(defcompiler-module program-xform family1
  (:input rep1 program)
  (:output rep1 program)
  (:function xform-program))

(defun xform-program (program)
  (format t "~&Xforming program ~a" program)
  program)

(defcompiler-module def-analyser family1
    (:input rep1 def)
    (:output rep1 def)
    (:before-function program print-before-program)
    (:after-function program print-after-program)
    (:function analyse-def))

(defun print-before-program ()
  (format t "~&Print before program in ~a" (current-module-name)))
(defun print-after-program ()
  (format t "~&Print after program in ~a" (current-module-name)))

(defun analyse-def (def)
  (format t "~&Analysing def ~a" def)
  def)

(defcompiler-module def-printer family1
  (:input rep1 def)
  (:function print-def)
  (:before-function program print-before-program)
  (:after-function program print-after-program))

(defun print-def (def)
  (format t "~&Printing def ~a" def))

(defcompiler compiler1 family1 ()
    def-parser
    program-xform
```

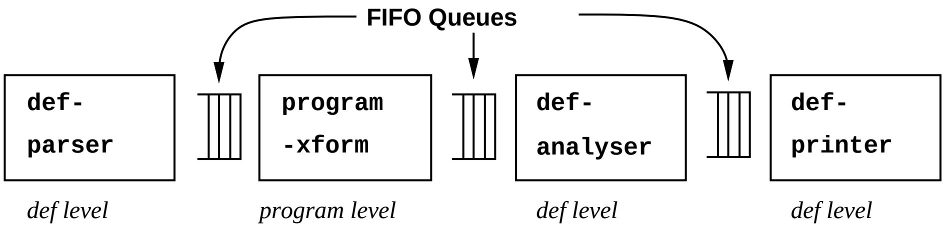


Figure 2.3: Structure of the Compiler `compiler1`.

```
def-analyser
def-printer)
```

The internal structure of the compiler `compiler1` is pictorially depicted in figure 2.3. It consists of four modules whose operating levels are as defined. Module `def-parser` is a generator generating `def`s (which are actually simply integers!). Note that since `def-parser` operates at a lower level than the highest `program` level, it must mark the end of a `program` after emitting a certain number of `def`s. We simply choose to do so after every 5 definitions emitted.

The `def`s emitted by the `def-parser` module have to be collected by the substrate to form a `program` and fed to the next module `program-xform` because that operates at the `program` level. Modules `def-analyser` and `def-printer` operate at the `def` level, therefore, the program emitted out from the `program-xform` module needs to be split into `def`s and supplied to them definition by definition. Finally, the module `def-printer` is a collector operating at the `def` level. The substrate inserts the virtual FIFO queues between the modules so that different operating levels can be matched.

A call to run the compiler `compiler1` produces the following output.

```
> (compiler1)
Printing before compile
Parsing def 1
Parsing def 2
Parsing def 3
Parsing def 4
Parsing def 5
Xforming program (1 2 3 4 5)
Print before program in DEF-ANALYSER
Analysing def 1
Print before program in DEF-PRINTER
Printing def 1
Analysing def 2
Printing def 2
Analysing def 3
Printing def 3
Analysing def 4
Printing def 4
Analysing def 5
```

```
Printing def 5
Print after program in DEF-ANALYSER
Print after program in DEF-PRINTER
Printing after compile
NIL
>
```

We discuss the various phases below.

1) First, the before function of the generator module `def-parser` is executed before the entire compilation.

2) Now the compiler enters a loop to execute modules in a depth-first fashion. Since there is nothing to do, the generator is triggered.

3) The generator produces a definition, but since module `program-xform` requires the complete program, this definition will just accumulate in the input FIFO queue of module `program-xform` until the complete program is available.

4) The generator must mark the end of the program, since there is no other way of knowing it. It will presumable do so after the end of its input file or stream. Once the last definition has been generated and the end of the program has been marked, the substrate has enough objects in the queue to invoke the module `program-xform` on the complete program. Note that, when a module operates at a higher level than its previous modules, it is not scheduled until all the objects comprising its operating level have been accumulated. This effectively amounts to a breadth-first scheduling of that module.

5) When module `program-xform` is done, it places the complete program on its output queue. At this point, a program level before-function for module `def-analyser` will be executed. The substrate now attempts to push the objects in the queue as far down into the compiler as possible. As soon as the first definition crosses the module `def-analyser`, it can be passed to the module `def-printer`, since that operates on the def level as well. Of course, before the module `def-printer` starts processing the first definition its program level before-function will be executed.

6) Definitions residing at the output queue of the module `program-xform` will be pushed through modules `def-analyser` and `def-printer` one by one until all of them are done.

7) Note that the after functions of the modules are executed when the appropriate level boundary passes through the module[2].

8) Finally, the after function of the generator is executed after the whole compilation.

[2]There is a slight difference between the scheduling of the after function and the corresponding before functions with respect to the scheduling of the intervening modules, as it may be evident from the given example. Whether this is a bug or a feature, it is not clear.

Chapter 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*[1], 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.

Selectors

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

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

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`.

`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`.

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.

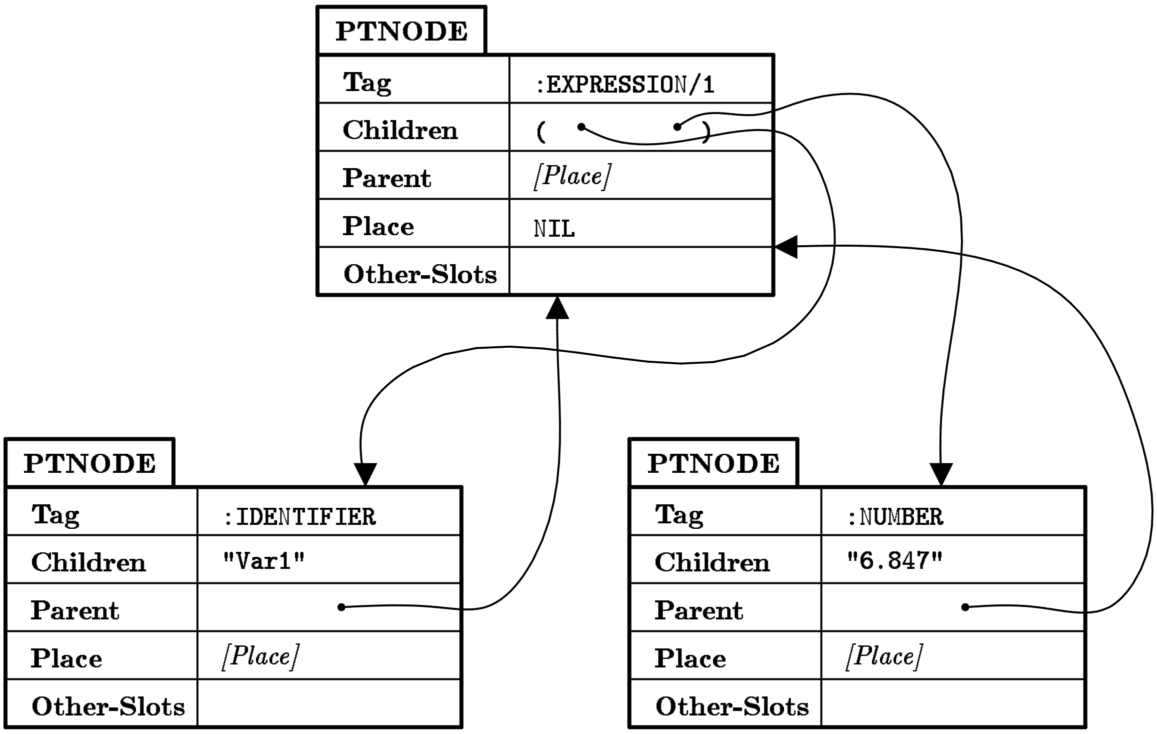


Figure 3.1: Parse Tree for `Var1 + 6.847`

Selectors

`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 *ptnode*s 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.*).

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`.

Mutators

In addition to the functions below, all of the selectors in Section 3.1.2 and any selectors defined by `define-ptnode-slot` may be used with `setf`.

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

Replaces the nth 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.)

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.

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.]

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.

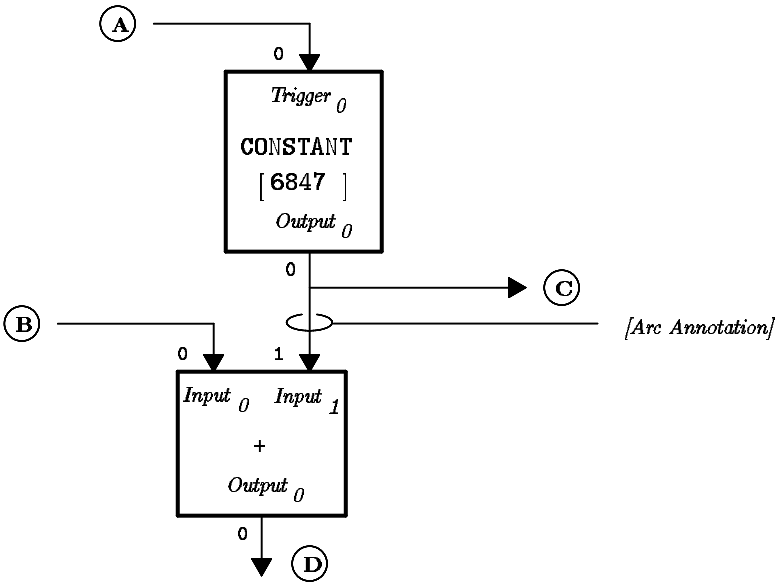
3.2.1 Instructions

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*.

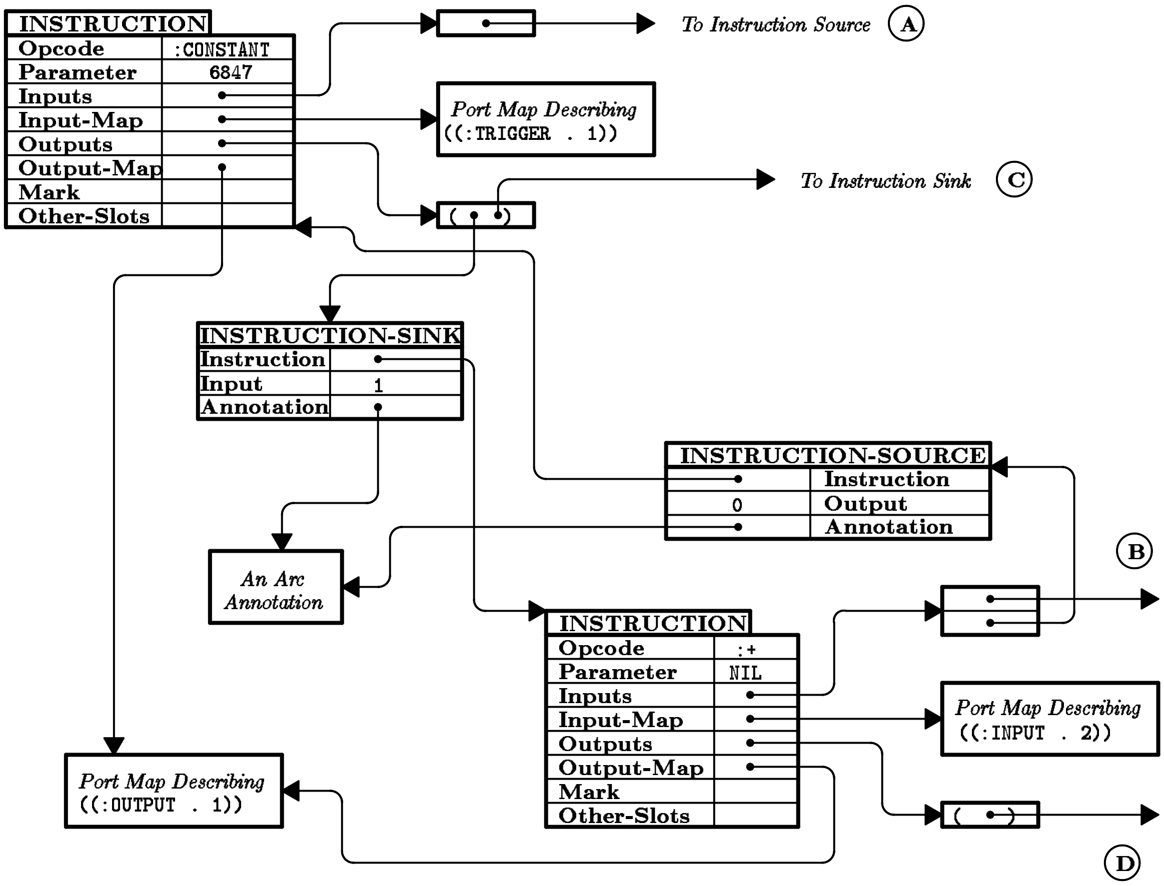
A dataflow graph is constructed by wiring the outputs of some instructions to the inputs of other instructions; it is not possible to wire outputs to outputs or inputs to inputs. An additional restriction is that while outputs may have any number of arcs leading away from them, each input may have only 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 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



(a)



(b)

Figure 3.2: (a) A Dataflow Graph Fragment; (b) Its Internal Representation

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, 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.

`remove-any-arc` *instruction input* [*Function*]

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

`remove-all-arcs` *instruction output* [*Function*]

If there are arcs from output *output* of instruction *instruction*, `remove-all-arc` removes them by removing all the sinks from *instruction*'s output and removing the sources from each of the destination instruction's inputs.

`move-any-arc-destination` *old-instruction old-input new-instruction new-input* [*Function*]

If there is any arc to input *input* of instruction *instruction*, `move-any-arc-destination` moves it by modifying the sink in the source instruction's output and moving the source from *old-instruction*'s input to *new-instruction*'s input *new-input*.

`move-all-arc-origins` *old-instruction old-output new-instruction new-output* [*Function*]
If there are arcs from output *output* of instruction *instruction*, `move-all-arc-origins` moves them by moving all the sinks from *old-instruction*'s output to *new-instruction*'s output *new-output*, and modifying the sources in each of the destination instructions' inputs.

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 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

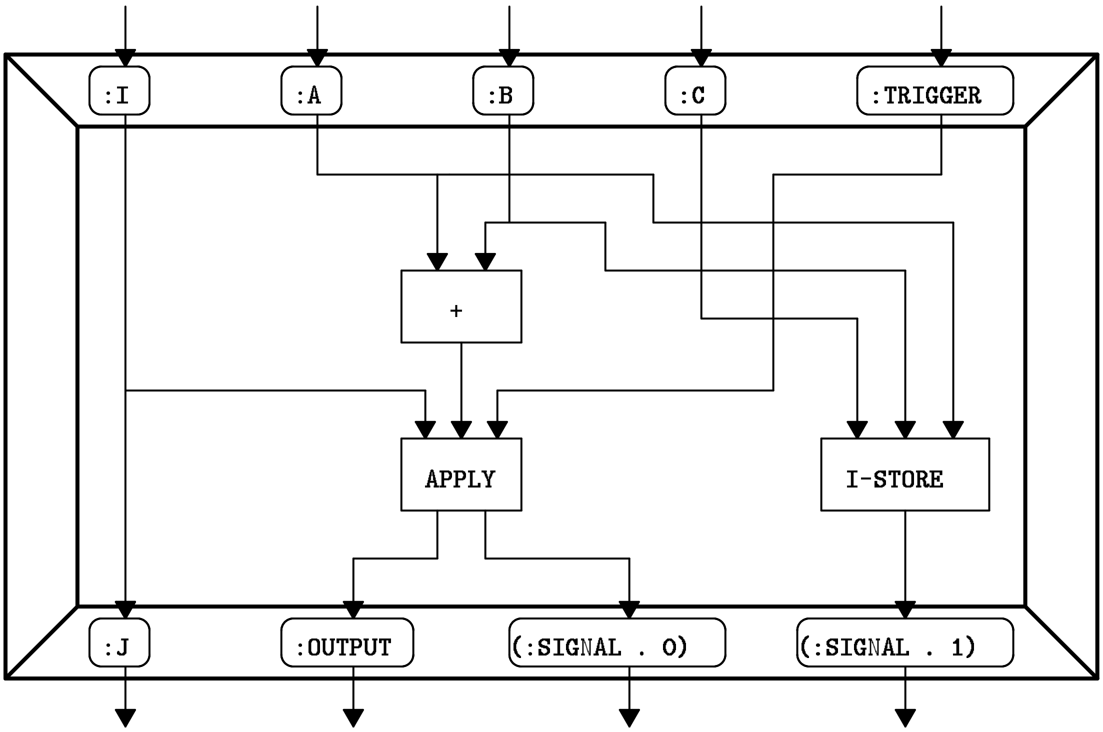


Figure 3.3: A Typical Frame

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` *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` *frame input instruction input* `&optional` *annotation* [*Function*]
`wire-frame-input-to-sink` *frame input sink* `&optional` *annotation* [*Function*]
`wire-instruction-to-frame-output` *instruction output-1 frame output-2* `&optional` *annotation* [*Function*]
`wire-instruction-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 39: 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`.

`with-instruction-array` (*var dataflow-graph* `&optional` *number-p*) `&body` *body* [*Macro*]

Within the body of the `with-instruction-array` form, the variable *var* will be bound to an array containing all of the instructions of *dataflow-graph*. If *number-p* is true, then the instructions will be numbered such that the `instruction-number` of the ith element of the instruction-array will be i. When control leaves the body of the `with-instruction-array` form in any way, the instruction-array will be deallocated. Therefore, the instruction array may not be passed outside the dynamic scope of the `with-instruction-array`.

Chapter 4

Syntax Directed Operations

4.1 Parse Trees and Grammars

In this section we will describe syntax-directed operations on parse-trees and parse-tree-nodes that can be performed in DFCS. As a running example, we will use the simple expression grammar given below (in abstract syntax):

$$
\begin{aligned}
\text{expression} &::= \text{plus-expression} \\[1em]
\text{plus-expression} &::= \text{plus-expression} + \text{mul-expression} \\
\text{plus-expression} &::= \text{plus-expression - mul-expression} \\[1em]
\text{mul-expression} &::= \text{mul-expression * prim-expression} \\
\text{mul-expression} &::= \text{mul-expression / prim-expression} \\[1em]
\text{prim-expression} &::= \text{number}
\end{aligned}
$$

4.1.1 Grammars

A *grammar* is a set of productions. A grammar has a name, and a grammarspec, the latter used to identify the grammar within a production specification, also called a *prodspec*. For example, consider the following prodspec:

