Control Flow Aspects of Semantics-Directed Compiling

RAVI SETHI
Bell Laboratories

This paper is a demonstration of a semantics-directed compiler generator. We focus on the part of a compiler between syntax analysis and code generation. A language is specified by adding semantic rules in a functional notation to the syntax of the language. Starting with a small sublanguage of while statements, statement constructs of the C programming language are added in stages. Using a small ad hoc code generator, a compiler is automatically constructed from the semantics. The semantic description is analogous to a syntax-directed construction of a flow diagram for a program. By analogy with grammars and parser generators, minimal knowledge of the underlying theory is required.

For the control flow aspects of languages, efficient compilers can quickly be generated.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*semantics*; D.3.4 [**Programming Languages**]: Processors—*compilers*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*denotational semantics*

General Terms: Languages

Additional Key Words and Phrases: Compiler generation, mathematical semantics, flow diagrams, continuations

1. INTRODUCTION

This paper focuses on the part of a compiler between syntax analysis and code generation. We demonstrate that efficient compilers can quickly be generated for the control flow aspects of typical languages. Starting in Section 3 with a language containing conditional and while statements, we gradually add: break and continue statements (Section 5), restricted gotos that do not jump into the bodies of while or conditional statements (Section 6), gotos that can go anywhere (Section 7), and a switch statement (Section 9). With the addition of two variants of the while statement in Section 8, the final language will contain all the statement constructs of the C programming language [27]. (No knowledge of C is assumed; C is just a convenient vehicle since it has statement types found in a number of other languages.) The actual specification from which a compiler has been generated is shown, along with some indication of how the generation takes place.

A condensed version of this paper was presented at the SIGPLAN 1982 Symposium on Compiler Construction, Boston, June 1982.
Author's address: Bell Laboratories, Murray Hill, NJ 07974

By analogy with parsing, a specification can be constructed with minimal knowledge of the underlying theory.

There is every indication that the approach used in this paper will extend to other aspects of compiling. Control flow is a convenient starting point since it is a common denominator for many languages.

1.1. Structure of a Compiler

Compilation starts with lexical and syntactic analysis, because source programs have to be read and understood, and ends with code generation, because object code has to be produced. It therefore comes as no surprise that available compiler development tools are aimed largely at constructing lexical analyzers [25, 30], syntax analyzers (too numerous to enumerate), and code generators (e.g., [13, 15, 20]).[1]

Speaking very generally, compilers have the following structure. The output of syntactic analysis is an internal representation of the syntax of the program in which issues such as precedence of operators have been sorted out. This internal representation will be called the *abstract syntax* of the program.[2] Various so-called "semantic routines" then walk up and down the abstract syntax, carrying out checks and collecting information that is used to guide subsequent code generation. Much of the meaning of a programming language is embedded in these semantic routines.

This paper explores an alternate approach to compiling. Instead of embedding the semantics of a language into routines we specify the semantics precisely and mechanically generate a compiler from the specification. The flexibility afforded by this approach is evident from the fact that a sequence of six sublanguages of increasing complexity will be compiled. Additions to a language are made by changing a specification and remaking the compiler, with no reprogramming being necessary. This approach is also efficient.

1.2. Formalizing Control Flow

The term "control flow" dates back at least to Goldstine and von Neumann who conjure up visions of flow: "[The computing instrument] will, in general, not scan the coded sequence of instructions linearly. It may jump occasionally forward or backward, omitting (for the time being, but probably not permanently) some parts of the sequence, and going repeatedly through others" [21]. Computing instruments perform basic operations; the flow of control through a program determines the sequence in which the basic operations of the program are performed.

The meaning of a sequence of operations can be given by focusing not on the operations themselves, but on their effect on the state of a machine. Mathematically, the meaning of control flow can be specified by composing the functions for the basic operations [42, 46, 44, 22]. If f_1 and f_2 represent the state transfor-

[1] The extensive literature on compiler writing can be accessed either through the surveys that have appeared from time to time [3, 8, 16, 28], or through books on the subjects [4, 9, 23, 31].

[2] UNIX is a trademark of Bell Laboratories.

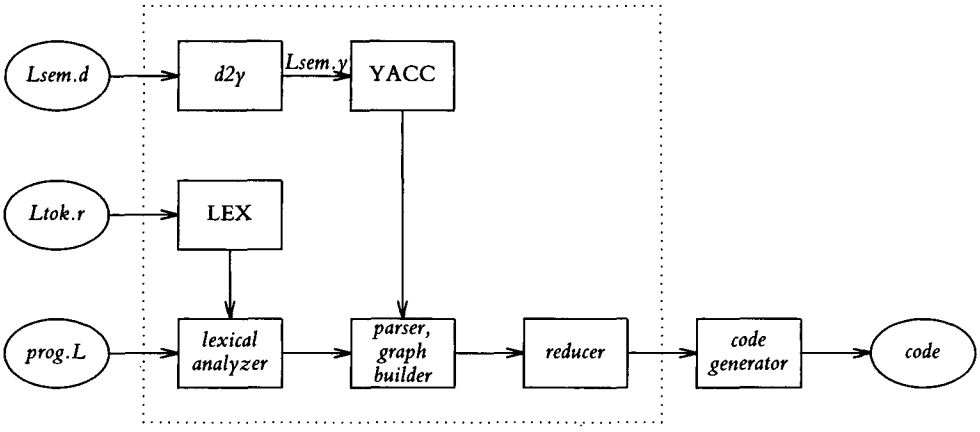


Fig. 1. Logical organization of a semantics-directed compiler generator. The dotted box shows the user interface. For a language L, the lexical and syntactic analyzers are constructed from the specifications *Ltok.r* and *Lsem.d*, respectively. *Ltok.r* contains regular expressions and their translations, while *Lsem.d* contains denotational semantic rules. The combination of *d2y* and Yacc constructs a syntax-directed translator: a progam *prog.L* is translated into a directed graph representing its semantics $[\![prog.L]\!]$. The *reducer* transforms the graph representation of $[\![prog.L]\!]$ into an equivalent more usable graph representation. The *code generator* linearizes and prints the graph representation. A similar figure appears in [43]. Christiansen and Jones [14] use different tools, but their logical organization has much in common with the above.

mations performed by stm_1 and stm_2, then $f_2 \circ f_1$ represents the state transformation function for $stm_1; stm_2$.

Goto statements disrupt the connection between syntax and control flow, so care is needed in composing functions. An intuitive understanding of the semantics of statements can be obtained by thinking of the flow diagram associated with a program. In an early paper, McCarthy [32] showed that corresponding to each flow diagram is a set of recursively defined functions. The semantics of statements is presented in terms of similar functions; it is almost as if we were mentally visualizing the flow diagram for a statement construct and then writing down the corresponding function. McCarthy's construction is reviewed in Section 1.5.

1.3. Semantics-Directed Compiling

A denotational semantics is a syntax-directed definition of the meaning of a program. For syntax, we will use the notation of the parser generator Yacc [24]. Associated with each syntactic rule is a semantic rule that specifies a function. The notation for building functions in Sections 1.4–1.5 has been given the name *Plumb* in order to distinguish it from other notations like DSL [36] and Meta-IV [11]. The name Plumb is inspired by "plumbing of functions" which itself comes from connecting functions with "pipes" (see Section 1.4).

Figure 1 shows the logical organization of a simple semantics-directed compiler generator. For a language L, the lexical analyzer is constructed from the specification *Ltok.r*, which describes the lexical structure of L in terms of regular expressions. *Lsem.d* consists of semantic rules in Plumb embedded into a syntactic specification of L in a form acceptable to Yacc.

Just as we deal with representations of numbers rather than the numbers themselves (3 + 5 is a representation of eight, for instance), we will deal not with functions, but with representations of them. For example, the expression $f_2 \circ f_1$ is a concrete representation of the state transformation function for stm_1; stm_2. We use the term *concrete semantics* to refer to an expression representing the semantics of a construct.

A program will be mapped into its concrete semantics in a syntax-directed manner by the combination of *d2y* and Yacc. The program *d2y* is a preprocessor for Yacc: it examines the semantic rules in *Lsem.d* and converts them into a C [27] program fragment that will cause a graph to be built for a program in *L*. The mapping performed by *d2y* is such that its output *Lsem.y* becomes the input of Yacc, so the parser cum graph builder in Figure 1 is constructed automatically from *Lsem.d*.

For the languages in this paper, the output of the *reducer* is a directed graph that is very close to a flow diagram for a given program. The code generator is a simple, quite small, ad hoc program for linearizing the graph and printing it.

If desired, a code optimization phase can be inserted before the code generator.

1.4 Pipes for Combining Functions

Since metalanguages for specifying denotational semantics deal with functions, mechanisms are needed for composing functions. One such mechanism, inspired by [37, 48] and pipes in the UNIX[3] operating system is reviewed here: further details may be found in [40]. The mechanism is suited to expressing the control flow aspects of sequential languages.

In its simplest form, the pipe mechanism is a form of function composition (the symbol | is called a *pipe*):

$$f \mid g = g \circ f.$$

For example, given suitable functions ***fetch*** and ***store***, the meaning of the assignment $a := b$ can be written as

$$\textbf{\textit{fetch}}(b) \mid \textbf{\textit{store}}(a).$$

The functions ***fetch*** and ***store*** will be used in Section 2 to give the semantics of expressions with embedded assignments. Given identifier a, ***fetch***(a) will map a state s to a value-state pair (v, s) where v is the value of a in state s. Since assignments may occur within expressions, ***store***(a) also returns a value-state pair: it maps (v, s) to (v, s'), where s' associates value v with a.

Proliferation of parentheses will be avoided by dropping the parentheses around b in ***fetch***(b) and writing ***fetch*** b.[4] In the following equalities which define

[3] Some static properties that are evident from looking at a program—for instance, whether a label decorates just one statement in a block—are not part of the syntax. (Terminology has changed little since Feldman and Gries [16] observed that often "syntax is taken to be precisely those aspects of language describable in the syntactic metalanguage under discussion.") Attribute grammars [29] have been used to formalize static properties of programs [39]. The abstract syntax is more precisely formulated as an initial algebra [1].

[4] The basic notation for representing the function application of expression E_1 to E_2 is to write E_1 followed by E_2, as in E_1E_2, or $E_1(E_2)$, since parentheses are used only for grouping. Expressions can be parsed by consistently associating function application to the left; both $f\ a\ b$ and $f(a)b$ are equivalent to $(f(a))(b)$.

operations ***fetch*** a and ***store*** a, s is some state, v is some value, and $s' = s[a := v]$ is the state that differs from s only at a, with $s'(a) = v$.

$$(\textbf{\textit{fetch}}\ a)(s) = (s(a), s)$$

$$(\textbf{\textit{store}}\ a)(v, s) = (v, s[a := v])$$

The semantic rule for the assignment $id := exp$ is based on the above example. The meaning of expression exp, which we write as $[\![exp]\!]$, is a function from states to value-state pairs;[5] $[\![id]\!]$ is the particular identifier represented by an instance of the syntactic variable id. The rule is:

$$[\![exp]\!] \mid \textbf{\textit{store}}[\![id]\!].$$

The next example motivates a more general definition for pipes. Consider the expression $b + c$. We would like to make sense out of a term beginning with

$$\textbf{\textit{fetch}}\ b \mid \textbf{\textit{fetch}}\ c.$$

The operational view is that each function connected by a pipe is handed a sequence of values. Each function takes zero or more arguments from the right end of the sequence, places zero or more results at the right end of the sequence, and passes the sequence to the next function. In the case where all the results of one function are arguments of the next, pipes merely compose functions.

