A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

By

Lawrence Paulson

December 1981

Acknowledgements

For the past three years, John Hennessy has assisted my study of programming languages. He suggested that I investigate semantics, and later noted that the compiler generator might be feasible. As the research progressed, John helped me to focus on the most important problems. He made extensive comments on numerous drafts of this thesis and other papers.

Zohar Manna and Gio Wiederhold have taken an interest in my career ever since I came to Stanford. Zohar introduced me to denotational semantics; recently, he got me several job offers. Sue Owicki made several insightful comments about the dissertation; Jeff Ullman pointed out that I had omitted some important material.

Wolf Polak informed me of people who were doing related work; Ole L. Madsen sent a wealth of research reports.

Of Stanford's capable and underpaid staff, Carolyn Tajnai kept me posted on all the deadlines and requirements, and was the friendliest person in the Department. Betty Scott resolved the quarterly funding emergencies.

I would like to thank my friends for all we shared during my years at Stanford. Special thanks to Bob and Anne. If not for Randy Gellerman, this work would have been completed much sooner.

This work was supported in part by Advanced Research Projects Agency Contract MDA 903-76-C-02306 and Joint Services Electronics Program Contract DAAG 29-79-C-0047.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

Language designers must compromise between their goals and resources, and reconcile conflicting concepts into a harmonious whole. They cannot try out their ideas on real programs, because of the cost and time required to write compilers. Their only feedback comes years later, as users evaluate the language. This thesis intends to make it easier to design, document, and implement programming languages.

There is no widely accepted notation for describing programming languages, so the designers generally use a mixture of Backus-Naur Form and English. The resulting document is often confusing, ambiguous, and tedious.

A bad document compounds the burden on the compiler writers. Before they can begin to implement the language, they must understand the document and resolve its ambiguities. No wonder compilers are so often incompatible with one another, that programs written in high-level languages are not transportable, and that it is so difficult to distribute new languages throughout the computer science community. Consider the experience with Algol 60, Pascal, and Fortran:

The Algol Report [43] is well-written, with plenty of examples. Yet Knuth [26] notes that Algol is ambiguous about side-effects, go to statements, for statements, parameters, and several other topics. This is not a poor showing; Hoare [15] comments: "When you can design a language with so few trouble spots, you can be proud."

Welsh, Sneerington, and Hoare [66] cite ambiguities in the Pascal Report [19] concerning types, sets, and scope rules. Arthur Sale [54] found many differences among seven Pascal compilers.

The Fortran* Standard [3] provides neither a formal description of the syntax nor a single example. A follow-up report by the standards committee lists fifty-two topics in need of clarification [2]. Peter Poole [47] exhibits several incompatibilities among Fortran compilers, including compilers written by the same manufacturer for the same machine.

The Department of Defense noticed these problems. In commissioning the language Ada, it required that all implementations be compatible [8, 17]. That goal may prove elusive due to Ada's sheer complexity. The defining report is twice as long as the reports of Fortran, Algol, and Pascal combined. Ada includes several advanced, ill-understood features: tasking, exception handling, and generic program units.

While a formal semantic definition cannot solve all the problems of language design, it can answer the myriad questions that implementors and programmers ask: when are parameters evaluated, in what order, etc. This thesis introduces a formal notation, the semantic grammar, which is both expressive and readable. Semantic grammars describe syntax and semantics together, without separate lists of formulas or rules that need to be put into correspondence. They handle both static and dynamic semantics — both compile- and run-time actions. They describe languages at a high level of abstraction, without needless detail. They are concise; Appendix D, a grammar describing the full syntax and semantics of Pascal, is only twenty-one pages long.

To provide language designers with quick implementations of their ideas, I have implemented a compiler generator that converts semantic grammars into compilers. It has generated compilers for Pascal, Fortran, and other languages. Using the Pascal grammar, it has executed an intricate seven-page program: an LR(0) parser constructor. It can handle unconventional languages, perform extensive static checking, and report semantic errors in programs.

The compiler generator is the starting point for many systems that translate programs into another formalism. For program verification, it can translate programs into verification conditions. For efficient compilation, it can translate programs into intermediate code, which a separate program could use to generate optimized code. The compiler generator can provide compatible compilers on different machines. It is transportable, since it is written in standard Pascal.

---

*In this thesis, "Fortran" means Fortran 66, not the new Fortran 77.

Summary of the thesis:

Related Work. Semantic grammars evolved from W-grammars, denotational semantics, and attribute grammars. Most compiler generators use one of these notations. Most generate only part of a compiler; few are efficient enough for production use.

Semantic Grammars. A semantic grammar contains domain definitions, expression definitions, and attribute grammar rules. Domains and expressions follow the conventions of denotational semantics, restricted for machine processing but still powerful. Attributes express semantic dependencies and constraints among the syntactic symbols in each rule.

Static Semantics. A semantic grammar may express compiler functions such as type-checking and symbol table management. Abstract syntax trees, definable as a domain, are a natural representation of types. Symbol tables, or environments, are mappings from identifiers to their meanings. Declarations modify the environment; representing procedure and parameter declarations is straightforward but tedious.

Dynamic Semantics. A semantic grammar may express the run-time execution of programs using denotational semantics. The first step of writing the semantics is defining the state of a computation, taking Fortran or Pascal as a model. There are two frameworks for control flow: a direct semantics can describe "structured" commands; a continuation semantics can describe any flowchart. A grammar can also define axiomatic or operational semantics.

The Compiler Generator. The compiler generator consists of a grammar analyzer, universal translator, and stack machine. The grammar analyzer converts a semantic grammar into a language description file that includes LALR(1) parse tables and attribute semantics. The universal translator reads the file and parses a program, producing a graph of attribute dependencies. It simplifies the graph into a single formula, while reporting semantic errors in the program. It compiles the formula into stack machine instructions for execution. The stack machine uses Landin's SECD architecture to execute lambda-calculus formulas.

Implementation Issues. The compiler generator embodies many design decisions. Rather than interpreting semantic formulas, it compiles them into more efficient SECD machine instructions. The attribute evaluator is efficient because it neither traverses nor stores a parse tree, unlike the evaluators used in other systems. The formula simplifier uses many transformations and optimizations; to assess them, performance statistics are presented.

Conclusions. Semantic grammars have concisely defined both Fortran and Pascal. The compiler generator allows the user to debug a grammar; to avoid stubborn bugs, the user should develop and test the grammar in stages. Several lines of research may lead to compilers efficient enough for production use. Separate formalisms for static and dynamic semantics may speed compilation; specifying operational rather than denotational semantics may speed execution. Even now, the compiler generator is efficient enough to aid language designers — it can run test programs several pages long.

Chapter 2
Related Work

Semantic grammars evolved from earlier notations for defining programming languages. Attempts to generate compilers from semantic descriptions date back over ten years.

2.1. Language Description Notations

Researchers have developed many language description notations over the years, striving for precision, clarity, generality, and power. There are far too many notations to discuss them all; I concentrate on those for static semantics.

Most research, including the debate over axiomatic, operational, and denotational semantics, has investigated such dynamic complexities as recursion, control structures, and storage sharing. Static semantics — symbol tables and types — seems trivial in comparison. But language designers are introducing overloading, parametrized types, abstract modules, and scope control. Readable definitions of static semantics have become essential.

Donahue [9] and Marcotty, Ledgard, and Bochmann [34] give extended comparisions of several description methods, static and dynamic.

2.1.1. A Specification Task

Consider a language that includes the following syntax rules:

```
type =    "integer" |
          "boolean" |
          "array" "[" number ".." number "]" "of" type

assignment-command =
          variable ":=" expression
```

The language has integer, boolean, and array types; the assignment command (statement) requires that the type of the variable equal the type of the expression. Type-checking is typical of the concepts that context-free syntax cannot express. It requires describing types and associating them with variables and expressions. Since a variable has no intrinsic type (this is not Fortran), the environments and scope rules also must be described.

