LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

An Introduction to the Id Compiler

Computation Structures Group Memo 328
May 7,1991

**Boon S. Ang, Alejandro Caro,
Stephen Glim and Andrew Shaw**

This research was supported in part by Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract no. N00014-89-J-1988. Stephen Glim is supported by a fellowship from the National Science Foundation.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

An Introduction to the Id Compiler

Boon S. Ang, Alejandro Caro,
Stephen Glim and Andrew Shaw

May 7,1991

Abstract

This memo describes the machine independent phases of the current Id compiler. It details the organization and implementation of the compiler's modules. It also describes how the DFCS abstractions for compiler building are used in the Id compiler.

Contents

List of Figures

Chapter 1

Introduction

This memo is an outgrowth of a workshop on the Id compiler held during the winter of 1991. The Id compiler has grown considerably since its first incarnation and documentation has not kept pace. At the time of the workshop, some essential features of the compiler were documented only in the source code, or not at all. Though we heartily agree that source level comments are essential for the compiler's maintenance and modification, they are no substitute for a general introduction to help novice compiler hackers navigate their way through the code. Before writing this memo, its authors were unfamiliar with the specific features of the Id compiler they documented. It was our intention to both document and demystify the code as much for our own benefit as for others. In the process, we've surely made mistakes and omitted some details but we hope we've provided a helpful introduction to the Id compiler for other programmers. Please note that the compiler is a rapidly moving target and features may have been added, removed or reorganized from what's described within. We don't intend to keep this memo up to date with the latest and greatest changes, but hopefully future compiler hackers will feel compelled to update relevant sections for posterity.

It was our original intention to document the three phases of the compiler and the top-level "glue" that binds it together. However, manpower constaints have forced us to restrict ourselves to only the implementation independent front and middle ends. The reader interested in a description of a back end for an ETS dataflow machine is referred to Jamey Hick's memo [5]

The reader is advised to have a copy of the DFCS documentation [10] handy while reading since we make frequent references to tools and structures described in it. A working knowledge of Common Lisp [8] is essential to understand the compiler's code. Also, Ken Traub's master's thesis [9] describing an initial version of the compiler might be read in parallel with this document.

We'd like to thank the compiler hacker's of CSG for giving presentations at the workshop and explaining things in general, including, Ken Traub, Jamey Hicks, Shail Adityah Gupta, Paul Barth and Andy Shaw. We'd also like to thank Christine Flood for helpful comments.

Chapter 2

General Structure of the Compiler

2.1 Introduction and Motivation

At first glance, the Id compiler seems to be an impenetrable piece of software. The fledgling compiler hacker can be easily intimidated by the number of modules and by the sheer size of the program's source code. Nevertheless, the Id compiler was conceived as an experimental tool, and consequently, one of its design goals was to make it easy to modify and to adapt. In order to enjoy the benefits of this design goal, it is important to understand the high level organization of the compiler, as well as its high level structuring facilities. This chapter provides a top-down description[1] of the structure of the Id compiler, and it will introduce several of the important software engineering concepts that make the compiler a flexible tool.

At this point, it is important to make one clarification. Although we have been referring to *the* Id compiler, there are actually many Id compilers! Some Id compilers compile Id source files to TTDA object files while other Id compilers compile regions of Id code in editor buffers to Monsoon object files. The reason why there are many Id compilers is that Id compilers are built on top of the Dataflow Compiler Substrate (DFCS) [10]. DFCS is a collection of data structures and abstractions that provide the general functionality for writing compilers for a dataflow language. In the framework provided by DFCS, a compiler is composed of a collection of *modules* and of a top-level procedure that "supervises the passing of control from module to module" [10].[2]

Thus, a compiler in the DFCS framework is really a very flexible software system. One can add modules to a compiler to implement new functionality or one can eliminate modules that become superfluous. Instead of building one monolithic compiler to handle different cases of input and output formats, optimizations levels, etc..., one can build many different compilers using DFCS. More importantly, each of these compilers is separate from others already defined. This prevents separate software development efforts from interfering with each other.

[1] This chapter should be regarded as a tutorial in its scope and depth. While it introduces many new concepts, it does not attempt to provide a definitive treatment of these. Alternative references are provided for readers interested in more detail.

[2] Since the top level structure of an Id compiler is defined by using the facilities provided by DFCS, this chapter will parallel much of the discussion found in Chapter 2 of the DFCS reference document [10]. However, this chapter takes a *top-down* approach, while the DFCS document is organized in a *bottom-up* fashion. In addition, this chapter will give specific examples of the use of DFCS in the context of building or modifying an Id compiler.

2.2 Defining a Compiler

So let's plunge right in. This section presents the DFCS macros used to define an Id compiler. It also presents a sample definition of a compiler as well as examples on how to modify the definition to add functionality. For a more detailed description of `defcompiler`, refer to [10].

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

This macro defines a compiler *name* belonging to the compiler family *family*. The compiler is composed of the given sequence of modules *modules*. The various optional clauses are used to build the top-level function that invokes this particular compiler.

The `:wrapper-macro` clause supplies the name of a "wrapper" macro for the whole compiler. A wrapper macro provides a lexical "enclosure" for the invocation of the compiler. The `:message-hook` clause provides the name of a function *hook-name* that is called whenever the compiler generates a message. This hook allows the compiler writer to customize the generation of messages depending on the compiler. The `:options` clause specifies a set of options that can be set to alter the behavior of the compiler on a particular invocation. Compiler options will be discussed later. The `:option-default` clause allows the default values of options to be changed for this particular compiler. The `:lambda-list` clause permits the specification of a LISP-like lambda list for the top-level function that invokes the compiler. If this clause is omitted, the `defcompiler` macro will generate a default lambda list using rules discussed later.

Figure 2.1 shows the definition[3] of a typical compiler.As its name implies, this particular compiler, `monsoon-id-compile-file`, compiles Id source files into Monsoon object code. The compiler belongs to the `id-compiler` family. Notice that only some of the optional clauses of the `defcompiler` macro are included. This compiler uses a wrapper macro and defines the compiler option `libraries`. Figure 2.2 shows the code for the wrapper macro.

A list of module names follows the list of clauses in the definition of this Id compiler. The name of a module usually reveals its function. For example, the first module of this compiler is the `file-parser` module. The purpose of this module, as we'll see later on, is to read an Id source code file and to produce a parse tree from it. The module following the parser, the `pre-scope-analysis-desugaring` module, receives a parse tree as input an replaces parts of the parse tree with semantically equivalent constructs. Notice that the type of the output of one module must match the type of the input of the following module. The `defcompiler` macro makes sure that the different representation of the Id program used by different modules match, as control is passed from module to module.

That's all there is to defining the top-level structure of an Id compiler. With the proper collection of modules, any particular type of dataflow compiler, whether for Id for some other language, can be implemented. The following section describes how modules are defined and implemented.

2.3 Defining a Module

As we saw in the previous section, an Id compiler is composed of a sequence of modules. While the `defcompiler` macro takes care of connecting modules and building the top-level Lisp function

[3]For expository purposes, this definition has been shortened somewhat. However, it retains the key features a typical use of `defcompiler`.

```
(defcompiler monsoon-id-compile-file
             id-compiler
        ((:wrapper-macro id-monsoon-compile-file-wrapper)
         (:options libraries))
  file-parser                  ; Beginning of front end.
  pre-scope-analysis-desugaring
  scope-analysis
  type-checking                ; Type inference module.
  overloading-translation
  lambda-lift                  ; Lambda lifting module.
  generate-program-graph       ; Beginning of middle end.
  manager-synchronization
  constant-propagation         ; Optimizations
  call-substitution            ;
  fetch-elimination            ;
  simple-fetch-elimination     ;
  dead-code-elimination        ;
  cse-and-hoisting             ;
  circulate-structures         ;
  constant-propagation         ;
  synchronize-release          ;
  partial-evaluation           ;
  constant-propagation         ;
  dead-code-elimination        ;
  thunk-splitting              ;
  inlinable-definitions        ;
  loop-constant-propagation    ;
  code-blocks
  signals-and-triggers
  generate-ets-graph           ; Beginning of back end.
  peephole
  add-fanouts
  merge-merges
  expand-system-functions
  merge-merges
  assign-fp-offsets
  add-successor-constraints
  assign-ip-offsets
  file-assemble-monsoon-graph  ; Assemble and produce object code.
 )
```

Figure 2.1: Definition of an Id compiler

```
(defmacro id-monsoon-compile-file-wrapper (&body body)
  `(let* ((*current-compilation-exsym-table* (make-exsym-table))
          (*the-copyright-notice* nil)
          (start-time (get-universal-time)))
      (format t "~&The Id Monsoon Compiler, Version 1.0.~
               ~&~A" (get 'id-compiler :copyright-notice))
      (monsoon-id-compile-file-exsym-table-prolog)
      ,@body
      (monsoon-id-compile-file-exsym-table-epilog)
      (format t "~&~%Compiled ~d procedure~P,~
                  ~%  in ~d seconds"
          (- (get-universal-time) start-time) )))
```

Figure 2.2: Compiler Wrapper Macro

that invokes the compiler, it is the compiler modules that actually do the work. According to the DFCS reference document, a compiler module "packages up a particular phase of compilation." In Figure 2.1, we saw that there was a single parser module, some graph generation modules, several optimization modules, and finally a single assembler module. Each of these is a separate phase of the compilation of an Id program, and each is implemented by a distinct module. In order to define a module, we must use one more DFCS facility: the `defcompiler-module` macro.

`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*)]

The `defcompiler-module` macro really defines the external interface of a module to the rest of the compiler. The `:input` clause informally describes the "type" of input data structure which the module expects. The type of this data structure is described by a *representation* and a *level*; these two concepts will be addressed later. Similarly, the `:output` clause describes the type of the output data structure of the module. In order to maintain the correct composability of modules, the types of the `:input` and `:output` data structures must be "equivalent" in a manner described below. The `defcompiler` macro checks this property and signals an error if the equivalence is not maintained.

The `:function` clause names the top-level Lisp function that implements the module. The `:before-function` and `:after-function` clauses provide hooks to setup and cleanup functions which are called before and after the invocation of the module.

The `:levels-marked` clause provides some rather subtle functionality which is beyond the scope of this document. Interested readers are referred to [10].

The `:options` clause specifies the compiler options used by this module. These options should have been defined using the `defcompiler-option` macro, which will be described in Section 2.4. Finally, the `:wrapper-macro` clause names the macro *macro-name* as the wrapper for this module.

2.3.1 Families, Representations, and Levels

Modules generate, transform, and pass on data structures. In order to be able to describe the interface of a module, we must be able to name these data structures. A *representation* is exactly that: a name for a data structure. Informally, a representation can be viewed as the type of a data structure. For example, the compiler uses two representations of programs as graphs during compilation: these are program graphs and machine graphs. Although these are different abstractions, they both happen to be implemented using the dataflow-graph data structure provided by DFCS.

However, as we mentioned in the previous section, the input and output interfaces of a module are described by a representation and a *level*. Often, the complete representation of a program may be too unwieldy for a particular module to process. For example, a module performing an optimization on loops does not need to know the structure of the whole program. Therefore, it becomes important to describe the program in a hierarchical manner. A level is exactly this: a name for a level in the hierarchical representation of a program. For example, an Id program is described using the following levels: a program is composed of definitions, each of which is composed of some procedures. Finally, each procedure is implemented by one or more code-blocks. Figure 2.3 shows an Id program and its hierarchical representation.

A representation must also describe the levels that it supports. For example, a `parse-tree` supports the `program`, the `def`, and the `procedure` levels. Similarly, the `program-graph` representation supports these three levels plus the `code-block` level. Since we have different data structures supporting the same levels, it becomes important to describe how they correspond to each other. We use the `defcompiler-family` macro to describe these relationships.

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

The `:representation` clauses declare the different program representations used in this compiler family. The format for these clauses includes the name of the representation followed by the names of the levels it supports, from smallest to largest. The `:equivalence` clauses establish equivalence classes among different representations. These equivalence classes, known as *units*, are used by the `defcompiler` macro when checking the connectivity of modules: the condition that must be met is that the output of one module must be in the same unit as the input of the next module.

Figure 2.4 shows the definition of the Id compiler family. Notice that the definition describes three representations for Id programs: the `parse-tree` representation, the `program-graph` representation, and the `machine-graph` representation. Different parts of the compiler use the different representations of the program to facilitate their work. Moreover, the definition of the Id compiler family also describes four units, one for programs, one for definitions, one for procedures, and one for codeblocks.

The real purpose of compiler families, however, is to implement the mechanism of sharing. Modularity without sharing amounts to reinventing the wheel for every new software project. The association of data structures, modules, and options with a compiler family allows the sharing of these resources among the different compilers of the family. For example, an Id compiler that compiles Id source files shares most of its internal data structures and modules with an Id compiler that compiles source code from Id editor buffers. Without the sharing facilities provided by compiler families, each of these two compilers would have to replicate substantial amounts of code.

Currently, the Id compiler family is the only compiler family defined at MIT. However, other users of DFCS have defined compiler families for languages such as Fortran.

PROGRAM TEXT

```
%%% This is the program foo.id


def foo x =
 { def subfoo z = z * z;
   def subfoo2 z =
    { result = 1;
     in
      { for i <- 1 to z do
          next f = f * i
        finally f }};
   a = x * x
  in
    subfoo (subfoo2 a) };

def bar x = x * 2;
```

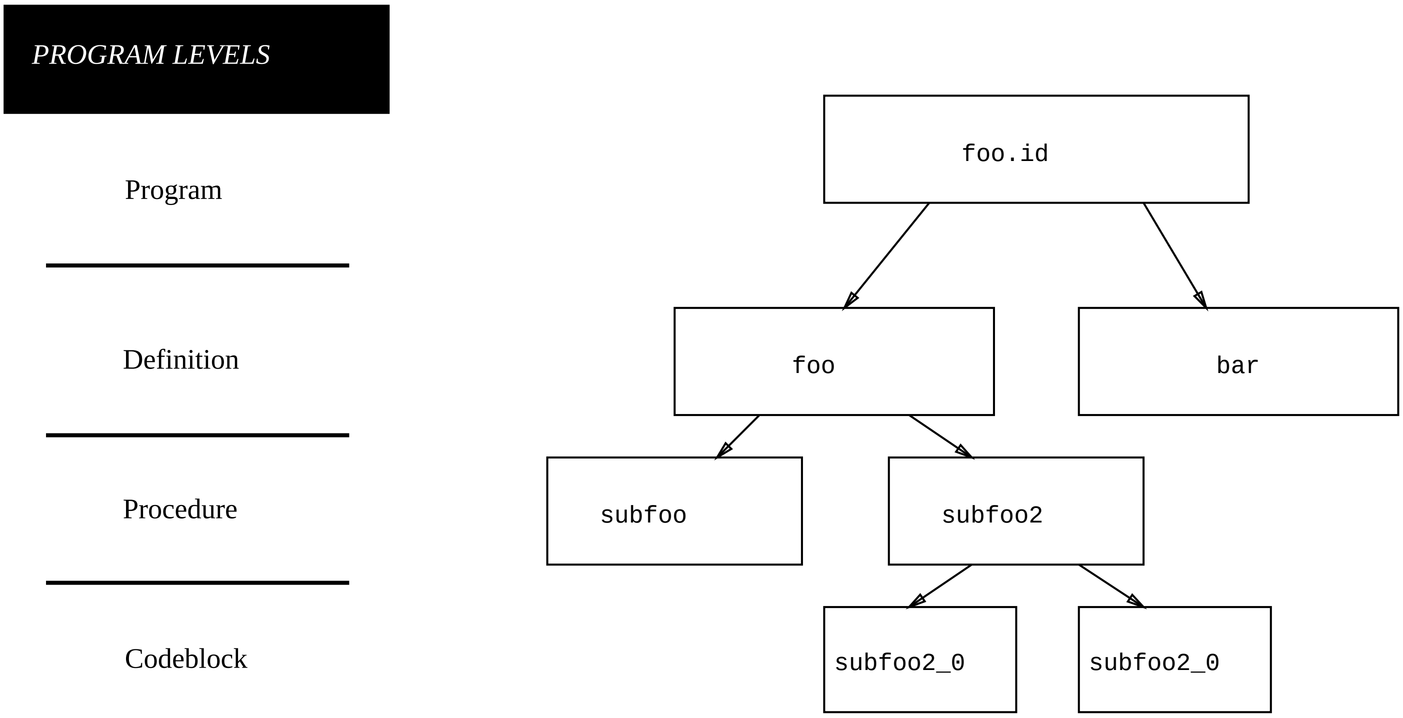

Figure 2.3: Hierarchical Representation of a Program

```
(defcompiler-family id-compiler
  (:representation parse-tree procedure def program)
  (:representation program-graph code-block procedure def program)
  (:representation machine-graph code-block procedure def program)
  (:equivalence (parse-tree program)
                (program-graph program)
                (machine-graph program))
  (:equivalence (parse-tree def)
                (program-graph def)
                (machine-graph def))
  (:equivalence (parse-tree procedure)
                (program-graph procedure)
                (machine-graph procedure))
  (:equivalence (program-graph code-block)
                (machine-graph code-block)))
```

Figure 2.4: Id Compiler Family Definition

2.3.2 Cycling the Compiler

The discussion of representations and levels naturally leads one to ask the question: how does the compiler control the level at which a particular module receives its input? After all, the user *declares* that a module will operate at a certain level, yet apart from equality between levels—as defined by the `defcompiler` macro—the user makes no explicit conversion between levels and representations. The answer to this question is known as *cycling*.