```
(expression  -> "(" expression ")")
```

The grammarspec in this example is the symbol `->`. By having different grammarspecs, several grammars can be in use; typical grammarspecs might be `-ID->` *etc.*

All productions have a tag which identifies them; the tag is a keyword symbol. For example, the above production may have a tag `:expression/1`. Grammar keeps a mapping from the prodspec to the production tag in a hash table. Finally, a grammar may have some attributes associated with it.

`defgrammar` *grammarname grammarspec* [*Macro*]

Define a grammar named *grammarname*, which is identified by the symbol *grammarspec* in production specifications.

4.1.2 Productions

A production is a structure for recording information about a reduction in the grammar. In addition to prodspecs and tags, a production may also have *properties* and *attributes*.

The tags of all productions are unique, even if they are in different grammars. This is for efficiency; if two productions in different grammars could have the same tag, then to identify a ptnode you would have to do a two-level dispatch on the tag and the grammar.

A prodspec is used to specify a production, and sometimes to associate names or other things with components of the production. A prodspec takes the following form:

```
(lhs grammarspec rhs-1 rhs-2 rhs-3 ...)
```

e.g.,

```
(EXPRESSION -> LET (LET-BINDING-LIST BLIST) IN (EXPRESSION))
```

Here, the grammarspec is the symbol `->`, and names a particular grammar. Some of the other components have associated values, such as the symbol `blist` for `let-binding-list`. The last right hand side component explicitly has no value. The left hand side has an implied value of the symbol `expression`; thus in a prodspec, *symbol* is equivalent to (*symbol symbol*). This means that `let` and `in` each have an implied value, but that will be irrelevent if they are terminals of the grammar (*i.e.*, have no corresponding children in ptnodes represented by this production). When determining the production a prodspec names, only the symbol part of prodspec components matters.

A special case of a prodspec is when some of the rhs components have values which are integers: this is called a *template*. Each production has a template where the integers indicate child numbers for the rhs components.

`defproduction` *tag grammar-name* `&clauses` [(`:template` *template*)] [*Macro*]
{(`:properties` {{*indicator value*}}*)}*

This defines a production with tag *tag* for grammar *grammar-name*. The *template* will be a prodspec, as defined above. The *properties* clauses define properties associated with this ptnode tag; these properties are accessible via `ptnode-get`.

If there is already a production with the same tag, grammar, and template, then the properties of the existing production are replaced, otherwise a new production is defined, which may require removal of an existing production with the same tag, and removing other tags pointing to the same production.

`tag &rest` *prodspec* [*Macro*]

The `tag` macro expands into the parse tree node tag associated with the production named by *prodspec*.

`ptnode-get` *ptnode indicator* `&optional` *default* [*Macro*]

This returns the value of the property *indicator* for the production associated with the tag of the given parse tree node *ptnode*. If no value is found, then *default* is returned.

4.1.3 A Grammar for the Expression Language

```
(defgrammar expression-language ->)
```

```
(defproduction :expression/1
   (:template (expression -> plus-expression)))

(defproduction :plus-expression/1
   (:template (plus-expression -> plus-expression "+" mul-expression)))
(defproduction :plus-expression/2
   (:template (plus-expression -> plus-expression "-" mul-expression)))

(defproduction :mul-expression/1
   (:template (mul-expression -> mul-expression "*" prim-expression)))
(defproduction :mul-expression/2
   (:template (mul-expression -> mul-expression "/" prim-expression)))

(defproduction :prim-expression/1
   (:template (prim-expression -> number)))

;;; number is a pseudo-terminal
(defproduction :number)
```

Note that the names of the non-terminals in the `:template` clause of the `defproduction` forms match the abstract grammar, not the particular tags given to individual productions.

4.1.4 Grammar Abbreviations

A grammar abbreviation is similar to a production, in that it has a unique *tag* and *template*. However, it may actually name several productions. In order for an abbreviation to make sense, the productions it abbreviates should be related in some fashion.

`define-grammar-abbreviation` *tag* `&clauses` [(`:template` *template*)] [*Macro*]
(`:productions` {*productions*}*)

This defines an abbreviation for one or more productions. The *template*, if present, can be used to access the children of all the specified productions. In this case, all the productions must contain the same number of nonterminals. Different shapes of productions can be abbreviated together only if a template is not desired.

4.1.5 Some abbreviations for the Expression Language

In many cases, we could treat the addition, subtraction, multiplication, and division productions of the expression grammar in a uniform fashion, and so we would like to define and abbreviation: `any-binary-expression`.