2.1.2. W-Grammars

One way to check types in the syntax is to have a separate rule for each type:

```
int-assignment-command =
    int-variable ": = " int-expression

bool-assignment-command =
    bool-variable ": = " bool-expression

int-array-assignment-command =
    int-array-variable ": = " int-array-expression
```

This requires an infinite number of rules, particularly to handle environments. A *W-Grammar* [62] does this through two levels of rules: *hyper-rules* and *metaproduction rules*. The hyper-rules are templates that are expanded into an infinite set of context-free rules. The metaproduction rules generate the data that fill the "slots" in the templates. Here is a hyper-rule for assignment:

```
ENV-assignment-command =
    ENV-variable-of-TYPE ": = " ENV-expression-of-TYPE
```

The upper-case names are *metanotions*, the slots to be filled in. The following metaproduction rule describes the metanotion TYPE, which is the internal representation of types.

```
TYPE ::   int |
          bool |
          array INT upto INT of TYPE
```

TYPE depends on a metanotion INT representing integers. W-grammars do not provide integers as a primitive. They must be explicitly defined, as well as essential integer operations such as division. Integer expressions are cumbersome; I will treat just the simplest case, non-negative integers represented in unary:

```
INT ::    EMPTY  |
          one INT
```

An environment is a mapping from names to their meanings, but W-grammars do not provide mappings as a primitive. The user must define a concrete representation of environments, such as an *association list* of name/type pairs, and must write hyper-rules to search a list for a name.

```
ENV ::    EMPTY  |
          map NAME to TYPE  ENV
```

W-grammars are unclear because they represent everything in low-level terms, ultimately character strings. They cannot really associate types with variables, but merely generate grammar symbols that contain the string "variable" and strings that represent types. However, they are powerful; the official definition of Algol 68 uses a W-grammar [62] to handle its intricate rules for coercion, scope, and type-checking.

2.1.3. Denotational Semantics

A *denotational* definition of a programming language consists of an abstract syntax, domain definitions, and function definitions. The domains describe the data structures for the semantic concepts of the language, like data types in Pascal. Each function describes the semantics of one nonterminal symbol. Functions are defined by cases: one case for each syntax rule rewriting the nonterminal. If our language had if and compound commands, then its abstract syntax would include:

```
command =   variable ":= " expression   |
            "if" expression "then" command      |
            "begin" command1 ";" ... ";" commandn "end"
```

The functions specify both static and dynamic semantics. Domains are introduced to represent abstract syntax trees. Brackets ⟦ and ⟧ enclose the syntactic arguments of functions. Below, the function *valid* specifies the static semantics of commands, relative to an environment *env*. The example omits the definition of *valid* for variables and expressions, and the definitions of the functions *compatible* and *type*.

valid⟦variable := expression⟧env =
    valid⟦variable⟧env and valid⟦expression⟧env and
    compatible(type⟦variable⟧env, type⟦expression⟧env)

valid⟦if expression then command⟧env =
    valid⟦expression⟧env and valid⟦command⟧env and
    compatible(type⟦expression⟧env, boolean)

valid⟦begin command$_1$; . . .; command$_n$ end⟧ =
    valid⟦command$_1$⟧ and . . . and valid⟦command$_n$⟧

Some denotational definitions are unreadable. They reduce a language to a formless list of function definitions; bad conventions are also to blame. Many authors favor meaningless names, with no similarity between the names of a variable and its domain, for example [61]:

Variable	*Domain*	*Meaning*
Γ	Com	commands
ρ	U	environments
θ	C	command continuations
π	Q	parameter continuations

Denotational definitions can treat a wide class of languages, including Algol 60, CLU, SNOBOL4, and Pascal [39, 55, 59, 61]. They can describe complex control structures like label variables and expressions that cause jumps. But they cannot easily handle tasking or concurrency.

Tennent [60] gives a brief introduction to denotational semantics. Gordon and Stoy cover more detail [11, 58]. Milne and Strachey [37] ascend far into the stratosphere.

2.1.4. Attribute Grammars

An attribute grammar [51] is a context-free grammar augmented with attributes and attribute equations, which propagate semantic information along the edges of the parse tree. *Inherited* attributes, prefixed by ↓, move information from a node down to its children. *Synthesized* attributes, prefixed by ↑, move information from the children up to the parent. Below, the nonterminal *variable* inherits an environment and synthesizes a type. The rule includes a *constraint* that the types of the variable and expression are equal; if the constraint does not hold, then the program has a semantic error.

```
command< ↓env > =
    variable< ↓env ↑type1 >
    ": = "
    expression< ↓env ↑type2 >
    constraint type1 = type2
```

There are different styles of writing attribute grammars. Knuth's original paper [27] suggests writing the attribute equations alongside rules, rather than embedded within; Wilner [67] defines the language Simula in this style. Watt and Madsen's *extended* attribute grammars [63] express the constraint $type_1 = type_2$ implicitly by using the same attribute *type* with both the variable and the expression. Such conventions shorten rules:

```
command< ↓env > =
    variable< ↓env ↑type >
    ": = "
    expression< ↓env ↑type >
```

Each attribute belongs to a particular data type, or domain. Extended attribute grammars provide domains for mappings, records, and recursive data structures. Here is one way to define a type as either boolean, integer, or array, and an environment as a mapping from variable names to types:

```
TYPE = ( bool | int | array(INT,INT,TYPE) )
ENV =  NAME → TYPE
```

Compare this with the W-grammar above. The attribute grammar expresses structure without reducing everything to character strings. Watt has written an extended attribute grammar that cleanly specifies all the context-senstive rules of Pascal [64].

2.1.5. Semantic Grammars

A semantic grammar is an extended attribute grammar where:

- The attribute domains are those of denotational semantics.
- Grammar rules specify dynamic as well as static semantics.
- The user may define auxiliary functions to use in the rules.

To illustrate how to embed dynamic semantics in a semantic grammar rule, I will use a simple

denotational description of assignment. It uses a function *var* representing variables and a function *exp* representing expressions. The assignment command evaluates the expression in the current state *s* and passes the result to var, which stores it in the state s:

λs. var(exp s)s

A traditional denotational definition separates the semantics from the syntax, re-establishing the context by explicitly providing var and exp with a syntactic construct and environment to operate on. By convention, a function definition f = λx.y is written f x = y. Here com is the semantic function for commands; it takes a command, environment, and state, producing a new state.

com⟦variable: = expression⟧env s = var⟦variable⟧env (exp⟦expression⟧env s) s

Embedding the function λs.var(exp s)s in the attribute grammar rule yields the semantic rule for the assignment command. The variable and expression synthesize their semantic functions var and exp; the rule combines these to produce the semantics of the command.

```
command< ↓env ↑ λs. var(exp s)s> =
    variable< ↓env ↑type ↑var>
    ": = "
    expression< ↓env ↑type ↑exp>
```

A semantic grammar need not specify denotational semantics. As Madsen [31] suggests, it can use operational or axiomatic methods too: instructions for an abstract machine, instructions for a real machine, execution traces, or predicate transformers.

2.2. Compiler Generators

There exist few genuine compiler generators, which automatically produce a compiler given a formal definition of a programming language. More common are *translator writing systems*, which generate only part of a compiler. Many generate only the parser; more advanced systems combine user-coded semantic routines into a compiler. Almost all require the user to code a major part of the compiler in some implementation language. They do not save the user much effort, but their compilers are more systematic, easier to maintain, and more likely to be correct than hand-written compilers.

A parser generator requires the user to program all of the semantics; it only processes the

syntax. In YACC [21], semantic code immediately follows each rule. The Stanford Pascal Parser Generator [14] translates programs into abstract syntax trees, which semantic routines process. The BOBS-System [10] calls a user procedure each time a reduction occurs. Each of these systems uses LALR(1) parsing [1], which efficiently handles complex grammars.