From the definition of ***fetch***, a given state s is mapped by ***fetch*** b to the sequence $s(b)$, s. Since ***fetch*** c takes only the state as an argument, the definition of pipes[6] is that $s(b)$ is left untouched, that is, s is mapped by ***fetch*** $b \mid$ ***fetch*** c to $s(b)$, $s(c)$, s.

Note that the values to be added are $s(b)$ and $s(c)$, which are not at the right end of the sequence. We therefore need to allow + to skip over the rightmost element in the sequence, which is denoted by $|_1$ instead of $|$. The expression

$$\textbf{\textit{fetch}}\ b \mid \textbf{\textit{fetch}}\ c \mid_1 +$$

maps a state s to the sequence $s(b) + s(c)$, s. The semantic rule for $exp_1 + exp_2$ is similar:

$$[\![exp_1]\!] \mid [\![exp_2]\!] \mid_1 +$$

CONVENTION. In order for sequences of functions connected by pipes to be precisely defined, we assume that pipes associated to the left:

$$f \mid g \mid h = (f \mid g) \mid h,$$

with the obvious extension allowing both $|$ and $|_1$ in place of $|$.

[5] In standard denotational semantics [33] it is usual to have a separate semantic function for each kind of meaning associated with a syntatic object. Thus there might be two separate semantic functions $\mathscr{V}$ and $\mathscr{L}$ for expressions, such that $\mathscr{V}[\![\text{exp}]\!]$ is a function from states to values and $\mathscr{L}[\![\text{exp}]\!]$ is a function from initial states to final states. The brackets $[\![$ and $]\!]$ are then simply "an aid to the eye" [42]. For ease of implementation we lump all the meanings of a syntactic object together so there is only one semantic function. In this case, it does not make sense to introduce a separate name for each kind of semantic function; we simply enclose a syntactic object between the brackets $[\![$ and $]\!]$ to denote its meaning.

[6] The formal definition of pipes [40] keeps track of the number of arguments of functions. Having a fixed number of arguments for a constructed function facilitates type checking.

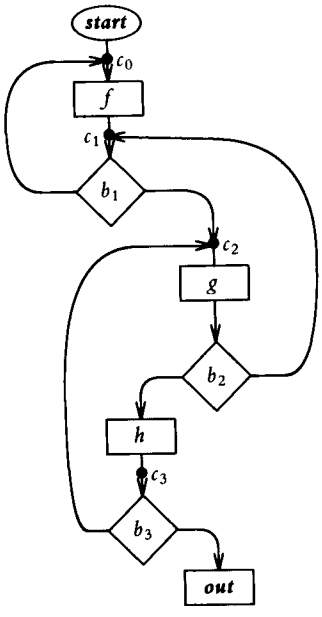


Fig. 2. The left and right exits of tests correspond to ***true*** and ***false***, respectively. Cutpoints $c_0, \ldots, c_3$ cut all loops in the flow diagram. Since loops have been cut, any path from a cutpoint leads either to another cutpoint, or to the ***out*** vertex.

1.5 Semantics of Iteration

The basic idea behind the semantics of iterative programs can be illustrated by reviewing McCarthy's construction of a set of recursive definitions for a flow diagram. Consider the flow diagram in Figure 2. The functions f, g, h perform some transformations on the state. The functions b_1, b_2, b_3 are meanings of expressions with side effects, that is, they map states to value-state pairs. Let ***out*** be a function that maps a state to whatever the result of the program is.

The construction "cuts" each loop by inserting a dummy vertex called a *cutpoint* along some of the edges. For example, there are four cutpoints, $c_0, \ldots, c_3$ in Figure 2. Associated with each cutpoint is a function: since no confusion can arise, we will use c_i to refer to both the cutpoint and the function associated with it. Starting at cutpoint c_0 in state s, cutpoint c_1 is reached in state $f(s)$. Therefore $c_0(s) = c_1(f(s))$. Using pipes,

$$c_0 = f \mid c_1.$$

Conditional branches will be handled using a function ***cond***, as in

$$c_1 = b_1 \mid \textbf{\textit{cond}}(c_0, c_2).$$

The arguments of ***cond*** are functions for two of the cutpoints: ***cond***(c_0, c_2) is itself a function that takes a value-state pair, and applies either c_0 or c_2 to the state, depending on the value.

The relationships between the functions for the cutpoints are

$$c_0 = f \mid c_1$$
$$c_1 = b_1 \mid \textbf{\textit{cond}}(c_0, c_2)$$
$$c_2 = g \mid b_2 \mid \textbf{\textit{cond}}(h \mid c_3, c_1)$$
$$c_3 = b_3 \mid \textbf{\textit{cond}}(c_2, \textbf{\textit{out}}).$$

The functions c_i at the cutpoints are called *continuations*. The semantics of statements will be given by associating a continuation with each program point.

2. PARTS OF A LANGUAGE DESCRIPTION

There are three parts to describing a language using Plumb and Yacc: declarations that help Yacc build a parser, declarations of basic semantic functions and domains, and the syntactic and semantic rules.

2.1. Declarations to Help Yacc

When Yacc builds a parser from a grammar, the terminal symbols of the grammar have to be distinguished from the nonterminals. A terminal is either a single character enclosed between a pair of apostrophes: for example, '(', or an identifier declared in a %token declaration:

```
%token    BREAK DO PLUS
```

Associativity and precedence of operators (in expressions) can be declared using %left and %right, as in

```
%right    ASS
%left     TIMES MOD
```

All identifiers on a %left or %right line have the same precedence and associate to the left or right, respectively. Successive %left and %right lines indicate increasing precedence; for instance, the assignment operator represented by ASS has lower precedence than TIMES and MOD in the above declarations.

We follow the convention of using upper case letters in terminal names and lower case letters in nonterminal names. Each terminal in a %left or %right declaration is implicitly declared to be a terminal, so %token declarations will be omitted for such terminals.

The terminals for the sublanguages L1–L6 are all declared by

```
%token    IF MAIN RETURN WHILE          ! MAIN: only procedure in L1
%token    BREAK CONTINUE                ! L2 has these keywords too
%token    GOTO                          ! L3: cant jump into ifs etc.
                                        ! L4: can jump into whiles
%token    DO ELSE FOR                   ! L5
%token    CASE DEFAULT SWITCH           ! L6

%token    ID NUM                        ! identifiers and integers
%right    ASS                           ! :=
%left     ORB                           ! | (bitwise or)
%left     ANDB                          ! & (bitwise and)
%left     EQ NE                         ! == !=
%left     LT GT LE GE                   ! < > <= >=
%left     PLUS MINUS                    ! + -
%left     TIMES DIV MOD                 ! * / %

%start    prog
```

2.2. Basic Operations and Domain Declarations

The metalanguage Plumb in which semantic rules will be written is a functional language in the tradition of ISWIM, with facilities for list manipulation, subsidiary expressions, function manipulation, and arithmetic. Except for the pipe construct (see Section 1.4), the constructs of Plumb have been borrowed from languages like DSL [36]. In this paper, very few of these constructs will be needed.

Domain declarations in Plumb are very much like domain declarations in standard denotational semantics [33]: they can be used to introduce variables, literals, and names for unspecified abstract domains all at the same time. Plumb allows literal variables to appear free in expressions, as long as their types (i.e., domains) have been declared. (Types are not presently checked.) A number of such literals will now be introduced. Note that none of the literals is built into Plumb.

Let us use the symbol V for a domain of values. (For technical reasons we will talk of domains rather than sets, but the informal view that a domain is a set will not mislead the reader.)

The meaning of a language construct is built up from a small collection of basic semantic operations. For example, corresponding to the operator symbol + is a function plus that maps pairs of values to a result value. The declaration of plus is

```
"plus":          [V,V] -> V;
```

As in [V, V] -> V, the operator -> is used to build functions from one domain to another, and lists of domains can be grouped between [and]. The grouping for A -> B -> C is A -> [B -> C].

The quotes in "plus" identify plus as being a literal (i.e., a particular) member of the domain following the colon in the above declaration. (The quotes are needed only in declarations.) In contrast, a declaration like

```
NUM:             V;
```

says that NUM is a variable representing an element of V. A given instance of NUM might represent the value 83, or 571 for that matter.

The declarations of arithmetic functions common to the sublanguages in this paper are shown below. Note that the domain V of values equals Int, the predeclared domain of integers.

```
:                V = Int;                    ! values are integers

"plus","minus","times","div","mod",          ! basic arithmetic functions
                                             ! 0 is false, nonzero true
"eq","ne","lt","gt","le","ge",               ! so relational operators
                                             ! return value 0 or 1, as do
"orb","andb":                                ! bitwise or, and
                 [V,V] -> V;

NUM:             V;
```

The simplest expressions are identifers from the domain Ide. States in S associate a value with each identifier. Expressions in C have side effects since

assignments may occur within expressions. Given a state s, an expression exp will yield a value and a modified state. Following Yacc, the symbol $ is used for "meaning of." Thus $exp is written instead of the non-ASCII ⟦exp⟧ that is more usual in denotational semantics.

```
ID:     Ide;                    ! identifiers and labels
s:      S = Ide -> V;           ! states
$exp:   S -> [V,S];             ! side effects may occur
```

The meaning of an identifier in an expression will be given using the basic function fetch. Since expressions return value-state pairs, as in Section 1.4, fetch will take a state, determine the value of the identifier in question, and return the value and the unchanged state. load plays a similar role for integer constants. The semantics of assignments will be specified using store and popv.

```
"fetch":    Ide -> S -> [V,S];      ! returns value+unchanged state
"load":     V  -> S -> [V,S];       ! load constant
                                    ! assignments occur in exp's so
"store":    Ide -> [V,S] -> [V,S];  ! store changes state but does
                                    ! not throw away value
"popv":     V -> [ ];               ! throws away value
```

Programs in this paper will consist of a single parameterless procedure called main. Statements in the C programming language must occur within the body of a procedure. A goto may jump anywhere within a procedure, but may not jump out of the procedure. A natural unit for specifying semantics is the sequence of statements that constitute a procedure body.