As we have seen, the definition of the Id compiler family includes a description of the hierarchical representation of programs. If we think of a representation at a certain level as a set whose elements are representations at a lower levels, then we can see that the compiler can automatically insert code to convert between different levels. This cycling code would consist of code to mapping over the elements of the set. For example, the `lambda-lift` module produces the `parse-tree` representation at the `def` level while the module the follows it, `generate-program-graph`, expects its `parse-tree` input at the `procedure` level. In this case, the compiler would cycle the `generate-program-graph` module over each of the procedures contained in each of the definitions produced by the `lambda-lift` module.

The opposite case arises when the module that occurs earlier in the compilation process produces a data structure that is at a lower level than that required by the following module. In this case, the compiler needs to *collect* the output of the earlier module into a data structure at the higher level so that it can be passed on the later module.

Readers interested in more details should consult the DFCS reference document [10].

2.3.3 Some Compiler Modules

A module can be classified as a member of one of four different classes of modules, according to inputs and outputs. The first type of module is the **generator** module. This module does not include an `:input` clause in its definition, and it is the first module of the compiler. There is only one generator module in a compiler. Figure 2.5 shows the definition of a file parser module for a file-based Id compiler.

```
(defcompiler-module file-parser id-compiler
  (:output parse-tree def)
  (:levels-marked program)
  (:before-function initialize-lexical-analyzer-for-file)
  (:function parse-def)
  (:wrapper-macro file-parser-wrapper)
  (:options input-file))
```

Figure 2.5: A Generator Module

```
(defcompiler-module call-substitution id-compiler
  (:input program-graph code-block)
  (:output program-graph code-block)
  (:options call-substitution)
  (:function cs-code-block))
```

Figure 2.6: A Standard Module

The second class of modules is the **standard** module class. This type of module receives its input at a certain representation and level and produces its output in the same representation and level. Modules that implement optimizations epitomize this class of modules since they typically perform some local analysis and perform small changes to their input. Figure 2.6 shows the definition of a module that performs the call-substitution optimization on `program-graphs` at the `code-block` level.

The third class of modules is the **translator** module class. Translator modules take input at one representation and level and produce output in another representation, but at the same level.[4] These modules mark the places where different parts of the compiler begin. For example, the parse-tree representation is used in the front-end of the compiler, program graphs are used in the middle-end, and machine graphs are used in the back end. The modules that appear on the boundaries of the conceptual "ends" of the compiler are translator modules.

Figure 2.8 contains a **collector** module. A collector module simply receives input. It produces no output apart that will pass to the rest of the compiler. A collector module is typically the last module of a compiler and produces a file or a stream of object code.

[4]Remember that conversions between levels are done automatically by the compiler. Therefore a module cannot change the vary the level of its input and output representations.

```
(defcompiler-module generate-machine-graph id-compiler
  (:input program-graph code-block)
  (:output machine-graph code-block)
  (:function gmg-dataflow-graph))
```

Figure 2.7: A Translator Module

```
(defcompiler-module assemble-monsoon-graph id-compiler
  (:input machine-graph code-block)
  (:wrapper-macro monsoon-assembler-wrapper)
  (:function assemble-monsoon-graph))
```

Figure 2.8: A Collector Module

```
(defcompiler-module display-graph id-compiler
   (:input program-graph code-block)
   (:output program-graph code-block)
   (:function display-program-graph)
   (:options reverse-video)
   (:wrapper-macro display-graph-wrapper))
```

Figure 2.9: Definition of a New Module

2.3.4 Example: Adding a New Module

Let's say we want to instrument our compiler so that it displays the intermediate representation of the program in a graphics window. A module such as this is useful when debugging the output of an optimization module, for example.

Due to the modular nature of the compiler, a module such as this is simple to add. Let's assume the task of our module is to display program graphs (rather than machine graphs) since most of the optimizations of the compiler at done with this program representation. Assume that the top-level function of our graph display module is called `display-program-graph`. Also, assume that we want to display graphs at the `code-block` level so that the graphs do not become too large. Figure 2.9 shows a possible definition for this new compiler module.

Notice that we specify the inputs and outputs to be the same representation and level. After all, this module simply displays graphs and does not modify them. Next, the `:function` clause specifies the name of the top-level function that implements this module, `display-program-graph`. The `:options` clause declares that this module makes use of the `reverse-video` option (which should have been defined already). Finally, the `:wrapper-macro` clause contains the name of the wrapper macro for this module.

Figure 2.10 shows a possible wrapper macro for this module. If the compiler is operating under the X Window System, for example, this wrapper macro takes care of opening a display connection and a top-level window.

2.4 Compiler Options

Options are additional compiler resources—shared by all compilers of a given family—that modules may use during their computations. Typically, options augment the environment in which a module operates. The `defcompiler-option` macro is used to define compiler options.

```
(defmacro display-graph-wrapper (&body body)
   `(let* ((*display* (OPEN-DISPLAY ``vinod:0''))
   (*screen* (first (DISPLAY-ROOTS *display*)))
   (*gcontext* (CREATE-GCONTEXT :drawable (SCREEN-ROOT *screen*)
                 :foreground (SCREEN-BLACK-PIXEL *screen*)
        :background (SCREEN-WHITE-PIXEL *screen*)
        :font font))
           (*top-level-window* (CREATE-WINDOW :parent (SCREEN-ROOT *screen*)
      :class :input-output
      :x 0
      :y 0
      :width 500
      :height 500
      :border-width 5
      :background (SCREEN-WHITE-PIXEL *screen*)
  )))
,@body))
```

Figure 2.10: Wrapper for New Module

`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*]

This macro adds the option *name* to the compiler family *family*. The `:type` of the option must be one of the types defined by `*option-types*`. These types are typically `:symbol`, `:string`, `:number`, `:integer`, `:date`, `:pathname`, `:boolean`, `:enumeration`, and `:stream`. The `:how-defined` clause specifies how the option is going to be defined when the lambda-list of the top-level function to invoke the compiler is constructed. The two possible values are `:keyword` or `:positional`. If the option is specified to be a `:keyword` option, then it must also have a `:default` value *default*. The `:documentation` clause specifies a string to describe the option. Finally, the `:importance` clause specifies a number between 0.0 and 1.0 that is used to sort the options when the lambda-list of the top-level function is constructed.

In the previous section, for example, we declared that the `display-graph` compiler module used the `reverse-video` option. A possible definition for this option is shown in Figure 2.11.

In order to access the value of a compiler option within a module, we use the `option` macro. In order to determine whether or not an option is valid for the current compiler, we use the `option-exists-p` macro.

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

```
(defcompiler-option reverse-video id-compiler
   (:type boolean)
   (:how-defined :positional)
   (:default nil)
   (:documentation ``This options defines whether or not graphics are
                    displayed in reverse video'')
   (:importance 0.3))
```

Figure 2.11: Definition of the `reverse-video` Option

```
(define-target-option Id-Compiler-Float-Format *Float-format*
   (:ttda 'single-float)
   (:monsoon 'double-float))
```

Figure 2.12: Definition of a Target Option

2.5 Target Options

Target options are used to set the value of variables that depend on the target architecture for which the compiler is compiling. These options do not use the `defcompiler-option` machinery defined by DFCS and described in the previous section. In particular, target options are not associated with a particular compiler family. Rather, they are defined and set with the following forms:

`define-target-option` *name variable* `&body` *alist* [*Macro*]
`set-target` *target* [*Function*]

The `define-target-option` macro defines the target-option *name* and the variable *variable* to hold the value of this target option. The body of the macro, *alist*, is an association list of targets and values. Figure 2.12 shows an example definition of a target option.

The `set-target` function sets all the options defined for a particular target to their appropriate values. In the example above, `(set-target :monsoon)` would set the value of `*Float-format*` to `'double-float`. The target options mechanism is really a tool that allows the compiler writer, specially those working on the back-end of the compiler, to write code that depends on the details of the target architecture in a reasonably portable manner.

2.6 Pragmas

While the Id compiler performs extensive optimizations on programs, there are certain situations which it cannot detect; control over these optimizations is left to the Id programmer. Currently, the compiler allows the Id programmer to control the way in which certain loops are compiled, in order to reduce overhead, and to "insert suggestions...to the compiler to generate code that releases or reuses data structure storage"[6]. The compiler provides a mechanism to define and implement these pragmatic annotations of programs (known as pragmas) in a straightforward way.

`define-definition-pragma` *keywords arglist properties* `&body` *body* [*Macro*]
`define-block-pragma` *keywords arglist properties* `&body` *body* [*Macro*]

```
(define-definition-pragma :inlineonly (name pragma dataflow-graph place) ()
  (declare (ignore name pragma place))
  (exsym-put *current-procedure-name* :inline-only t *current-compilation-exsym-table*)
  (setf (dataflow-graph-get dataflow-graph :inline-only) 't)
  (setf (dataflow-graph-get dataflow-graph :inlinablep) 't))
```

Figure 2.13: A Definition Pragma

```
(define-block-pragma :release process-release-pragma (:expression-variables-free T))
```

Figure 2.14: A Block Pragma

`define-toplevel-pragma` *keywords arglist properties* `&body` *body* [*Macro*]

There are three different kinds of pragmas supported by the Id compiler: definition pragmas, block pragmas, and toplevel pragmas. Definition pragmas apply only to the definition of the function, and they occur between the formal parameters and the "=" of a definition. Several definition pragmas can appear together; they are simply enclosed in braces. For example, the **`@inlineonly`** pragma is a definition pragma. Figure 2.13 contains the definition of this pragma, whose purpose is to allow the Id programmer to declare that the Id procedure being defined should always be inlined in its callers' code.

Block pragmas are found in bindings lists. They apply to the region of the program within the braces that enclose the pragma's binding list. A pragma in a loop binding list, for example, applies to the predicate of the loop and to the FINALLY expression of the loop. The **`@release`** is a block pragma which allows the programmer to specify that the storage for a data structure should be released (reused) upon termination of the current control region.[5] Figure 2.14 illustrates the definition of **`@release`**.

The final type of pragma is the toplevel pragma. These pragmas affect the global behavior of the compiler; they are not associated with a particular definition or binding list. The**`@include`** pragma is an example of a toplevel pragma. **`@include`** tells the compiler to include the exsym information (see Section 2.7) for a particular file during the current compilation. The inclusion of exsym information allows certain optimizations to be performed since the compiler has more information available. Figure 2.15 shows the defintion of **`@include`**.

The development of new pragmas requires a deep understanding of the compiler; the reader should not be alarmed if the topic seems confusing at first.

2.7 Exsyms

The Id compiler supports separate compilation of functions. In order to be able to support this feature, the compiler must maintain a database of information about top-level symbols. Each entry in this database is known as an *exsym*[6][7]

[5] A control region is simply a region of the program, such as a procedure body, loop iteration, or conditional expression, for which the compiler detects termination.

[6] The term exsym comes from EXternal SYMbol.

[7] The material in this section is also covered in Chapter 5 of [10].

```
(define-toplevel-pragma :include (name expression place)
                        (:processing-module pre-scope-analysis-desugaring)
  (declare (ignore name))
  (if (null expression)
      (message :warning place
       "INCLUDE pragma requires a string expression.")
      (grammarcase expression
(:string
  (let ((file (parse-library-filename (unslashify (ptnode-value expression)))))
    (let ((exsym-table (file-exsym-table file t)))
      (when exsym-table
(pushnew exsym-table (cdr *exsym-search-path*))))))
(otherwise
  (message :warning place
   "INCLUDE pragma requires a string expression."))))
  nil)
```

Figure 2.15: A Toplevel Pragma

2.7.1 Exsym Properties

Exsym entries store information about top-level symbols in *properties*. An exsym for a top-level function, for example, contains information such as the type signature of the function, the arity of the function, and whether or not the function is "inlinable". In order to access a property from an exsym entry, the following functions are used.

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

The `exsym-get` function retrieves the property *indicator* from the exsym entry for *symbol*. It searches for the exsym entry in the set of exsym tables indicated by *search-path*. For example, if one wanted to find the arity of the function `foo`, one would use the following form:

```
(exsym-get :foo :arity *exsym-search-path*)
```

The `exsym-put` function inserts a property value in an exsym entry. For example, if the compiler was processing a function of two arguments named `bar`, it would eventually use the following form to insert the arity of `bar` into the exsym database.

```
(exsym-put :bar :arity 2 *exsym-search-path*)
```

Exsym properties are defined using `define-exsym-property`.

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

This macro defines the property *indicator*. This property can be placed in exsym entries using `exsym-put`. The `:consistency-predicate` clause is necessary. It specifies the name of a predicate that checks the consistency of this property. The next section introduces the concept of consistency checking, which uses these predicates. The last two clauses, `:consistency-encoder` and `:consistency-printer` are optional.

The following functions manipulate individual exsyms.

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

The function `install-exsym` stores *exsym* into *exsym-table* under the name `(exsym-name` *exsym*`)`. `Find-exsym` finds the exsym for *symbol* in *search-path*. An exsym search path is simply a list of exsym tables. `Exsym-exists-p` is a predicate which returns `t` if an exsym for *symbol* exists in *exsym-table-or-search-path*. Finally, `copy-exsym` returns a copy of *exsym*.

The following functions manipulate collections of exsyms (exsym tables).

`make-exsym-table` [*Function*]
`map-exsym-table` *function exsym-table* [*Function*]

`Make-exsym-table` constructs a new, empty exsym table. Exsym tables essentially implement a mapping from keyword symbols (representing top-level Id identifiers) to exsym structures. `Map-exsym-table` applies *function* to each entry in *exsym-table*. The function is called for its side-effects, and it is passed the *name* of the exsym and the *exsym* structure itself as arguments.

2.7.2 Assumptions

Exsym tables store not only properties about identifiers but also *assumptions* about properties. Any time the compiler uses an exsym property of one identifier to compile code for another identifier, it records an assumption in the "assumer's" exsym. For example, if in the course of compiling function `foo`, the compiler encounters a reference to function `bar`, it looks up the `:arity` property in `bar`'s exsym in order to determine whether or not to compile a general application or a direct application.[8] The compiler records an assumption in `foo`'s exsym about `bar`'s arity. This record of assumptions is then used by the consistency checking functions to determine whether or not the functions that were compiled separately were compiled with a consistent set of assumptions about each other.

The following functions implement the assumption and consistency checking mechanisms of exsyms.

`exsym-assume` *assumer assumer-table assumee assumee-indicator assumee-search-path* [*Function*]
`exsym-assume-value` *assumer assumer-table assumee assumee-indicator assumee-search-path assumee-value* [*Function*]
`consistency-summary` *root-exsym-names search-path* [*Function*]
`describe-consistency` *root-exsym-names search-path* [*Function*]

The `exsym-assume` and `exsym-assume-value` functions are used to record assumptions into exsyms. Following the example above, the form below would record an assumption about `bar`'s arity in `foo`'s exsym.