Affix grammars, which are similar to attribute grammars, are the input language for the CDL compiler-compiler [28, 29]. Each nonterminal is an action or predicate, defined by a macro, with parameters and local variables. CDL translates the affix grammar into a recursive-descent compiler that parses programs top-down, "executing" the nonterminals of syntax rules from left to right. There are two versions of CDL: the high-level version writes compilers in languages like Algol 60; the low-level version writes assembly language compilers, based on a standard machine model and predefined operations to control it.

NEATS is a compiler writing system that accepts extended attribute grammars [20, 63]. It provides a fixed set of domains to represent environments, parameters, types, and other language concepts. During compilation, it translates the source program into an output stream, calling a user procedure every time an output symbol is generated. The attribute grammar may specify translation from the source language into intermediate code, which the user procedure translates into machine instructions. The NEATS attribute evaluator, which I have adopted, is fast and general.

Räihä's [49, 50] Helsinki Language Processor (HLP) has generated compilers, assemblers, and preprocessors for a dozen languages, including Simula, Euclid, and PL360. It constructs a parse tree and evaluates attributes in alternating passes [48]. Attributes are Burroughs Algol variables, computed using Algol statements. HLP compiles Pascal programs at only fifteen tokens per second, and consumes 90,000 words when compiling a one-page Euclid program; it is more efficient when used as an assembler or preprocessor.

SIS, by Peter Mosses, is the first compiler generator that does not need user-coded semantic routines [40, 41, 42]. Instead it uses formal descriptions of the syntax and denotational semantics of the language to be compiled. It constructs the parse tree of a program, applies semantic functions to it, and interprets the result. Its input language is a formal version of the traditional notation for denotational semantics. An untyped lambda-calculus, extended with tuples and lists, represents source programs, object programs, and compilers. SIS has handled several small languages such as LOOP and M-Lisp. It faithfully processes

denotational formulas; its "call-by-need" interpreter handles a wider range of semantics than my compiler generator. Mosses reports that executing even short programs requires several minutes of computer time. Despite this inefficiency, SIS proves that compilers can be generated automatically from high-level language descriptions.

Ravi Sethi [56, 57] is experimenting with semantics-directed compilation. His simplifier performs beta-reductions and looks up identifiers in environments. It can resolve references to labels in goto-programs, eliminating the environment. It produces a circular expression that matches the control flow of the program. His system uses YACC to parse programs. It has processed languages similar to Mosses's, and can handle all of the control statements of the C programming language. It does not execute programs; the user must provide a procedure that translates circular expressions into imperative code. Sethi uses an algebraic formulation of denotational semantics.

Like Sethi, Jones and Christiansen [23] use algebra to handle control flow. Their compiler generator translates a language definition into a *compiling semantics*, which specifies what actions to perform during parsing to compile a program. The target machine executes sequences of instructions that update a run-time state. The user must define the machine's instruction set and the structure of its state, using denotational semantics.

Denmark's Aarhus University has conducted much research on semantics-directed compiler generation: extended attribute grammars, BOBS, SIS, NEATS, and a conference [22]. Work is continuing in several directions [23, 32].

Martin Raskovsky's compiler generator [52, 53] has converted Stoy's example language definition [58] into a compiler. In a series of steps, it translates a standard denotational definition into an low-level definition, then into the programming language BCPL. The compiler generates instructions for the PDP-10 computer. There is little documentation on this project.

Where does my compiler generator fit among the others? Of those that use denotational semantics, it is the only one that has produced compilers for complete, realistic languages, such as Pascal and Fortran. Of those that use attribute grammars, it is the only one that provides a full set of primitives for defining semantic data structures. It accepts a functional, rather than procedural, semantic specification; if attribute evaluation can cause side-effects, then the user is forced to know the order in which attributes are evaluated. Most of these

compiler generators are not efficient enough to compile a one-page program, while mine has executed a seven-page program. It provides the essential compiler functions of static semantics and error reporting, which most other research has neglected.

Chapter 3
Semantic Grammars

Most denotational definitions are too informal for computer processing, as Mosses discovered when he implemented SIS [42]. Semantic grammars are simple enough for the compiler generator to process, yet rich enough to describe most programming language concepts. A grammar contains domain definitions, expression definitions, and semantic rules.

Appendix F is a semantic grammar that formally defines the syntax and static semantics of this notation.

3.1. Domains and Expressions

The domains and expressions are those of denotational semantics. Domains represent semantic data types, such as mappings, tuples, and tree structures. It is possible to define new function, product, and union domains. An expression denotes an element of some domain. Operators only accept operands of the correct domains; there are no automatic coercions between domains.

3.1.1. Basics

Domain names are written in UPPER CASE. The variables of a domain have the same name in lower case, possibly followed by digits. For example, the variables list, list0, and list435 belong to the domain LIST. To use an arbitrary variable name, specify its domain when declaring it; lambda-, let-, letrec-, and case-expressions declare variables. An example is limit:INT.

The lattice theory underlying denotational semantics [58] augments each domain with an element $\perp$, read "bottom." Intuitively, $\perp$ represents the result of erroneous or looping computations. Most operators return $\perp$ if any operand is $\perp$.

The standard domain BOOL contains the truth values **true** and **false**. Boolean expressions may use the operators **and**, **or**, and **not**. The conditional expression

if p **then** x **else** y **fi**

denotes x if p = **true**, y if p = **false**, and ⊥ if p = ⊥. The expressions x and y must belong to the same domain.

The standard domain INT contains the integers 0, 1, −1, 2, ... Expressons may contain the following infix operators, which all have the same precedence and require INT operands:

Symbol	*Meaning*	*Result Domain*
+	sum	INT
−	difference	INT
•	product	INT
div	quotient	INT
mod	remainder	INT
lt	less than	BOOL
gt	greater than	BOOL
le	less or equal	BOOL
ge	greater or equal	BOOL

The standard domain NAME contains all character strings enclosed in quotes, such as "i", "*&(horatio". It represents identifiers that appear in programs.

If x and y belong to the same standard domain, then x **eq** y is the expression testing whether x and y denote the same truth value, number, or string. If x or y is ⊥ then the result is ⊥; it is impossible to write a function testing whether a value equals ⊥.* The expression x **ne** y denotes **not** (x **eq** y).

Examples of expressions and their domains:

Expression	*Domain*
"revolucionario"	NAME
int + 7	INT
if int **gt** 0 **then** "pos" **else** "neg" **fi**	NAME
(int1 **le** int2) **and** (int2 **le** int3)	BOOL

*In the lattice theory, a function f can not be continuous if f(⊥,⊥) = true, f(⊥,0) = false, and f(0,0) = true. Because ⊥ ⊆ 0 in the lattice ordering, monotonicity implies true ⊆ false ⊆ true. Thus true = false, a contradiction. Pragmatically, testing for ⊥ requires solving the halting problem.

3.1.2. Functions

Functions are mappings from one domain into another, representing the notions of *array* and *computable function*. The domain X→Y contains mappings from X to Y. Functions may not be compared for equality.

3.1.2.1. Abstraction

If an expression y contains a variable x, then the value of y depends upon that of x. The lambda notation λx.y expresses this dependence as a function; x is called the *bound variable* and y is called the *body*. The body extends to the right as far as possible, so λint.int + 1 denotes λint.(int + 1), not (λint.int) + 1. Examples:

Expression	*Meaning*	*Domain*
λint. int **gt** 0	test if int is positive	INT→BOOL
λy:INT. y*y	square y	INT→INT

3.1.2.2. Application

The expression f x denotes the value of the function f applied to the argument x. If f belongs to the domain X→Y, then x must belong to X and the result f x belongs to Y. Example:

(λint.int*int) 5 denotes 25.