Like expressions, procedures return values and change the state. For bookkeeping reasons, we define an abstract domain A of procedure answers, and an explicit conversion function return that converts a value-state pair to an answer. The meaning of a procedure is now an element of S -> A, abbreviated by C, and called the domain of continuations. Control leaves a procedure in the C language either on execution of a return statement, or an "falling of the end" of the procedure. In the latter case, the basic function fall will be used.

```
                                    ! procedures return values +
                                    ! modified states, as do exp's
:           A;                      ! abstract domain of answers

"return":   [V,S] -> A;             ! eases checks and code gen.
c0,c1,c2, $prog:  C = S -> A;       ! continuations
"fall":     C;                      ! fall is the continuation for
                                    ! an empty program; from fall
                                    ! off the end

"cond":     [C,C] -> [V,S] -> A;    ! [C,C] are true and
                                    ! false continuations
```

As in Section 1.5, the literal cond is told the program points for the true and false exits by being given a pair of continuations; then a value and a state are

provided; if the value corresponds to true, then the answer yielded by branching to the true exit is returned, otherwise the answer from the false exit is returned.

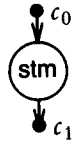
2.3 Syntactic and Semantic Rules

In the following rules for expressions, if the code enclosed between { and } is deleted, a syntactic specification of expressions will remain. Since the precedence of operators has already been specified in the declarations in Section 2.1, precedence considerations do not enter into the syntactic rules. The semantic rules are as discussed in Section 1.4. All operators are treated in the same manner.

```
exp     : ID
                { fetch $ID }
        | NUM
                { load $NUM }
        | ID ASS exp
                {  $exp | store $ID }
        | '(' exp ')'
                { $exp }
        | exp ORB exp
                { $exp.1 | $exp.2 |1 orb }
        | exp ANDB exp
                { $exp.1 | $exp.2 |1 andb }
        | exp EQ exp
                { $exp.1 | $exp.2 |1 eq }
        | exp NE exp
                { $exp.1 | $exp.2 |1 ne }
        | exp LT exp
                { $exp.1 | $exp.2 |1 lt }
        | exp GT exp
                { $exp.1 | $exp.2 |1 gt }
        | exp LE exp
                { $exp.1 | $exp.2 |1 le }
        | exp GE exp
                { $exp.1 | $exp.2 |1 ge }
        | exp PLUS exp
                { $exp.1 | $exp.2 |1 plus }
        | exp MINUS exp
                { $exp.1 | $exp.2 |1 minus }
        | exp TIMES exp
                { $exp.1 | $exp.2 |1 times }
        | exp DIV exp
                { $exp.1 | $exp.2 |1 div }
        | exp MOD exp
                { $exp.1 | $exp.2 |1 mod }
        ;
```

3. SEMANTICS OF THE CORE LANGUAGE

The semantic rules for statements can be visualized in terms of flow diagrams. The textual position of a statement in a program determines an entry and exit point for it. The part of a flow diagram corresponding to a single statement can be pictured as follows:

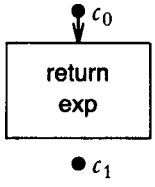


Here c_0 and c_1 are cutpoints corresponding to the single entry and exit of a statement stm. As in Section 1.5, c_0 and c_1 will also be used for the continuations associated with the cutpoints; they are related by an equation of the form

$$c_0 = \$stm(c_1). \tag{3.1}$$

that is, $stm, maps continuations to continuations, and statements are *continuation transformers.*

At first, the expression of c_0 in terms of c_1 as in eq. (3.1) may seem backward, but it allows the semantics of return statements to be specified. The flow diagram for a return statement by itself is



Since control never reaches c_1, the continuation c_0 does not depend on c_1 (i.e., $stm in eq. (3.1) is a function that is independent of its argument c_1. To sum up: textual position correctly identifies the entry point of a statement, but does not always identify the exit point, so it makes sense to determine the continuation for the entry point in terms of the properties of the rest of the flow diagram.

3.1 Rules

The rules in this subsection will be interspersed with explanatory text.

Recall from eq. (3.1) that the meaning $stm of a statement will be a continuation transformer. Following Yacc, $$ represents the meaning of the nonterminal on the left hand side of a given syntactic rule. For example, in a rule like

```
stm     : IF '(' exp ')' stm
                { $$ c1 = ... $stm ... }
```

$$ represents the meaning of the instance of stm on the left-hand side, while $stm represents the meaning of the instance of stm on the right-hand side of the syntactic rule. The semantic rules are related to eq. (3.1) as follows:

```
c0 = $$ c1
```

Return Statement. A procedure returns the value of an expression to its caller by means of the return statement:

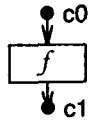
```
stm        : RETURN exp ';'
                  { $$ c1 = $exp | return }
```

The basic function *return*, (see Section 2.1), converts the value-state pair yielded by the expression into an answer. Note that the continuation c1 is ignored, thereby indicating that, on reaching a return statement, the expression is evaluated and the resulting value-state pair becomes the answer of the procedure.

Null Statement. For a null statement, $stm in eq. (3.1) will be the identity function. The syntax and semantics of null statements are given by

```
| ';'
        { $$ c1 = c1 }
```

Expression Statement. As in Section 1.5, the continuations in the following diagram are related by c0 = f | c1.



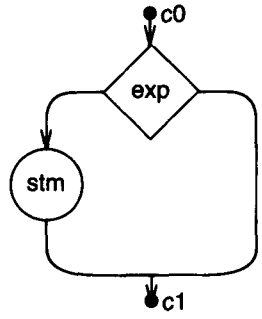
In an expression statement, the expression is evaluated for its side effect; the value of the expression is discarded. The function f in the above diagram corresponds to $exp | 1 popv where popv throws away the rule of the expression. Note that |1 is needed since popv must skip over the state to find the value to be thrown away.

```
| exp ';'
        { $$ c1 = $exp |1 popv | c1 }
```

Conditional Statement. The conditional statement has the syntax

```
| IF '(' exp ')' stm
```

The expression in a conditional statement is evaluated, and if it is nonzero, the substatement is executed.



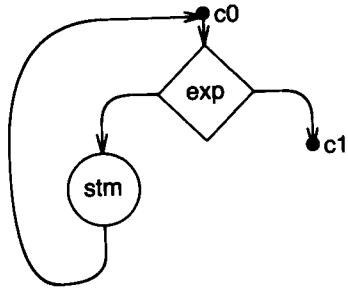
The semantic rule for conditionals can be read off the diagram.

```
{ $$ c1 = $exp | cond( $stm c1 , c1 ) }
```

While Statement. The while statement has the syntax

```
| WHILE '(' exp ')' stm
```

The substatement stm is executed repeatedly so long as the value of the expression remains nonzero. The test takes place before each execution of the statement.



The relationship between c0 and c1 is given by

```
c0 = $exp | cond( $stm c0 , c1 )
```

Equalities like the above in which the identifier on the left hand side is defined in terms of itself must be prefixed with either of the synonymous keywords cyclic and rec. The semantic rule for while statements is

```
{ $$ c1 =
        cyclic c0 = $exp | cond( $stm c0 , c1 )
}
```

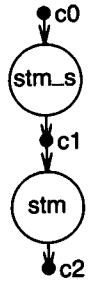
Compound Statement. A sequence of statements enclosed between { and } can be used wherever a statement is expected. The rules for a sequence of statements stm_s are given below.

```
| '{' stm_s '}'
        { $stm_s }
```

An empty sequence of statements is just like a null statement.

```
stm_s    : ! empty
                { $$ c1 = c1 }
```

The diagram for a sequence stm_s followed by stm is as follows.



Working backward from the continuation c2, the following rule is obtained.

```
| stm_s stm
        { $$ c2 = $stm_s ( $stm c2 ) }
```

3.2 Summary of Rules

For completeness, the specification of the core language L1 is shown below. Lines beginning with #include cause the contents of a file to be substituted for the line. The files ydecs, basics, and exp contain the parts of the specification discussed in Sections 2.1–2.3, respectively. The %% lines separate the three parts of a language description: declarations to help Yacc; domain declarations; and, the syntactic and semantic rules.

```
#include "ydecs"
%%
#include "basics"

$stm, $stm_s:       C -> C;                          ! continuation transformers

%%
#include "exp"
prog        : MAIN '(' ')' stm
                    { $stm fall }
            ;
stm         : RETURN exp ';'
                    { $$ c1 = $exp | return }
            | ';'
                    { $$ c1 = c1 }
            | exp ';'
                    { $$ c1 = $exp |1 popv | c1 }
            | IF '(' exp ')' stm
                    { $$ c1 = $exp | cond( $stm c1 , c1 ) }
            | WHILE '(' exp ')' stm
                    { $$ c1 =
                            cyclic c0 = $exp | cond( $stm c0 , c1 )
                    }
            | '{' stm_s '}'
                    { $stm_s }
            ;
stm_s       : ! empty
                    { $$ c1 = c1 }
            | stm_s stm
                    { $$ c2 = $stm_s ( $stm c2 ) }
            ;
```

4. TRANSLATION INTO PLUMB TERMS

The examples in this section explore the Plumb terms that arise for the semantics of expressions and statements, that is, for $exp and $stm. We show that there is a close connection between the Plumb terms for $stm and flow diagrams. This

close connection allows the code generator to be a simple routine for printing out the internal representation of these terms in a suitable linearized form. Details of the printing routine are given in Appendix A.

The examples are based on the following program for computing the greatest common divisor of two integers.

```
main()
{
        while( m >= 0 & n >= 0 )
        {
                r := m;
                while( r > n ) r := r - n;
                if( r == 0 ) return(n);
                m := n; n := r;
        }
}
```

4.1 Expressions

The translation of a simple expression like m >= 0 can be read from the semantic rules in Section 2.3 to be

```
fetch m | load 0 |1 ge
```

Sequences of pipes associate to the left (see Section 1.4), so the term is parenthesized as

```
( fetch m | load 0 ) |1 ge
```

When possible, the *reducer* (see Section 1.3) constructs such a *left linear* form, in which the right operand of each pipe operator is simple. There is an immediate analogy between the left linear form and a sequence of operations, so code generation is easier from such a form. For example, the expression

```
m >= 0 & n >= 0
```

translates into the left linear term

```
fetch m | load 0 |1 ge | fetch n | load 0 |1 ge |1 andb
```

For completeness, the remainder of this subsection describes how terms are converted into left linear form using the associative rule

$$f|(g|_1 h) = (f|g)|_1 h.$$

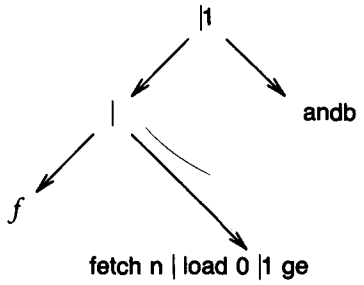
A more general associative rule exists,[7] but is not needed for the language here.