```
(define-grammar-abbreviation any-binary-expression
   (:template (e -> (e0 0) op (e1 1)))
   (:productions
     (plus-expression -> plus-expression "+" mul-expression)
     (plus-expression -> plus-expression "-" mul-expression)
     (mul-expression -> mul-expression "*" prim-expression)
     (mul-expression -> mul-expression "*" prim-expression)))
```

The prodspec's `any-binary-expression` and `(e -> e0 op e1)` will both match against any of the `:plus-expression/1`, `:plus-expression/2`, `:mul-expression/1` or `:mul-expression/2` productions. As we shall see later, the template of the abbreviation can be very useful in these cases.

4.1.6 Traversing Parse Trees

Two mechanisms are provided to traverse a given parse tree. The first is *structural* traversal which directly accesses the parse tree's parent children links and is independent of the grammar. The second is *abstract* traversal that provides an abstract means of identifying each ptnode with its production in the associated grammar and descends through the tree according to the production reduced at a given parse tree node. Both methods have their own merits. It is expected that modules that perform structural editing of the parse tree may use the former more naturally, while the modules that annotate the tree with properties may find the latter more useful.

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

`let-ptnode-children` is used to structurally descend in the parse tree by one level. It evaluates the given forms as an implicit `progn` in a environment where each child of the ptnode *ptnode* is bound to the corresponding variable *var*.

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 *var*s. While is it is legal for *ptnode* to have more children then there are *var*s, this usage is discouraged.

The following macros can be used to perform an abstract traversal of the parse tree.

`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 (the prodspec), rather than its tag. This feature may only be used when the grammar is known at compile time and has been loaded into the Lisp world.

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
  (:number
   e0)
  ((plus-expression -> plus-expression "+" mul-expression)
   e1)
  (((mul-expression -> mul-expression "*" prim-expression)
    (mul-expression -> mul-expression "/" prim-expression))
   e2)
  (otherwise
   (print-error "Unknown ptnode tag")))
```

This is equivalent to the following:

```
(grammarcase (ptnode-tag x)
  (:number
   e0)
  (:plus-expression/1
   e1)
  ((:mul-expression/1 :mul-expression/2)
   e2)
  (otherwise
   (print-error "Unknown ptnode tag")))
```

Note that the actual symbols used in place of `:plus-expression/1`, `:mul-expression/1` and `:mul-expression/2` would depend on the grammar being used at the time.

`grammarbind` ({({*key*}*)} | *key*) *ptnodeform* `&rest` *body* [*Macro*]
`ptnode` *symbol* `&optional` *index* [*Macro*]
`n-ary-n` [*Macro*]

`Grammarbind`, like `let-ptnode-children`, sets up an environment in which one may access the children of a ptnode by name. However, in `grammarbind` the names are derived from the keys in the key list. Each key is a prodspec: either the template of a production defined by `defproduction` or the template of an abbreviation defined by `define-grammar-abbreviation`.

The form *ptnodeform* must evaluate to a ptnode. The `ptnode` macro is used within the scope of the `grammarbind` to access the children of the ptnode. The argument to ptnode is the name of one of the non-terminal children of the ptnode.

Each item in the keylist must be a list of the form (*symbol* `->` *symbol-or-string symbol-or-string ..*). This form is provided so that the children of a production may be referred to by the production's appearance, rather than by position, as in `let-ptnode-children`. `grammarbind` may only be used when the grammar is known at compile time and has been loaded into the Lisp world. Unlike in `grammarcase`, the keys in the keylist must have some similarity — each key must have the same number of non-terminals, and the corresponding non-terminals must have the same names. Furthermore, the names of each non-terminal child must be distinct — there cannot be two children of the ptnode which have the same name. For instance, if two non-terminals were named `expression`, it would be ambiguous which child is referred to by `(ptnode expression)`.

The macro `n-ary-n` is useful when the productions named in the keylist have a `$separated-by` key within them. This is a special key that permits n-way branching in the parse tree to be represented by a single production. As an example, a tuple expression may be parsed using the following production:

```
(defproduction :tuple-expression/1
   (:template (tuple-expression -> plus-expression $separated-by ",")))
```

The macro `n-ary-n` returns the number of iterations of the right hand side used at a particular parse tree node corresponding to this production. For example, the value of `(n-ary-n)` inside the scope of a `grammarbind` applied to a tuple expression parse tree node used to parse the expression `2+3,4*5,1` will be 3.

`grammarcasebind` *keyform* {({({*key*}*) | *key*} {*form*}*)}* [*Macro*]
This is a combination of `grammarcase` with a `grammarbind` in each clause.

4.2 Parse Tree Attributes

4.2.1 Attribute Declarations

`define-ptnode-attribute` *selector-name grammar-name-or-names* `&clauses` [*Macro*]
(`:type` *type*) [(`:storage` *storage*)]
{(`:productions` {*productions*}*)}*

`define-ptnode-attribute` defines a ptnode attribute. The attribute is created and added to the system, and a selector macro is defined for use by the user. The selector name must begin with `ptnode-`, for consistency with ptnode slots. As with user defined slots, the name is imported into and exported from the `dfcs` package.

The same attribute may apply to several grammars, therefore *grammar-names* is a list of the grammars for which it is defined.

Type is either `:synthesized` or `:inherited`. Synthesized attributes are computed bottom-up; they are initialized at the terminal nodes of the parse tree and the attribute value for the parent node is computed from that of its children at each intermediate parse tree node. Inherited attributes are computed top-down. They are initialized at the root of the parse tree and the attribute values of the children are computed using that of the parent at each intermediate parse tree node.

Storage is either `:ephemeral`, `:memoized`, or `:permanent`. An `:ephemeral` attribute is not memoized at all: every time an ephemeral attribute of a ptnode is fetched it is recomputed. The advantage is that it occupies no space within ptnodes. The value of a `:memoized` attribute is recorded within a ptnode after it is computed, and so multiple fetches of its value will cause at most one computation. `:memoized` attributes are invalidated if the structure of the parse tree changes, and are therefore recomputed when fetched after a change has been made to the parse tree topology. `:permanent` attributes are like `:memoized` attributes, but their values are retained even if the structure of the parse tree changes. `:permanent` attributes are intended for applications where the values of the attributes can only be computed at a particular time, but once computed continue to have meaning even when the structure of the parse tree changes. An example would be unique names for identifiers. Because new nodes can be added to a parse tree after permanent attributes have been computed, it is allowed to `setf` the values of `:permanent` attributes, for once the structure of the parse tree changes, values of already computed `:permanent` attributes will have to be maintained manually.

`:productions` clause supplies a list of production specifiers, tags, templates, or abbreviations, that specify all the productions on which this attribute is defined. There may be more than one such clause. If there is no `:productions` clause, then the attribute is defined for all productions in the specified grammars.

`defattributes` {({({*key*}*) | *key*} {*form*}*)}* [*Macro*]
{((*attribute-name ptnode* `&optional` *index-variable*) {*value-form*}*)}*
`defattributes` {({({*key*}*) | *key*} {*form*}*)}* [*Macro*]
{((`values` {(*attribute-name ptnode* `&optional` *index-variable*)}*) {*value-form*}*)}*

This macro defines the method for computing the value of attribute *attribute-name* for the the productions named by the keylist. If `values` is used on the left hand side of a clause, then the *value-form* must return the same number of values.

On the right hand sideof each clause of `defattributes` specification, the non-terminals of the productions are named and accessed in the same way as in the body of a `grammarbind`, with `(ptnode` *nonterminal-name*`)` or `(ptnode` *nonterminal-name index*`)`. The parent ptnode of the production may also be accessed using the `ptnode` macro.

The macro `n-ary-n` is also available for use when the productions named have a `$separated-by` key within them.

On the left hand side, the attribute value being computed can be named by `(attribute-name` *nonterminal-name*`)` or `(attribute-name` *nonterminal-name index*`)`. We also allow the syntax `(`*attribute-name* `(ptnode` *ptnode [index-variable]*`))` for consistency with the rhs.

The key

```
($lhs -> $rhs-component $separated-by)
```

is used to define the default production of an attribute (for an example grammarspec "`->`"). The value computed here is used for an attribute when there is no `defattributes` specification for that production.

4.2.2 Attribute Example

As an example, we will compute a synthesized attribute that counts the number of leaves and binary internal nodes in the parse tree for the example grammar given in section 4.1.3.

```
(define-ptnode-attribute ptnode-node-count expression-language
```

```
  (:type synthesized)
  (:storage :memoized))

(defattributes (e -> e0 op e1)
  ((ptnode-node-count e)
   (+ 1 (ptnode-node-count (ptnode e0))
        (ptnode-node-count (ptnode e1)))))

(defattributes :number
  ((ptnode-node-count :number)
   1))

(defattributes ($lhs -> $rhs-component $separated-by)
  ((ptnode-node-count $lhs)
   (loop for i from 0 below (n-ary-n)
         sum (ptnode-node-count (ptnode $rhs-component i)))))
```

The synthesized attribute `ptnode-node-count` is defined to have memoized storage. Note that we have used the grammar abbreviation defined in section 4.1.5 to collectively define the attribute computation for all the binary productions. Also, the default attribute definition picks up any production not covered explicitly.

4.3 Graph Attributes (Unimplemented)

[Note that this section proposes an unimplemented feature of DFCS.]

In this section, we will define a framework for declaring and computing attributes for individual input and output ports of dataflow instructions. In spirit, it is similar to the syntax directed attribute system for parse trees described in the last section.