[8] General applications involve building a closure; they are used when the arity of a function is unknown or when the function application is a partial one. Direct applications are compiled when the arity of a function is known and satisfied. Direct applications are more efficient than general applications since they do not build closures!

```
(exsym-assume :foo exsym-table-1 :bar :arity *exsym-search-path*)
```

Notice that an exsym for the assumee must exist; otherwise, the value of the property (the arity, in this example) could not be determined. The difference between `exsym-assume` and `exsym-assume-value` is that the latter allows for the assumed value to be provided directly. This obviates the need for the assumee to have an exsym.

The consistency checking functions determine whether the assumptions recorded in the exsym table are consistent with each other. As an example of an inconsistency, suppose that `bar` gets recompiled, after `foo` has been compiled, so that it takes one more argument than it did before. If a direct application of `bar` was compiled into `foo`, due to the value of `bar`'s `:arity` property at the time `foo` was compiled, the resulting state of the compiled code in the machine would be incorrect! If foo were to be executed, the new version of `bar` would not receive enough arguments! When called with the appropriate "root" set of exsyms, the `consistency-summary` function would detect this problem and would report it. The `consistency-summary` applies the consistency predicate, specified when the exsym property was defined, to the assumed value stored in the assumer's exsym and to the actual value, stored in the assumee's exsym. If these values fail to pass the consistency predicate's test, `consistency-summary` records them and returns them as part of its result.

The `describe-consistency` function prints a detailed description of the inconsistencies found by *consistency-summary*, using the consistency printers defined with each exsym property.

Typically, execution managers call `consistency-summary` before executing functions to ensure that no error will occur due to separate compilation problems.

2.8 User Features

The following sections discuss features of the compiler that are more "user oriented" than "compiler-writer oriented". They are included here because they should form part of the reader's mental model for the Id compiler.

2.8.1 Incremental Compilation

Traditional compilers operate in a batch-oriented manner: the user "submits" a program to the compiler, the compiler attempts to compile the program, and finally, the compiler produces object code, assuming there were no syntax or type errors in the source program. While the Id Compiler can operate in this way, it can also operate *interactively* and *incrementally*. Incremental compilation allows the user to compile a program in small pieces until a complete program is compiled. In the case where a particular function needs to be recompiled, the user need only recompile *that* function. The work done to compile all the other functions in the program does not need to be repeated uselessly.

When compiling incrementally, the Id compiler may take its input from an editor buffer, for example. By selecting a region of text, the user may request that a particular function be compiled, that a set of functions be compiled, or that the whole buffer be compiled. In any case, localized changes to the source program require only localized recompilation of functions.

As a consequence of its ability to support incremental compilation, the Id compiler also supports *separate compilation*. As mentioned previously, separate compilation allows groups of functions that refer to each other to be placed into separate files at compile time. This allows for a more modular programming style and provides the ability to build useful libraries of functions. References from functions in one file to functions in another file are resolved when the object code for the functions is loaded into the machine.

2.8.2 Librarian

The previous sections discussed the ability of the Id compiler to support separate compilation and mentioned that references to functions and other top-level identifiers must be resolved at load time. The compiler generates information about these symbols in `.etb`[9] files. The purpose of the librarian is to manage the exsym table information stored in `.etb` files. As Id files as compiled, the librarian keeps a cache of the exsym information produced by previous compilations so that subsequent invocations of the compiler can use this information.

2.9 Summary

This chapter has attempted to present a high level overview of the Id compiler by introducing a broad spectrum of topics and by keeping the discussions of each of these topics short and concise. These quick glimpses into the structure of the compiler should provide the reader with some intuition behind its underlying ideas. Later chapters will develop the topics touched in this chapter and will provide more precise detail.

[9]The suffix `.etb` stands for Exsym Table (Binary) format.

Chapter 3

Front-end

The front-end of the Id compiler is organized as shown in figure 3.1:

The front-end currently consist of the following modules:

- file-parser
- pre-scope-analysis-desugaring
- scope-analysis
- type-checking
- overloading-translation
- lambda-lift

The first module, file-parse, reads in the source code and performs lexical analysis and parses the input. The output from the lexer is a stream of tokens or lexemes, which is fed into the parser. The parser outputs a parse tree, which is the data-structure that the rest of the front-end modules operate upon. The file-parser can be called repeatedly on the same input file to obtain several parse trees if the input file contains the code for more than one parse trees. The functions of the other modules in front-end are described in the next chapter.

The lexical token and parse tree data structures are discussed in great detail in Chapter 3 of *DFCS: Dataflow Compiler Substrate Manual*[10]. Note that parse trees are slightly different from syntax trees. They represent the same information in a more compact form by suppressing syntax tree nodes that do not contain any additional information. The differences and motivations for doing this are again described in Chapter 3 of *DFCS: Dataflow Compiler Substrate Manual*[10]. In the following sections, we will first describe the tools used in building the lexer and the parser, give some examples, and proceed with the description of the rest of the modules in the Id compiler.

3.1 PAGEN

PAGEN is a lexer and LALR parser generator for Common Lisp[8]. Given a definition of a grammar, PAGEN generates the tables used by the table driven lexer and parser. In addition, it also generates the `defgrammar` and `defproduction` code for the input grammar. For more details about `defgrammar` and `defproduction`, please refer to Chapter 4 of *DFCS: Dataflow Compiler Substrate Manual*[10]. Both PAGEN itself and the lexer/parser are written in Common Lisp.

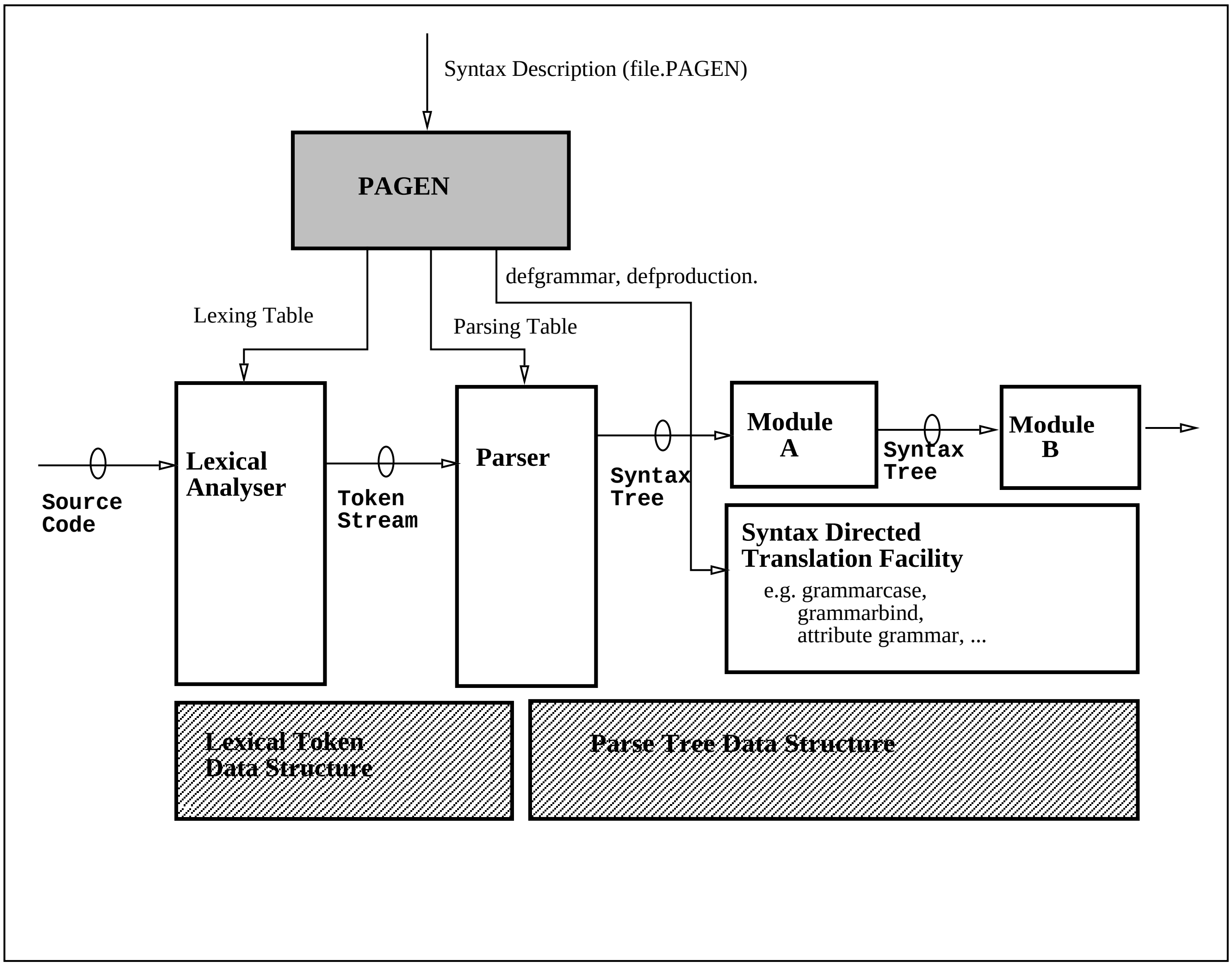

Figure 3.1: Id compiler front-end

3.1.1 PAGEN Input

A PAGEN grammar definition is a Backus-Nauer Format (BNF) definition with a few extensions for common cases. The PAGEN input file must have a filename ending with a `.Pagen` suffix. It consists of a series of statements that do such things as tell what machines to generate scanners for, what package the resulting file should be in, *etc.*It is interesting to note that the grammar for the PAGEN input language itself is defined by a PAGEN file, and PAGEN generated the parser for the PAGEN input language.

In this document, we will give the grammar for PAGEN in BNF. Appendix pagen-pagen contains the PAGEN file for the grammar of PAGEN's input language. Terminals will be written as `terminal`.

3.1.2 Grammar

A PAGEN grammar definition consists of a series of statements. Each statement starts with a PAGEN keyword, and may be on more than one line. Keywords in PAGEN start with a '$' sign and are case-insensitive. PAGEN statements can be divided into 2 categories. Those used for specifying the lexemes, and those used for specifying the parse trees. Note however that in PAGEN, both go into the same input file. (In contrast, in C/UNIX, LEX handles the former while YACC the latter.) We will first describe those used for specifying lexemes and give some examples, before going on to those used for specifying parse trees. In the course of the description, non-terminals productions of the PAGEN input language will be written in small-caps fonts, *e.g.,*TARGET-STATMENT.

3.1.3 Lexical Analyzer

The lexical analyzer, or lexer, is defined by a series of lexical statements.

Target Statement

The `target` statement specifies what target system to make lexers for. Some targets have different character sets from others, and a mapping from native character codes to internal character codes will be made for each target. In addition, characters that are indistinguishable in the grammar (*e.g.,*whitespace characters) will be mapped to the same internal code in order to compact the scanner tables.

TARGET-STATEMENT $\rightarrow$ `$target identifier`

Currently the available targets are: `lispm` and `unix`.

Let Statements

A `let` statement maps an identifier to a regular expression for use in definition of lexical tokens.

LET-STATEMENT $\rightarrow$ `$let identifier` = REGULAR-EXPRESSION

The identifier defined is a meta-variable used in other lexical statements, i.e. the identifier is a variable of the input to PAGEN.

Regular Expressions

Lexical tokens are defined by regular expressions. The lexer reads the input stream, finds the longest string that matches one of the regular expressions, and then returns the corresponding lexical token. If the longest string matching any regular expression matches more than one regular expression, the regular expression appearing earliest in the `.PAGEN` file takes priority.

Regular expression specification in the PAGEN input language is defined by the following grammar (The following is an ambiguous grammar that specifies the same language as the unambiguous grammar used by PAGEN, but is easier to read):

REGULAR-EXPRESSION	→	REGULAR-EXPRESSION ”| ” REGULAR-EXPRESSION
REGULAR-EXPRESSION	→	REGULAR-EXPRESSION REGULAR-EXPRESSION
REGULAR-EXPRESSION	→	REGULAR-EXPRESSION ”*”
REGULAR-EXPRESSION	→	REGULAR-EXPRESSION ”+”
REGULAR-EXPRESSION	→	PRIMARY-REGULAR-EXPRESSION
PRIMARY-REGULAR-EXPRESSION	→	`identifier`
PRIMARY-REGULAR-EXPRESSION	→	`set`
PRIMARY-REGULAR-EXPRESSION	→	`string`
PRIMARY-REGULAR-EXPRESSION	→	`caseless-string`
PRIMARY-REGULAR-EXPRESSION	→	”~” PRIMARY-REGULAR-EXPRESSION
PRIMARY-REGULAR-EXPRESSION	→	”(” PRIMARY-REGULAR-EXPRESSION ”)”

where `identifier`, `set`, `string`, and `caseless-string` are defined as:

`identifier`	=	ALPHA {(ALPHA | DIGIT)}*
`set`	=	“{” {(SET-CONTENTS | “\” ANY)}* “}”
`string`	=	“"” {(STRING-CONTENTS | “\” ANY)}* “"”
`caseless-string`	=	“‘” {(CASELESS-STRING-CONTENTS | “\” ANY)}* “’”

As in BNF, “`*`” means zero or more occurences of the preceding regular expression (Kleene recursion), and “`+`” means at least one occurence of the preceding regular expression. The definitions of ALPHA, and DIGIT are self explanatory. ANY means any character in the character set. SET-CONTENTS is any character except right brace (`}`) and backslash, unless these are preceded by the escapes character, backslash. STRING-CONTENTS is any character except double-quotes and backslash. Again, these charaters can be included by preceding them with the backslash escape character.CASELESS-STRING-CONTENTS is any character except close-quote and backslash. As before, a preceding backslash allows us to include these characters in CASELESS-STRING-CONTENTS. A few examples will make the meaning of these terminals clear.

Examples

1) An `identifier` is a meta-variable defined in a `$let` statement. When used on the RHS, the corresponding expression that the `identifier` denotes is substituted in its place. For example, if we have

```
$let digit = 0123456789
```

then the regular expression for number is

```
digit*
```

and is equivalent to

$\{0123456789\}^*$

2) A `set` represents any single character string within the set. An example of a `set` is

```
{abc}
```

and is equivalent to

```
"a"|"b"|"c"
```

3) A `string` is any sequence of characters enclosed by double quotes. An example is:

```
"aBc"
```

which only matches the string

```
"aBc"
```

4) A `caseless-string` on the other hand ignores the case of the letters. So

```
'aBc'
```

matches any one of the following strings:

```
"abc"|"ABC"|"Abc"|"aBc"|"abC"|"ABc"|"aBC"|"AbC"
```

The productions of REGULAR-EXPRESSION are self-explanatory and similar to BNF grammar. The only exception is that involving `~`. This is only used for `~exp` where `exp` must match only single characters. `~exp` matches any character other than those in `exp`.

Lexical Tokens

There are three kinds of lexical tokens in Pagen, *terminals*, *pseudo-terminals*, and *discards* and they are defined by their respective statements:

TERMINAL-STATEMENT	→	`$terminal identifier` = REGULAR-EXPRESSION
PSEUDO-TERMINAL-STATEMENT	→	`$pseudo identifier` = REGULAR-EXPRESSION
DISCARD-STATEMENT	→	`$discard identifier` = REGULAR-EXPRESSION

A *terminal* is a lexical token used by the parser. No ptnode is built by the parser for a terminal lexical token. The string making up a terminal is not saved. *Discard* tokens are recognized by the lexer and discarded. They are not passed on to the parser. A *pseudo-terminal*, like a *terminal*, is used by the parser, but also becomes a ptnode itself. During lexical analysis, the string making up a pseudo-terminal is saved, and subsequently used by the parser. For instance, identifiers in an input program are pseudo-terminals — the `ptnode-value` of an identifier is the string of characters that names the identifier. Pseudo-terminals have values that are used during the static-analysis phase of compilation.

Before proceeding with the description of those parts of PAGEN concerned with parsing, we give a complete example of an input file to PAGEN specifying a lexer.