A direct translation of m > = 0 & n > = 0 yields the term

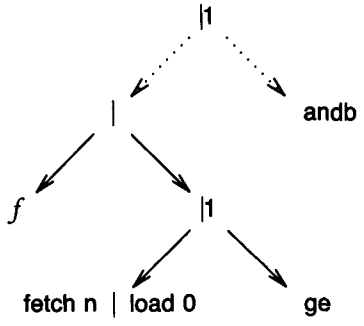
```
fetch m | load 0 |1 ge | ( fetch n | load 0 |1 ge ) |1 andb
```

[7] $f|_i(g|_j h) = f|_i g|_{i+j} h$, where i, j are integers and $|_0$ is just | [40]. Recall that pipes associate to the left, so the parentheses around $f|_i g$ have been dropped.

which can be drawn as



The subdiagram at which the associative rule is applicable shows up more clearly in



The first application of the associative rule linearizes | 1 ge; the next application linearizes | load 0 leading to the left linear form

f | fetch n | load 0 | 1 ge | 1 andb

Substituting for *f*, we get

fetch m | load 0 |1 ge | fetch n | load 0 |1 ge |1 andb

The above example generalizes to show that all instances of $exp will be put into left linear form. In rules containing $exp.1 | $exp.2, the parentheses around the left linear form for $exp.2 can be dropped, thereby yielding a left linear form for the entire expression.

4.2 Basic Blocks

Left linear terms for expressions lead immediately to left linear terms for expression statements. For example, if the computation following m := n; has continuation c, then the term for the assignment is

fetch n | store m |1 popv | c

By analogy with the use of *basic block* in [7] for straight line sequences of code, let a left linear sequence of pipes for a statement be called a basic block. The *entry* of a basic block is the root, which will be the last pipe operator in the block since the block is in left linear form.

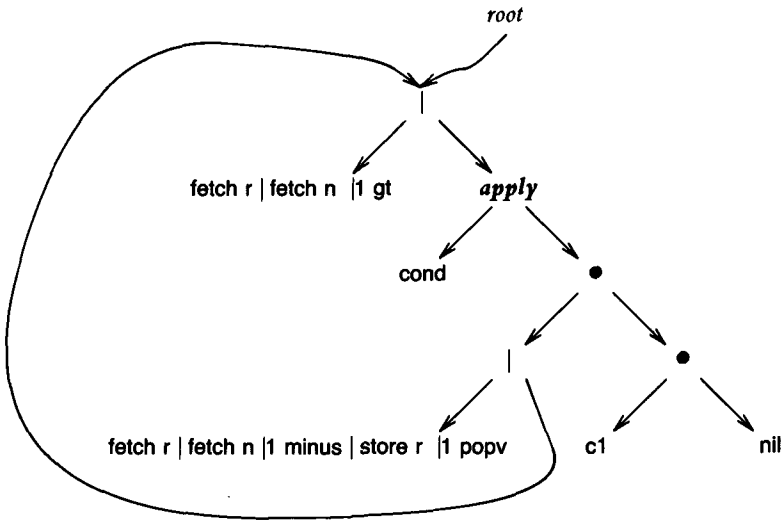


Fig. 3. Internal graph representation of the meaning of the statement

while (r > n) r := r − n.

Vertices marked with • represent cons vertices that construct a list; nil represents the empty list. ***apply*** is a Plumb operator that applies the function represented by its left argument to its right argument.

Sometimes basic blocks can be combined to form a single larger one. For example, the direct translation of m := n; n := r; is formed by substituting an appropriate continuation for c in the above translation of m := n;. If c1 is the continuation for the computation following the pair of assignments, the following is obtained.

fetch n | store m |1 popv | (fetch r | store n |1 popv | c1)

The *reducer* will automatically linearize this term into a basic block.

4.3 Edges Between Basic Blocks

Since Plumb is a functional language, any common subterms can be shared in a graph representation. For example, the semantic rule for conditional statements requires both the true and the false exits to use the same continuation: in operational terms, control flows to the same point from both the true and false parts; in the internal graph representation there will be edges to the same vertex. This vertex will be the entry point of some basic block.

In order to share subterms, the *reducer* applies the associative rule only to tree-like subterms, that is, the rule is not applied if application leads to copying of subterms because of multiple edges into subterms. These multiple edges become edges between basic blocks.

Consider for example the statement

while (r > n) r := r − n;

Assuming that c1 is the continuation for the exit of this while statement, a direct

translation yields

```
cyclic c0 =
        fetch r | fetch n |1 gt |
        cond( fetch r | fetch n |1 minus | store r |1 popv | c0 , c1 )
```

The internal graph representation of this term is suggested by Figure 3.

4.4 Conclusion

The semantics of expressions and statements is such that the internal graph representation of $stm corresponds closely to a flow diagram. Left linear sequences of pipes correspond to boxes in a flow diagram with edges between sequences becoming the edges of the diagram, as in Figure 3.

The process of code generation consists of taking graphs like the one in Figure 3 and printing them in a suitable linearized form. For the program in the beginning of this section, a routine in Appendix A prints the following code (integer labels correspond to the start of basic blocks, and are generated from node numbers in the internal representation). The lines between labels 81 and 102 are generated from the subgraph in Figure 3. Label 81 corresponds to the root of the figure, while label 102 corresponds to the continuation c1 in the figure.

```
140:    fetch m
        load 0
        ge
        fetch n
        load 0
        ge
        andb
        onfalse goto 23
        fetch m
        store r
        popv
81:     fetch r
        fetch n
        gt
        onfalse goto 102
        fetch r
        fetch n
        minus
        store r
        popv
        goto 81
102:    fetch r
        load 0
        eq
        onfalse goto 128
        fetch n
        return
128:    fetch n
        store m
        popv
        fetch r
        store n
        popv
        goto 140
23:     fall off the end
```

5. BREAK AND CONTINUE STATEMENTS

The meaning of break and continue statements can be suggested by the following program fragment: a break within the while statement is equivalent to a branch to hbrk; a continue to a branch to hcon.

```
            while (...)
            {
                    ...
            hcon:   ;
            }
hbrk:       ;
```

The semantics of break and continue statements will be given using implicit labels hbrk and hcon that are hidden from the programmer.

In order to determine where control flows to from a break statement, some sort of symbol table is needed to keep track of the continuation for hbrk. Symbol tables are formalized by functions called *environments* that map labels to continuations. Labels and identifiers will be lumped into the domain Ide, to avoid arbitrary restrictions on the form of labels and identifiers. Environments map identifiers to continuations:

```
!           new declarations to handle break and continue

"hbrk","hcon":      Ide;                    ! implicit labels set by
                                            ! break and continue
e:                  Env = Ide -> C;         ! label environments
"mte":              Env;                    ! (mt==empty) environment

$stm, $stm_s:       Env -> C -> C;          ! continuation transformers
```

Note that the meaning of a statement takes an environment as an additional parameter so that the meanings of embedded breaks and continues can be determined. Otherwise, few changes to the semantic rules in Section 3 are needed. The following rules carry over immediately:

```
prog        : MAIN '(' ')' stm
                    { $stm mte fail }
            ;
stm         : RETURN exp ';'
                    { $$ e c1 = $exp | return }
            | ';'
                    { $$ e c1 = c1 }
            | exp ';'
                    { $$ e c1 = $exp |1 popv | c1 }
            | IF '(' exp ')' stm
                    { $$ e c1 = $exp | cond( $stm e c1 , c1 ) }
```

When a break is encountered, a branch to the hidden label hbrk is formalized by using the continuation entered into the environment at hbrk: given environment e, this continuation is just e(hbrk).

```
| BREAK ';'
        { $$ e c1 = e(hbrk) }
| CONTINUE ';'
        { $$ e c1 = e(hcon) }
```

The rule for while statements must now set up environments appropriately: the environment e′, given by e[hbrk := c1][hcon := c0], agrees with e everywhere except at hbrk and hcon, which it maps to c1 and c0, respectively. This new environment e′ is used for the body of the while and does not affect any other statements.

```
| WHILE '(' exp ')' stm
        { $$ e c1 =
                cyclic c0 =
                        let     e' = e[hbrk:=c1][hcon:=c0];
                        in      $exp | cond( $stm e' c0 , c1 )
        }
```

The printing routine in Appendix A is used without change while constructing a compiler for L2 from the above specification.

The test program of Section 4 has been modified below to show that the meaning of continue statements is specified correctly. A continue statement has been added inside the inner while loop to illustrate that the environments handle nested while statements. If, as expected, the continue statement sends control to the beginning of the while loop, then the statement

```
junk := garbage;
```

should be skipped. The generated compiler for L2 does indeed map the following program to the code given at the end of Section 4, (modulo changes in label numbers).

```
main()
{
        while( m >= 0 & n >= 0 )
        {
                r := m;
                while( r > n )
                {
                        r := r - n;
                        continue;
                        junk := garbage;
                }
                if( r == 0 ) return(n);
                m := n; n := r;
        }
}
```

6. RESTRICTED GOTO STATEMENTS

There is little difference between the handling of break and goto statements—in each case an environment is consulted to determine where control should flow to. Explicit labels result in changes to the semantic rules from Section 5. Consider for example the semantic rule for while statements in that section. Since break and continue statements must be enclosed within a while statement, the environ-

ment needed to determine the meanings of embedded breaks and continues was set up as part of the semantics of the while. Explicit labels on the other hand may occur anywhere, so something special has to be done to set up the environment for goto statements. Clearly, it is necessary to determine the labels that actually occur. The continuations associated with these labels must also be determined. The treatment in this section is similar to that in [22].

6.1 Encapsulating Labels

It will simplify the semantic rules if we restrict goto statements from jumping into the bodies of conditional or while statements. Let us introduce the concept of a block which is a statement into which gotos cannot jump. The bodies of whiles and conditionals will then be blocks. Statements which might contain visible labels will be generated by a new nonterminal l_stm, leaving stm to generate statements without visible labels. More precisely,

```
$stm, $block:    Env -> C -> C;          ! labels inside stm and block
                                         ! are not visible outside
```

The semantic rules for stm are just as in Section 5, so let us include them here.

```
stm     : RETURN exp ';'
                { $$ e c1 = $exp | return }
        | ';'
                { $$ e c1 = c1 }
        | exp ';'
                { $$ e c1 = $exp |1 popv | c1 }
        | BREAK ';'
                { $$ e c1 = e(hbrk) }
        | CONTINUE ';'
                { $$ e c1 = e(hcon) }
        | GOTO ID ';'
                { $$ e c1 = e($ID) }
        | IF '(' exp ')' block
                { $$ e c1 = $exp | cond( $block e c1 , c1 ) }
        | WHILE '(' exp ')' block
                { $$ e c1 =
                        cyclic c0 =
                                let     e' = e[hbrk:=c1][hcon:=c0];
                                in      $exp | cond( $block e' c0 , c1 )
                }
        ;
```

Note that the rule for goto statements is similar to that for break and continue statements, and that the bodies of while and conditional statements are blocks.