A *graph attribute* is affiliated with one or more graph representations (abstractions on dataflow-graphs) in a compiler family. A graph attribute can be specified for either all the input ports of an instruction, or all the output ports of an instruction, or both input and output ports of an instruction. Once defined for a graph representation, an attribute specification applies to every instruction in that representation. We cannot specify attributes for subsets of instructions.

Graph attributes are essentially property slots defined for the input and/or output ports of an instruction instance. A new attribute slot is set aside for each relevent port whenever a new instruction instance is generated. There are general mechanisms to control the time of actual allocation of an attribute slot. One may choose to always pre-allocate storage for an attribute in each instruction instance generated, or one may allocate storage on demand only for those instructions that actually access the attribute. There are general mechanisms to initialize the attribute slots once they are allocated. Of course, it is possible to access and modify the attribute values residing at each port individually. We will describe all these mechanisms in detail shortly.

Apart from providing some control over the allocation and initialization of the attributes, the responsibility of computing, propagating, and maintaining the graph attributes is left to the user. There is no general mechanism to specify the automatic semantic computation of the attribute value of a port using the graph connectivity existing around it. The user must explicitly traverse the graph and set the attribute values. This situation is likely to change in the future when some attribute grammar style mechanism may be devised to allow automatic definition of attribute values of a port with respect to the graph it belongs to. In the very least, such a mechanism will have to deal with cyclic graphs and fixpoint calculations for such graphs.

`define-dfg-attribute` *selector-name* `&clauses` [*Macro*]
{(`:affiliation` *representation family*)}+ (`:type` *type*)
(`:storage` *storage*)
[`:initial-element` *initial-element* [*initial-element-type*]]

Defines a dataflow graph attribute with the name *selector-name*. The name of the attribute must be of the form *prefix*`-port-`*name*. By convention, *prefix* is used to specify the kind of graphs (the representation) this attribute belongs to; *name* is the name of the attribute. For example, the *type* attribute for each port of a program graph instruction may be named as `pgi-port-type`. The name of the attribute is also used as a selector macro for that attribute with an instruction, a port type (`:input` or `:output`), and a port name as parameters[1]. It can also be used with `setf` to modify the attribute value stored at a given port of a given instruction.

The `:affiliation` clauses record the names of the graph representation and the compiler family that this attribute belongs to. There should be at least one affiliation clause.

The `:type` clause specifies the type of ports this attribute is applicable to. It must be one of `:input`, `:output`, or `:input-output`, which mean that the given attribute is defined for the input ports only, the output ports only, or both input and output ports of an instruction, respectively.

The `:storage` clause specifies the time of allocation of storage for the attribute slots on the ports. It must be one of `:pre-allocated` or `:on-demand`. `:pre-allocated` means that an array of slots will be allocated whenever a new instance of an instruction is generated. `:on-demand` means that the array of slots will be allocated only when a selection or modification operation is attempted for that attribute on some port of the given instruction instance. This saves considerable space if the attribute is relevent for only some instructions, but may incur a slight additional runtime overhead in testing whether the attribute array has already been allocated or not.

The `:initial-element` clause, if present, specifies the initialization mechanism of the attribute when allocated. If the clause is absent, each attribute slot is initialized to the keyword symbol `:uncomputed`. When the clause is present, its interpretation depends on the *initial-element-type*. This must be one of `:static` or `:dynamic` and defaults to `:static`. If the initial element type is inferred to be `:static`, then *initial-element* must be an immediate constant. It is evaluated once at the time of attribute definition and used to initialize the port array slots whenever they are allocated. If the initial element type is specified to be `:dynamic`, then the initial element must name a function with one argument that will be called once at the time of allocating an instruction instance with that instance as an argument. That function can fill up the slot array as desired.

`initialize-dfg-attributes` *instruction representation family* [*Function*]

This initializes all the `:pre-allocated` attributes of the given instruction. Only the attributes belonging to the given representation and compiler family are initialized.

[1] We should really have just the instruction and the port name as the parameters. We have to look into this more carefully.

`wire-attribute-from-port-to-port` (*from-attribute from-instruction from-port-type from-port*) (*to-attribute to-instruction to-port-type to-port*) `&optional` *wire-function* [*Macro*]

This macro wires the value of the attribute *from-attribute* from the specified source instruction port to the attribute slot for *to-attribute* in the destination instruction port using the given wiring function. The wiring function, if unspecifed, defaults to `setf`.

`map-on-input-port-values` *function instruction attribute* [*Function*]
`map-on-output-port-values` *function instruction attribute* [*Function*]

These functions map the given function on all the input/output port values of the given instruction at the given attribute. The mapping function is called with 2 arguments, the value of the port slot and the port number. The results of the mapping are not accumulated.

`call-on-input-port-instruction` *function instruction input-port* [*Function*]
`map-on-output-port-instructions` *function instruction output-port* [*Function*]

These functions apply the given function on all the instruction ports connected to a given input/output port of a given instruction. There can be only one source instruction wired to an input port, while there may be several sink instructions wired from an output port. Therefore, the former results in a single call over the source instruction, while the latter maps the supplied function over all the sink instructions. In each case, the function is called with 2 arguments, the connected instruction and the port to which the connection is made (this is a port number rather than a name).

`with-all-port-values-of-dataflow-graph` (*var dataflow-graph attribute*) `&body` *body* [*Macro*]

This collects all the port values on all instructions of a dataflow graph for the given attribute into an array accessible through *var*. The array is available only within the scope of this macro and should not be passed outside it.

Chapter 5

Exsym Tables: Separate Compilation Support

Exsym-Tables, for EXternal SYMbol tables, are provided to support separate compilation. Exsym-tables map Lisp symbols to *exsym*s, where an exsym is a structure that maps property indicators to values. Functions are provided to create exsyms, exsym-tables, and to set and get the property values of exsyms. Exsym-tables also allow the compiler (or some other program) to ensure that a consistent set of exsyms is being used, by recording assumptions on an exsym.

5.1 Creating Exsym Tables

`make-exsym-table` [*Function*]
`exsym-table` [*Type*]

The function `make-exsym-table` makes an empty exsym-table. An exsym maps keyword symbols to exsyms.

`map-exsym-table` *fcn exsym-table* [*Function*]

Applies the function *fcn* to each entry in *exsym-table*. The function is called for side-effect with argument *name* and *exsym* for each exsym in *exsym-table*. No value is returned.

5.2 Exsyms

`find-exsym` *symbol search-path* [*Function*]
`exsym-exists-p` *symbol exsym-table-or-search-path* [*Function*]
`install-exsym` *exsym exsym-table* [*Function*]
`copy-exsym` *exsym* [*Function*]

The function `find-exsym` finds and returns the exsym named *symbol* in *search-path*, which is a list of exsym-tables. The function `exsym-exists-p` returns `t` if an exsym for *symbol* is present in *exsym-table-or-search-path*, which may be an exsym-table or list of exsym-tables. Precedence of exyms is by the order in which the containing exsym-tables occur in the *exsym-search-path*. `Install-exsym` store *exsym* in the table *exsym-table* under the name of `(exsym-name` *exsym*`)`. `Copy-exsym` returns a copy of *exsym*.

`exsym-clear` *symbol exsym-table* [*Function*]

Clears all properties and assumptions on the exsym bound to *symbol* in *exsym-table*.

5.3 Exsym Properties

`define-exsym-property` *indicator* `&clauses (:consistency-predicate` *pred*`)` [*Macro*]
`[(:consistency-encoder` *encoder*`)]`
`[(:consistency-printer` *printer*`)]`

The macro `define-exsym-property` defines a property that may be used in later calls to `exsym-get`, `exsym-put`, `exsym-assume` and `exsym-assume-value`. The consistency-predicate clause is required, but the consistency-encoder and consistency-printer clauses are optional.

Note that all values stored in an exsym either as the value of a property or as an encoding in an assumption must have CIOBL[1] read/write methods if exsym-tables or exsyms are to be written to CIOBL streams.

`exsym-get` *symbol indicator search-path* [*Function*]

Finds the exsym for *symbol* in *search-path* and returns the value of property *indicator* for that exsym. Two values are returned: the first is the value of the property or `nil` if the property is undefined or the exsym does not exist in *search-path*, the second is `t` if the exsym exists and `nil` otherwise.

`exsym-put` *symbol indicator value exsym-table* [*Function*]

Finds the exsym for *symbol* in *exsym-table*, and sets the value of *indicator* property to be *value*. An error is signaled if the exsym does not exist, or if *indicator* is not the name of a valid exsym-property.

When a property of an exsym is set, then an encoding of the value is also recorded. The default encoder function is `identity`, but other encoders can be provided by supplying a `:consistency-encoder` clause to `define-exsym-property`.

5.4 Exsym Assumptions

`exsym-assume` *assumer assumer-table assumee assumee-indicator* [*Function*]
assumee-search-path

`exsym-assume-value` *assumer assumer-table assumee assumee-indicator* [*Function*]
assumee-search-path assumee-value

The functions `exsym-assume` and `exsym-assume-value` record an assumption on the exsym for *assumer* in the *assumer-table* exsym-table. The assumption consists of *assumee*, the name of the exsym whose property is being used, *assumee-indicator*, the name of the property being used, and the value of *assumee*'s *indicator* exsym-property as found in *assumee-search-path*. An error is signaled if an exsym for *assumee* does not exist in *assumee-search-path*. `Exsym-assume-value` is similar to `exsym-assume`, except that *assumee-value* is provided directly and that an exsym for *assumee* need not exist in *assumee-search-path*.

[1]CIOBL stands for "Compiler Input/Output Base Language", and is discussed in chapter 6 in detail.

5.5 Consistency Checking

Two functions are provided for checking the consistency of a set of exsyms.

`consistency-summary` *root-exsym-names search-path* [*Function*]
`describe-consistency` *root-exsym-names search-path* [*Function*]

These functions check the assumptions recorded on each of the exsyms corresponding to *root-exsym-names* in *search-path*. An assumption about the value of property *indicator* on exsym *assumee* is checked by applying the consistency-predicate for exsym-property *indicator* to *actual-encoded-value* and *assumed-encoded-value*, where *assumed-encoded-value* was stored in the assumptions of *assumer* and *actual-encoded-value* was found in the *search-path*.

The function `describe-consistency` prints a detailed account of the inconsistencies found using the consistency-printers for each exsym-property to `*standard-output*`.

Chapter 6

External Representation

CIOBL stands for "Compiler Input/Output Base Language", and is pronounced "CHO-bul", as if it were an Italian word. It refers to a language for representing various types of data in files. The "Compiler" refers to the Id Compiler, but CIOBL has applications beyond communicating with the Id Compiler, such as representing GITA statistics. It is designed to be flexible enough to accomodate three broad categories of files:

1) Files that have structure to their data but do not imply a particular representation for the data within programs that manipulate it. Example: TTDA object code files produced by the Id Compiler. TTDA object code files have a particular structure determined in part by the Tagged-Token Dataflow Architecture, but do not require a program that uses object code — GITA, for example — to represent object code internally in any particular way.