Example

The following is a specification of a lexer for mathematical expressions:

```
$let            alpha = {ABCDEFGHIJKLMNOPQRSTUVWXYZ}|
                        {abcdefghijklmnopqrstuvwxyz_-}
$let            digit = {0123456789}

$terminal       left-parenthesis = "("
$terminal       right-parenthesis = ")"
$terminal       plus = "+"
$terminal       minus = "-"
$terminal       times = "*"
$terminal       divide = "/"
$terminal       semicolon = ";"
$terminal       equal = "="

$pseudo         id = alpha (alpha | digit)*
$pseudo         number = digit+

$discard        whitespace = (space | newline | tab | return | page)+
```

Given the input: `offset = 3 + index` ,

the above lexer recognizes a pseudo-terminal `id` with value `offset`, the `terminal '='`, another pseudo-terminal `number` with value `3`, followed by another `terminal '+'`, and finally a last pseudo-terminal `id` with value `index`.

3.1.4 The Parser

The parser is defined by a series of grammar statements. There are five kinds of grammar statements:

Acceptance Statement

The acceptance statement tells the parser what to return as a parse tree. When the parser is called, it will parse until it processes the accept production, and return the parse tree. There are two forms: *accept* and *accept-each*. The *identifier* below names a non-terminal in the grammar being defined.

ACCEPT-STATEMENT	$\rightarrow$	`accept` *identifier*
ACCEPT-EACH-STATEMENT	$\rightarrow$	`accept-each` *identifier*

The *accept* form means that the whole file or stream must parse into exactly one parse-tree. The *accept-each* form means that the parser can be called repeatedly until the file is empty. Each time the parser is called, it will parse and return one parse-tree.

significant

This defines a normal production. When the parser reduces this production it makes a ptnode whose tag is derived from the name of this production, and leaves the new ptnode on the stack.

no-node

This defines a *linear* production, that is, the right hand side contains only one non-terminal. It is used for parsing, but it really conveys no useful information beyond the parser, so no ptnode is made for this production. For example:

```
$no-node primary-regular-expression -> identifier
```

When the parser reduces the identifier to a primary-regular-expression, it just leaves the identifier on the stack instead of making a primary-regular-expression ptnode out of it.

n-ary

This is an extension to the normal syntax of context-free grammars. For example, the statement

```
$n-ary   tuple-expression -> expression $separated-by ","
```

defines *tuple-expression* to be any number of *expressions* separated-by commas. This production takes the place of a number of simpler productions. The ptnode formed will have as its children a list of all the child ptnodes that were separated by the separator tokens.

To specify a number of non-terminals separated by whitespace, use the following format.

```
$n-ary   production-rhs -> production-rhs-primary $separated-by
```

n-ary-1

The `n-ary-1` is a combination of `n-ary` and `no-node`. If there is at least one separator token, then it behaves like an `n-ary` production. If there is no separator, then it behaves like a `no-node` production. This is what we use for *tuple-expressions* if we allow a single expression to be considered a degenrate case of *tuple-expression*:

```
$n-ary-1  tuple-expression -> expression $separated-by ","
```

Example

The following is a grammar specification for a parser, where `id` and `number` are as defined in the lexer example:

```
$significant         proc -> "def" id id "=" expr ";

$no-node             expr -> add-expr

$no-node             add-expr -> mul-expr
$significant         add-expr -> add-expr "+" mul-expr
$significant         add-expr -> add-expr "-" mul-expr

$no-node             mul-expr -> primary-expr
$significant         mul-expr -> mul-expr "*" primary-expr
$significant         mul-expr -> mul-expr "/" primary-expr

$no-node             primary-expr -> "(" expr ")"
$no-node             primary-expr -> block-expr
```

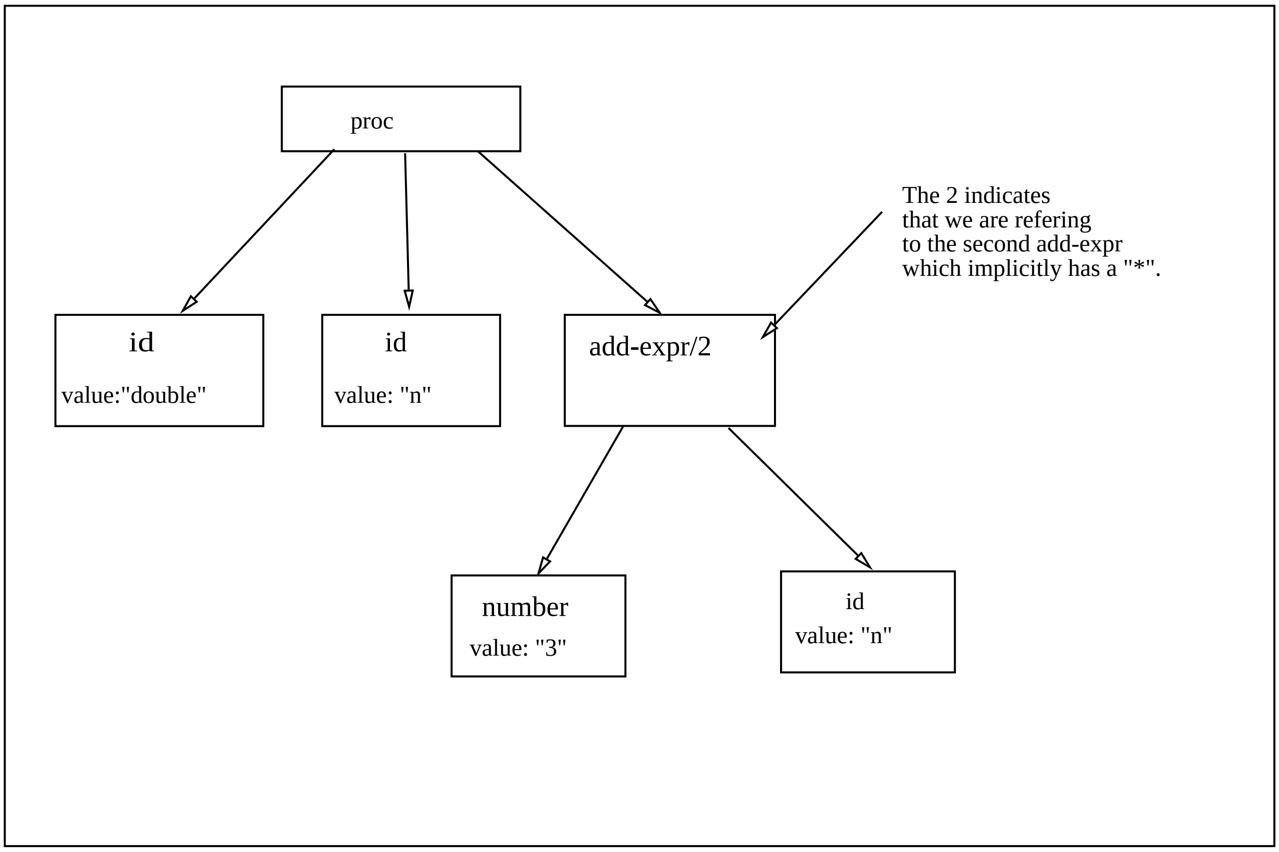

Figure 3.2: Parse tree for: def double n = 2 * n;

```
$no-node            primary-expr -> id
$no-node            primary-expr -> number

$significant        block-expr -> "{" blist in expr "}"

$n-ary              blist -> binding $separated-by ";"

$significant        binding ->
$significant        binding -> id "=" expr

$accept             proc
```

Given the input file:

```
def double n = 2 * n;
```

the parser produces the parse tree shown in figure 3.2

3.1.5 Running Pagen

First, make a grammar definition, in a pagen file (with suffix "`.pagen`"). Next bring up a LISP process and do the following to load PAGEN:

```
(load "/df/id-world/utilities/011/loop")
(load "/df/id-world/utilities/011/out-file-system")
(load "/df/id-world/utilities/011/defprogram")
(load "/df/id-world/csg/registry/021/registry")
(make-program :pagen :noconfirm)
```

Finally:

(pagen:pagen *file*)

will generate a file with the extension "`.pagen-output`" that contains the tables for the scanner and parser. The functions that perform the shifts and reductions in the parser resides in `parser.lisp` in the PAGEN directory and need to be added manually. The "`.pagen-output`" file will include an `in-package` statement naming the package specified by the package-statement in the PAGEN file.

Using the Parser

The parser should be in a package that *uses* the Lisp and DFCS package. See the *DFCS: Dataflow Compiler Substrate Manual*[10] for support manipulating parse trees. Some examples are given in the next section.

3.2 Working with parse tree data structures

Selectors, constructors and mutators for working with parse tree data structures are defined in DFCS and documented in *DFCS: Dataflow Compiler Substrate Manual*[10]. We will not repeat the definitions here, but will instead give some examples which illustrate use.

3.2.1 Examples

Example 1

In the first example, we make use of the DFCS functions: `block-place, ptnode-tag, ptnode-child, ptnode-value, ptnode-children, ptnode-place, ptnode-line, parse-tree-root`. The example operates on a block-structured language and prints out the place where each variable binding occurs. The place in this case refers to the line number of the beginning of the enclosing block. Both `:identifier` and `:block-expr/1` are tags associated with productions.

```
;; top level function.
(defun print-binding-occurences (parse-tree)
   (pbo-expr (parse-tree-root parse-tree) nil))


;; pbo-expr recrusively walks down a parse tree from the root.
;; If the current node is an :identifier, we reach a base case.
;;   Nothing is done in the base case as there no binding takes place.
;; If the current node is a :block-expr/1, the 0th child of the
;;   node is a subtree of bindings, while the 1st child is a subtree
;;   containing the body of the block.  pho-expr calls pbo-blist to
;;   print out the binding-occurences and recursively calls itself on
```

```
;;   the body of the block.
;; For any other internal ptnodes, pbo-expr calls itself recursively
;;   on every subtree.
(defun pbo-expr (ptnode block-place)
   (case (ptnode-tag ptnode)
     (:identifier
      nil)
     (:block-expr/1
      (let ((blist (ptnode-child 0 ptnode))
            (in-expr (ptnode-child 1 ptnode)))
         ;; process bindings
         (pbo-blist blist (ptnode-place ptnode))
         ;; process "in" expr
         (pbo-expr in-expr block-place)))
     (otherwise
      (dolist (child (ptnode-children ptnode))
         (pbo-expr child block-place)))))


;; blist is a list of bindings.
;; block-place is the 'place' where these bindings occur.
;; pbo-blist prints out the information that the bindings in
;;   in blist occurs at block-place.
(defun pbo-blist (blist block-place)
   (dolist (binding (ptnode-children blist))
      (pbo-binding binding block-place)))

;; binding is a the ptnode of a single binding.
;; block-place is the 'place' where the binding occurs.
;; pbo-binding prints out the information that the binding
;;   occurs at block-place.
(defun pho-binding (binding block-place)
   (let ((lhs (ptnode-child 0 binding))
         (rhs (ptnode-child 1 binding)))
      ;; print this identifier
      (format t "Block at ~a; identifier ~a~%"
              (ptnode-line block-place)
              (ptnode-value lhs))
      ;; process the rhs
      (pbo-expr rhs block-place)))
```

When the above is run on the following input:

```
def examp x =
  {y = x + 3;
   z = {q = y + 7;
        r = q + 3;
        in
          r};
```

```
    in
      w}
```

we get the output:

```
Block at 2; identifier y
Block at 2; identifier z
Block at 3; identifier q
Block at 3; identifier r
```

Example 2

The definition of `pbo-expr` can be recoded as follows using `grammarcase`. `Grammarcase`,`grammarbind`, and `gramarcasebind` used in this and the following example are described in chaper 4 of DFCS.

```
(defun pbo-expr (ptnode block-place)
   (grammarcase ptnode
     (:identifier
      nil)

     ((block-expr -> "{" blist in expr "}")
      (let ((blist (ptnode-child 0 ptnode))
            (in-expr (ptnode-child 1 ptnode)))
         ;; process bindings
         (pbo-blist blist (ptnode-place ptnode))
         ;; process "in" expr
         (pbo-expr in-expr block-place)))

     (otherwise
      (dolist (child (ptnode-children ptnode))
         (pbo-expr child block-place)))))
```

`Grammarcase` allows us to use the prodspec instead of the tag. It also takes care of taking the tag of ptnode. A prodspec is a representation of a production. In the above example,

```
block-expr -> "{" blist in expr "}"
```

is a prodspec. It is possible to give a list of prodspec in place of one prodspec if the same code is excuted for all of them. Note that we can still use keyword tags (such as `:identifier`) within a grammarcase statement.

Example 3

The definition of `pbo-binding` can be recoded to use `grammarbind`:

```
(defun pho-binding (in-binding block-place)
   (grammarbind (binding -> id "=" expr)
                in-binding
      ;; print this identifier
```

```
    (format t "Block at ~a; identifier ~a~%"
            (ptnode-line block-place)
            (ptnode-value (ptnode id))) ; (ptnode id) returns the child
                                        ; of in-binding corresponding to id.

    ;; process the rhs
    (pbo-expr (ptnode expr) block-place))) ; (ptnode expr) is similar to
                                           ; (ptnode id)
```

In cases where a production has repeated occurence of the same non-terminal, we need to explicitly name prodspec components. For instance, if we have the following production:

```
add-expr -> add-expr "+" mul-expr
```

and we use it in `grammarbind`, `(ptnode add-expr)` is ambiguous. The way to get around it is to explicitly name the components. We can use

```
add-expr -> (add-expr op1) "+" (mul-expr op2)
```

in a `grammarbind` expression, and use `(ptnode op1)` to obtain the ptnode for the add-expr on the RHS.

Example 4

The last example here illustrates the use of `grammarcasebind` which combines the functionality of `grammarcase` and `grammarbind`:

```
(defun pbo-expr (ptnode block-place)
   (grammarcasebind ptnode
     (:identifier
      nil)

     ((block-expr -> "{" blist in expr "}")
      ;; process bindings
      (pbo-blist (ptnode blist)
                 (ptnode-place ptnode))
      ;; process "in" expr
      (pbo-expr (ptnode expr)
                block-place))

     (otherwise
      (dolist (child (ptnode-children ptnode))
         (pbo-expr child block-place)))))
```

Multiple prodspec's can be used as a pattern for `grammarcasebind` matching. Note however that they must have the same number of ptnodes on the RHS, and syntactically identical names must be used for corresponding components. Explicit naming as explained in the previous example can be used obtain identical names. For instance, the following list can be used as multiple prodspec. Access to ptnode will be through `(ptnode lb)` and `(ptnode e)`.

```
(((block-binding lb) -> call (expression e))
 ((block-binding lb) -> (keyword-expression e))
 ((loop-binding lb) -> call (expression e))
 ((loop-binding lb) -> (keyword-expression e)))
```

3.2.2 Attribute Grammar

In the course of working with parse trees, we often want to associate information with each ptnode. Attribute grammar provides a simple way of doing this when the information can be computed in a structured way from other ptnodes of the production. With an attribute grammar, we associate with each production rules for computing an attribute of the ptnodes related by the production. For example, we can associate an attribute *free* with each ptnode that denodes the free variables in the subtree of the syntax tree rooted at that ptnode. Then the rule for computing the *free* attribute for a production such as:

```
add-expr1 -> add-expr2 "*" mul-expr
```

is

free(add-expr1) = *free*(add-expr2) ∪ *free*(mul-expr)

Attributes are either *synthesized* or *inherited*. An attribute is *synthesized* if all definitions of the attribute are for LHS of productions. In term of the parse tree, the attribute of a ptnode is calculated from the attributes of the children of that ptnode. One the other hand, an attribute is *inherited* if all definitions of the attribute are for RHS of productions. The attribute of a ptnode in a parse tree is, in this case, calculated from the attributes of its parent, hence the name inherited.

The tools for working with attribute grammars are defined in DFCS and documented in Chapter 4 of *DFCS: Dataflow Compiler Substrate Manual*[10]. Again as in previous sections, we will only give examples of their use. The reader should refer to the DFCS document for details. In particular, the reader should look at the 3 different storage methods for attributes, `:ephemeral, :memoized, and :permanent`, which are discussed in detail there.