6.2 Labeled Statements

It is at the level of a block that the environment containing continuations for labels is set up. Labels visible in the block must therefore be known along with their associated continuations (which formalize program points). The meaning of l_stm will be twofold: just like $stm, $l_stm will be a continuation transformer;

in addition, $l_stm will yield a list of visible labels and their associated continuations:

```
$l_stm, $l_stm_s:                                   ! possibly labeled statements

                    Env -> C -> [C,LC*];            ! C in [C,LC*] is as in $stm;
                                                    ! LC* contains visible labels
                                                    ! in l_stm + their continuations

:                   LC = [Ide,C];                   ! label–continuation pairs
list:               LC*;
```

The semantic rules for l_stm are as might be expected. The list of label-continuation pairs from an unlabeled statement is empty. When a label is encountered, a new label-continuation pair is added to the existing list using the Plumb operator cons. In a sequence of statements the lists of the elements of the sequence are concatenated using the Plumb operator cat.

```
l_stm       : stm
                    { $$ e c1 =
                              let       c0 = $stm e c1;
                                        list = ();
                              in        ( c0, list )
                    }
              | ID ':' l_stm
                    { $$ e c1 =
                              let       p = $l_stm e c1;
                                        c0 = p.1;
                                        list = ($ID,c0) cons p.2;
                              in        ( c0, list )
                    }
              | '{' l_stm_s '}'
                    { $l_stm_s }
              ;
l_stm_s     : ! empty
                    { $$ e c1 =
                              let       c0 = c1;
                                        list = ();
                              in        ( c0, list )
                    }
              | l_stm_s l_stm
                    { $$ e c2 =
                              let       q = $l_stm e c2;
                                        c1 = q.1;
                                        p = $l_stm_s e c1;
                                        c0 = p.1;
                                        list = p.2 cat q.2;
                              in        ( c0, list )
                    }
              ;
```

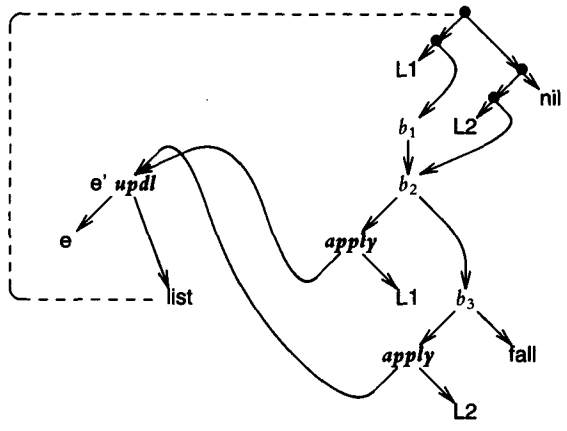


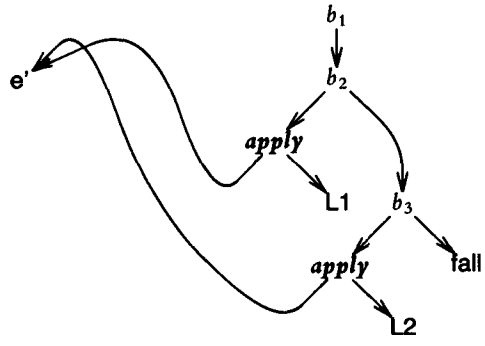
Fig. 4. The dashed line indicates that list is the same as the list of label-continuation pairs at the top of the figure. ***updl*** is a Plumb operator for updating a function with a list of pairs. Vertices marked with • represent cons vertices that construct a list; nil represents the empty list. Vertices labeled b_1, b_2, and b_3 abbreviate subgraphs with edges entering and leaving as shown.

6.3 Example

The following program fragment will be used to show how the environment is set up for goto statements.

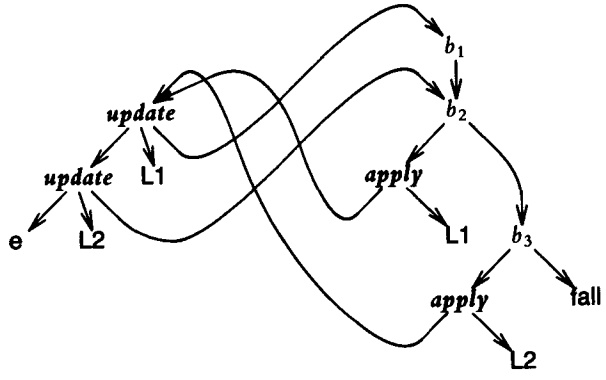
```
main()
{
L1:     u := v;
L2:     w := x;
        if (p) goto L1;
        y := z;
        if (q) goto L2;
}
```

Suppose that an environment e′ containing appropriate continuations for labels has already been determined. Then the continuation for the entire program is given by the node labeled b_1 in the following diagram. The symbols b_1, b_2, and b_3 are placeholders for the three basic blocks that the term for the program will have. Note that the meanings of the goto statements are determined by applying the environment e′ to the label gone to.

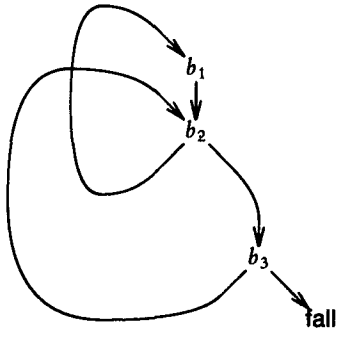


The suitable environment is determined as shown in Figure 4. At the top of the figure is a list of label-continuation pairs. This list will be called a *mini-environment* because it contains label bindings for the current block. At the bottom left hand corner is the environment e containing label bindings from outer blocks. Labels bindings from the current block have to be entered into e in order to determine e′.

The construction of the environment in Figure 4 takes a little getting used to. Perhaps it will help if we show how the environment is used. The ***updl*** operator is such that Figure 4 is equivalent to the following diagram in which the environment e is updated directly with continuations for the two labels: ***update***(e, L1, c) is the same as e[L1 := c].



It is a simple observation that the indirection through the environment to determine the continuations for L1 and L2 can be eliminated; the *reducer*, (see Section 1.3) ends up with the following diagram.



Once the compiler is constructed from the specification in this section, the output for the above program will be

```
44:     fetch v
        store u
        popv
164:    fetch x
        store w
        popv
        fetch p
        onfalse goto 256
        goto 44
```

```
256:    fetch z
        store y
        popv
        fetch q
        onfalse goto 23
        goto 164
23:     fall off the end
```

6.4 Setting up the Environment

The semantic rule for blocks can be abstracted out of Figure 4. The vertex labeled b_1 represents the continuation for the l_stm that constitutes the block. This continuation and the list of label-continuation pairs is determined when $l_stm is applied to the environment e[list] and the continuation c1, which happens to be fall in Figure 4. The dotted line indicates that the definition of list is circular, which is to say

```
c0,list = $l_stm (e[list]) c1
```

At present Plumb requires identifiers on the left hand side of a circular definition, so the above definition can be rewritten as

```
p = $l_stm (e[p.2]) c1
```

The following rule for blocks sets up the environment to be used for the block body and then yields the continuation for the statements in the block body.

```
block     : l_stm
                { $$ e c1 =                                  ! setup environment for
                                                             ! enclosed labels
                        let cyclic p = $l_stm (e[p.2]) c1;
                                c0 = p.1;
                        in      c0
                }
          ;
```

7. UNRESTRICTED GOTO STATEMENTS

In a freewheeling atmosphere in which goto statements can jump into the middle of while statements there are enough statements with visible labels that we will drop the distinction of the last section between unlabeled and labeled statements: once again, stm is the only nonterminal for statements. A goto will be allowed to jump anywhere within the main procedure, so the environment for goto statements will be set up at the level of a program instead of at the block level. $stm is declared by

```
:                   LC = [Ide,C];                  ! label–continuation pairs
$stm, $stm_s:       Env –> C –> [C,LC*];           ! like $l_stm in Section 6
```

The semantic rule for a program differs slightly from that of blocks (presented in the last section) because the inherited environment and continuation are

known to be mte and fall. Rules for statements without visible labels are also given here: such statements yield an empty label-continuation pair list.

```
prog    : MAIN '(' ')' stm
                {       ! setup environment for enclosed labels
                        let cyclic p = $stm (mte[p.2]) fall;
                                c0 = p.1;
                        in      c0
                }
stm     : RETURN exp ';'
                { $$ e c1 = ( $exp | return , () ) }
        | ';'
                { $$ e c1 = ( c1, () ) }
        | exp ';'
                { $$ e c1 = ( $exp |1 popv | c1, () ) }
        | BREAK ';'
                { $$ e c1 = ( e(hbrk), () ) }
        | CONTINUE ';'
                { $$ e c1 = ( e(hcon), () ) }
        | GOTO ID ';'
                { $$ e c1 = ( e($ID), () ) }
```

A conditional statement is not labeled, but there may be labels within the body of the conditional that have to be propagated.

```
| IF '(' exp ')' stm
        { $$ e c1 =
                let     p = $stm e c1;
                        c0 = $exp | cond( p.1 , c1 );
                        list = p.2;
                in      ( c0, list )
        }
```

A jump into a while loop bypasses the loop test, but otherwise follows the test-execute structure of the loop.

```
| WHILE '(' exp ')' stm
        { $$ e c1 =
                cyclic p =
                        let     e' = e[hbrk:=c1][hcon:=p.1];
                                q = $stm e' p.1;
                                c0 = $exp | cond( q.1 , c1 );
                                list = q.2;
                        in      ( c0, list )
        }
```

Since control flows from the body of a while loop to its beginning, the continuations for the labels in the body depend on the continuation for the entire whole

loop. For this reason, the list of label-continuation pairs is determined along with the continuation for the while loop.

The remainder of the specification is essentially like that of the language in Section 6.

```
        | '{' stm_s '}'
                { $stm_s }
        | ID ':' stm
                { $$ e c1 =
                        let     p = $stm e c1;
                                c0 = p.1;
                                list = ($ID,c0) cons p.2;
                        in      ( c0, list )
                }
        ;
stm_s   : ! empty
                { $$ e c1 = ( c1, () ) }
        | stm_s stm
                { $$ e c2 =
                        let     q = $stm e c2;
                                c1 = q.1;
                                p = $stm_s e c1;
                                c0 = p.1;
                                list = p.2 cat q.2;
                        in      ( c0, list )
                }
        ;
```

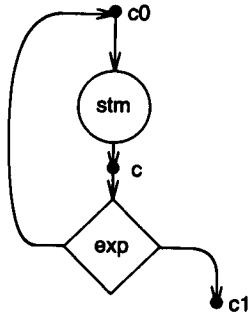
The discussion of environments and the elimination of indirection through the environment (see Section 6.3) carries over to unrestricted goto statements.

8. VARIANTS OF THE WHILE STATEMENT

One advantage of generating a compiler from a specification is that it is fairly easy to add variants of constructs that are already in the language. We illustrate by adding do and for statements.