2) Files that have a structure and also a particular representation for certain components, but whose representation is simple enough to be supported by a variety of implementations. Example: GITA statistics files. The main parts of these have a particular representation as arrays of data, but arrays are a simple enough data type that implementations other than Common Lisp can make use of them.

3) Files that have a structure and a particular representation in a particular Common Lisp program. Example: Id Compiler internal program graph files. These files contain data structures that are represented by objects defined within the Id Compiler code, and are not meant to be (easily) read by programs other than the Id Compiler.

Throughout this document, these three applications will be referred to Category I, Category II, and Category III.

The main feature of CIOBL is that it provides three different *encodings* for files, each useful for different applications. These encodings are:

Standard An encoding which uses only Common Lisp standard characters (the 94 printing characters of ASCII, plus space and newline), and which is readable enough to be edited manually by humans.

Compressed An encoding which also uses only Common Lisp standard characters, but uses a variety of tricks to greatly reduce the number of characters required, at the expense of human readability. [The Compressed encoding is currently unimplemented.]

Binary An encoding which uses the same compression techniques as the Compressed encoding, but which is composed of 8-bit bytes rather than standard characters. This makes it even more compact than the Compressed encoding.

The Standard and Compressed encodings are useful for transmission over media which only transmit standard characters, such as electronic mail. The Standard encoding is also useful for making manual adjustments to CIOBL files.

The CIOBL software provides a uniform interface between files and programs by defining a set of *CIOBL objects* which can appear in CIOBL files. Programs that do I/O deal only with CIOBL objects, and the CIOBL software attends to the details of how the objects are represented in each of the three encodings. A program need only be aware of the encodings when opening a file, for at that time it must select which of the three encodings is to be used.

6.1 CIOBL Objects

A CIOBL file consists of a sequence of CIOBL objects. CIOBL objects include the following *primitive* objects:

Integers These are distinct from floats which happen to have integral values.

Floats A "float" is a floating point number. As described later, CIOBL uses a textual representation for floating point numbers, and so makes no assumptions about base or precision.

Characters The only characters allowed in CIOBL files are Common Lisp standard characters, which consist of the 94 printing characters of ASCII plus space and newline. (This restriction to standard characters has nothing to do with the fact that the Standard and Compressed encodings use only standard characters. Instead, it stems from the need to make sure that CIOBL character objects have a representation in all implementations of CIOBL.)

Strings A string is a (possibly empty) character string. The characters of a string are limited to the 96 Common Lisp standard characters.

Symbols A symbol is a pair of names, the *package* name and the name of the symbol itself. Each name is composed of Common Lisp standard characters.

CIOBL places no practical limit on the magnitude of integers and floats, or on the length of strings.

Symbols seem superfically similar to strings (or a pair of strings), but are usually used for different purposes. One reason is that an implementation of CIOBL often represents strings and symbols differently: while strings are generally represented as arrays of characters, symbols are usually mapped into addresses or serial numbers, with all symbols with the same pair of names being mapped into the same address or serial number. Strings are often decomposed into their component characters, while symbols rarely are. Instead, the names of symbols are important only insofar as they distinguish between symbols that are the same and those that are different.

The foregoing explanation might seem strange to readers familiar with Lisp, who already know why strings and symbols are different. On the other hand, readers not familiar with Lisp might wonder what the package name of a symbol is for. Symbols in Common Lisp are

[Figure to be included]

partitioned into "packages", each of which is a namespace for identifying symbols. The package name of a CIOBL symbol is usually only important for Category III files, which use CIOBL symbols to represent Lisp symbols. When the package of a symbol is not important, which is usually the case in Category I and Category II files, the name `KEYWORD` is generally used as the package.[1]

In addition to the above primitive objects, CIOBL objects include the following *compound objects*:

Lists A list is a (possibly empty) delimited sequence of CIOBL objects. **In CIOBL, the empty list is different from the symbol `NIL`**, although programs may of course choose to treat them the same in some situations.

Dotted Lists A dotted list is like a list, but also includes a special object at the end. Dotted lists are mainly used in Category III files; they correspond to non-`NIL` terminated lists in Common Lisp.

Arrays Like a list, an array is a sequence of CIOBL objects. Unlike a list, an array carries an indication of how many objects it contains. Also, arrays may be multidimensional, up to seven dimensions. Each dimension has subscripts running from zero up to but not including the size of the dimension.

User Defined Objects Arbitrary compound objects may be defined by user applications.

For category III files, the difference between arrays and lists is important, as these are two different kinds of Common Lisp objects. For category II files, the difference can also be important since arrays can be multidimensional (although this can be simulated by lists of lists). For category I files, the choice between arrays and lists is fairly arbitrary. Lists have the advantage when writing a file that the contents may be written without knowing the length of the list. When reading a file, this aspect may be a disadvantage.

The next section describes how compound objects appear in CIOBL files.

6.2 CIOBL Tokens

The contents of a CIOBL file can be viewed from three different levels of abstraction. At the lowest level, a CIOBL file is just a sequence of characters (or 8-bit bytes, in the case of Binary encoding). At the highest level, a CIOBL file is a sequence of CIOBL objects, as previously discussed. At the middle level, a CIOBL file is a sequence of *CIOBL tokens*.

CIOBL tokens are the smallest indivisible components of CIOBL files, and are independent of the encoding chosen. The set of CIOBL tokens consists of all of the primitive CIOBL objects, as well as some *punctuation tokens* which serve to identify compound objects. Defining a token layer allows the portions of a CIOBL implementation which understand the various encodings to be separated from the portions which understand how to put together and pick apart compound objects. This is illustrated below.

[1] If you don't know Common Lisp, don't ask why.

Two punctuation tokens are used to represent lists: `list-begin` and `list-end`. A CIOBL list appears as the token `list-begin`, followed by the CIOBL objects comprising the list (or none if the list is empty), followed by the token `list-end`. A CIOBL dotted list makes use of another puncutation token called `list-dot`. A dotted list appears as `list-begin`, followed by one or more CIOBL objects, followed by `list-dot`, followed by exactly one CIOBL object, followed by `list-end`.

One punctuation token is used to represent arrays: `array-begin`. A CIOBL array appears as the token `array-begin`, followed by the dimensions, followed by the elements of the array. There is no terminating puncuation token. The dimensions are either a list of integers, or just an integer itself, the latter being equivalent to a list of one integer. The rank of the array is the number of integers in the dimension list; each dimension has subscripts from zero, inclusive, to the integer given in the dimensions list, exclusive. The elements of the array appear in row-major order. Here is an example of a two dimensional array:

`array-begin list-begin` 3 2 `list-end` $a(0,0)$ $a(0,1)$ $a(1,0)$ $a(1,1)$ $a(2,0)$ $a(2,1)$

Two more puncuation tokens are used to represent user-defined objects: `user-defined-begin`, and `user-defined-end`. A user-defined object appears as the token `user-defined-begin`, followed by a symbol, followed by some number of CIOBL objects, followed by the token `user-defined-end`. The symbol immediately after the `user-defined-begin` indicates which user defined object is being represented; when reading such an object, the symbol is used to dispatch to user-written code which reads the remaining objects up to the `user-defined-end`[2].

There is another kind of token that is not associated with any object. Called the version token, it indicates what version of CIOBL software was used to write the file in which it appears. It also indicates in which of the three encodings the file is expressed. Whenever a CIOBL file is opened for writing, a version token is immediately written. Thus, every file has a version token at the beginning. A file may have more version tokens within, if the file was ever opened for appending. When a file is read, the version token is used to make sure the file uses the expected encoding, and that the file was written with a compatible version of CIOBL.

To summarize, there are twelve kinds of CIOBL tokens: the five primitive CIOBL objects, the six punctuation tokens, and the version token.

6.3 CIOBL Streams

CIOBL streams are streams to which CIOBL objects are written and from which CIOBL objects are read. The following sections describe how to create CIOBL streams, how to read and write CIOBL objects, and how to define readable/writable CIOBL objects.

6.3.1 Creating CIOBL Streams

`make-ciobl-stream` *stream encoding* [*Function*]
`ciobl-stream` [*Type*]
`ciobl-stream-p` *object* [*Function*]

`Make-ciobl-stream` creates and returns a CIOBL stream of the appropriate type from a Common Lisp stream, after first making sure the encoding chosen is compatible with the underlying

[2]The `user-defined-end` token is not strictly necessary, but is included to help catch errors in user-written handler code.

stream type. It ensures that the version token has been read, so that `read-ciobl`, `skip-ciobl` do not have to. The encoding must be one of `:standard`, `:binary` or `:compressed`.

The function `ciobl-stream-p` returns `t` if *object* is a CIOBL stream and `nil` otherwise.

`close-ciobl` *ciobl-stream* [*Function*]

The function `close-ciobl` just does a Common Lisp `close` on the underlying stream.

`with-ciobl-stream (`*var stream binaryp*`) &body` *body* [*Macro*]

The macro `with-ciobl-stream` binds a variable to the result of making a CIOBL stream out of a given stream, and makes sure always to call `close-ciobl` on it, even if errors occur.

`with-ciobl-file (`*var filename binaryp* `&rest` *open-options*`) &body` *body* [*Macro*]

The macro `with-ciobl-file` combines `with-ciobl-stream` and `with-open-file`.

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

The functions `finish-ciobl-output`, `force-ciobl-output`, and `clear-ciobl-output` just call the appropriate Common Lisp function on the underlying stream.

6.3.2 Reading and Writing CIOBL Streams

`read-ciobl` *ciobl-stream* `&key (eof-error-p t) (return-list-p t)` [*Function*]
`(list-punctuation-error-p t)`
`read-ciobl-1` *ciobl-stream eof-error-p return-list-p list-punctuation-error-p* [*Function*]

The function `read-ciobl` tries to read a returnable element of CIOBL. It can always return types `:symbol`, `:string`, `:integer`, `:float`, or a user defined type. If *eof-error-p* is false, it can return also `:eof`. If *list-punctuation-error-p* is false it can also return `:list-dot` or `:list-end`. If *return-list-p* is false it can also return `:list-begin`, otherwise it can also return `:list`. An error is signalled in every other case.

The function `read-ciobl-1` is the same as `read-ciobl`, except that it does not incur the overhead of parsing keyword arguments. The time spent parsing keyword arguments can be considerable when reading CIOBL streams.

`skip-ciobl` *ciobl-stream* `&key (eof-error-p t) (return-list-p t)` [*Function*]
`(list-punctuation-error-p t)`
`skip-ciobl-1` *ciobl-stream eof-error-p return-list-p list-punctuation-error-p* [*Function*]

The function `skip-ciobl` tries to skip a returnable element of CIOBL. It can always return types `:symbol`, `:string`, `:integer`, `:float`, or a user defined type. If *eof-error-p* is false, it can return also `:eof`. If *list-punctuation-error-p* is false it can also return `:list-dot` or `:list-end`. If *return-list-p* is false it can also return `:list-begin`, otherwise it can also return `:list`. An error is signalled in every other case. As with `read-ciobl-1`, `skip-ciobl-1` is the same as `skip-ciobl`, except that it does not use keyword arguments.