Example 1

The following example defines the rules for 5 attributes for the default production. Default production refers to the special production

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

that PAGEN matches to any production. The attribute `ptnode-non-generic-frame` is an inherited attribute, while all the others, `ptnode-type`, `ptnode-defined-type-env`, `ptnode-idnodes`, and `ptnode-nexified-variables`, are synthesized attributes.

```
(defattributes ($lhs -> $rhs-component $separated-by)
   ((ptnode-type $lhs)       ; defining attribute ptnode-type for $lhs
    (ptnode-type (ptnode $rhs-component 0)))
                             ; <ptnode $rhs-component 0> returns the
                             ; 0th ptnode on the RHS.

   ((ptnode-defined-type-env $lhs)
    nil)
```

```
((ptnode-idnodes $lhs)
 nil)

((ptnode-non-generic-frame $rhs-component i)
                            ; defining attribute ptnode-non-generic-frame
                            ; for ith component of the RHS
 (ptnode-non-generic-frame (ptnode $rhs))
                            ; <ptnode $rhs> returns the ptnode on the RHS.

((ptnode-nexified-varibales $lhs)
 nil))
```

Our attribute system computes attributes in a lazy fashion, and can handle any attribute not defined circularly. In addition, it also invalidates `:ephemeral` attributes when the structure of the parse tree changes in such a way as to affect the value of that attribute.

Example 2

We can also define rules that apply to a specific set of productions. In this case, all the productions have to have the same number of children. In addition, the LHS and RHS must be renamed in the same way as when we use multiple prodspec's in `grammarcasebind` (Example 4 of the previous section).

```
(defattributes (((block-binding lb) -> call (expression e))
                ((block-binding lb) -> (keyword-expression e))
                ((loop-binding lb) -> call (expression e))
                ((loop-binding lb) -> (keyword-expression e)))

   ((ptnode-idnodes lb)
    (let ((lb-node (ptnode lb)))
      (make-idnode-list nil (ptnode-free lb-node) lb-node))))
```

This concludes the description of PAGEN and the attribute grammar tools in DFCS. The next chapter will describe the other front-end modules.

Chapter 4

Front End Modules

4.1 Introduction

Unlike some compilers which generate low-level code directly, the Id compiler uses parse-trees as one of its intermediate representations for programs. More detailed documentation about the parse-tree data structure can be found in [10]. The front end of the Id compiler is the group of modules which operates on the parse-tree representation.

Id Source File
$\downarrow$
`file-parser` *or* `stream-parser`
$\downarrow$
`pre-scope-analysis-desugaring`
$\downarrow$
`scope-analysis`
$\downarrow$
`type-checking`
$\downarrow$
`overloading-translation`
$\downarrow$
`lambda-lift`
$\downarrow$
Id Compiler Middle End

Figure 4.1: Structure of Current Id Compiler Front End (3/18/91)

4.2 The Modules

Each module description briefly describes some high-level characteristic of the module, including:

Module: The `defcompiler` module definition.

Options: A description of the module's options.

Description: A declarative description of the module's effect on the parse tree.

Strategy: The algorithms used by the module.

Author: Primary person responsible for coding.

Caveats: Known limitations, warnings, points of interest, etc.

Each module in the front end of the compiler reads or modifies a number of global data structures in addition to its explicit parse-tree argument and result. The following entries, for each module, provide information about its usage of global data structures:

Parse-Tree Properties: Parse-tree node properties.

Parse-Tree Slots: Parse-tree node slots.

Not each module has all the entries filled in. If a module is the first to define an entry in a global structure or is the exclusive user of an entry then the entry's name is in **boldface** followed by a description. Otherwise, just the name is listed.

4.2.1 Parsing

Module:

```
(defcompiler-module file-parser id-compiler
  (:output parse-tree def)
  (:levels-marked program)
  (:before-function initialize-lexical-analyzer-for-file)
  (:function parse-def)
  (:wrapper-macro file-parser-wrapper)
  (:options input-file))

(defcompiler-module stream-parser id-compiler
  (:output parse-tree def)
  (:levels-marked program)
  (:before-function initialize-lexical-analyzer-for-stream)
  (:function parse-def)
  (:wrapper-macro stream-parser-wrapper)
  (:options input-stream initial-character
            initial-line initial-column input-stream-truename))
```

Options:

- `input-file` is the filename of the Id file to be compiled. This is a positional argument which has the highest importance (meaning that this argument will appear first in the argument list of the compile function). This option is only used for file-compiles.

- `input-stream` is the stream that is passed to the compiler. This is a positional argument which also has the highest importance. (This argument will also appear first in the argument list of the compile function. However, there is no conflict with the `input-file` option, because only one of these two options is ever specified for any compile function.) This option is only used for editor-compiles.

- `initial-character` is the initial setting for the counter that the parser uses to keep track of where each lexeme comes from in the editor buffer. Eventually, this information is used for error messages and source debugging information. This option is only used for editor-compiles, and the editor calculates the character offset from the beginning of the buffer. (For file-compiles, the character counter is automatically set to 0, so there is no need for an option.)

- `initial-line` is similar to the `initial-character` option, except that this is the line offset from the beginning of the buffer.

- `initial-column` is similar to the previous two options, except that this is the column offset from the current line.

- `input-stream-truename` is only used for editor-compiles. This is the truename of the file which is associated with the buffer in which the region or definition is being compiled.

Description: Only one of these two modules (`file-parser` and `stream-parser`) are used, depending upon whether we are compiling a file or an editor buffer. `file-parser` reads from a file stream, and `stream-parser` reads from an editor stream.

The two modules are similar: they read the parser tables generated by PAGEN and parse the input stream. The *only* function of these modules is to produce parse-trees from Id source.

Strategy: The parser is a LALR(1) parser. More information about parsing can be found in [1].

The chapter on PAGEN describes how to change the grammar of the language. Both `file-parser` and `stream-parser` are just interpreters for the parser tables generated by PAGEN.

Author: Ken Traub, May 1986

4.2.2 Desugaring

Module:

```
(defcompiler-module pre-scope-analysis-desugaring id-compiler
  (:input parse-tree def)
  (:output parse-tree def)
  (:options loop-bound-mode)
  (:function pre-scope-analysis-desugar-def))
```

Options:

- `loop-bound-mode` selects whether `for` loops will be bounded or unbounded. This option is set to either `:bounded-for-loops` or `:unbounded-for-loops`. If the option is set to `:bounded-for-loops`, then the compiler will generate the bounded loop schema, which has slightly more overhead than the unbounded loop schema.

  The default is `:unbounded-for-loops`.

Description: Several types of desugaring are done in this module:

1. List comprehensions, array comprehensions, accumulator comprehensions, and stream comprehensions are translated into loops.
2. Array literals are desugared into either a while loop or a list of assignments, depending upon whether its bounds are known at compile time.
3. Managers are desugared to abstract types.
4. Multi-clause `def`'s are desugared to case statements.
5. Lambda's (`fun`'s) are desugared to be internal `def`'s (which are eventually lifted to the top-level by the lambda lifting module).
6. `for` loops are transformed into `while` loops.

Strategy: Recursively descend the parse tree. When you find a pattern that matches one that you wish to desugar, cut and paste that pattern.

Author: Ken Traub, January 1987

4.2.3 Scope Analysis

Module:

```
(defcompiler-module scope-analysis id-compiler
  (:input parse-tree program)
  (:output parse-tree program)
  (:function scope-analyze-program))
```

Description: `variable` objects are added to parse tree nodes relating valriable definitions and their uses (def-use information). This analysis is done using the attribute grammar which is described in more detail in [10]. More information about attribute grammars can be found in [1] and in the previous chapter.

Strategy: To compute def-use information, a synthesized pt-node attribute is defined for the definition point of each variable use, and an inherited attribute is defined for the use point of each variable definition. Each node is then touched to calculate the value of each attribute variable.

The type inference system also uses attribute grammars to calculate the types.

Author: Ken Traub, January 1987

4.2.4 Type Inference and Checking

Module:

```
(defcompiler-module type-checking id-compiler
  (:input parse-tree program)
  (:output parse-tree program)
  (:function type-check-program)
  (:options type-check))
```

Options:

- `type-check` determines whether the compiler will type infer and type check. If this option is set to `t`, then the compiler will infer and check the types of the program; if this option is set to `nil`, then the parse tree will pass through this module without alteration.

  The default is `t`.

Description: The type-inference system determines the type of each variable and function. A more detailed description of the type-inference system can be found in Shail Aditya's Master's thesis [4]. The type-inference system is based upon the Hindley/Milner type-inference system used in ML, and has been extended to work with separate compilation. A clear exposition of the Hindley/Milner type-inference algorithm can be found in [3].

Strategy:

Author: Shail Aditya, July 1988

Caveats: The type-inference module has the same limitations that the Hindley-Milner system has. It has exponential complexity, and limits the polymorphism of let-block assigned variables.

In addition, in order to resolve the type of certain functions, sometimes separate compilation will not work appropriately. Although information is preserved between separate compilations, if that information changes, then enough context must be compiled in order to resolve the types of the changed functions.

4.2.5 Overloading Resolution

Module:

```
(defcompiler-module overloading-translation id-compiler
  (:input parse-tree program)
  (:output parse-tree program)
  (:function translate-overloaded-program))
```

Description: Most of the work for this module is actually done in the `type-checking` module. This module takes the information deduced in the `type-checking` module and cuts and pastes parse tree nodes so that each overloaded operator is transformed into the correct type instance of that operator.

After this module, each operator has an unambiguous type. For example, all `+` operators will be translated to either `plus~float` or `plus~int`. If the operator cannot be resolved, an error message is printed.

Strategy: With the type information calculated in the `type-checking` module, the context of each overloaded operator resolves the type of the operator. Each node is visited, and if that node is an overloaded operator, the type of the operator is looked up from information calculated in the `type-checking` module and that operator is replaced with the correct type instance of the operator.

Author: Shail Aditya Gupta, July 1990

Caveats: The `overloading-translation` module does not have much more functionality than the `type-checking` module. These two modules are separate because the `type-checking` module does not do any cut and paste on the parse tree, and `overloading-translation` only does cut and paste. In the future, this module may be incorporated into the type-checking module.

4.2.6 Lambda Lifting

Module:

```
(defcompiler-module lambda-lift id-compiler
  (:input parse-tree def)
  (:output parse-tree def)
  (:function lambda-lift-def))
```

Description: Nested `def`'s are lifted to the outermost scope, making every function definition a top-level function. Extra arguments are added to the previously internal function definition to pass along variables that the internal function references from its lexical scope.

Strategy: The algorithm for lambda lifting is taken from [2].

Author: Ken Traub, October 1986

Caveats: Lambda lifting is implemented to support nested function definitions. Scheme and Lisp use environments in order to support nested function definitions. It is not clear that lambda lifting is more efficient than environments. Perhaps we should do an environment-based implemention in order to clarify the various issues involved in the relative efficiency of these two implementations.

Chapter 5

Middle End Data-Structures and Tools

5.1 Introduction

This chapter and the following chapter discuss the middle end of the Id compiler. In theory, the middle end of the compiler encompasses a machine independent data structure for representing Id programs and a set of modules that transform one instance of the data structure into another semantically equivalent (but hopefully more efficient) instance. However, one should always be wary of sweeping claims. As in most optimizing compilers, our current implementations of semantics preserving optimizations have not always been rigorously checked (not of course, due to lack of desire, but these problems are still open areas of research). Making claims that certain machine independent code transformations result in more efficient execution (in time or space) on a particular machine is even harder to decide. Therefore, the reader is encouraged to adopt a critical attitude for the next two chapters and towards machine independent compiler optimizations in general until the field reaches firmer ground.

Disclaimers aside, the overall philosphy of machine independent optimizations seems appropriate. There are many well known program transformations that work best when applied before the details of specific machine implementations obscure the landscape of a program. For the Id compiler these include: optimizing accesses to statically known data (either through constant folding and propagation or fetch elimination from static structures), simplifying algebraic expressions, unrolling procedure calls and loops, and hoisting code out of loops among others.

Also, over time, features that have more to do with specific machine models than machine independent optimizations have crept into middle end modules. Hence the middle end is an amalgam of modules that perform general optimizations as well as add features that make programs easier to implement in the dataflow model or on a specific machine.

Current wisdom dictates that graphs are the most appropriate data structure for middle end transformations. The Id compiler has the fortune to use very similar graph structures for both the middle *and* the ETS back end (described in [5]), both program graphs (in the middle end) and ets machine graphs (in the back end) are implemented on top of the same low level structure provided by DFCS. Unfortunately, sometimes the underlying details of the low-level data structure obscure the higher level abstraction used by the middle end (largely for reasons of efficiency). In particular, it's worthwhile to pay extra attention to the details of *surfaces* which can confuse the novice.

5.2 Program Graphs

The Id compiler's middle end modules manipulate program graphs, a graph data structure for representing Id programs. Program graphs are a generalization of the dataflow graph data structure provided by DFCS. Conceptually, progam graphs consist of instructions wired together into a graph:

Definition 1 *A* **Program graph** *contains*

1. *A set of* **instructions***, each with the following components:*
   (a) *An* **opcode***, represented by a keyword symbol.*
   (b) *An* **external surface** *with a set of* **input and output ports***. The name of the external surface is the keyword symbol* `:exterior`*.*
   (c) *Zero or more* **internal surfaces***, each with their own sets of input and output ports. The name of each internal surface is represented by a keyword symbol.*
   (d) *Various other fields for characterizing an instruction's properties and keeping bookkeeping information.*
2. *A set of* **directed arcs** *connecting instruction output ports to instruction input ports:*
   - *No input ports may have more than one incoming arc.*
   - *Output ports have no limitation on the number of outgoing arcs.*
3. *An (implicit) notion of the* **graph's inputs and outputs***:*
   - *A graph's inputs are the set of its unconnected instruction inputs.*
   - *A graph's outputs are the set of its unconnected instruction outputs.*

In the dataflow model, where arcs carry tokens from outputs to inputs, every dataflow instruction produces a complete set of output tokens when presented with a complete set of input tokens. Program graph instructions have dataflow instruction behavior. Dataflow behavior can be generalized to graphs of instructions as well.

Definition 2 *A* **Basic Block** *(or just a block) is a program graph where if a token arrives at every input:*

1. *Every output emits a token.*
2. *Every instruction in the graph executes exactly once.*

Its useful to classify program graph instructions based on the number of internal surfaces they have:

Definition 3 **Simple instructions** *have no internal surfaces.*

Simple instructions represent primitive Id operators (like `+` or `i-store`). A simple instruction's input ports represent arguments to the operator and its outputs represent the results returned by the operator. Some simple instructions can be classified by their use of input and output ports:

- **Purely functional instructions** (like `+`) depend only on their inputs and have no observable effects outside the results they return.

- **Purely side-effecting instructions** (like `i-store`) have no outputs.

- **Constant instructions** have no inputs.

Definition 4 Encapsulating instructions *(called encapsulators) have at least one internal surface.*

Each internal surface of an encapsulator connects to different, disjoint basic blocks. Encapsulators are used to represent Id operators with complex translations from parse trees to blocks of machine graph instructions (called translation schemas). The opcode of the encapsulating instruction represents the invariant part of the schema and the blocks connected to each internal surface of the instruction correspond to the variant subgraphs of the schema.

Alternatively, encapsulators can be thought of as simple instructions whose semantics are completely determined by their opcode and internal blocks. They represent black boxes, hiding internal surfaces from the outside. Like simple instructions, they produce a full set of output tokens on their external outputs when presented with a full set of tokens on their external inputs.

For example, the Id `if` expression is represented by an `if` encapsulator with the following components:

1. An external input taking the output of the predicate expression.

2. n external inputs for the arguments (free variables) of the `then` and `else` expressions.

3. m external outputs.

4. Two internal surfaces named `:then` and `:else` which connect to blocks of n inputs and m outputs.

Note, after being presented with a full set of input tokens, the `if` encapsulator produces a full set of output tokens, though only one of its internal blocks executes.

5.3 Port Names

The input and output ports on each surface of an instruction can be referred to either by number or symbolic name. If a surface has n input ports then each input port can be referred to as a number in the range $\{0, \dots, n-1\}$ (similarly for output ports). Alternatively, ports can be referred to by symbolic names (which map onto port numbers) using the following conventions:

- A **Symbolic Name** is either
  1. A keyword symbol (a simple name).
  2. `(keyword . i)` (a subscripted name), where i is a non-negative integer.