8.1 Do Statements

The basic difference between do and while statements is that while statements have a test-execute cycle, but do statements have an execute-test cycle. The meaning of a do can be explained in terms of the following diagram:



Any breaks within stm transfer control to the point for continuation c1, that is, to a point just after the do. Any continues within stm transfer control to the point for continuation c, that is, to just before the test. The following relationships exist between the continuations

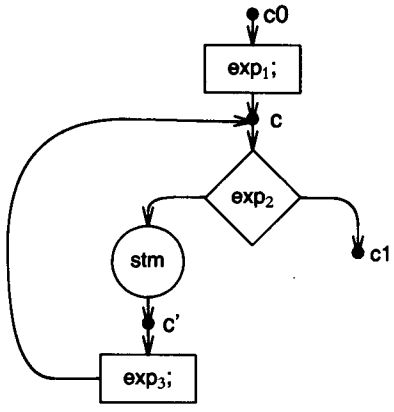
```
c = $exp | cond( c0 , c1 )

c0 = δ ( e[hbrk:=c1][hcon:=c] ) c
```

where δ is the continuation transformer part of $stm. The rule is

```
| DO stm WHILE '(' exp ')' ';'
        { $$ e c1 =
                cyclic p =
                        let     c = $exp | cond( p.1 , c1 );
                                e' = e[hbrk:=c1][hcon:=c];
                        in      $stm e' c
        }
```

8.2 For Statements

The first expression in a for statement specifies initialization that is performed once on entry to the for; the second specifies a test, as in a while loop; the third is executed at the end of each iteration generally to perform incrementation. The following diagram applies.



As usual, a break within stm sends control to the loop exit, so continuation c1 is used. A continue on the other hand causes the third expression to be evaluated before starting the next iteration, more precisely, c′ is to be used.

The semantic rule first determines the continuation c and the list of label-continuation pairs; c0 is then determined from c and (c0, list) is returned.

```
| FOR '(' exp ';' exp ';' exp ')' stm
        { $$ e c1 =
                let     cyclic p =
                        let     c' = $exp.3 |1 popv | p.1;
                                e' = e[hbrk:=c1][hcon:=c'];
                                q = $stm e' c';
                                c = $exp.2 | cond( q.1 , c1 );
                                list = q.2;
```

```
        in      ( c, list );
        c0 = $exp.1 |1 popv | p.1;
        list = p.2;
in      ( c0, list )
        }
```

8.3 Else Clauses

For completeness, we add else clauses to conditionals. In the following rule, the lists of label-continuation pairs from the true and false parts are concatenated to take labels from both arms into account. It is assumed that a label is declared only once, so the order in which the lists are concatenated does not matter. The question of checks like those for multiply defined labels will be taken up in a separate paper.

```
| IF '(' exp ')' stm ELSE stm
        { $$ e c1 =
                let     p = $stm.1 e c1;
                        q = $stm.2 e c1;
                        c0 = $exp | cond( p.1 , q.1 );
                        list = p.2 cat q.2;
                in      ( c0, list )
        }
```

9. SWITCH STATEMENTS

Switch statements have been saved until the end for a purpose. The compilers for C are very careful about the code generated for switches, and the question of integrating a special purpose code generator with the approach in this paper arises. Appendix B gives a simple-minded routine for generating code for switches. The routine fits into the printing routine of Appendix A and can easily be replaced by a fancier one. The details are in Appendix B.

9.1 Informal Description of Switches

A switch in C is a multiway branch depending on the value of an expression. The syntax is

```
| SWITCH '(' exp ')' stm
```

Substatements within stm may be of the form

```
| CASE NUM ':' stm
| DEFAULT ':' stm
```

The C reference manual has the following description:

> When the switch statement is executed, its expression is evaluated and compared with each case constant. If one of the case constants is equal to the value of the expression, control is passed to the statement following the matched case prefix. If no case constant matches the expression, and if there is a default prefix, control passes to the prefixed statement. If no case matches and if there is no default then none of the statements in the switch is executed.
>
> case and default prefixes in themselves do not alter the flow of control, which continues unimpeded across such prefixes.
>
> A break in a switch causes control to pass to the statement following the switch. [27, p. 203; copyright © 1978, Bell Telephone Laboratories, Inc., reprinted by permission.]

9.2 Case Numbers Are Like Labels

Case numbers have to be extracted out of a statement in the same manner that labels were extracted in Sections 6 and 7. After all, in order to determine if the expression value matches one of the case numbers, we have to know what the case numbers are. A new domain VC containing case number-continuation pairs is therefore introduced.

```
:                   VC = [V,C];
$stm,$stm_s:        Env -> C -> [C,LC*,VC*];          ! VC* added here
```

The only change to the existing rules in Section 8 is that lists of case number-continuation pairs have to be propagated just as lists of label-continuation pairs are propagated. We avoid listing the rules here since the final specification is given in Appendix C.

The new rule for case prefixes is similar to that for label prefixes:

```
| CASE NUM ':' stm
        { $$ e c1 =
                let     p = $stm e c1;
                        c0 = p.1;
                        lc = p.2;
                        vc = ($NUM,c0) cons p.3;
                in      ( c0, lc, vc )
        }
```

Default prefixes will be handled just like case prefixes: a special-value vdef distinct from all other values will be introduced for the purpose.

```
"vdef":             V;

        | DEFAULT ':' stm
                { $$ e c1 =
                        let     p = $stm e c1;
                                c0 = p.1;
                                lc = p.2;
                                vc = (vdef,c0) cons p.3;
                        in      ( c0, lc, vc )
                }
```

The last rule that will be discussed here is that for switch statements themselves. Once the list in the domain VC* is constructed, a mechanism is needed for finding the case number matching the value of the expression in the switch. We abstract the details of this mechanism by defining a basic function switch. In addition to the list in VC*, switch has to be supplied with the continuation on exit from the switch statement since this continuation is used if there is no default in the switch body.

```
"switch":           [VC*,C] -> [V,S] -> A;          ! VC*: cases + continuations
                                                    ! C: used if no default
                                                    ! [V,S]: value of exp
```

```
| SWITCH '(' exp ')' stm
        { $$ e c1 =
                let     e' = e[hbrk:=c1];
                        p = $stm e' c1;
                        c0 = $exp | switch( p.3, c1 );
                        lc = p.2;
                        vc = ();
                in      ( c0, lc, vc )
        }
```

10. DISCUSSION

10.1 Summary

Within the limitations of the medium, this paper is a demonstration of a semantics-directed compiler generator. All the details of the input specification and much of the background information have been shown. Notes on running times are given below.

A language specification consists of: lexical information (not discussed); a syntactic specification for the parser generator Yacc [24]; and semantic rules in a functional metalanguage called Plumb. In a syntax-directed manner, a program in the language being defined can be translated into an expression in Plumb. The expression in Plumb is a concrete representation of the semantics of the program and is referred to as the *concrete semantics* of the program.

Compilation is based on the concrete semantics. Informally, the semantics of control flow is in terms of a flow diagram—the diagram is actually a functional counterpart of the usual flow diagrams. Plumb contains a "pipe" construct for combining functions [40] that permits diagrams to be constructed easily. Central to the construction of the diagram is the representation of recursive definitions by cycles as in [47, 43] and the static elimination of environments that associate program points with labels.

A code generator has to be supplied by the compiler writer. A trivial one is used in this paper. Specialized code can be inserted if desired (see the handling of switch statements in Section 9, for example). Since the concrete semantics of a program is essentially a flow diagram, it should be easier to base code optimization on the concrete semantics than on the abstract syntax.

There are two reasons for incrementally specifying the semantics of the statement constructs of C. The first is that it is easier to introduce the concepts one by one. The second is to show that semantics-directed compiler generation provides flexibility to a language designer. Consider for example the problem of restricting the scope of labels so that goto statements are more disciplined. In a hand-crafted compiler it may not be easy to make the change. The difficulty of making the change with a compiler generator can be assessed by comparing the specifications in Sections 6 and 7, where restricted and unrestricted goto statements are considered.

10.2 Running Times

Execution times (in seconds on a PDP 11/70) are given relative to the logical organization in Figure 1. The times were obtained for the specification in Appen-

dix C of all the control flow constructs of C: Lex 8.8; *d2y* 3.4; Yacc 7.0; C compiler (on the outputs of Lex and Yacc, the reducer, and the code generator) 84.1. The generated compiler (41,318 bytes long) took less than 0.5 seconds on each of the examples in this paper.

10.3 Influences on the Implementation

Scott and Strachey closed their seminal paper [42] with the claim that a mathematical semantics will provide a "standard against which to judge an implementation." The Semantics Implementation System (SIS) of Mosses [36] goes further since it translates programs into a concrete semantics and then interprets that concrete semantics. The SEMANOL system [5, 6] is an interpreter generator based on operational semantics.

SIS stimulated a lot of work on compiler generation: see for example the papers in [26] and the survey [19]. The prospect of generating an implementation from any denotational semantics is exciting. Unfortunately, SIS is inefficient: "a large portion of the inefficiency of SIS derives from its method for parsing" [12]. Paulson's compiler generator avoids some of these difficulties: a Pascal compiler generated by Paulson "is twenty-five times slower than the regular Pascal compiler" [38]. Efficient and practical implementations are claimed to have been obtained by Raskovsky [41] by starting with a denotational semantics, hand-transforming it into an implementation-oriented semantic description, and then automatically generating a compiler. For the transformations to apply, the starting semantics must follow certain conventions, so the method sacrifices some generality. A more modular use of manual assistance occurs in the implementations of Gaudel [18] and Christiansen and Jones [14], that are based on abstract data types. See also [10, 17].

For some time Mosses [37] has argued that an algebraic approach would make denotational semantics easier to understand and process. In fact, a lot of advice has been given on structuring compilers and proving them correct using an algebraic approach [35, 2, 37, 48, 14]. This advice has had an indirect influence on the present work: Figure 1 is closest to Wand's proposed structure [48]. In addition to the overall structure, the metalanguages of Mosses [37] and Wand [48] were studied. In [48] clever representations of continuation-style operators are used to construct concrete semantics that looks like machine code. The formalization of pipes in [40] allows direct operators to be used in clever representations of continuation semantics. There is also an easy way of translating terms involving pipes into more standard continuation-style terms [40].

The connection between control flow and continuations dates back to McCarthy [32], who constructed recursive equations from flow diagrams as discussed in Section 1.5. Although continuations have been rediscovered a number of times, Morris [34] and Wadsworth [45] are given the credit for constructing syntax-directed rules for determining the continuation for a program. The reverse process of doing code generation by translating continuations into control flow is done in a fairly ad hoc way in [41].

The use of an ad hoc code generator in this paper allows the programs in the dotted box in Figure 1 to be used as compiler construction tools. One of the first questions that was asked about this work was, "How are you going to compile switches?" As mentioned in Section 9, the compilers for C very carefully generate

good code for switches. Rather than compete with each compiler writer's favorite compiling technique, the decision was made to provide an opportunity for a compiler writer to do the actual code generation.

A final question raised by the referees concerns code generation for other machines. It should be clear from the examples that the code generated in this paper is actually pseudocode for a stack-oriented machine. Similar code is produced at an intermediate stage by a number of compilers, so it should not be difficult to send the pseudocode in the examples to a "real" code generator that worries about the idiosyncrasies of "real" machines.

11. CONCLUSION

Gaudel's survey of compiler generation [19] ends with: "Finally a last, pragmatic and decisive reason is that even a slow, memory-consuming compiler generator, producing correct and reasonably efficient compilers, would be better than several man–years of coding a possibly incorrect compiler, provided that the specification method is not too difficult to use." I believe that the results in this paper show that the adjectives "slow" and "memory consuming" can be dropped for the control flow aspects of programming languages.