A *higher-order* function is one that returns a function as its result. It is like a function of several arguments, but can accept its arguments one at a time. The following syntactic conventions involve higher-order functions:

Construct	*Meaning*
X→Y→Z	X→(Y→Z)
λx y . z	λx. (λy. z)
f x y	(f x) y

The application of a function to its argument, f x, may also be written f:x. The colon associates to the right, so f:g:x denotes f(g(x)). This spares some parentheses, especially in expressions that define continuations (Chapter 5).

Applying a lambda-expression to an argument is formally equivalent to substituting the argument for the bound variable throughout the body of the lambda-expression. This substitution is called *beta-reduction*. Example with two arguments:

(λint1 int2. int1 + int2) 3 5
(λint2. 3 + int2) 5
(3 + 5)
8

3.1.2.3. Modification

The updating-expression [x→y]f denotes

λu. **if** u **eq** x **then** y **else** f(u) **fi**,

a function that maps x to y but otherwise is the same as f. (The domain of x must allow the equality operator **eq**.) This models how assignment commands update the state, and how declarations update the environment. For instance, if env represents an environment, then ["fred" → y]env represents a new environment that maps "fred" to the value y. An iterated updating-expression describes a table; a table mapping p to x, q to y, r to z, and otherwise undefined, is

[p→x][q→y][r→z]⊥

3.1.2.4. Recursion and Fixedpoints

The lattice theory models recursive functions as fixedpoints of functionals. If f is a function, then fix(f) is defined to be the least value satisfying f(fix(f)) = fix(f).

A definition of the factorial function, using an explicit **fix** operator, is

fix λfactorial :INT→INT.
  λint. **if** int **eq** 0 **then** 1 **else** int * factorial(int – 1) **fi**

The compiler generator requires the argument of **fix** to be a lambda-abstraction. The sequence **fix**λ is treated as a single token, similar to LISP's operator LABEL for defining recursive functions [30].

3.1.2.5. Local Definitions

The let clause

let x = y **in** z

denotes the expression (λx.z)y. Intuitively, this stands for z where x takes on the value of y. The definition x = y can not be recursive, because y is outside the scope of x.

The letrec clause

letrec x = y in z

denotes the expression (λx.z)(fix λx.y). Letrec defines recursive functions, such as:

```
letrec factorial :INT→INT =
  λint. if int eq 0 then 1 else int * factorial(int - 1) fi
in factorial(9)
```

3.1.3. Tuples

If X and Y are domains, the *product domain* X × Y contains all pairs (x,y) where x belongs to X and y belongs to Y.

The operators **left** and **right** extract components from pairs:

left (x,y) = x and **right** (x,y) = y.

N-tuples are iterated pairs:

the domain X × Y × Z means X × (Y × Z),
the tuple (x, y, z) means (x, (y, z)).

The bound variable of a function may be a tuple of variables. This is shorthand for using **left** and **right** to extract components of the bound variable, and is useful for defining functions of several arguments. For example,

λ(int1,int2). int1 **lt** int2

is equivalent to

λpair:INT×INT . (**left** pair) **lt** (**right** pair)

and belongs to the domain (INT × INT)→BOOL.

Tuples may be compared for equality. The expression

pair1 **eq** pair2

denotes

```
((left pair1) eq (left pair2))
and
((right pair1) eq (right pair2))
```

3.1.4. Unions

If $D_1, \ldots, D_n$ are domains and $tag_1, \ldots, tag_n$ are distinct identifiers, then the *union domain*

$$[tag_1[D_1] + \ldots + tag_n[D_n]]$$

contains the following values:

$tag_1[d_1]$ for all d_1 in D_1
...
$tag_n[d_n]$ for all d_n in D_n

A particular tag name may be used in only one union domain. If each of $D_1, \ldots, D_n$ allow testing for equality, then so does their union.

A domain need not be given for every tag. A tag without a domain adds just one value to the union. The extreme case, where no tag has a domain, is like Pascal's enumerated types [19]. For instance, the domain [red + yellow + blue + green] contains four values denoted red, yellow, blue, and green. The domain [errorVal + intVal[INT] + boolVal[BOOL]] is an example of the mixed case, and contains the values:

```
errorVal
intVal[0], intVal[-1], intVal[1], ...
boolVal[false], boolVal[true]
```

An expression such as intVal[int + 2] creates a union value and is called an *injection*. Its inverse is *projection*, denoted with a bar:

```
intVal[3] | intVal      = 3
intVal[3] | boolVal     = ⊥    (of BOOL)
```

The operator is checks the tag of a union value:

```
intVal[3] is intVal     = true
intVal[3] is boolVal    = false
```

The case-expression selects among several expressions according to the tag of a union value; the following example converts colors into integers:

```
case color of red.1, yellow.2, blue.3, green.4 esac
```

The case-expression has an "arm" for each tag of the union. If a tag has a domain, then the

arm may refer to the projected value of the union. Below, if x has the form intVal[int], then the value of the case-expression depends on x|intVal, which is the value of int.

```
case x of
  errorVal.          "err",
  intVal[int].       if int gt 0 then "plus" else "minus" fi,
  boolVal[bool].     if bool then "true" else "false" fi
esac
```

3.2. Attributes

Look again at the semantic rule for the assignment command:

```
command< ↓env ↑ λs. var(exp s)s> =
    variable< ↓env ↑type ↑var>
    ":="
    expression< ↓env ↑type ↑exp>
```

The symbol *command*, and its attributes, are on the left side of the equals sign; the other symbols and attributes are on the right side. In a parse tree, the nodes for the right side symbols are the children of the node for the left side symbol. Attributes depend upon others in various ways:

- The expression λs. var(exp s)s specifies the semantics of the command in terms of the functions *var* and *exp*, supplied from below in the parse tree.
- The variable *env* appears three times. In command, the parse tree supplies the value of env from above. The variable and expression pass env down into the tree.
- Both variable and expression define the variable *type*. Since an attribute can't have two values, the types of the variable and expression must be equal.

Compiling a program requires evaluating attributes and checking that their constraints hold. Every attribute must yield a proper value; ⊥ indicates an error. This section explains how a rule defines, uses, and constrains attributes.

3.2.1. Defined Attributes

Any inherited attribute on the left side of a rule "sees" a value from above in the parse tree. Likewise, any synthesized attribute on the right side sees a value from below. These are *defined* attributes. In the rule below, the defined attributes are underlined.

```
command< ↓env ↑ λs. var(exp s)s> =
    variable< ↓env ↑type ↑var>
    ": = "
    expression< ↓env ↑type ↑exp>
```

A defined attribute may both specify the values of variables in a rule and impose constraints on the rule. This complexity arises because a defined attribute may be an expression, not just a variable.

Suppose a defined attribute sees a value *val*. The effect recursively depends upon the form of the expression, which must be one of the following:

a variable v	makes v denote val. If the same variable v is defined more than once in a rule, adds constraints that the definitions are all equal.
a constant c	adds a constraint that val = c.
a pair (x,y)	both x and y are themselves defined attributes; x sees left(val) and y sees right(val). The domain of val must be a product. (Checked during compiler generation, not compilation.)
an injection tag[x]	x is a defined attribute that sees the projection val | tag; adds the constraint that val is tag.

Watt and Madsen [63] first used expressions for defined attributes. The next chapter gives other examples.

3.2.2. Applied Attributes

Any synthesized attribute on the left side of a rule sends a value up into the parse tree. Likewise, any inherited attribute on the right side sends a value down. These are *applied* attributes. An applied attribute may contain any expression, as long as all of its free variables are defined elsewhere in the same rule. The applied attributes are underlined in the rule below:

```
command< ↓env ↑ λs. var(exp s)s> =
    variable< ↓env ↑type ↑var>
    ":="
    expression< ↓env ↑type ↑exp>
```

3.2.3. With Clauses

A rule may contain clauses of the form:

```
with x = y
```

This defines x to denote y in the rule. Strictly speaking, x is a defined attribute that sees the value y, an applied attribute. Using a with clause to extract the embedded expression in the rule for the assignment command yields an equivalent rule:

```
command< ↓env ↑com> =
    variable< ↓env ↑type ↑var>
    ":="
    expression< ↓env ↑type ↑exp>
    with com = λs. var(exp s)s
```

3.3. Putting It All Together

A semantic grammar consists of domain definitions, expression definitions, attribute declarations, semantic rules, and a resolution part. The symbol **end** terminates the grammar. Comments may appear anywhere; they begin with a number sign (#) and continue to the end of the line.