- An input or output port may have at most one symbolic name.

- No two input ports or output ports of the same instruction may have the same symbolic name although an input port and an output port may have the same name.

- If `(keyword . n)` is a port name then so is `(keyword . i)` for all i from zero through n.

- Subscripted names with consecutive subscripts always map to consecutive port numbers.

The following functions translate between port names and numbers:

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

These functions return nil if the name or number is ill-defined for the given *instruction*. The names returned are always subscripted.

5.4 Arcs, Sources and Sinks

Definition 5 **Sources** *and* **sinks** *allow the programmer to specify a specific output or input port of an instruction:*

- *A* **source** *points to an instruction's output port.*
- *A* **sink** *points to an instruction's input port.*

Definition 6 *An* **Arc** *is a source-sink pair.*

Arcs are implemented as pairs of sources and sinks. The source of the pair points to the output port attached to the arc's tail. The sink of the pair points to the input port attached to the arc's head.

Notice that instructions never point directly to other instructions. Instead, sources and sinks provide a doubly linked list corresponding to arcs. For example, if instruction A points to instruction B then A points to a sink pointing to an input of B and B points to a source pointing to an output of A.

Sources, sinks and arcs can also carry **annotations** which are either nil or a pointer to some data-structure.

Constructors

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

Creates and returns an instruction source or sink referring to an input or output *port* of *instruction*. *Port* may be either a symbolic name or a port number.

Selectors

`instruction-sink-instruction` *instruction-sink* [*Function*]
`instruction-source-instruction` *instruction-source* [*Function*]

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

`instruction-sink-input` *instruction-sink* [*Function*] `instruction-source-output` *instruction-source* [*Function*]

Returns the port number referred to by *instruction-sink* or *instruction-source*. Note that the value returned is always a number. These functions are discussed further in section 5.6. May *not* be used with `setf`.

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

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

5.5 Parameters, Slots and Properties

Instruction Parameters

Instruction parameters are used when one opcode stands for a 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 `:constant` instruction's parameter determines what constant it emits.

Instruction Slots

It's possible to define extra slots for program graph instructions. These extra slots can store additional information about a given instruction. Also, slots can be added and removed without the entire compiler being recompiled. Section 5.13 describes slots used in the compiler's current implementation.

Instruction Properties

Read-only slots called `properties` can be set up that store information for all instructions with a particular opcode. See section 5.7 and the explanation of the `define-program-graph-instruction` macro for the details of defining instruction properties. Section 5.12 describes properties used in the compiler's current implementation.

5.6 Using Program Graph Instructions

Now that the basic components of program graph instructions have been defined, we can describe the Lisp functions operate on instructions.

Selectors

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

Returns the contents of the `opcode` slot of *instruction*.

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

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

`instruction-encapsulator-p` *instruction* [*Function*]

Returns true if the instruction contains internal surfaces.

`instruction-block-names` *instruction* [*Function*]

Returns a list of the names of *instruction*'s surfaces.

`instruction-`<*property-name*> *instruction* [*Function*]

Returns the value of the property for *instruction*. May not be used with `setf`.

`define-program-graph-instruction-slot` *selector-name* `&key copyable-p` [*Macro*]

Sets up a new instruction slot with name *selector-name*.

`instruction-`<*slot-name*> *instruction* [*Function*]

Returns the value of the slot for *instruction*. May be used with `setf`.

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

Returns the number of exterior inputs or outputs for *instruction*.

5.6.1 Surface Related Functions

For historical and efficiency reasons, an encapsulator's internal surfaces are manipulated with the same functions used for manipulating a simple instruction's external surface. In actuality, input and output ports are assigned to different surfaces only by convention. For simple instructions, an instruction's input and output ports are the same as the ports for its exterior surface. For encapsulators, functions are provided to map between the absolute ports of the instruction (considering all ports to be on the `:exterior` surface) and the ports for each surface.

Fact 1 *A* **simple instructions** *input and output ports are the same as the ports on its exterior surface.*

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

Returns an array containing the inputs of *instruction*. Each element of the array is an instruction source and each element's index is its port number. May *not* be used with `setf`.

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

Returns an array containing the outputs of *instruction*. Each element of the array is a *list* of instruction sinks and each elements index is its port number. May *not* be used with `setf`.

Fact 2 *An* **encapsulating instruction** *has an indirect mapping between the ports of its surfaces and its absolute ports referenced by* `instruction-inputs` *and* `instruction-outputs`.

There is an indirect mapping between: (<surface-name>, <surface-port-num>) and <absolute-port-num>

Mapping from Surface to Absolute Ports

The following functions implement the mapping from (<*surface-name*>, <*surface-port-num*>) to <*absolute-port-num*>.

`instruction-surface-lowest-input` *instruction surface* [*Function*]
`instruction-surface-lowest-output` *instruction surface* [*Function*]

`instruction-surface-highest-input` *instruction surface* [*Function*]
`instruction-surface-highest-output` *instruction surface* [*Function*]

These functions return the number of the lowest (or highest) numbered port belonging to a given surface, or `nil` if there is no such surface. Note that a number is returned in the case where a surface exists but has no ports.

Note: <*absolute-port-num*> = <*lowest-surface-absolute-port-num*> + <*surface-port-num*>.

Mapping from Absolute to Surface Ports

The following functions implement the mapping from <*absolute-port-num*> to (<*surface-name*>, <*surface-port-num*>).

`instruction-input-surface` *instruction port* [*Function*]
`instruction-output-surface` *instruction port* [*Function*]

These return the surface on which a given port of *instruction* lies, or `nil` if the instruction has no such port.

5.6.2 Using Internal Surfaces

The definition of interior surfaces, as implemented above, usually confuses novice users. Though program graph instructions make no distinction between internal and external ports, a user's mental picture of an instruction does! In the diagrams we draw to represent an encapsulating instruction (see figure **??**), external inputs connect to the "top" of the instruction and outputs extend from its "bottom." However, for internal blocks, outputs hang off the "top" of the internal block and inputs enter from the "bottom" of the internal block. It's vital to keep in mind that the *outputs* from an interior surface feed the inputs of an internal block (and vice versa for the *inputs* of an interior surface).

An Example

Here are some Lisp functions using instruction surfaces:

```
;;;-------------------------------------------------------------------------
;;; multiple value giving number of lowest & highest port for given surface

(defun instruction-surface-input-numbers (instruction surface)
  (values (instruction-surface-lowest-input instruction surface)
  (instruction-surface-highest-input instruction surface)))

(defun instruction-surface-output-numbers (instruction surface)
  (values (instruction-surface-lowest-output instruction surface)
  (instruction-surface-highest-output instruction surface)))

;;;-------------------------------------------------------------------------
;;; Return sequence that corresponds to the subsequence of the port array
;;; for a given surface

(defun instruction-surface-inputs (instruction surface)
  (multiple-value-bind (l h)
```

```
      (instruction-surface-input-numbers instruction surface)
    (and l h (coerce (safe-subseq (instruction-inputs instruction) l h) 'list))))

(defun instruction-surface-outputs (instruction surface)
  (multiple-value-bind (l h)
      (instruction-surface-output-numbers instruction surface)
    (and l h (coerce (safe-subseq (instruction-outputs instruction) l h) 'list))))
```

5.7 Defining Program Graph Instructions

The `define-program-graph-instruction` macro provides a convenient way of declaring program graph instructions. It sets up a blueprint that defines all instructions of a given opcode. It declares a parameterized constructor function to create instances of the instruction having the specified opcode. It defines the symbolic names of input and output ports, interior surface names, and what ports are associated with what surfaces. It also defines properties associated with any instructions created by the constructor.

The syntax is as follows:

`define-program-graph-instruction` *opcode* `(`*arg1 arg2* ...`)` `&clauses` [*Macro*]
{`(:inputs` {*input-name input-expr*}∗`)`}
{`(:outputs` {*output-name output-expr*}∗`)`}
[`(:constructor-name` *constructor-name*`)`]
[`(:blocks` {*block-name input-names output-names*}∗`)`]
[`(:properties` {*indicator value*}∗`)`]

`input-expr` :integer
`output-expr` :integer
<*input-names*> ::= (*keyword**)
<*output-names*> ::= (*keyword**)

- The `:inputs` and `:outputs` clauses are required; all others are optional.
- The numbers of inputs and outputs are determined by evaluating the expression for each port name, where these expressions may refer to the arguments.
- The `:blocks` clause may be used to declare that certain inputs and outputs belong to interior subgraphs; there is always a surface named `:exterior` to which all ports not mentioned in the `:blocks` clause belong. If the `:blocks` clause is omitted entirely, then the only surface is the `:exterior`.
- Ports within a surface are always assigned consecutive numbers. The lowest numbers are assigned to the `:exterior` surface, and the remaining numbers are assigned to surfaces in the order in which they appear in the `:blocks` clause. Within each surface, the ports are ordered as they appear in the `:inputs` or `:outputs` clause.
- The `:properties` clause can be used to associate properties to all instructions of a given opcode; Note that the properties are really assigned to opcodes, rather than individual instructions.

Evaluating the macro defines the lisp constructor:

`make-`<*opcode*>`-instruction` {*arg*}* [*Function*]

The constructor function for the instruction. The name may be overridden by the `constructor-name` clause. The argument values may be used to parameterize the numbers of inputs and outputs.

<*opcode*>`-instruction-arg`<*i*> *instruction* [*Function*]

Returns the ith argument to the constructor creating the instruction

Examples

Here's an example defining a simple instruction.

```
(define-program-graph-instruction make-tuple ()
  (:constructor-name make-make-tuple-instruction-internal)
  (:inputs :trigger 1 :anchor 1)
  (:outputs :output 1 :signal (if *resource-manager-signal-p* 1 0))
  (:properties :generate-ets-function 'g-ets-make-functional-tuple-instruction
       :side-effecting-p t
       :create-like-p t
       :k-allocator :make_k_tuples
       :k-deallocator :free_k_tuples))

;;;
;;; Constructor for make-tuple instruction. Notice make-...-internal is used
;;; to construct a tuple instruction with no parameter. The parameter of
;;; resulting instruction is set to be size. Parameters should be definable
;;; with define-program-graph-instruction, but apparently they must be
;;; set by hand!
;;;
(defun make-make-tuple-instruction (size)
  (let ((make-tuple-instruction (make-make-tuple-instruction-internal)))
    (setf (instruction-parameter make-tuple-instruction) size)
    make-tuple-instruction))

;;;
;;; Selector for the size of the tuple the instruction creates
;;;
(defun make-tuple-instruction-size (make-tuple-instruction)
  (instruction-parameter make-tuple-instruction))
```

Here's an example defining an encapsulating instruction:

```
 (define-program-graph-instruction loop (n-variables n-constants)
   (:inputs :predicate 1
            :loop-constant n-constants
            :predicate-output n-variables
            :loop-input n-variables
            :body-output n-variables)
```

```
(:outputs :predicate-input n-variables
          :body-input n-variables
          :loop-output n-variables)
(:blocks :predicate (:predicate-output :predicate) (:predicate-input)
         :body (:body-output) (:body-input))
(:properties :iterator-p t))
```

The example above defines the following functions:

```
(make-loop-instruction n-variables n-constants)
(loop-instruction-n-variables loop-instruction)
(loop-instruction-n-constants loop-instruction)
(instruction-iterator-p instruction)
```

5.8 @!-opcode program graph instructions

To be filled in.

5.9 Wiring Instructions Together

For now see the DFCS manual for a description of how to wire together dataflow graphs.

5.10 Traversing Program Graphs

Instruction Marks and Numbers

When traversing program graphs it's often useful to mark instructions as they are processed, to avoid following loops in the graph. Instructions may also be numbered with non-negative integers. Algorithms which traverse program graphs cannot mark some instructions and number others since marks and numbers are stored in the same location within an instruction structure.

The functions below control marking a program graph:

`reset-all-instruction-marks` [*Function*]
`mark-instruction` *instruction* [*Function*]
`instruction-marked-p` *instruction* [*Function*]
`unmark-instruction` *instruction* [*Function*]

The functions below control numbering a program graph:

`reset-all-instruction-numbers` [*Function*]
`number-instruction` *instruction number* [*Function*]
`instruction-number` *instruction* [*Function*]
`unnumber-instruction` *instruction* [*Function*]

Iterating Over Blocks

The `with-basic-block-array` macro provides an efficient way to point to every instruction in a basic block. The macro stores the simple and encapsulating instructions of a block in two arrays (in no particular order). Iterating over the arrays provides access to every instruction in the block.

`with-basic-block-array` (*bb-array encaps-array encapsulator bb-name*) *body* [*Macro*]

<*bb-array*> ::= *symbol*
<*encaps-array*> ::= *symbol*
<*encapsulator*> ::= <*lisp form evaluating to an encapsulator instruction*>
<*bb-name*> ::= <*keyword denoting a basic block in encapsulator*>

- *bb-array* is bound to an array whose elements are the instructions comprising the block named *bb-name* in the encapsulator.
- *encaps-array* is bound to an array containing all instructions in it bb-array which are themselves encapsulators.
- The original encapsulator is not included in either array.
- Any values returned by the body are returned as the result of the entire macro.
- `with-basic-block-array` changes instruction marks.

`With-basic-block-array` is a macro because it recycles the arrays after executing the body. Therefore, **no pointers to the arrays must exist when the body exits!** If necessary, the body can make copies of these arrays.

An Example

Here's a code fragment lifted from the `circulate-structures` module. `Circ-basic-block` traverses a basic block within an encapsulator from the bottom-up, applying `circ-loop-instruction` to every `loop` encapsulator in the block.

```
(defun circ-basic-block (encapsulator surface)
  (with-basic-block-array (instruction-array encapsulator-array encapsulator surface)
    instruction-array
    (map nil #'circ-encapsulator encapsulator-array)))

(defun circ-encapsulator (encapsulator)
  (loop for block in (instruction-block-names encapsulator) do
(circ-basic-block encapsulator block))
  (when (eq (instruction-opcode encapsulator) :loop)
    (circ-loop-instruction encapsulator)))
```

Iterating Without Blocks

The `with-instruction-array` stores all instructions in a program graph into an array, not separating encapsulators from simple instructions and not heeding block boundaries.

`with-instruction-array` (*instruction-array dfgraph*) *body* [*Macro*]

<*instruction-array*> ::= *symbol*
<*dfgraph*> ::= <*lisp form evaluating to a dataflow graph*>

Instruction-array is bound to an array whose elements are all the instructions in the *dfgraph*. *Instruction-array* contains a pointer to every instruction in the graph, including those hidden by encapsulators. The restrictions for array reuse in `with-basic-block` apply to `with-instruction-array`. Note, that when dealing with program graphs, `with-basic-block` is preferred.

5.11 Anchors, Signals and Triggers

Though program graph's are machine independent, they are also the preferred site for adding some machine dependent features. The fact that these transformations occur in the machine-independent "middle-end" is just a matter of semantics. In fact, it's more correct to view the middle-end as a mixture of machine independent optimizations and some machine dependent transformations. I describe these concepts here because the modules that implement them work on program graphs as defined above.

5.11.1 Signals and Triggers

The abstract model of dataflow graph execution, though useful for reasoning about graph transformations, has flaws that make it hard to realize directly in executable machine instructions. The requirement that every instruction in a basic block execute exactly once, when all inputs are presented, needs clarification. In a block, reachable instructions with inputs fire after inputs are presented to the basic block since tokens will travel to every reachable instruction. However, constant instructions with no inputs, are not reachable from the block's inputs and have no local method of determining when to fire. Similarly, it's often necessary to know when every instruction in a block has executed since the resources required by the graph's execution can be deallocated. But again, instructions with no outputs don't contribute to the outputs of the block making a simplistic check of the outputs insufficient.