APPENDIX A. CODE GENERATION

A routine to print the internal representation of graphs like the one in Figure 3 must have some knowledge of the internal representation. At the moment no attempt has been made to encapsulate the details of the internal representation. Experience with writing code generators will be used to guide the design of a suitable interface at some future date.

A1 Nodes

The following C declaration shows the structure of a graph node.

```
struct gnode
{
        int             lab;        /* label e.g. PIPE, LIST, ... */
        int             knd;        /* 0–1 for PIPE; id. number, ... */
        int             mark;       /* keep track of visits */
        struct gnode    *l, *r;     /* left and right sons */
        ...
};
```

Pointers to graph nodes can be declared using gnodep, itself declared by

```
typedef struct gnode *gnodep;
```

There are two useful functions: gp2n(p) converts a pointer p into a unique integer; mfathers(p) returns true if the node pointed to by p has multiple fathers, and false otherwise. In the output an instruction label will be generated using gp2n(p) if mfathers(p) returns true.

While there are node labels corresponding to each construct in Plumb, only a few of the label types are needed here. Ugly as the prefix d2 may seem on some

of the identifiers below, it keeps the identifiers distinct from the ones a compiler writer might want to use.

d2ELEM. The ELEM comes from "element" of a domain. There will be a d2ELEM node for each basic function like plus. An identifier is generated by adding the prefix d2 to the name of a basic function; this identifier has an integer value that is entered in the knd field of the node for the basic function. The C routine putid applied to, say, d2plus, puts the characters plus into the standard output.

d2PIPE. The knd field will be 0 or 1 depending on whether | or | 1 is intended.

d2APP. For the application of the left son to the right.

d2LIST. Lists are in right linear form. In cond(c, d), the pair (c, d) will be represented as a list containing two d2LIST nodes, with a special node pointed to by gnil being the right son of the second list node.

A2 Printing Routine

The compiler writer supplies a file d2main.c containing a main routine, and any other desired routines. main must first call yyparse, for the parser and reducer to place a pointer to the root of the internal graph in the global variable rootp. On return from yyparse, the mark field of each node is initialized to d2NO, a negative integer. The routine yyerror prints a string in the error output and returns control to the caller.

The routine d2main.c for the languages in Sections 3 and 5–8 is shown below. Nodes corresponding to continuations will have their mark field set to d2YES, a predefined positive value. An integer label will be printed for a continuation node if there is more than one father for the node. Subsequent visits to the marked node generate a goto to the integer label.

```
main()
{
        yyparse();
        gnprint(rootp); printf("\n");
}

void gnprint(p)
gnodep p;
{
        if( p->mark != d2NO )                    /* happens only for continuations */
        {
                printf("\tgoto %d",gp2n(p));
                return;
        }
        switch(p->lab)
        {
        default:
                yyerror("unexpected label");
                break;
```

```
        case d2ELEM:
                if( p->knd == d2fall )
                {
                        p->mark = d2YES;
                        if( mfathers(p) ) printf("%d:",gp2n(p));
                        printf("\tfall off the end");
                }
                else
                {
                        printf("\t"); putid(p->knd);
                }
                break;
        case d2PIPE:
                p->mark = d2YES;
                if( mfathers(p) ) printf("%d:",gp2n(p));
                gnprint(p->l); printf("\n"); gnprint(p->r);

                break;
        case d2APP:
                if( p->l->lab != d2ELEM) yyerror("misapplication");

                else if( p->l->knd == d2fetch )
                {
                        printf("\tfetch "); putid(p->r->knd);
                }
                else if( p->l->knd == d2store )
                {
                        printf("\tstore "); putid(p->r->knd);
                }
                else if( p->l->knd == d2load )
                {       printf("\tload %d",p->r->knd);
                }
                else if( p->l->knd == d2cond )
                {
                        gnodep truep, falsep;
                        truep = p->r->l; falsep = p->r->r->l;
                        printf("\tonfalse goto %d\n",gp2n(falsep));
                        gnprint(truep);
                        if( falsep->mark == d2NO )
                        {
                                if( mfathers(falsep) ) printf("\n");
                                else printf("\n%d:",gp2n(falsep));
                                gnprint(falsep);
                        }
                }
                break;
        }
}
```

APPENDIX B. CODE FOR SWITCHES

The list of case number-continuation pairs from the body of a switch has to be converted into code for locating the appropriate case. The arguments of the switch basic function are the list and the exit continuation to be used if there is no default prefix. The following routine makes two passes over the list. If a list entry is for the default prefix, then the corresponding continuation is overwritten for the exit continuation in def in the routine. Otherwise, a simple test for equality of the case number with the expression value is generated. Since an equality test pops the values of its arguments, the expression value has to be explicitly copied using copyv. The second pass ensures that the vertices for all continuations are visited so that code for them can be generated.

```
else if( p->l->knd == d2switch )
{
        gnodep def, q, t;
        def = p->r->r->l;
        for( q = p->r->l; q != gnil; q = q->r)
        {
                if( q->l->l->lab != d2ELEM );
                else if( q->l->l->knd != d2vdef );
                else
                {       def = q->l->r->l;
                        continue;
                }
                printf("\tcopyv\n");
                printf("\tload %d\n\teq\n",q->l->l->knd);
                printf("\tontrue goto %d\n",gp2n(q->l->r->l));
        }
        printf("\tpopv\n");
        printf("\tgoto %d",gp2n(def));
        for( q = p->r->l; q != gnil; q = q->r )
        {
                t = q->l->r->l;
                if( t->mark != d2NO ) continue;
                if( mfathers(t) ) printf("\n");
                else printf("\n%d:",gp2n(t));
                gnprint(t);
        }
}
```

The above routine has to be inserted into d2main.c in Appendix A, right after the part for d2cond. A sample program and its output follow.

```
main()
{
        switch(p)
```

```
        {
        case 3: case 1:        a:=b;
        case 5:                c:=d; break;
        default:               e:=f;
        case 12:               g:=h;
        }
}
```

```
295:    fetch p                        popv
        copyv                          goto 196
        load 3                 55:     fetch b
        eq                             store a
        ontrue goto 55                 popv
        copyv                  120:    fetch d
        load 1                         store c
        eq                             popv
        ontrue goto 55         23:     fall off the end
        copyv                  196:    fetch f
        load 5                         store e
        eq                             popv
        ontrue goto 120        244:    fetch h
        copyv                          store g
        load 12                        popv
        eq                             goto 23
        ontrue goto 244
```

Clearly a more sophisticated routine can be used instead of the above simple-minded routine.

APPENDIX C. FINAL SPECIFICATION

```
#include "ydecs"
%%
#include "basics"

"hbrk","hcon":    lde;                    ! implicit labels set by
                                           ! break and continue
e:                Env = lde -> C;         ! label environments
"mte":            Env;                    ! (mt==empty) environment

:                 LC = [lde,C];           ! label–continuation pairs

:                 VC = [V,C];             ! case number–cont. pairs
                                           ! for switches

"vdef":           V;                      ! special default value
"switch":         [VC*,C] -> [V,S] -> A;  ! VC*: cases + continuations
                                           ! C: used if no default
                                           ! [V,S]: value of exp
```

```
$stm, $stm_s:     Env -> C -> [C,LC*,VC*];        ! Env: used by embedded gotos
                                                  ! C: normal exit continuation
                                                  ! [C: entry continuation
                                                  ! LC*: visible labels+contin.
                                                  ! VC*: visible cases+contin.
%%
#include "exp"
prog      : MAIN '(' ')' stm
                  {       ! setup environment for enclosed labels
                          let cyclic p = $stm (mte[p.2]) fall;
                                  c0 = p.1;
                          in      c0
                  }
          ;
stm       : RETURN exp ';'
                  { $$ e c1 = ( $exp | return , (), () ) }
          | ';'
                  { $$ e c1 = ( c1, (), () ) }
          | exp ';'
                  { $$ e c1 = ( $exp |1 popv | c1, (), () ) }
          | BREAK ';'
                  { $$ e c1 = ( e(hbrk), (), () ) }
          | CONTINUE ';'
                  { $$ e c1 = ( e(hcon), (), () ) }
          | GOTO ID ';'
                  { $$ e c1 = ( e($ID), (), () ) }
          | IF '(' exp ')' stm
                  { $$ e c1 =
                          let     p = $stm e c1;
                                  c0 = $exp | cond( p.1 , c1 );
                                  lc = p.2;
                                  vc = p.3;
                          in      ( c0, lc , vc )
                  }
          | IF '(' exp ')' stm ELSE stm
                  { $$ e c1 =
                          let     p = $stm.1 e c1;
                                  q = $stm.2 e c1;
                                  c0 = $exp | cond( p.1 , q.1 );
                                  lc = p.2 cat q.2;
                                  vc = p.3 cat q.3;
                          in      ( c0, lc , vc )
                  }
          | WHILE '(' exp ')' stm
                  { $$ e c1 =
                          cyclic p =
                                  let     e' = e[hbrk:=c1][hcon:=p.1];
                                          q = $stm e' p.1;
```

```
                                        c0 = $exp | cond( q.1 , c1 );
                                        lc = q.2;
                                        vc = q.3;
                                in      ( c0, lc , vc )
        }
| DO stm WHILE '(' exp ')' ';'
        { $$ e c1 =
                cyclic p =
                        let     c = $exp | cond( p.1 , c1 );
                                e' = e[hbrk:=c1][hcon:=c];
                        in      $stm e' c
        }
| FOR '(' exp ';' exp ';' exp ')' stm
        { $$ e c1 =
                let cyclic p =
                        let     c' = $exp.3 |1 popv | p.1;
                                e' = e[hbrk:=c1][hcon:=c'];
                                q = $stm e' c';
                                c = $exp.2 | cond( q.1 , c1 );
                                lc = q.2;
                                vc = q.3;
                        in      ( c, lc , vc );
                        c0 = $exp.1 |1 popv | p.1;
                        lc = p.2;
                        vc = p.3;
                in      ( c0, lc , vc )
        }
| ID ':' stm
        { $$ e c1 =
                let     p = $stm e c1;
                        c0 = p.1;
                        lc = ($ID,c0) cons p.2;
                        vc = p.3;
                in      ( c0, lc , vc )
        }
| CASE NUM ':' stm
        { $$ e c1 =
                let     p = $stm e c1;
                        c0 = p.1;
                        lc = p.2;
                        vc = ($NUM,c0) cons p.3;
                in      ( c0, lc, vc )
        }
| DEFAULT ':' stm
        { $$ e c1 =
                let     p = $stm e c1;
                        c0 = p.1;
```

```
                                    lc = p.2;
                                    vc = (vdef,c0) cons p.3;
                            in      ( c0, lc, vc )
                    }
            | SWITCH '(' exp ')' stm
                    { $$ e c1 =
                            let     e' = e[hbrk:=c1];
                                    p = $stm e' c1;
                                    c0 = $exp | switch( p.3, c1 );
                                    lc = p.2;
                                    vc = ();
                            in      ( c0, lc, vc )
                    }
            | '{' stm_s '}'
                    { $stm_s }
            ;
stm_s       : ! empty
                    { $$ e c1 = ( c1, (), () ) }
            | stm_s stm
                    { $$ e c2 =
                            let     q = $stm e c2;
                                    c1 = q.1;
                                    p = $stm_s e c1;
                                    c0 = p.1;
                                    lc = p.2 cat q.2;
                                    vc = p.3 cat q.3;
                            in      ( c0, lc , vc )
                    }
            ;
```

ACKNOWLEDGMENTS

Comments by Al Aho and Bjarne Stroustrup on the presentation are appreciated. The referees were very thorough in their reading of this paper and made numerous suggestions for its improvement. Discussions with Neil Jones, Peter Mosses, and Mitchell Wand have been most profitable; it is hard to be more specific.