3.3.1. Domain Definitions

The domain definitions declare all the domains used to describe the semantics.* Definitions may be recursive, such as LIST, VAL, and TYPE below.

---

*Semantic grammars do not use syntactic domains, which define abstract syntax trees in traditional denotational definitions.

```
domain
    LIST = [nil + cons[INT × LIST]];          # lists of integers
    VAL = [intV[INT] + arrV[INT → VAL] ];     # values: integers and arrays
    ENV = NAME → TYPE;                        # environments: types of variables
    TYPE = [intTy + arrayTy[TYPE] ];          # types
    S = NAME → VAL;                           # states: values of variables
    EXP = S → VAL;                            # expressions
    COM = S → S;                              # commands: state transforms
    VAR = VAL → COM;                          # variable assignments
```

The domain LIST deserves special mention, for it illustrates how to define list domains in terms of union domains. (The compiler generator does not provide lists as a primitive.) A list is either nil, or has the form cons[int,list]; a list of n integers is

$$\text{cons}[\text{int}_1, \ldots, \text{cons}[\text{int}_n, \text{nil}] \ldots]$$

To introduce an "abstract" domain, with no definition, declare it **unspec**. Since the structure of the domain is unspecified, its only values are $\perp$ and **unspec** expressions. You may find **unspec** definitions useful while developing a grammar, but you must remove them all before executing programs on the compiler generator.

```
DATA = unspec;                # abstract data elements
ANS = unspec;                 # final answers
C = S → ANS;                  # command continuations
```

When comparing two domains for compatibility, a domain name is considered to be a synonym for its definition. In the above example, C and S→ANS are the same domain. Each **unspec** domain is unique, so DATA and ANS are different domains.

3.3.2. Expression Definitions

The expression definitions declare expressions that help describe the semantics. Most grammars define functions to check types or combine declarations, and define structured constants representing the initial environment. Definitions may be recursive or **unspec**. If a name is referenced before its definition, it must appear in the **forward** declarations, along with its domain. The function *append* is an example of list manipulation.

```
forward
    append : (LIST × LIST) → LIST;

define
    append = λ(list1,list2).    # append two lists
      case list1 of
       nil. list2,
       cons[int,list]. cons[int, append(list,list2)]
      esac;

    aList = unspec : LIST;    # aList is an unspecified LIST constant

    abort = λs.⊥;
```

3.3.3. Attribute Declarations

The attribute declarations list every nonterminal symbol in the grammar, along with the domains of its attributes. A dot separates inherited from synthesized attributes. In the following example, the symbol *identifier* has an inherited attribute of domain ENV, and synthesized attributes of domains NAME and TYPE:

```
attribute
    identifier<ENV.NAME,TYPE>;          # variable identifiers
    expression<ENV.TYPE,EXP>;           # expressions
    variable<ENV.TYPE,VAR>;             # variables
    command<ENV.COM>;                   # commands
```

Four symbols are built in, for use only on the right side of rules:

number<.INT>	represents an integer number, a string of digits.
ident<.NAME>	represents an identifier, an alphanumeric string beginning with a letter.
where<BOOL.>	represents the empty string; adds a constraint that the boolean condition is **true**.
uniqueName<.NAME>	represents the empty string; each instance in the parse tree generates a distinct name; useful for generating arbitrary labels.

3.3.4. Rules

The rules describe the syntax and semantics of a programming language. The rule part begins by naming the start symbol of the syntax:

rule *start-symbol*

Terminal symbols, either alphanumeric reserved words or combinations of special characters, are enclosed in quotes:

"begin" " + " ": = "

Many of the example rules in this thesis use arrows ↑ and ↓ to indicate whether an attribute is synthesized or inherited, but the compiler generator expects rules in which commas separate the attributes. (The **attribute** declarations specify the types of attributes.) The assignment rule becomes:

```
command<env, λs. var(exp s)s> =
    variable<env, type, var>
    ": = "
    expression<env, type, exp> ;
```

There is no way to specify the lexical conventions of a language; the current implementation assumes the following:

- the braces { and } enclose comments in programs
- spaces, newlines, and comments separate numbers and identifiers
- keywords are reserved
- there are no string constants

3.3.5. Resolution Part

The resolution part assigns binding powers and associativities to terminal symbols, for eliminating syntactic ambiguities [1]. It can resolve the dangling-else problem and specify operator precedence. Operators can be left-, right-, or non-associative; each **left**, **right**, or **nonassoc** declaration defines a group of operators with the same binding power. Each declaration specifies a higher binding power than the next declaration. The terminals not mentioned in the resolution part have the lowest binding power.

Examples: The declaration

```
left "*" "/"
```

gives * and / equal priorities and makes them associate to the left, so that x*y/z means (x*y)/z. The following resolution part reflects Pascal's operator precedences:

```
resolution
  nonassoc   "not";                    # most binding
  left       "*" "/" "and";
  left       "+" "-" "or";
  nonassoc   "<" ">" "=";              # least binding
```

Chapter 4
Static Semantics

To clearly describe a programming language using a semantic grammar, you must dissect the language into concepts more carefully than you would when writing a compiler. A compiler need only work properly, but a semantic grammar should be revealing as well as correct. If you intend to process the grammar through the compiler generator and run programs using it, then you must also consider efficiency and other limitations of the compiler generator.

Static properties are those evident from the program text without execution of the program. They are also called context-sensitive or compile-time properties; the domains and expressions that describe them resemble the data structures and algorithms used in a compiler. Types and symbol tables are typical examples. Watt and Madsen investigated these concepts while writing an attribute grammar for the static semantics of Pascal [63, 64].

A major strength of the compiler generator is its ability to process static semantics and report errors in user programs. Modern languages, such as Pascal, CLU, and Ada, require complex static analysis. Yet the literature on denotational semantics rarely mentions static semantics.

4.1. Syntax

The context-free syntax of a language defines its operators and delimeters, and how phrases nest inside one another. It is the framework upon which a semantic grammar is built. For each rule, the semantic grammar shows the semantic relationships among the syntactic constituents.

If a grammar is to be processed through the compiler generator, its syntax must be unambiguous — in fact LALR(1). Unfortunately, the most natural description of a language is often ambiguous [1]. Appendix A shows how to eliminate simple ambiguities, such as operator precedence and dangling else, using resolution declarations.

Watt's attribute grammar for Pascal performs type-checking using a set of rules that generate the empty string if all of their semantic conditions are satisfied, and generate no strings otherwise — the *blind alley* technique used to define Algol 68 [62]. Syntactically, these rules are highly ambiguous, generating the empty string in many different ways. In a semantic grammar, you can define functions to perform semantic checks without using blind alley rules.

Other ambiguities are harder to eliminate. In Pascal, an identifier may be parsed as an expression through two different nonterminals, constant or variable:

```
expression = variable | constant | . . .
variable = identifier | . . .
constant = identifier | number
```

An actual parameter can be a variable either directly, or through an expression:

```
parameter = variable | expression
expression = variable | . . .
```

In Fortran, the expression A(I,J) may be either an array reference or a function call, and a statement beginning with A(I,J) = may be either an assignment statement or the definition of a statement function.