In order to solve the above problems and make dataflow graph evaluation easier to implement, two new software abstractions are added to our model of program graphs: signals and triggers. In essence, adding signals and triggers to the program graph gives every instruction at least one input and one output, making the question of when an instruction fires simple to determine. As a first approximation, trigger inputs are added to instructions with no other inputs and signal outputs are added to instructions with no other outputs:

Definition 7 *A* **Trigger** *is an additional input to an instruction which must receives a token whenever a set of inputs arrive (the trigger may itself be the only input).*

Definition 8 *A* **Signal** *is an additional output of an instruction which emits a token whenever the instruction fires and a set of outputs appear (the signal may itself be the only output).*

With these definitions in hand, we can define triggers and signals for basic blocks:

Definition 9 *A* **Block Trigger** *is an additional input to a basic block that receives a "trigger token" whenever inputs arrive at the block. One token is received for every complete set of inputs that arrive.*

Definition 10 *A* **Block Signal** *is an additional output from a basic block that must emit a "signal token" after every instruction with a signal output fires.*

Note, a block signal indicates only that instructions with signal outputs have fired. **All** instructions in a block have fired *only* after all outputs from the block (including the block signal) emit tokens.

Definition 11 *A* **Signal Tree** *is a graph of instructions with n-inputs and one output that fires after receiving all it's inputs.*

A block signal is normally computed by connecting it to the output of a signal tree whose inputs are connected to all the signal outputs in the block. Though the addition of signals and triggers is closely related to a particular machine implementation, it's easier to add them at the program graph level

5.11.2 Anchors

Signals and triggers make the semantics of basic blocks explicit by providing paths from a block's inputs to all its instructions and from all its instructions to its outputs. However, signals and triggers also complicate the graph considerably and are usually added just before leaving the middle end. Before their addition, blocks can contain subgraphs with no paths to the block's inputs or outputs (for example, constant instructions connected to purely side-effecting instructions). During optimizations it's easier to work with fully connected graphs, so extra inputs are provided to "anchor" all instructions with either no inputs or outputs.

Definition 12 *An* **Anchor** *is an additional input to an instruction with either no input's or output's that serves to keep blocks connected.*

Definition 13 *A* **Block Anchor** *is an additional input to a basic block that connects to every instruction in the block with no inputs or outputs*

Semantically, anchors are superfluous. They are merely a convenience to the programmer, providing a simple way to keep blocks connected.

5.12 Program Graph Properties

What follows is a list of some currently used program graph properties:

constant-p :boolean True for constant instructions and loop constants.
propagatable-constant-p
strict-p :boolean True for instructions which are strict in all their arguments.
folding-function : $value^n -> value^m$ Function which "executes" an instruction, taking n inputs and returning m outputs.
side-effecting-p :*boolean* True for instructions that are non-functional. Includes those that allocate unfilled storage for aggregate structures as well as explicit store instructions.
apply-like-p :boolean True for any instruction that matches the type signature of the `apply` instruction. **Apply-like** instructions are used to apply a function whose arity won't be satisfied by the given arguments.
encapsulator-propagation-alist
create-like-p :*boolean* True for instructions that return a pointer to an aggregate structure that can be fetched from or stored into
store-like-p :(list *Input-port Output-port fe-Match-p*) Instructions that store into part of an aggregate data structure. All store-like instructions are side-effecting.
fetch-like-p :(list *Input-port Output-port fe-Match-p sfe-Match*) Instructions that retrieve part of an aggregate DS created by a create-like instruction.
Where:
fe-Match-p = (*store-inst* x *fetch-inst* $->$ *boolean*)
is the type of a function which returns true if the fetch instruction retrieves the same item put in by the store instruction (ie storing and fetching on the same index).

sfe-Match = (*make-inst* x *fetch-inst* $->$ *input-Port*)
is the type of a function which returns the *make-inst*'s input port name that corresponds to the object the fetch instruction is fetching.
For example, if (`match` :*sfe-match*),(`cons` :*make-inst*) and (`head` :*fetch-inst*) then `(match cons head)` returns `'(:input . 0)`.
associative-p :boolean True for associative instructions.
generate-ets-function :*pg-inst* $->$ *ets-machine-graph* Generates an ets machine graph for a given program graph instruction.
sr-before-function :

5.13 Program Graph Slots

What follows is a list of currently used program graph slots:

instruction-unfold-count :copyable-p t
instruction-peel-count :copyable-p t
instruction-loop-type :copyable-p t

Chapter 6

Middle End Modules

6.1 Introduction

The program graph structure is extremely flexible and expressive. Constraining middle end modules to work on program graphs is hardly a restraint since the number of different types of instructions in a program graph is indefinitely extendable. In retrospect, the design of the Id compiler is complicated by an intermediate language that is *too* flexible (though extensions to the core of the intermediate language are easier to express, if not implement). Without a more restrictive notion of an intermediate language, adding new program graph instructions can impinge on every module in the middle end. For ease of development, it's probably best, to define several intermediate languages, each a small perturbation of the other, which have modules that work exclusively on them. Other, more general modules can be defined to work with every perturbation. In actuality, the Id compiler implements this scheme, but its boundaries are hard to define without close examination of the compiler's code.

6.2 Ordering the Modules

The middle end modules can be divided into two categories: *tranformation* and *optimization*. Optimization modules perform machine independent optimizations within a given intermediate language. They can be used in any order, any number of times (including not at all). Transformation modules, convert one intermediate language to another. Usually they perform a small perturbation of the intermediate language by either excising opcodes, adding opcodes or enforcing some machine dependent constraint on the graph. They *must* be used exactly once in a sensible order.

- The following modules transform the program graph:

`generate-program-graph`	Creates a program graph from a parse tree.
`manager-synchronization`	Connects `release` instructions.
`call-substitution`	Stores inlinable functions in the Exsym table.
`synchronize-release`	Translates the `synch` instruction.
`thunk-splitting`	Translates the `thunk` instruction.
`inlineable-definitions`	Inlines functions.
`loop-constant-propagation`	Inserts loop-constant instructions.
`code-blocks`	Splits loop's into seperate code blocks.
`loop-analysis`	Determine's when a while loop can be used as a for loop.
`unroll-loops`	Unroll and unfold loops.
`signals-and-triggers`	Insert signals and triggers into the graph.

The above ordering of transformation modules gives a middle end with a minimum number of modules.

- The following modules optimize the program graph:

`fetch-elimination`	Remove fetch's when a corresponding store found.
`simple-fetch-elimination`	Remove fetch's from functional data-structures.
`dead-code-elimination`	Remove dead code.
`constant-propagation`	Propagate constants through encapsulators and bodies.
`cse-and-hoisting`	Common subexpression eilmination and hoisting out of loops.
`partial-evaluation`	Simplify some algebraic identities.
`circulate-structures`	Hoist allocate's and deallocate's out of loops.

Note, a minimal middle end won't necessarily compile quickly since some optimizations shrink the the program graph's size considerably.

6.3 The Modules

Each module description briefly describes some high-level characteristic of the module, including:

Module: The `defcompiler` module definition.

Options: A description of the module's options.

Description: A declarative description of the module's effect on the program graph.

Strategy: The module's method of traversing the graph.

Author: Primary person responsible for coding.

Caveats: Known limitations, warnings, points of interest etc.

Each module in the middle end of the compiler reads or modifies a number of global data structures in addition to its explicit program graph argument and result. The following entries, for each module, provide information about its usage of global data structures:

Exsym Properties: External symbol properties.

Graph Properties: Dataflow graph properties.

Instruction Properties: Program graph instruction properties.

Instruction Slots: Program graph instruction slots.

Not each module has all the entries filled in. If a module is the first to define an entry in a global structure or is the exclusive user of an entry then the entry's name is in **boldface** followed by a description. Otherwise, just the name is listed.

6.3.1 Generate Program Graph

Module:

```
(defcompiler-module generate-program-graph id-compiler
  (:input parse-tree procedure)
  (:output program-graph procedure)
  (:options check-bounds)
  (:function gpg-definition))
```

Description: The primary function of this module is to transform the parse-tree into a program graph. Pattern matching is also compiled out at this stage, although it should probably be done at an earlier module.
[1]

Strategy: The information obtained from the `scope-analysis` module is used to wire together instructions, given the parse-tree.

Author: Ken Traub, October 1986.

Caveats: Pattern matching should logically being in the desugaring module in the front end. At this point, it would probably be too difficult to change move this code there.

6.3.2 Fetch Elimination

Instruction Properties: store-like-p, fetch-like-p

Module:

```
(defcompiler-module fetch-elimination id-compiler
  (:input program-graph code-block)
  (:output program-graph code-block)
  (:options fetch-elimination)
  (:function fe-code-block))
```

Description: Fetch elimination traverses the program graph searching for instructions storing values into data aggregates. When a "store-like" instruction is found it looks back from it's inputs for the create instruction allocating the aggregate. If found it follows the aggregate's other uses looking for other fetch instructions retrieving information placed in the aggregate by the store. These fetches can be eliminated by rewiring from the store instruction's input.

[1]Nikhil and Arvind figured out the algorithm for doing pattern matching regardless of the order of the patterns. The algorithm for this is described in the file: `/jj/nikhil/haskell/pattern-matching.text` Unfortunately, this file has not been published – perhaps it should be turned into a CSG memo. Other pattern matching algorithms can be found in Simon Peyton Jones's book [7].

Strategy: Uses `with-instruction-array` to traverse every instruction in the graph, hence it doesn't do fetch elimination across encapsulator boundaries[2]. Fetch elimination is defined for i-structures, tuples, sums and cons.

Caveats: Fetch elimination doesn't look across encapsulator boundaries.

6.3.3 Simple Fetch Elimination

Instruction Properties: fetch-like-p

Module:

```
(defcompiler-module simple-fetch-elimination id-compiler
  (:input program-graph code-block)
  (:output program-graph code-block)
  (:options fetch-elimination)
  (:function sfe-code-block))
```

Description: Traverse the program graph searching for instructions fetching from a data aggregate. When the fetch is from an aggregate allocated by a functional create instruction (determined by the result of sfe-match) then the fetch instruction can eliminated by wiring directly from the create instruction's input.

The following pairs of functional <*constructor, selector*> instructions can be "short-circuited" (ie. an input of the constructor instruction can be wired around the selector function. The, no longer connected, selector function can be removed.

```
<make-functional-tuple, tuple-fetch>
<make-functional-sum, sum-fetch>
<make-functional-sum, disjunct-number>
```

Strategy: Uses `with-instruction-array` to traverse every instruction in the graph, hence it doesn't look across encapsulator boundaries.

Author: Jonathan Young, September 1988.

Caveats: Simple fetch elimination doesn't look across encapsulator boundaries.

6.3.4 Dead Code Elimination

Instruction Properties: side-effecting-p, create-like-p

[2]To `with-instruction-array` an encapsulator's subgraphs are disjoint from the graph containing the encapsulator. Logical connections between the two are determined by the semantics of the encapsulator and are captured, for simple cases by an encapsulator's `encapsulator-propagating-alist` property

Module:

```
(defcompiler-module dead-code-elimination id-compiler
  (:input program-graph code-block)
  (:output program-graph code-block)
  (:options dead-code-elimination)
  (:function dce-code-block))
```

Description: There are three types of dead code elimination:

1. Remove non-encapsulating, non-side-effecting instructions with unused outputs.
2. Remove arcs from encapsulators.
   (a) If there is an `if` output not used, then disconnect the corresponding arcs feeding the interior `then-output` and `else-output`.
   (b) If there is an `if` input or `loop` constant not used within any interior blocks, disconnect the corresponding arc feeding the exterior.
3. Find structure creating instructions that only feed structure storing instructions. Remove both the creating and storing instructions.

Strategy: Traverse the program graph doing 2a on the way down for each level. On the way up do 1 for all non-SE instruction in the basic-block (searching out the largest subgraphs with unused outputs) and 3 for all create-like instructions. Finish each level by doing 2b.

Author: Ken Traub, June, 1987

Caveats: Step 2 should be generalized to other encapsulators. Neither entire encapsulators, useless circulating variables from loops, or dead but cyclic portions of the graph are eliminated. Not all the dead code is caught in one pass.

6.3.5 Constant and Loop Constant Propagation

The constant propagation module and the loop constant propagation module use the same internal functions and differ only in the way they handle propagation of loop constants.

Instruction Properties: folding-function, inhibit-constant-propagation.

Module:

```
(defcompiler-module constant-propagation id-compiler
  (:input program-graph code-block)
  (:output program-graph code-block)
  (:function cp-dataflow-graph)
  (:options constant-folding constant-propagation)
```

Options:

> **constant-folding** :*boolean* Whether to evaluate constant expressions at compile time.
> **constant-propagation** :*boolean* Whether to propagate constants through encapsulators at compile time.

Description: Constant propagation,

1. Evaluates expressions involving only constants.

2. Pushes constants through encapsulators.

3. Removes encapsulators that choose which internal block to execute based on a constant expression by unencapsulating the appropriate block.

Strategy: Traverse the program graph top-down. At each level identify constant instructions and either evaluate (fold) them (creating a new constant instruction) or, depending on where on an encapsulator they are connected, push them in or use them to unencapsulate a block. If any blocks are unencapsulated then iterate the process again at the same level. Otherwise, recurse on the next lower level of the graph.

Module:

```
(defcompiler-module loop-constant-propagation id-compiler
  (:input program-graph code-block)
  (:output program-graph code-block)
  (:function lcp-dataflow-graph)
  (:options constant-folding constant-propagation
```

Description: The loop constant propagation module, uses the same code as the constant propagation module except it performs an extra transformation on loop encapsulators. In addition to the constant propagation optimizations, it removes constant arcs from the predicate and body blocks of loop encapsulators and makes them the first variable arcs, splitting each arc with a loop-constant instruction. Only constants actually used within the body or predicate are retained.

Caveats: The loop-constant conversion is a machine specific optimization. It turns free-variables occurring inside loops into constants that must be evaluated before any of the loop's iterations begin. Conceptually, it belongs in the back end.

6.3.6 Code Block Partitioning

Graph Properties:

> **code-block-summary** : See below.
> **n-loops** :
> **constant-area-size** :

Program Graph Instructions:

> The **fastcall-apply** instruction applies a code-block to n args and get m results.
> The **fastcall-def** instruction defines a code block taking n args and returning m results.

Instruction Properties: side-effecting-p

Module:

```
(defcompiler-module code-blocks id-compiler
  (:input program-graph procedure)
  (:output program-graph procedure)
  (:function cb-procedure))
```

Description: The code block partitioning module takes a procedure's program graph and splits off each of the graph's loop encapsulators into their own code block. These code blocks are contained inside `fastcall-def` encapsulators and linked into from the existing code blocks with `fastcall-apply`. For efficiency, both the loop encapsulator and any of its constant inputs are lifted into the `fastcall-def`. As configured, the module limits every code block (including the top-level `def`) to at most one loop.

Naming Conventions: If the top level procedure is named `F` and has n code-blocks lifted out then their names are `F-0`, `F-1`, ..., `F-(n-1)`. Similarly, if F-i has n code blocks lifted out then their names are `F-i-0`, `F-i-1`, ..., `F-i-(n-1)`, etc.

1. Each code block has the same `:procedure-name` property as the outer code block.

2. The outer code block has a `:code-block-summary` property describing the names of code blocks lifted out and their hierarchy: `(F (F-0) (F-1 (F-1-0)))` says `F` has three code blocks lifted out, two from the body of F and one from the body of `F-1`.

Strategy: Code block partitioning proceeds top-down, splitting off all loop instructions at a given level before proceeding into the next level of encapsulators. It differentiates between loops at the top-most level and ones nested inside other loops. The first top-level loop is kept in the original code block and all others are split out. When a loop instruction is found and split out, all the constants attached to the loop's inputs are taken along with the loop.

This module should be called immediately prior to signal and trigger generation.

Author: Ken Traub, May, 1986.

Caveats: The module's name should say something about loops! Also, it uses a hack to find the constant instructions connected to a loop's inputs which doesn't always give the best results. The coding style might benefit from some graph cut and paste abstractions.

6.3.7 Inlinable Definitions

Exsym Properties:

:inlinable-definition :(*string* $\rightarrow$ *dfgraph*) maps procedure name to copy of dataflow-graph

Graph Properties: :inlinablep, :procedure-name

Module:

```
(defcompiler-module inlinable-definitions id-compiler
  (:input program-graph procedure)
  (:output program-graph procedure)
  (:function ild-procedure))
```

Description: For every code-block with the **:inlinablep** property true, make an entry in the current exsym table associating the procedure name of the code-block with a copy of the code-block.