REFERENCES

1. ADJ: GOGUEN, J.A., THATCHER, J.W., WAGNER, E.G., AND WRIGHT, J.B. Initial algebra semantics and continuous algebras. *J. ACM 24*, 1 (Jan. 1977), 68–95.
2. ADJ: THATCHER, J.W., WAGNER, E.G., AND WRIGHT, J.B. More on advice on structuring compilers and proving them correct. *Theor. Comput. Sci. 15*, (1981), 223–249.
3. AHO, A.V. Translator writing systems: Where do they now stand? *Computer 13*, 8 (Aug. 1980), 9–14. This paper is a guest editor's introduction and overview of a special issue on Translator Writing Systems.
4. AHO, A.V., AND ULLMAN, J.D. *Principles of Compiler Design.* Addison-Wesley, Reading, Mass. (1977).
5. ANDERSON, E.R., BELZ, F.C., AND BLUM, E.K. SEMANOL (73): A metalanguage for program-

ming the semantics of programming languages." *Acta Inf. 6* (1976), 109–131.

6. ANDERSON, E.R., BELZ, F.C., AND BLUM, E.K. Issues in the formal specification of programming languages. In *Formal Description of Programming Concepts*, E. J. Neuhold, Ed., Elsevier North-Holland, New York (1978), pp. 1–30.
7. BACKUS, J.W., BEEBER, R.J., BEST, S., GOLDBERG, R., HAIBT, L.M., HERRICK, H.L., NELSON, R.A., SAYRE, D., SHERIDAN, P.B., STERN, H., ZILLER, I., HUGHES, R.A., AND NUTT, R. The Fortran automatic coding system. *Western Joint Computer Conference*, pp. 188–198 (1957).
8. BAUER, F. L. Historical remarks on compiler construction, In *Compiler Construction: An Advanced Course, 2nd ed*, F.L. Bauer and J. Eickel, Eds., Lecture Notes in Computer Science 21, Springer-Verlag, New York (1976), pp. 603–621.
9. BAUER, F.L. AND EICKEL, J. Eds. *Compiler Construction: An Advanced Course, 2nd ed*, Lecture Notes in Computer Science 21, Springer-Verlag, Berlin (1976).
10. BJØRNER, D. Programming languages: Formal development of interpreters and compilers. In *International Computing Symposium 1977*, E. Morlet and D. Ribbens, Eds., Elsevier North-Holland, New York (1977), pp. 1–21.
11. BJØRNER, D. AND JONES, C.B. *The Vienna Development Method: The Meta-Language.* Lecture Notes in Computer Science 61, Springer-Verlag, New York (1978).
12. BODWIN, J., BRADLEY, L., KANDA, K., LITLE, D., AND PLEBAN, U. Experience with an experimental compiler generator based on denotational semantics. In *Proc. SIGPLAN '82 Symp. Compiler Construction, SIGPLAN Notices 17*(6), pp. 216–229 (June 1982).
13. CATTELL, R.G.G. Automatic derivation of code generators from machine descriptions. *ACM Trans. Program. Lang. Syst. 2*, 2 (Apr. 1980) 173–190.
14. CHRISTIANSEN, H. AND JONES, N.D. Control flow treatment in a simple compiler generator. In *Formal Description of Programming Concepts II*, D. Bjørner, Ed., Elsevier North-Holland, New York (1982), pp. 38–62. The page numbers refer to the preliminary proceedings of the IFIP TC-2 Working Conference, Garmisch, West Germany, June 1982.
15. DAVIDSON, J.W., AND FRASER, C.W. The design and application of a retargetable peephole optimizer, *ACM Trans. Program. Lang. Syst. 2*, 2 (Apr. 1980), 191–202.
16. FELDMAN, J. AND GRIES, D. Translator writing systems. *Commun. ACM 11*, 2 (Feb. 1968), 77–113.
17. GANZINGER, H. Transforming denotational semantics into practical attribute grammars. In *Semantics Directed Compiler Generation*, N.D. Jones, Ed., Lecture Notes in Computer Science 94, Springer-Verlag, New York (1980), pp. 1–69.
18. GAUDEL, M.C. Specification of compilers as abstract data type representations. In *Semantics Directed Compiler Generation*, N. D. Jones, Ed., Lecture Notes in Computer Science 94, Springer-Verlag, New York (1980), pp. 140–164.
19. GAUDEL M.C. Compiler generation from formal definition of programming languages: a survey. In *Formalization of Programming Concepts, Intl. Colloquium, Peniscola, Spain*, Lecture Notes in Computer Science 107, Springer-Verlag, New York (Apr. 1981), pp. 96–114.
20. GLANVILLE, R.S. AND GRAHAM, S.L. A new method for compiler code generation (extended abstract). In *Proc. Fifth Annual Symp. Principles of Programming Languages.* (Jan. 1978), pp. 231–240, ACM, New York.
21. GOLDSTINE, H.H., AND VON NEUMANN, J. Planning and coding problems for an electronic computing instrument, Part II, Vol. 1. In *John von Neumann: Collected Works, Vol. V*, Macmillan, New York (1963) pp. 80–151. The report was prepared for the U.S. Army Ordnance Department in April 1947.
22. GORDON, M.J.C. *The Denotational Description of Programming Languages.* Springer-Verlag, New York (1979).
23. GRIES, D. *Compiler Construction forDigital Computers.* Wiley, New York (1971).
24. JOHNSON, S.C. Yacc—yet another compiler compiler. *Computer Science Tech. Rep. 32*, Bell Laboratories, Murray Hill, N.J. (July 1975). See the *UNIX Programmer's Manual 2*, Section 19 (January 1979), Bell Laboratories, Murray Hill, N.J.
25. JOHNSON, W.L., PORTER, J.H., ACKLEY, S.I., AND ROSS, D.T. Automatic generation of efficient lexical processors using finite state techniques, *Commun. ACM 11*, 12 (Dec. 1968), 805–813.
26. JONES, N.D. Ed. *Semantics-Directed Compiler Generation*, Lecture Notes in Computer Science 94, Springer-Verlag, New York (1980).
27. KERNIGHAN, B.W., AND RITCHIE, D.M. *The C Programming Language*, Prentice-Hall, Englewood Cliffs N.J. (1978).

28. KNUTH D.E. History of writing compilers. In *Proc. 1962 ACM National Conference*, (Syracuse, N.Y., September 4–7), ACM, New York, pp. 43.
29. KNUTH, D.E. Semantics of context-free languages. *Math. Syst. Theory 2*, 2 (June 1968) 127–145 Correction in Vol. 5, no. 1 (1971) pp. 95–96.
30. LESK, M.E. Lex—a lexical analyzer generator. *Computer Science Tech. Rep. 39*, Bell Laboratories, Murray Hill NJ (October 1975). See the version by M.E. Lesk and E. Schmidt in the UNIX *Programmer's Manual 2*, Section 20 (Jan. 1979), Bell Laboratories, Murray Hill, N.J.
31. LEWIS, P.M., ROSENKRANTZ, D.J. AND STEARNS, R.E. *Compiler Design Theory*. Addison-Wesley, Reading, Mass (1976).
32. MCCARTHY, J. Towards a mathematical science of computation. In *Information Processing 1962*, C. M. Popplewell, Ed., Elsevier North-Holland, New York (1963), pp. 21–28.
33. MILNE, R.E. AND STRACHEY, C. *A Theory of Programming Language Semantics*. Chapman and Hall, London, and J. Wiley, New York (1976).
34. MORRIS, F.L. The next 700 programming language descriptions. Unpublished manuscript (Nov. 1970).
35. MORRIS, F.L. Advice on structuring compilers and proving them correct. In *Proc. ACM Symp. Principles of Programming Languages*, (Boston, Mass., Oct. 1–3, 1973), ACM, New York, pp. 144–152.
36. MOSSES, P.D. SIS—semantics implementation system: Reference manual and user guide. DAIMI MD-30, Dept. Computer Science, University of Aarhus, Denmark (Aug. 1979).
37. MOSSES, P.D. Abstract semantic algebras! In *Formal Description of Programming Concepts II*, D. Bjørner, Ed., Elsevier North-Holland (1982) pp. 63–88. The page numbers refer to the preliminary proceedings of the IFIP TC-2 Working Conference, Garmisch, West Germany, June 1982.
38. PAULSON L. A semantics-directed compiler generator. In *Proc. Ninth Annual ACM Symposium on Principles of Programming Languages* (Albuquerque N.M., Jan. 25–27, 1982), ACM, New York, pp. 224–233.
39. RAIHA, K.-J. Bibliography on attribute grammars, *SIGPLAN Notices 15*, 3 (Mar. 1980), pp. 35–44.
40. RAOULT, J.-C., AND SETHI, R. Properties of a notation for combining functions. In *Automata, Languages and Programming, 9th Colloquium, Aarhus, Denmark*, Lecture Notes in Computer Science 140, Springer-Verlag, New York (July 1982). Revised version in *J. ACM. 30*, 3 (July 1983) 595–611.
41. RASKOVSKY, M. Denotational semantics as a specification of code generators. In *Proc. SIGPLAN '82 Symposium on Compiler Construction, SIGPLAN Notices 17*, 6 (June 1982), pp. 230–244.
42. SCOTT, D.S. AND STRACHEY, C. *Towards a mathematical semantics for computer languages*, In *Proc. Symp. Computers and Automata*, Polytechnic Press, Brooklyn, New York (April 1971), pp. 19–46.
43. SETHI, R. Circular expressions: elimination of static environments. *Science of Computer Programming 1*, 3 (May 1982), pp. 203–222.
44. STOY, J.E. *Denotational Semantics*, MIT Press, Cambridge, Mass. (1977).
45. STRACHEY, C. AND WADSWORTH, C. Continuations: a mathematical semantics which can deal with full jumps. Technical Monograph PRG-11, Programming Research Group, Oxford University (1974).
46. TENNENT, R.D. The denotational semantics of programming languages. *Commun. ACM 19*, 8 (Aug. 1976), 437–453.
47. TURNER, D.A. A new implementation technique for applicative languages. *Softw. Prac. Exper. 9*, 1 (Jan. 1979), 31–49.
48. WAND, M. Deriving target code as a representation of continuation semantics. *ACM Trans. Program. Lang. Syst. 4*, 3 (July 1982), 496–517.

Received October 1981; revised December 1982; accepted January 1983