Resolve such ambiguities as you would for any parser generator. A solution to the first example is letting a constant identifier be parsed as a variable, extending the semantics of variables to handle constants:

```
expression = variable | number | . . .
variable = identifier | . . .
```

A solution to the second example is merging expression and variable into a single nonterminal expVar, which carries a flag indicating whether an expression is in fact a variable, and if so the semantics of that variable.

```
parameter = expVar
expVar = variable | . . .
```

While the solutions of the previous examples are messy and annoying, the Fortran example is intractable. The right hand side of an assignment statement must be treated differently from that of a statement function definition; subscripts must be treated differently from parameters. Rules handling every possibility would be unreadable. My Fortran grammar omits statement functions, and uses a non-standard syntax for function calls. The best solution is representing expressions as abstract syntax trees, and defining auxiliary functions to interpret the trees in the various ways.

Hand-written compilers resolve syntactic ambiguities using the symbol table: to see if an identifier is a variable or constant, if a parameter is expected to be a variable or expression, if a name is dimensioned or not. Can semantics resolve ambiguities automatically? Watt [65] and others [24, 32, 63] suggest methods of letting the attributes control the parsing of an ambiguous syntax; Milton, Kirchhoff, and Rowland [38] use attributes to resolve LL(1) parser conflicts, producing one-pass compilers. Unfortunately, many languages cannot easily be compiled in a single pass. The problem is a messy case-analysis; both the ambiguities and the resolving attributes may only have certain forms.

An unambiguous syntax provides a solid foundation for a semantic grammar. One benefit is that my generated Pascal compiler recovers from some semantic errors more robustly than the regular Pascal compiler does. The regular compiler depends upon semantic information for parsing; if that information is denied due to an error, an avalanche of unwarranted error messages can result. Most ambiguities can be avoided by careful language design, making programs more readable to humans as well as machines.

4.2. Static Environments

Environments are the formal equivalent of symbol tables in a compiler. The issues involved in implementing a symbol table [1] are twofold: logical considerations of what to include in the table, and practical considerations of fast search and storage management. In a semantic grammar, the logical considerations are paramount.

4.2.1. Modes

An environment records the meanings of the identifiers in a program. Depending on the language, identifiers may denote variables, procedures, types, constants, labels, COMMON blocks, etc. After defining a domain MODE to represent these possibilities,* an environment will be a mapping from names to modes:

*Watt and Madsen use "mode" for the meaning of an identifier; the standard term is "denotable value."

```
TYPE = ...;

PARAMLIST = [nilP + consP[TYPE × PARAMLIST]];

MODE = [ variableMode[TYPE] +
         typeMode[TYPE] +
         labelMode +
         procedureMode[PARAMLIST]];

ENV = NAME → MODE;
```

4.2.2. Identifiers

The compiler generator provides the lexical symbol ident<↑name> for scanning names; it is useful to define a nonterminal symbol identifier<↓env ↑mode> for looking up names in environments. Identifier inherits an environment and synthesizes the mode of the name that it parses.

```
identifier<↓env ↑mode> = ident<↑name> with mode = env(name)
```

An equivalent, more concise version is:

```
identifier<↓env ↑env(name)> = ident<↑name>
```

Identifiers may denote types. Using the above definition of MODE, the following rule checks that an identifier denotes a type and returns that type. There might be other rules to define array and record types, as discussed in the next section.

```
dataType<↓env ↑type> = identifier<↓env ↑typeMode[type]>
```

4.2.3. Declarations

Declarations create and update environments. Consider variable and type declarations in a simple language:

```
declaration = "variable" ident ":" type |
              "type" ident "=" type
```

In a semantic grammar, declarations inherit an environment and synthesize a new one that contains the effect of the declarations.

```
declaration<↓env ↑[name→variableMode[type]] env> =
        "variable" ident<↑name>
        ":" dataType<↓env ↑type> ;

declaration<↓env ↑[name→typeMode[type]] env> =
        "type" ident<↑name>
        "=" dataType<↓env ↑type> ;
```

Once you have declarations, concatenating them into declaration lists is trivial:

```
declarationList<↓env ↑env> = ;

declarationList<↓env ↑env2> =
        declarationList<↓env ↑env1>
        declaration<↓env1 ↑env2> ;
```

In a language with block structure, there is no need to delete local declarations from the environment after leaving a block. Instead, throw the local environment away and resume using the previous global environment. Since there are no side-effects in semantic formulas, the global environment will be the same as before it was used as the basis for another environment.

The obvious representation of an empty environment is the undefined element ⊥. But if you make "undefined" a separate tag of MODE, and use λname.undefined as the initial environment, then you will be able to check whether or not a name has been declared:

env(name) is undefined

for preventing multiple declarations. Remember that block-structured languages allow a local declaration to override a global declaration, but not another local declaration.

4.2.4. Procedures

Consider procedures with only value parameters. The domain PARAMLIST, defined above as a list of types, represents formal parameter lists.

```
procedure = "procedure" ident formalParamList "begin" statement "end"

formalParamList = empty | "(" formalParams ")"

formalParams = parameter | parameter ";" formalParams

parameter = ident ":" type
```

This syntax illustrates some weak spots of semantic grammars. Most syntax notations include shorthand for indicating alternative, optional, or repetitive elements. Semantic grammars do not, because it is difficult to incorporate semantics. (Mosses [40] provides a shorthand for repetition, using the parse tree as the interface between syntax and semantics.)

It is easier to handle zero-or-more repetitions of a construct than one-or-more repetitions, because the empty list is trivial. You must define formalParams using a recursive rule that defines lists of one-or-more parameters. The nonterminal *parameter* is needed to avoid duplicating the string *ident ":" type*; there may be duplication of semantics as well.

Semantic rules for formal parameters must not only build the parameter list, but also enter the parameters into the local environment of the procedure. I omit the details, which are straightforward.

Consider the matching of actual with formal parameters in procedure calls:

```
statement = identifier actualParamList

actualParamList = empty | "(" actualParams ")"

actualParams = expression | expression "," actualParams
```

The following semantic rules pass the formal parameter list as an inherited attribute to the actual parameter list, which checks that each actual parameter has the same type as the corresponding formal parameter. The formal parameter list must end at the same time as the actual parameter list, implying that the numbers of formal and actual parameters are equal. Although the compiler generator allows left recursion, the recursive structure of PARAMLIST dictates that actualParams be right recursive.

```
statement<↓env> =
        identifier<↓env  ↑procedureMode[paramList]>
        actualParamList<↓env ↓paramList> ;

actualParamList<↓env ↓nilP> = ;

actualParamList<↓env ↓paramList> =
        "("  actualParams<↓env ↓paramList> ")" ;

actualParams<↓env  ↓consP[type,nilP]> =
        expression<↓env ↑type> ;

actualParams<↓env  ↓consP[type,paramList]> =
        expression<↓env ↑type> ","
        actualParams<↓env ↓paramList> ;
```

Watt and Madsen [63] handle Pascal procedures without requiring declaration before use. Each procedure inherits two environments. The first contains only the declarations before the procedure, and is only used for processing the formal parameter list. The second contains the parameter lists of all the local procedures, and is used for processing the procedure bodies. Similarly, Watt [64] uses two environments to handle Pascal's forward references in definitions of pointer types — at the cost of two additional attributes on every type. These are examples of right-to-left information flow in a grammar.

The run-time semantics of labels and recursive procedures can not be stored in static environments. They require *dynamic* environments, discussed in the next chapter, because they involve fixedpoints of declarations. The compiler generator cannot handle circular attribute grammars, which are the only way to represent the fixedpoint of a semantic relationship expressed using inherited and synthesized attributes. However, if the language prohibits recursion, then it is simple to process procedure declarations one by one, inserting the run-time semantics of each into the environment. The Pascal grammar uses a dynamic environment for procedures; the Fortran grammar handles procedures in the static environment, using a dynamic environment for labels.