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

`write-ciobl-list-dot` *ciobl-stream* [*Function*]
`write-ciobl-newline` *ciobl-stream* [*Function*]

The function `write-ciobl` writes a CIOBL object. It calls a variety of functions, one for each different CIOBL token. The functions `write-ciobl-list-begin` and `write-ciobl-list-end` write the appropriate delimiter to the given stream. These functions are exported, so we do a little extra error checking. `Write-ciobl-list-dot` writes the puncutation indicating that the final `cdr` of a list follows. The function `write-ciobl-newline` writes a newline to a standard encoding stream, and is a no-op for a binary encoding stream. It can be used to get prettier standard encoding files, if desired.

6.3.3 User Defined Objects

One adds new object encodings to CIOBL's repertoire by defining read and write methods for objects of a given type. The macros `defciobl-read` and `defciobl-write` define read and write methods, respectively, for a Lisp type. Note that CIOBL read/write methods are defined for all of the types defined by DFCS.

`defciobl-write` *type* (*object-var ciobl-stream-var*) `&body` *body* [*Macro*]
`defciobl-read` *type* (*ciobl-stream-var*) `&body` *body* [*Macro*]

The macro `defciobl-write` defines a write method for objects whose type is *type*. The body of the ciobl-write-method should write out a representation of the object to which *object-var* is bound using calls to `ciobl-write`.

The body of the ciobl-read-method should read the representation of an object whose type is *type* using calls to *read-ciobl*. A ciobl-read-method *must* read exactly as much as the corresponding write-method wrote to the CIOBL stream.

6.4 Encodings

In the previous section, the translation between CIOBL tokens and CIOBL objects was described. Here, we describe how CIOBL tokens are encoded in the three types of encodings.

6.4.1 Standard Encoding

Standard encoding is the most easily read by humans. In fact, it is closely related to the syntax for Common Lisp objects used by the Lisp reader and printer. Beware, however, for CIOBL Standard encoding is not compatible with Lisp. Each contains objects not present in the other. Furthermore, CIOBL Standard encoding has some restrictions that must be observed by CIOBL output routines, in order that the job of CIOBL input routines may be made easier.

Here is how the twelve CIOBL tokens are represented in the Standard Encoding:

Integers Integers are represented as they are written in base 10: a sequence of digits, at least one digit long, and immediately preceded by a hyphen if negative. Leading zeros are never present (except when the integer is itself zero).

Floats Floats are represented as a sequence of digits containing exactly one period, with at least one digit on each side of the period. This may optionally be preceded by a hyphen, and optionally followed by the letter `E` (never lowercase `e`) and a sequence of digits, possibly with a hyphen appearing between the `E` and the digits. The number after the `E` indicates by what power of ten the number preceding the `E` should be multiplied.

Characters The 94 printing characters are represented by the three character sequence `#\`*char*, where *char* is the character to be represented. For example, the character `A` is represented as `#\A`, the character backslash as `#\\`, etc. The character "space" is represented as `#\Space`, and the character "newline" is represented as `#\Newline`.

Strings A string is represented by a quotation mark (`"`), followed by the characters composing the string, followed by another quotation mark. Each quotation mark or backslash within the string is itself preceded by a backslash. For example, the string `A backslash ("\")` appears as `"A backslash (\"\\\")"` in a Standard encoding file.

Symbols Symbols whose package name is `KEYWORD` are represented as a colon followed by the name of the symbol, *e.g.*, `:FRED`. Symbols accessible in the `DFCS` package are represented by just the name of the symbol, while all other symbols are represented by the name of the package followed by two colons, followed by the name of the symbol.

Names of symbols and packages are represented by a sequence of the characters composing the name, except that a backslash is inserted before each character that is not an uppercase letter, a digit, or one of the characters `+`, `-`, `*`, `/`, `\`, `$`, `%`, `^`, `&`, `_`, `<`, `>`, or `~`. Furthermore, the first character is preceded by a backslash if it is a digit or a hyphen (this is so that an object begins with a digit or hyphen if and only if it is an integer or float). Finally, if the name has no characters (*i.e.*, is the empty string), it is represented as two vertical bars.

`list-begin` `list-begin` appears as a left parenthesis (`(`).

`list-end` `list-end` appears as a right parenthesis (`)`).

`list-dot` `list-dot` appears as a period.

`array-begin` `array-begin` appears as the two-character sequence `#A`.

`user-defined-begin` `user-defined-begin` appears as as the two-character sequence `#[`.

`user-defined-end` `user-defined-end` appears as the two-character sequence `#]`.

Version The version token is a four-character sequence: `#`, followed by `V`, followed by a character indicating the version, followed by `S`. A CIOBL version number is an integer from 1 through 63, inclusive, and is represented in the Standard encoding using the *value* column of Section 6.4.4. The fourth character indicates that this is a Standard encoding file.

Whitespace may appear between any adjacent pair of CIOBL tokens, and is required between any adjacent pair of tokens which are also primitive objects. Whitespace is any non-empty sequence of spaces or newlines. The last character of a primitive object is defined to be the last character preceding the next whitespace, punctuation, or end of file.

CIOBL takes no position as to what character codes are used in Standard encoding files. This is because such files are character files, so it is assumed that an implementation will use whatever codes for characters are appropriate, and that file transfer programs will take care of any necessary code conversions when transferring Standard encoding files between machines.

6.4.2 Compressed Encoding

[The Compressed Encoding is currently unimplemented.]

6.4.3 Binary Encoding

Binary encoding is the most compact of the three encodings. Unlike the other encodings, binary encoding is based on 8-bit bytes rather than characters. A variety of techniques are used to reduce the space required by a file.

The following describes how the twelve CIOBL tokens are represented in the Binary encoding. All tokens begin with a special byte that identifies its type. Throughout the discussion, these bytes are represented by a name enclosed in angle brackets, for example, <b-pos-integer-8>. The numerical values of these bytes are given later.

Integers Integers are represented by a punctuation byte followed by a certain number of bytes which indicate the magnitude of the integer. The punctuation byte indicates the sign of the integer as well as how many magnitude bytes follow; the magnitude bytes appear least significant byte first. The punctuation bytes are:

Magnitude Range	*Punctuation Byte*
$-2^{32} < x \leq -2^{24}$	<b-neg-integer-32>
$-2^{24} < x \leq -2^{16}$	<b-neg-integer-24>
$-2^{16} < x \leq -2^{8}$	<b-neg-integer-16>
$-2^{8} < x < 0$	<b-neg-integer-8>
$0 \leq x < 2^{8}$	<b-pos-integer-8>
$2^{8} \leq x < 2^{16}$	<b-pos-integer-16>
$2^{16} \leq x < 2^{24}$	<b-pos-integer-24>
$2^{24} \leq x < 2^{32}$	<b-pos-integer-32>

For example, the number -4000 is represented as <b-neg-integer-16> 160 15, and zero is represented as <b-pos-integer-8> 0. (Throughout, all bytes are given in base ten.)

If the magnitude of the integer is 2^{32} or greater, one of the two punctuation bytes <b-pos-long-integer> or <b-neg-long-integer> is used, followed by a byte giving the number of bytes in the magnitude, followed by the bytes which comprise the magnitude. For example, five trillion (5×10^{12}) appears as <b-pos-long-integer> 6 0 80 57 39 140 4. Integers with magnitudes greater than or equal to 256^{256} cannot be represented (fortunately).

Characters A character appears as two bytes: <b-character> followed by the character itself. Because Binary encoding files are not character files, CIOBL must explicitly specify how characters are encoded into bytes. The 94 printing characters are encoded as they are in the ASCII character set. The remaining two standard characters, Space and Newline, are encoded as 32 and 10, respectively.

This encoding was chosen because it is exactly the encoding used in some systems (*e.g.*, Unix), and very close to the encoding used in most others (*e.g.*, the Lisp Machine, where Newline is 141). CIOBL implementations on non-ASCII systems will have to explicitly translate when reading or writing Binary encoded files.

Strings Strings of length 255 or less are encoded as <b-string>, followed by a byte giving the length of the string, followed by the characters of the string itself. Paragraph 6.4.3 explains how characters are translated into bytes.

Strings whose length is greater than 255 are encoded as <b-long-string>, followed by a byte giving the number of bytes which will represent the length, followed by the bytes comprising the length, followed by the characters of the string itself. As with integers, the bytes comprising the length appear least significant byte first. Strings with length 256^{256} or longer cannot be represented (not that your file system has room for all those characters, anyway).

Floats Floats are not encoded into a two's complement or similar representation. Instead, they appear just like strings, except that the bytes `<b-float>` and `<b-long-float>` are used instead of `<b-string>` and `<b-long-string>` (note that the latter is extremely unlikely). For example, the number -6.023×10^{-23} appears as `<b-float> 10 45 54 46 48 50 51 69 45 50 51`.

Symbols The symbol `NIL` (not the same as `NIL` in the `KEYWORD` package) is represented by a single byte, `<b-nil>`.

Keyword symbols are represented by encoding their names in a similar manner as strings (see Paragraph 6.4.3), except that the bytes `<b-keyword-symbol>` and `<b-long-keyword-symbol>` are used instead of `<b-string>` and `<b-long-string>`, respectively.

All other symbols are represented by encoding their names in a similar manner as strings, except that the bytes `<b-symbol>` and `<b-long-symbol>` are used instead of `<b-string>` and `<b-long-string>`. The package name is not included in the name that follows the `<b-symbol>` or `<b-long-symbol>` byte. Instead, the package name of a symbol read from a Binary encoded file is taken to be the "current" package name. The current package name is changed by including in the file one of the bytes `<b-set-current-package>` or `<b-long-set-current-package>`, which are used like `<b-string>` and `<b-long-string>` to encode the package name.

For example, if the following sequence appeared in a Standard encoding file:

```
:A  NIL  B::C  :NIL  NIL  B::D  Q::R
```

It would be rendered in a Binary encoding file as:

```
<b-keyword-symbol> 1 65 <b-nil> <b-set-current-package> 1 66
<b-symbol> 1 67 <b-keyword-symbol> 3 78 73 76 <b-nil>
<b-symbol> 1 68 <b-set-current-package> 1 81 <b-symbol> 1 82
```

The first symbol in a Binary encoding file that is not a keyword or `NIL` is always preceded by a set-current-package directive. Note that the set-current-package directive is not a CIOBL token, but only controls the interpretation of symbols that follow it.

One additional trick is used to encode symbols. If the same symbol (same package name and symbol name) appears more than once in the same file, only the first occurence is encoded as described earlier. All future occurences are encoded as an integer which indicates position within the file of its first occurence. These remarks do not apply to the symbol `NIL`, which is always encoded as the byte `<nil>`.