Caveats: Inlinable definitions must come before call substitution.

6.3.8 Call Substitution

Exsym Properties: :arity, :inlinable-definition, :recursive-set.

Instructions:

> **:direct-apply** :*instruction* Known arity application (is not apply-like and is side-effecting).
> **:apply-unsatisfied** :*instruction* Unfulfilled arity application (is apply-like and not side-effecting).

Instruction Properties: side-effecting-p, apply-like-p.

> **apply-like-p** :*boolean* True for apply or apply-unsatisfied instructions
> **propagating-input-p** :(*encapsulator x input* $\rightarrow$ *boolean*) True if input of encapsulator propagates to each block.

Instruction Slots:

> **instruction-cs-forwarding** :? forwarding pointer from instruction which has all its connections moved to another instruction.

Module:

```
(defcompiler-module call-substitution id-compiler
  (:input program-graph code-block)
  (:output program-graph code-block)
  (:options call-substitution)
  (:function cs-code-block))
```

Description:

1. Replaces a chain of applications of correct arity with :direct-apply (works across encapsulator boundaries) taking a :code-block literal (as the function) and n arguments.

2. Replaces applications of incorrect arity with :apply-unsatisfied

3. Performs inlining for identifiers having the :inlinable-definition property

(a) If the identifier is a constant then plug in its definition

(b) If the identifier defines a function, then replace a :direct-apply by the code-block for the function (if it's not in the :recursive set of the top-level definition, to avoid infinite unwinding). Only full arity applications will be inlined.

Strategy: Traverse the program graph top-down. For each encapsulator, attach a source to the encapsulator's anchor output. For every block in the encapsulator, first traverse every simple instruction, searching for identifier literals. Inline inlinable constant identifiers. For every non-constant identifier, follow its application chain (perhaps into an encapsulator) and convert it to either a direct-apply on a code-block literal or a chain of apply-unsatisfieds. For every inlinable code-block literal, inline the function body (subject to the recursive constraint) and recursively do call substitution on its body. Finally, do call-substitution for every encapsulator in the current block.

Caveats: The code for cs-inline-function could use some abstractions. A function application can only be inlined if the function's arity is satisfied by its arguments. It seems like there may be a cleaner and more efficient way of handling this optimization.

6.3.9 Common Subexpression Elimination and Code Hoisting

Instruction Properties: side-effecting-p, associative-p.

Options:

cse :*boolean* Whether to eliminate common subexpressions
hoist :*boolean* Whether to hoist invariant subexpressions from loops

Module:

```
(defcompiler-module cse-and-hoisting id-compiler
  (:input program-graph code-block)
  (:output program-graph code-block)
  (:function csech-dataflow-graph)
  (:options cse hoist))
```

Description: This module combines two interrelated but conceptually independent optimizations, Common Subexpression Elimination (CSE) and Code Hoisting (CH)

CSE Common Subexpression Elimination (CSE) merges duplicated expressions that compute the same function on the same inputs. CSE is done only within a given code block and only for purely functional expressions composed entirely of simple instructions (see caveats). There is no attempt to detect and eliminate equivalent encapsulators (the rationale being that this case won't occur often).

Combine Equivalent Inputs to Encapsulators CSE may make some inputs of a loop or seq encapsulator redundant and they should be eliminated before CH or else code may be duplicated when lifted out through the encapsulator inputs.

CH Code hoisting (CH), lifts invariant computations out of `loop` and `seq` encapsulators:

seq Non-constant (ie having at most one input), pure functional instructions connected to the `seq` inputs and their successors are hoisted out of the encapsulator "through their inputs."

loop Loop expressions are reassociated and hoisted from both the predicate and body:

1. Find loop invariant expressions (called loop "constants" in the code), pure instructions (and their successors) connected to "constant inputs" or literal values.
2. Find loop variant expressions, instructions that have any input from the loop variable inputs or any non-functional instruction.
3. Chains of n associative instructions with the same opcode are re-associated to form a tree of log(n) depth with invariant and variant inputs partitioning the leaves from left to right.
4. Hoist loop invariant expressions outside the encapsulator.

Strategy: CSE and CH occurs bottom-up:

1. Recursively CSE-CH all the encapsulators (hoisting out invariants, which must be simple and functional).
2. CSE all the simple, functional instructions in the basic block.
3. Combine equivalent inputs for all encapsulating instructions in the block.

Author: Ian Lance Taylor, May '87

Caveats:

1. Application is assumed to always be side-effecting.
2. Encapsulators are assumed to always be side-effecting.
3. Reassociation is a general notion that should be applied everywhere, not just in loops.
4. Invariants aren't hoisted out of `if` encapsulators.

There should be a general abstraction to make lifting instructions out or pushing them into an encapsulator easy.

6.3.10 Loop Analysis

Instruction Slots:

loop-parameters :(*keyword init-source upper-source step-source*) Identifies the type of loop and sources for its bounds (see below).

Module:

```
(defcompiler-module loop-analysis id-compiler
  (:input program-graph code-block)
  (:output program-graph code-block)
  (:function la-dataflow-graph))
```

Description: For every loop instruction, `Loop-analysis` examines the predicate and body to determine if the loop is equivalent to a simple `for` construct. If so, the module builds a piece of graph which computes the number of iterations and wires it into the loop as a new loop constant. Also, set up the loop-parameter slot to hold the sources for the loop's initial value, upper-limit and increment.

Strategy: Proceed top-down, analyzing loop instructions.

6.3.11 Loop Unfolding and Unrolling

Instruction Slots:

> **peel-count** :integer - number of times to peel loop
> **unfold-count** :integer - number of times to unfold loop
> **sequential-loop-p** :boolean - set true if loop is sequentialized

Module:

```
(defcompiler-module unroll-loops id-compiler
  (:input program-graph code-block)
  (:output program-graph code-block)
  (:function ul-dataflow-graph))
```

Description: Performs loop unfolding and unrolling. An unrolled loop executes k copies of the loop's body (ie k copies of the program graph corresponding to the loop's body wired in series) before entering the loop itself. Inother words, it unroll's k iterations from the start of the loop. Loop unfolding executes k copies of the loop's body for every iteration of the loop. Of course, care must be taken that termination during an unfolded or unrolled body exits the loop correctly (which is accomplished by analyzing the loop's behavior or by inserting checks of the loop's predicate around every body).

Strategy: Proceed bottom up, replacing loop instructions with their unrolled or unfolded equivalents when the unroll-count or unfold-count is non-nil. `While`loops use the most general unfolding strategy, nesting additional conditionals around every new body graph. `For` loops are compiled without this additional overhead by changing the loop's predicate and adding a series of (k-1) nested conditionals.

Author: Jamey Hicks, September 1989

Caveats: The terms unpeeling and unrolling are used interchangeably.

6.3.12 Circulate Structures

Instruction Properties: side-effecting-p, create-like-p, k-allocator, k-deallocator

Module:

```
(defcompiler-module circulate-structures id-compiler
  (:input program-graph code-block)
  (:output program-graph code-block)
  (:options circulate-structures)
  (:function circ-code-block))
```

Description: Attempts to find structures that are allocated and deallocated within a given loop body. If a structure with these properties exists, its allocation and deallocation code is lifted out of the loop. [A k-bounded loop will have k copies of the structure allocated before starting?]

Strategy: Circulate structures proceeds bottom-up through the graph hoisting allocation and deallocation code out of loop encapsulators at each level on the way up.

Author: Jamey Hicks

6.3.13 Algebraic Identities (misnamed partial-evaluation)

Module:

```
(defcompiler-module partial-evaluation id-compiler
  (:input program-graph code-block)
  (:output program-graph code-block)
  \#+DEBUG (:before-function program initialize-partial-evaluation)
  (:function pe-dataflow-graph)
  \#+DEBUG (:after-function program print-partial-evaluation-statistics)
  (:options partial-evaluation))
```

Description: The algebraic identities module defines a set of rewriting rules for program graph fragments. A program graph fragment is some subgraph of the program graph all of whose instructions can be reached from a single instruction. A pattern is a parenthesized expression that can match a graph fragment:

<*pattern*> ::= (<*opcode*> { <*input*>* |<*every-input*> } <*output*>* [<*triggers*>])
<*simple-pattern*> ::= *variable* |*constant* |<*pattern*>
<*opcode*> ::= *opcode* [$parameter <*sexp*>]
<*input*> ::= `:input` <*simple-pattern*>
<*every-input*> ::= ($every <*sexp*>)
<*output*> ::= <*simple-pattern*>
<*triggers*> ::= `:triggered-by` <*pattern*>*

For example,
`(:intimes :input 1 :input x)` represents an integer multiply instruction with two inputs.
`(:make-functional-tuple ($every :component i) (:tuple-fetch $parameter i :tuple x))` represents a tuple creating instruction connected to a tuple fetch instruction.

The module compiles a set of rewrite rules which describe how to rewrite existing patterns into new patterns. Sets of rewrite rules are defined by the `defevaluator` macro:

`defevaluator` *name input-pattern output-pattern* {*constraint*}* [*Macro*]

<*name*> ::= *symbol*
<*input-pattern*> ::= ($or <*pattern*>*) |<*pattern*>
<*output-pattern*> ::= <*pattern*>
<*constraint*> ::= <*lisp form*>

The module searches for instructions matching the applicable rewrite rules. If any instruction matches against the input pattern of a `defevaluator` then that graph fragment is replaced by the output pattern if all the constraints evaluate to true.

For example, a `defevaluator` for additive integer identity might look like:

```
(defevaluator plus-zero
  ($or (:intplus :input 0 :input x)
       (:intplus :input x :input 0))
   x)
```

Notice that repeated variable names refer to the same instruction in the scope of one `defevaluator`. Currently, there are `defevaluator`s to do:

1. Algebraic simplification for integers and floats:

   $x + 0 = 0 + x \to x$
   $x * 0 = 0 * x \to 0$
   $x * 1 = 1 * x \to x$

2. Reduction in strength for exponentiation

   $x^2 \to x * x$
   $x^3 \to x * x * x$

3. Algebraic simplification for booleans:

   $x \wedge x \to x$
   $true \wedge x = x \wedge true \to x$
   $false \wedge x = x \wedge false \to false$
   $x \vee x \to x$
   $true \vee x = x \vee true \to true$

4. Fetch elimination for functional data-structures

5. Removal of gates controlled by constants

6. Swapping of `form-address` with `set-ip` instructions

   `(:set-ip :input (:form-addr :input x :input offset) :input ip)` →
   `(:form-addr :input (:set-ip :input x :input ip) :input offset)`

7. Composition of `form-address` instructions

   `(:form-addr :input (:form-addr :input x :input offset1) :input offset2)` →
   `(:form-addr :input x :input (:intplus :input offset1 :input offset2))`

Strategy: The algebraic identities module proceeds top down, trying first to rewrite graph fragments in the current basic block, before proceeding to basic blocks within encapsulators. New instructions created by the rewrite rules are added to the list of instructions to traverse within the block.

Author: Ken Traub

Caveats: The module searches the rewriting rules in order of their definition. This may not be the best strategy. Pathological rewrite rules might cause the compiler to enter an infinite loop.

6.3.14 Thunk Splitting

Graph Properties: Adds new code blocks to the current graph.

Module:

```
(defcompiler-module thunk-splitting id-compiler
  (:input program-graph procedure)
  (:output program-graph procedure)
  (:function ts-proc))
```

Description: Delay encapsulators take arguments to a delayed expression and an address to store the result of the expression when evaluated. Thunk-splitting implements the delay abstraction by removing delay encapsulators and replacing them with lower level instructions. In detail, thunk-splitting implements the following: it encapsulates the delayed expression in a defthunk encapsulator and places it in its own code-block with the name " i-THUNK- ." It replaces the delay encapsulator with a pair of make-thunk/store-thunk instructions which take the arguments to the delayed expression, the address to store the delayed value in (when it's lazily computed) and the name of the thunk'd expressions code-block.

Strategy: Proceed bottom-up, transforming delay encapsulators into make-thunk/store-thunk pairs and add one defthunk code-block, for the delayed expression, to the current set of code-blocks.

Author: Jamey Hicks and ?

Caveats: Thunk splitting is very implementation dependent and should be considered part of the back end.

6.3.15 Synchronize Release

Instruction Properties: encapsulator-propagation-alist

> **sr-before-function** :? Only defined for loop instructions. Function that given a loop instruction, seems to peel it once if it contains a loop-release and hasn't been unrolled already. Jamey says that this has something to do with releasing nextified variables that are garbage after at least one iteration.

Instruction Slots:

instruction-unrolled-p :*boolean* Set for an unrolled loop instruction by sr-before-function

Module:

```
(defcompiler-module synchronize-release id-compiler
  (:input program-graph code-block)
  (:output program-graph code-block)
  (:function sr-dataflow-graph))
```

Description: Synchronize-release implements the `synch` instruction in terms of the `seq` encapsulator and other simple instructions. Basic blocks containing `synch`s are wrapped in `seq` encapsulators. Conceptually, `synch` instructions and the instructions that depend on their outputs are moved outside the seq and `synchs` are turned into gates controlled by the `seq`'s completion signal.

Strategy: Synchronize-release proceeds bottom-up, (partially) wrapping every basic block containing a `synch` instruction with a `seq` encapsulator. It finds all the outputs of the block that don't depend on synch instructions and wires them as the body outputs of the `seq`. All `synch` instructions and their successors are moved outside the encapsulator and replaced by gates controlled by the `seq`'s termination signal.

Author: Jamey Hicks

6.3.16 Manager Synchronization

Description: `Manager-synchronization` implements finer grain control over mutable structures than is possible using only block signals. Using Id's *enter* construct, mutable data-structures can be atomically read and modified using a locking mechanism. `Manager-synchronization` makes the "unlocking" of the structure occur after all its uses within the *enter* block, but before the block itself terminates.

Strategy: Generate-program-graph creates `enter` and `release` instructions that delimit the scope of Id's *enter* construct. The arc corrsponding to the imperative data-structure named in the *enter* statement is wired to the input of the `enter` instruction by `generate-program-graph`. Also, `enter` instruction's outputs are wired to every use of the data-structure. In `manager-synchronization`, the `release` instruction (wired to the block's signal by gpg), receives a signal tree taking its inputs from all the uses of the data-structure.

Author: Paul Barth

Caveats: This could (and probably should) be done by desugaring.

6.3.17 Signals and Triggers

Module:

```
(defcompiler-module signals-and-triggers id-compiler
  (:input program-graph code-block)
  (:output program-graph code-block)
  (:function st-dataflow-graph))
```

Instruction Properties: strict-p,legal-unconnected-inputs,legal-unconnected-outputs

Description: Adds signals and triggers to the program graph, as described in the previous chapter.

Strategy: `Signals-and-triggers` proceeds top-down, adding block signals and triggers to every block in every encapsulator, then wiring them to the appropriate instruction signals and triggers.

Author: Ken Traub

Caveats: I'm not sure how much optimization goes on here, but there should be ample opportunities.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison Wesley, Reading, Massachusetts, USA, 1986.

[2] Lennart Augustsson. A compiler for lazy ml. In *Proc. 1984 ACM Conf. on Lisp and Functional Programming, Austin, Texas*, pages 218–227. ACM, August 1984.

[3] L. Damas and R. Milner. Principle Type Schemes for Functional Programs. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[4] Gupta, Shail Aditya. An Incremental Type Inference System for the Programming Language Id. Technical Report MIT/LCS/TR-488, Laboratory for Computer Science, 545 Technology Square, MIT, Cambridge, MA 02139, November 1990. First published as the author's Master's thesis.

[5] Jamey Hicks. Id compiler back end for ets and monsoon. Technical report, Laboratory For Computer Science, 1990.

[6] Rishiyur S. Nikhil. Id version 90.0: Reference manual. Technical report, MIT Laboratory for Computer Science, July 1990.

[7] Simon Peyton-Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[8] Guy L. Steele Jr. *Common Lisp, The Language.* Digital Press, 1990. The Common Lisp Definition.

[9] 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.

[10] Kenneth R. Traub, James Hicks, and Shail Aditya. A dataflow compiler substrate. Technical Report CSG Memo 261-1, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, January 1991. Revised by Jamey Hicks and Shail Aditya.