4.3. Data Types

Many languages associate a fixed type with every variable, specifying its values, structure, and operations. Types include scalars, such as integers and booleans; arrays; and other data structures such as records, pointers, sets, and files. Fortran and Algol provide several basic types that may be used as scalars or as arrays of specified dimensions. A type is then a pair (scalar,dimensions) where scalar determines the element type and dimensions is a possibly empty list of integers.

```
TYPE = SCALAR X DIMENSIONS;
SCALAR = [integer + real + logical];
DIMENSIONS = [nilD + consD[INT X DIMENSIONS]];
```

Modern languages provide more data structures and allow them to be nested arbitrarily. Instead of multi-dimensional arrays, there are arrays of arrays. There are records with named fields, each of which has a type. The language syntax for types might be the following:

```
dataType = "integer"   |
           "boolean"   |
           identifier  |
           "array" "[" number ".." number "]" "of" dataType  |
           "record" fieldList "end"

fieldList = ident ":" dataType ";" fieldList  |
            empty
```

The following recursive domain holds all the necessary information about types. It is nothing more than abstract syntax trees.

```
TYPE = [integerType +
        booleanType +
        arrayType[INT X INT X TYPE] +
        recordType[FIELDS]];

FIELDS = [nilF + consF[NAME X TYPE X FIELDS]];
```

The examples in this section use the above definition of Pascal types. To keep things simple, they ignore run-time semantics.

4.3.1. Type Definitions

Given a domain representing types, you must write semantic rules that generate types from type definitions. Since TYPEs are abstract syntax trees, this is easy. (The pre-declared lexical symbols ident<↑name> and number<↑int> scan identifiers and integer constants, synthesizing their values.)

```
dataType<↓env ↑integerType> = "integer";

dataType<↓env ↑booleanType> = "boolean";

dataType<↓env ↑type> =  identifier<↓env ↑typeMode[type]>;

dataType<↓env ↑arrayType[int1,int2,type]> =
              "array" "[" number<↑int1> ".." number<↑int2> "]"
              "of" dataType<↓env ↑type>;

dataType<↓env ↑recordType[fields]> =
              "record" fieldList<↓env ↑fields> "end";

fieldList<↓env ↑consF[name,type,fields]> =
              ident<↑name> ":" dataType<↓env ↑type> ";"
              fieldList<↓env ↑fields>;

fieldList<↓env ↑nilF> = ;
```

4.3.2. Type Checking

Suppose the language has a unary **odd** operator that requires an integer operand and produces a boolean result. The following rule contains an explicit test that the operand's type is **integer**, using a where-clause:

```
expression<↓env ↑booleanType> =
              "odd" expression<↓env ↑type>
              where<type is integerType>;
```

The check can be made implicit, using the properties of defined attributes and constraints discussed in section 3.2.1. Because type is a synthesized attribute on the right side of the rule, it is a defined attribute; replacing it with the tag integerType implicitly specifies the constraint "type is integerType," which the compiler generator will check:

```
expression<↓env ↑booleanType> = "odd" expression<↓env ↑integerType>;
```

General type-checking for the assignment statement can be subtle. If the domain TYPE is simple enough to allow equality testing with **eq**, then the following suffices:

```
statement = variable<↓env ↑type1>  ":="  expression<↓env ↑type2>
            where<type1 eq type2>;
```

Using implicit constraints you can write:

```
statement = variable<↓env ↑type>  ":="  expression<↓env ↑type>;
```

To do more complex type-checking, write a recursive function

compatible: (TYPE × TYPE) → BOOL

that traverses a pair of types as a compiler would. However, some of the tricks compilers use are not available in semantic grammars, such as omitting the traversal of a pair of structures if the pointers to them are equal. When you need to compare pointers, you must simulate them by labelling each type with distinct integers (or identifiers generated by uniqueName). Then you can define type-checking by name compatibility, rather than structural compatibility [66].

4.3.3. Types of Variables

Pascal provides access to elements of arrays and records, using a general notion of *variable* with a syntax representing simple variables, subscripted variables, and field access:

```
variable = identifier  |
           variable "[" expression "]"  |
           variable "." ident
```

The semantic rule for simple variables returns the type of the variable identifier:

```
variable<↓env ↑type> = identifier<↓env ↑variableMode[type]>
```

The rule for arrays checks that the variable is an array, gets its element type, and checks that the subscript expression has type **integer**. Constraints do the checks automatically; arrayType[int1,int2,type] is an example of a complex defined attribute. Its subexpressions (int1,int2,type) and (int2,type) are also defined attributes. Ultimately, the attribute variables int1, int2, and type are all defined.

```
variable<↓env ↑type> =
            variable<↓env ↑arrayType[int1,int2,type]>
            "["  expression<↓env ↑integerType>  "]";
```

The rule for records requires a function

lookup : (NAME X FIELDS) → TYPE

that looks up a name in a field list and returns the corresponding type. A definition of lookup is

```
lookup =
  λ(name0,fields0).
    case fields0 of
      nilF. ⊥,                                  # No such field in record
      consF[name,type,fields].
        if name eq name0 then type              # Found the field
        else lookup(name0,fields) fi            # Continue search
    esac;
```

Now the semantic rule for record access can be written; it includes an implicit check that the variable's type is indeed a record.

```
variable<↓env ↑lookup(name,fields)> =
              variable<↓env ↑recordType[fields]>  "."  ident<↑name> ;
```

Chapter 5
Dynamic Semantics

Dynamic properties concern the execution of programs; many authors have studied their denotational semantics. Given a semantic grammar for the static semantics of a language, it is simple to augment it with dynamic semantics.

The denotational approach uses a single framework, the *standard semantics*, for all languages. As Gordon [11] explains, "If we describe languages using fixed standard techniques then comparisons between languages are made easier. The disadvantage is that for any particular language the 'fit' of the technique may not be perfect." The disadvantage is serious. The standard semantics is a poor fit for most languages because it is too general, simulating the Von Neumann computer to a level of detail that resembles assembly language. Denotational definitions, instead of being revealing, have a widespread reputation for being obscure.

The first step of writing a revealing language definition is adopting a suitable framework for the language, not necessarily the standard semantics. A reader can understand the basic concepts of a language by looking at just the domains representing states and commands.

5.1. Program States

A computer executes programs by repeatedly updating its store of machine words. The high-level analog of the store is the *program state*, which contains the values of all the variables in a program. The state is updated by assigning a new value to a variable. The usual denotational representation of the state mimics the hardware implementation, using a mapping from locations to a class of "storable values" that can fit into machine words. This defeats the purpose of formal definition; it obscures the organization of the state, which varies from language to languge.

5.1.1. Fortran

Fortran [3] specifies storage layout in complete detail, including the size of each data element and the address calculation used in array subscripting. Its EQUIVALENCE statement allows the programmer to overlay variables of various types and dimensions. But even the Fortran state, low-level as it is, has more structure than the machine store. Fortran partitions the state into independent, named regions, called COMMON blocks, each of which is an addressable array of values:

```
STATE = NAME → BLOCK;
BLOCK = INT → VALUE;
```

The domain VALUE, representing Fortran data values, remains to be defined. Only scalar values need be considered, because Fortran arrays are collections of scalar values stored in consecutive locations. Scalars include integer, real, character, and logical values. The compiler generator does not yet support reals or characters, leaving:

```
VALUE = [integerVal[INT] + logicalVal[BOOL]];
```

5.1.2. Pascal

Pascal's arrays and other data structures are defined without mention of memory locations. Pointer types are restricted to an invisible storage area, the *heap*, separate from the program variables. Programs manipulate variables, not locations. Pascal deserves a formal definition on the same high level as its defining report. Donahue [9] makes a similar criticism of locations.

When two variables share the same storage, an assignment to one changes the other, a situation called *aliasing*. In Pascal, aliasing can only occur in procedures that have variable parameters. Variable parameters are intended to pass results back to the calling program, and are usually implemented by passing a reference to the actual variable (call-by-reference). Aliasing occurs because an assignment to the formal parameter simultaneously alters the actual parameter, via the reference. The Report [19] seems to require call-by-reference: "In the case of a *variable parameter*, the actual parameter must be a variable, and the corresponding formal parameter represents this actual variable during the entire execution of the procedure."