As a Binary encoded file is written, a table is maintained which associates symbols and serial numbers. When a non-`NIL` symbol is to be written, it is looked up in the table. If an entry for that symbol is present, its serial number is written in a format to be described shortly. If an entry is not present, the symbol is written in the format described earlier, preceded by a set-current-package directive if necessary. The symbol is then assigned the next highest serial number, and entered in the table for future reference. The unique non-`NIL` symbols in a file are assigned consecutive serial numbers beginning with zero. Note that when a symbol is encoded as a serial number, it is never necessary to issue a set-current-package directive, as the serial number identifies both components of the symbol's name.

A predefined symbol is encoded as one of the three bytes `<b-predefined-symbol-8>`, `<b-predefined-symbol-16>`, or `<b-predefined-symbol-24>`, followed by one, two, or three bytes of serial number, respectively, least significant byte first.

Another example: suppose the following appeared at the beginning of a Standard encoding file:

```
:D :A  NIL  B::C  :NIL  NIL  Q::R  B::C  B::D
```

It would be rendered in a Binary encoding file as:

```
<b-keyword-symbol> 1 68 <b-keyword-symbol> 1 65 <b-nil>
<b-set-current-package> 1 66 <b-symbol> 1 67
<b-keyword-symbol> 3 78 73 76 <b-nil>
<b-set-current-package> 1 81 <b-symbol> 1 82
<b-predefined-symbol-8> 2 <b-set-current-package> 1 66
<b-symbol> 1 68
```

Only the first 2^{24} different non-`NIL` symbols in a file can be encoded by serial number.

The Version Token The version token appears in a Binary file as the four-byte sequence `35 86` *version* `66`, where *version* is the version number plus 32. The last byte indicates that this is a Binary file.

When reading a Binary encoding file, `163` (35 plus 128) should also be accepted as the beginning of a version token. In other words, only the lower seven bits are used in recognizing the beginning of the version token. The number 35 was chosen for the version token because it is the ASCII code for `#`, so that a version will be recognized even when read from a file in the wrong encoding (at least on systems that use ASCII to represent characters). This facilitates early detection of an attempt to read the wrong kind of file.

Other Tokens The CIOBL tokens `list-begin`, `list-dot`, `list-end`, `array-begin`, `user-defined-begin`, and `user-defined-end` are represented in Binary encoding as the bytes `<list-begin>`, `<list-dot>`, `<list-end>`, `<array-begin>`, `<user-defined-begin>`, and `<user-defined-end>`, respectively.

Run Length Encoding Long sequences of repeated tokens are represented in the Binary encoding using one of two special bytes, `<b-repeat-8>` or `<b-repeat-16>`. A `<b-repeat-8>` byte is followed by one byte, which indicates how many times the previous token is to be repeated, from 0 through 255. `<b-repeat-16>` is similar, except that it is followed by two bytes giving the repeat count, up to 65,535. For example, the following seuence:

```
<b-pos-integer-8> 34 <b-pos-integer-8> 34 <b-pos-integer-8> 34 <b-nil>
<b-nil> <b-nil> <b-nil> <b-nil> <b-nil> <b-list-close> <b-list-close>
<b-list-close>
```

could instead appear as

```
<b-pos-integer-8> 34 <b-repeat-8> 2 <b-nil> <b-repeat-8> 5
<b-list-close> <b-repeat-8> 2
```

Note that the repeat bytes indicate the repetition of *tokens*, not objects.

Values of Binary Punctuation Bytes The following table gives the values of each of the 30 binary punctuation bytes.

Name	*Value*	*Name*	*Value*
`<b-symbol>`	0	`<b-pos-long-integer>`	24
`<b-keyword-symbol>`	1	`<b-neg-integer-8>`	25
`<b-long-symbol>`	2	`<b-neg-integer-16>`	26
`<b-long-keyword-symbol>`	3	`<b-neg-integer-24>`	27
`<b-nil>`	4	`<b-neg-integer-32>`	28
`<b-set-current-package>`	5	`<b-neg-long-integer>`	29
`<b-long-set-current-package>`	6	`<b-float>`	30
`<b-predefined-symbol-8>`	7	`<b-long-float>`	31
`<b-predefined-symbol-16>`	8	`<b-version>`	35
`<b-predefined-symbol-24>`	9	`<b-list-begin>`	40
`<b-string>`	10	`<b-list-end>`	41
`<b-long-string>`	11	`<b-list-dot>`	42
`<b-character>`	15	`<b-repeat-8>`	45
`<b-pos-integer-8>`	20	`<b-repeat-16>`	46
`<b-pos-integer-16>`	21	`<b-user-defined-begin>`	50
`<b-pos-integer-24>`	22	`<b-user-defined-end>`	51
`<b-pos-integer-32>`	23	`<b-array-begin>`	55

6.4.4 Character Codes

The following table lists the 96 standard characters. The *byte* column gives the representation (in base 10) for each character in the Binary encoding. The *value* column gives the value assigned to characters when used to encode integers in the Compressed encoding, and when used as part of the version token in both the Compressed and the Standard encoding.

Char.	*Byte*	*Value*	*Char.*	*Byte*	*Value*	*Char.*	*Byte*	*Value*
Space	32	0	@	64	32	`	96	
!	33	1	A	65	33	a	97	
"	34	2	B	66	34	b	98	
#	35	3	C	67	35	c	99	
$	36	4	D	68	36	d	100	
%	37	5	E	69	37	e	101	
&	38	6	F	70	38	f	102	
'	39	7	G	71	39	g	103	
(	40	8	H	72	40	h	104	
)	41	9	I	73	41	i	105	
*	42	10	J	74	42	j	106	
+	43	11	K	75	43	k	107	
,	44	12	L	76	44	l	108	
-	45	13	M	77	45	m	109	
.	46	14	N	78	46	n	110	
/	47	15	O	79	47	o	111	
0	48	16	P	80	48	p	112	
1	49	17	Q	81	49	q	113	
2	50	18	R	82	50	r	114	
3	51	19	S	83	51	s	115	
4	52	20	T	84	52	t	116	
5	53	21	U	85	53	u	117	
6	54	22	V	86	54	v	118	
7	55	23	W	87	55	w	119	
8	56	24	X	88	56	x	120	
9	57	25	Y	89	57	y	121	
:	58	26	Z	90	58	z	122	
;	59	27	[	91	59	{	123	
<	60	28	\	92	60	|	124	
=	61	29	]	93	61	}	125	
>	62	30	^	94	62	~	126	
?	63	31	_	95	63	Newline	10	

Chapter 7

Miscellaneous

7.1 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.

`:fatal` An error which forces the termination of the currently executing module. The effect is to drop the offending source code input unit of that module. The compiler may continue processing the units that follow.

`: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.

`:log` A message that is only inserted in the log file, if any.

7.1.1 Message Hooks

`*message-output*` [*Internal Variable*]

`*message-output*` is a stream synonym for `*error-output*`.

When `message` is invoked, it normally prints a message to the stream `*message-output*`. Particular compilers may also want to perform other actions when message is invoked. DFCS supports this by allowing compilers to provide *message-hooks*. A message-hook is a function that takes a type, place, module-name, format-string and format-args as arguments and performs an application-specific operation. After `message` prints a message to `*message-output*`, it invokes the message-hook, if one was provided. The functions `standard-message-string` and `fill-and-indent` may be useful in the definition of message hooks.

`standard-message-string` *type place module-name format-string format-args* [*Function*]

`Standard-message-string` is the function that constructs the standard message string that is printed to the stream `*message-output*` when `message` is called. This function may be useful in message hooks.

`fill-and-indent` *string line-length indent* [*Function*]

Fills and indents string *string* so that it fits into *line-length* characters and each line begins with *indent* characters of white-space.

`*message-break-characters*` [*Variable*]

Characters at which `fill-and-indent` will break lines. Internal variable.

`*handle-lisp-errors*` [*Internal Variable*]
`*message-types-debugged*` [*Internal Constant*]
`*unrecoverable-catch-active-p*` [*Internal Variable*]
`*fatal-catch-active-p*` [*Internal Variable*]

`*handle-lisp-errors*` is an internal variable which is normally set to `t`. This instructs the internal compiler error catcher wrapper to prevent the user from entering the debugger when there are lisp errors (bugs in the compiler). Compiler hackers almost always set it to `nil` so as to invoke the lisp debugger when a bug is caught.

Adding a message type to the list `*message-types-debugged*` causes a break to be entered when that type of message is processed.

`*unrecoverable-catch-active-p*` and `*fatal-catch-active-p*` are dynamic variables that are bound to `t` by the error catching wrappers. If during compiler execution (or outside the cycling of compiler) this binding is set to `nil`, then the `:unrecoverable` and `:fatal` messages cause a break to take place.

7.2 Performance Metering

The Dataflow Compiler Substrate has a limited performance metering capability apart from any that may be offered by the native LISP system. These performance meters give an idea of how much time does the compiler spend in each module and how much consing does it do among its various data structures, *i.e.*, ptnodes, instructions, and frames.

The performance meters are not usually loaded into the system. They have to be loaded separately into the DFCS system. The file required to be loaded for this facility is given in appendix A.

7.2.1 Space Meters

The following consing statistics are collected.

`*instructions-consed*` [*Internal Variable*]
`*instruction-words-consed*` [*Internal Variable*]
`*ptnodes-consed*` [*Internal Variable*]
`*ptnode-words-consed*` [*Internal Variable*]
`*frames-consed*` [*Internal Variable*]
`*frame-words-consed*` [*Internal Variable*]
`*n-wirings*` [*Internal Variable*]

`*instructions-consed*` counts the number of dataflow graph instructions allocated.

`*instruction-words-consed*` gives a rough idea about the size of the instructions being allocated. It counts 12 words per instruction allocated and 1 word each per the basic input and output ports of an instruction. This is a VERY coarse measure of the amount of basic dataflow graph storage used. This does not include any extra storage required by the `instruction-slot` properties.

`*ptnodes-consed*` counts the number of ptnodes allocated. `*ptnode-words-consed*` counts 11 words of constant storage per ptnode allocated, and 1 per permanent or memoized attribute slot of that ptnode. Again, this is an extremely coarse description of the actual storage used.

`*frames-consed*` counts the number of frames allocated. `*frame-words-consed` counts 17 words per frame allocated along with the size of the input and the output hash tables.

`*n-wirings*` counts the number of wirings done in a dataflow graph.

`reset` [*Internal Function*]

Resets the internal space statistics collection counters.

`print-performance-meters` [*Internal Function*]

Prints the values of various space statistics counters.

7.2.2 Time Meters

A useful performance meter is the amount and the percentage of time spent within each module during a compilation. This is an abstract and useful measure of the time taken by each phase of the compiler. The time is computed in seconds of real time spent within each module and is printed as both actual time and as a percentage of the total. The Common LISP function `get-internal-real-time` is used for this purpose.

`reset-compiler-times` *compiler-name* [*Function*]

Resets the time counters associated with each module of the compiler *compiler-name*.

`print-compiler-times` *compiler-name* [*Function*]

Prints the time taken in seconds and as a percentage of the total by each module of the compiler *compiler-name*.

7.3 Sxhash Tables

`sxhash-table` [*Defstruct*]

An SXHASH table is essentially an EQL hash table that uses SXHASH to compute the hash codes. Collisions are handled by using alists as buckets. SXHASH-TABLEs are intended to be used when the keys are symbols, for in that case they end up being quite a bit faster than any of the Common Lisp hash table varieties, at least on the Lisp Machine, because SXHASH-TABLEs don't do rehashing upon GC or locking out of other processes. Also, they can be made smaller than the Lisp Machine is willing to make Common Lisp hash tables. We represent SXHASH-TABLEs as defstructs. The ENTRIES-ARRAY is indexed by SXHASH code mod the size.

`clrsxhash` *sxhash-table* [*Function*]

`getsxhash` *key sxhash-table* `&optional` *default* [*Function*]

`remsxhash` *key sxhash-table* [*Function*]

`mapsxhash` *fcn sxhash-table* [*Function*]

`Mapsxhash` calls *fcn* with two arguments, a key and its value, for every entry in the sxhash-table. It returns `NIL`.

7.4 Miscellaneous Functions

`pointer` *object* [*Internal Function*]

Returns a number for a structure. It is useful in printing structure in order to distinguish them from similar structures.

`record-source-file-name` *function-spec* `&optional` *type no-query* [*Function*]

System Independent implementation of `record-source-file-name`.

`write-verbose-dataflow-graph` *dataflow-graph stream* [*Function*]

Writes the dataflow graph *dataflow-graph* on stream *stream* in a tabular format. First the property list of the dataflow graph is printed. Then all the instructions are printed one by one starting from the root set.

An entry describes the instruction offset, its opcode, and for each output name, its destination instruction offset and the input name it is wired to. If there is an annotation associated with the arc, its first 26 characters are also shown.

Appendix A

Files of the Dataflow Compiler Substrate

Note:: The files described in this section correspond to the state of the DFCS system as on the date of release of this document. This section MUST be updated whenever there is an addition or deletion of files to the DFCS system. (Hopefully this would not be very often.)

Figure A.1 describes the files currently present in the DFCS system. Those marked with an asterisk (*) are ususally loaded as part of the definition of the `dfcs` program. Others are present as aids and tools. The references at the end point to the chapters or sections of this document or other documents which describe most (or relevent) parts of that file.

Dataflow Compiler Substrate System

System Definition File : `dfcs`

In Use	File Name	Description	Reference
*	`ciobl`	Implements CIOBL.	Chapter 6
*	`common`	Implements common utilities for DFCS; *sxhash-tables*, clause validation, and miscellaneous functions.	Section 7.4
*	`dataflow-graph`	Dataflow Graph Abstraction.	Section 3.2
*	`defcompiler`	Utilities for defining and running Compilers.	Chapter 2
*	`external-symbols`	Implements External Symbols Abstraction.	Chapter 5
*	`file`	CIOBL read/write functions for Dataflow Graphs.	
*	`frame`	Implements Frames (except for wiring functions).	Section 3.2.4
*	`parse-tree`	Implements Lexical Tokens and Parse Tree manipulations.	Section 3.1
	`performance-meters`	Implements Performance Metering.	Section 7.2
*	`storage`	Implements explicit management of adjustable vectors with fill pointers.	
*	`syntax-directed`	Implements Grammars and Parse Tree Attribute management.	Section 4.1 and Section 4.2
	`test`	Some random test functions for DFCS.	
*	`wiring`	Implements wiring functions to and from Instruction/Frame Sinks and Sources.	Section 3.2
	`write-verbose-dataflow-graph`	A utility to view a dataflow graph in a tabular form.	Section 7.4

Figure A.1: List of files for Dataflow Compiler Substrate System

Appendix B

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.

Bibliography

[1] Guy Lewis Steele Jr. *Common Lisp: The Language.* Digital Press, Burlington, Massachusetts, 1984.

[2] Guy Lewis Steele Jr. *Common Lisp: The Language, Second Edition.* Digital Press, Bedford, Massachusetts, 1990.

[3] Kenneth R. Traub. A Compiler for the MIT Tagged-Token Dataflow Architecture. Technical Report LCS TR-370, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, August 1986.

Index

add-input-to-frame, 42
add-next-input-to-frame, 42
add-next-output-to-frame, 42
add-output-to-frame, 42
after-function, 16
annotation, 9
arc annotation rules, 39
arc annotation rules, 43
arc, 32
arc, 38
arc-annotation, 39
assembler, 9
before-function, 16
call-on-input-port-instruction, 54
ciobl-stream, 62
ciobl-stream-p, 62
clear-ciobl-output, 63
close-ciobl, 63
clrsxhash, 74
collector module, 15
compiler family, 11
consistency-summary, 57
copy-exsym, 55
current-module-name, 20
dataflow graph, 32
dataflow graph, 9
dataflow-graph-get, 44
dataflow-graph-plist, 44
dataflow-graph-root-set, 44
defattributes, 51
defattributes, 51
defciobl-read, 64
defciobl-write, 64
defcompiler, 18
defcompiler-family, 13
defcompiler-module, 16
defcompiler-option, 14
defgrammar, 45
define-dfg-attribute, 53
define-exsym-property, 56
define-grammar-abbreviation, 47
define-instruction-slot, 35
define-ptnode-attribute, 50
define-ptnode-slot, 30
defproduction, 46
describe-consistency, 57
exsym-assume, 56
exsym-assume-value, 56
exsym-clear, 56
exsym-exists-p, 55
exsym-get, 56
exsym-put, 56
exsym-table, 55
fatal-catch-active-p, 72
fill-and-indent, 72
find-exsym, 55
finish-ciobl-output, 63
force-ciobl-output, 63
frame input, 39
frame output, 39
frame point, 41
frame, 39
frame-input-exists?, 42
frame-input-names, 42
frame-number-of-inputs-with-symbol, 43
frame-number-of-outputs-with-symbol, 43
frame-output-exists?, 42
frame-output-names, 42
frames-consed, 73
frame-words-consed, 73
free token list, 25
free token list, 26
free token list, 26
free token list, 26
generator module, 15
getsxhash, 74
grammar, 45
grammarbind, 49
grammarcase, 48
grammarcasebind, 50

grammarspec, 45
graph attribute, 52
handle-lisp-errors, 72
inherited, 50
initialize-dfg-attributes, 53
input map, 34
install-exsym, 55
instruction sink, 36
instruction source, 36
instruction, 32
instruction-input, 37
instruction-input-name-to-number, 35
instruction-input-number-to-name, 36
instruction-inputs, 37
instruction-n-inputs, 35
instruction-n-outputs, 35
instruction-opcode, 35
instruction-output, 37
instruction-output-name-to-number, 35
instruction-output-number-to-name, 36
instruction-outputs, 37
instruction-parameter, 35
instructions-consed, 73
instruction-sink-annotation, 38
instruction-sink-input, 37
instruction-sink-instruction, 37
instruction-source-annotation, 38
instruction-source-instruction, 38
instruction-source-output, 38
instruction-words-consed, 73
intermediary module, 15
keyword terminal, 27
LALR parser, 25
LALR parser, 9
let-ptnode-children, 48
level, 12
lexical analyzer, 25
lexical analyzer, 9
lexical token, 25
lexical token, 9
machine graph, 32
machine graph, 9
make-ciobl-stream, 62
make-dataflow-graph, 44
make-exsym-table, 55
make-frame, 42
make-frame-input-sink, 43
make-frame-input-source, 43
make-frame-output-sink, 43
make-frame-output-source, 43
make-instruction, 34
make-instruction-sink, 36
make-instruction-source, 36
make-parent-ptnode, 29
make-parse-tree, 31
make-place, 31
make-port-map, 34
make-ptnode, 29
make-token, 26
map-exsym-table, 55
map-on-input-port-values, 54
map-on-output-port-instructions, 54
map-on-output-port-values, 54
mapsxhash, 74
Marking a level, 15
mark-level, 17
mark-level, 17
message, 71
message-break-characters, 72
message-output, 72
message-types-debugged, 72
move-all-arc-origins, 39
move-any-arc-destination, 38
n-ary-n, 49
non-terminal, 27
n-wirings, 73
option, 14
option-exists-p, 14
option-types, 14
output map, 34
parse tree node, 26
parse tree, 26
parse tree, 7
parse-tree-get, 31
parse-tree-plist, 31
parse-tree-root, 31
performance metering, 72
place, 25
place, 27
place, 30
place-character, 31
place-column, 31
place-line, 31
point, 41
pointer, 74
port map, 32

port naming rules, 34
port, 32
print-compiler-times, 73
print-performance-meters, 73
prodspec, 45
program graph, 32
program graph, 9
pseudo-terminal value, 27
pseudo-terminal, 27
ptnode tag, 27
ptnode, 26
ptnode, 49
ptnode-children, 29
ptnode-get, 46
ptnode-parent, 29
ptnode-place, 29
ptnodes-consed, 73
ptnode-tag, 29
ptnode-value, 29
ptnode-words-consed, 73
read-ciobl-1, 63
read-ciobl, 63
record-source-file-name, 74
remove-all-arcs, 38
remove-any-arc, 38
remove-arc, 38
remsxhash, 74
replace-ptnode-child, 30
replace-ptnode-children, 30
representation, 12
reset, 73
reset-compiler-times, 73
return-token, 26
sink, 36
skip-ciobl-1, 63
skip-ciobl, 63
source, 36
standard-message-string, 72
suppressed productions, 27
sxhash-table, 74
synthesized, 50
tag, 46
template, 46
token-allocation-quantum, 26
token-class, 25
token-place, 26
token-value, 26
unit, 12
unrecoverable-catch-active-p, 72
virtual arc, 41
virtual arc, 43
wire-attribute-from-port-to-port, 54
wire-frame-input-to-frame-input, 43
wire-frame-input-to-frame-output, 43
wire-frame-input-to-instruction, 43
wire-frame-input-to-sink, 43
wire-frame-output-to-frame-input, 43
wire-frame-output-to-frame-output, 43
wire-frame-output-to-instruction, 43
wire-frame-output-to-sink, 43
wire-instruction-to-frame-input, 43
wire-instruction-to-frame-output, 43
wire-instruction-to-instruction, 36
wire-instruction-to-sink, 36
wire-source-to-frame-input, 43
wire-source-to-frame-output, 43
wire-source-to-instruction, 36
wire-source-to-sink, 36
with-all-port-values-of-dataflow-graph, 54
with-ciobl-file, 63
with-ciobl-stream, 63
with-instruction-array, 44
write-ciobl, 63
write-ciobl-list-begin, 63
write-ciobl-list-dot, 64
write-ciobl-list-end, 63
write-ciobl-newline, 64
write-verbose-dataflow-graph, 74