A different implementation that avoids aliasing is call-by-value-result, where the final value of the formal parameter is copied into the actual parameter after the procedure terminates. Call-

by-reference is sometimes more efficient, but may be regarded as an optimization — that occasionally produces incorrect results. I believe that Pascal's designer never intended to overspecify the implementation of variable parameters;* my semantics represents call-by-value-result.

With aliasing eliminated, the state can be defined without mentioning locations:

STATE = NAME → VALUE

Unlike the Fortran state, the Pascal state can not represent arrays as scalars in consecutive locations. Instead, the domain VALUE must represent arrays. The high-level notion of an array is a function from subscripts to elements. Define VALUE recursively to be either an integer, a boolean, or an array of values:

VALUE = [intVal[INT] + boolVal[BOOL] + arrayVal[INT → VALUE]];

Most of the examples in this chapter use these definitions of STATE and VALUE. The rules abbreviate "state" as "s." Most omit static semantics and injections/projections involving VALUE.

5.1.3. Block Structure

Block-structured languages pose the problem that several variables may have the same name. In the following example, the procedure bothx can see two variables named x: its local x, and the global x, obtained via the function getx:

```
var x;
    function getx;  getx := x;
    procedure bothx;  var x; . . . x . . . getx . . . ;
```

You can still use STATE = NAME→VALUE, but with semantic rules that rename every variable in the program to eliminate duplicates. Generate the new name using *uniqueName*, and store it in the environment with the variable's type. The example variable declaration from section 4.2.3 becomes

*If Pascal implementors had the freedom to pass small arguments by value-result, then they could eliminate the exasperating restriction on passing components of packed structures as variable parameters. The restriction reflects the impossibility of obtaining a reference to a partial word.

```
declaration<↓env  ↑[name1→variableMode[type,name2]] env> =
                "variable"  ident<↑name1>  uniqueName<↑name2>
                ":"  type<↓env·↑type> ;
```

Another way of handling block structure is to partition the state into lexical levels, so that x at level 1 is a different variable from x at level 2. This recalls the "displays" used in the implementation of block-structured languages [1]. Partitioning the state makes it easier to describe recursive procedures, which must save the initial values of their variables and restore them upon returning. With a partitioned state, procedures can save and restore their entire lexical level as a unit, without dealing with each variable individually.

```
STATE = LEVEL → NAME → VALUE;
LEVEL = INT;
```

5.1.4. Extra Components

The state must hold the entire state of the computation, including input and output files. The usual representation of files attaches extra components to the state:

```
STATE = STORE × INPUT × OUTPUT;
STORE = . . .;                    # program variables
INPUT = . . .;                    # remaining input to be read
OUTPUT = . . .;                   # output that has been written
```

This approach is sensible unless you are writing the grammar for the compiler generator. During execution, programs reference the state frequently; if the state is complex, the generator's stack machine will waste a great deal of time detaching and attaching the various components. To achieve fast running times, you must define the state as simply as possible, even if this entails a slight distortion of its structure.

Extra components are not needed to model Pascal input/output, where files are ordinary variables. The Pascal idea can be used in other languages: extend the domain VALUE with a representation of a file, and keep input and output files in special variables, such as "$input" and "$output." Since identifiers cannot have the character $, user programs cannot reference these variables, but the semantic formulas for input/output can.

The same principle applies to the Pascal heap, the storage area for pointer variables.

Programs will run faster if the heap is not a separate component of the state. My Pascal grammar represents pointers as integers, stores the heap in a global array variable named "$heap," and stores the heap size in the integer variable "$heaplimit." Pascal pointers are not the "locations" of denotational semantics — they can denote neither local variables, nor components of heap variables, but only entire heap variables.

5.2. Commands and Expressions

There are two frameworks for dynamic semantics. A *direct* semantics is simple and clear, but is too weak to handle most programming languages. A *continuation* semantics is complex and low-level, but is powerful enough to describe any flowchart program.

5.2.1. Dynamic Environments

Most authors use a single environment, denoted ρ. Tennent [61] divides his Pascal semantics into static and dynamic parts, each with its own environment; Scheifler [55] does the same for CLU. In a semantic grammar, the static environment is an inherited attribute of most symbols, and holds all of the compile-time information about a program; the dynamic environment is a bound variable of most semantic functions, and holds the run-time semantics of labels and recursive procedures. The definition of the dynamic environment is closely tied to that of commands and expressions.

5.2.2. Direct Semantics

Consider a simple language without jumps or expression side-effects. Executing a command (statement) changes the state, by assigning new values to variables. Evaluating an expression produces a value, but leaves the state unchanged. The value may depend on the state, because expressions may contain variables. The domains for a direct semantics are

```
COM = STATE → STATE;        # semantics of commands
EXP = STATE → VALUE;        # semantics of expressions
```

Expressions directly use the values of their subexpressions, as the rules for constants, negation, and addition show:

```
expression<λs.int> = number<int>;

expression<λs.not exp(s)> = "not" expression<exp>;

expression<λs.exp1(s) + exp2(s)> =
    expression<exp1> "+" expression<exp2>;
```

Control flows directly from one command to the next, as the rules for if and compound commands show:

```
command<λs.if exp(s) then com(s) else s fi> =
    "if" expression<exp> "then" command<com>;

command<λs.com2(com1 s)> =
    command<com1> ";" command<com2>;
```

The semantic function of any iterative command must be recursive. The while command satisfies the equivalence

while exp do com = if exp then (com; while exp do com)

leading to a semantic rule containing a fixedpoint:

```
command<fix λwh:COM. λs.if exp(s) then wh(com s) else s fi> =
    "while" expression<exp> "do" command<com>;
```

The next section discusses the direct semantics of variables and the assignment command. Gordon [11] shows how to define a direct semantics for side-effects, using the definition EXP = STATE→(VALUE × STATE). The rules become messy.

The Pascal grammar includes a dynamic environment to hold the semantics of procedures; its domains are:

```
DE = NAME → STATE → STATE;        # dynamic environments
EXP = DE → STATE → VALUE;         # expressions
COM = DE → STATE → STATE;         # commands
```

You should use direct semantics whenever possible. Compared to a continuation semantics, it is simpler, more readable, and allows faster compilation: fewer beta-reductions during simplification.

5.2.3. Continuation Semantics

The idea of continuations dates back to McCarthy [36], who suggested a method of converting a flowchart program into a set of recursive functions. Define a function for each box of the flowchart, such that each function calls its successor in the flowchart. The functions are called continuations; each denotes the final answer the flowchart would produce if started at the corresponding box.* The result depends only on the initial state of the program variables.

Leaving the domain of answers unspecified, the domain of command continuations is

```
ANSWERS = . . .;                    # final outputs of programs
C = STATE → ANSWERS;                # command continuations
```

Thinking of continuations as functions can be confusing; think of them as program points, or labels, that can be jumped to.

To handle expressions, generalize the notion of flowchart to include boxes that can compute values and pass them to other boxes. Normally an expression computes a value and passes it to a box that expects a value. However, an expression may branch to a different box without computing a value — if an exception occurs, or if a function executes a non-local goto command.

Now the flowchart contains boxes that expect a value before they can execute. If we start the flowchart program at such a box, the final answer will depend on the value we provide, as well as the state. The label of the box is called an expression continuation, because it is the destination of an expression's value.

```
EC = VALUE → STATE → ANSWERS;
```

The semantic function of an expression requires, as an argument, the destination to jump to after evaluating the expression; this argument is an expression continuation. The semantic function yields a continuation describing the expression's effect: produce a value, then pass it to the destination label. A command does not produce a value, so its argument (destination label) is a command continuation.

---

*If the flowchart program goes into an infinite loop instead of terminating, its final answer is $\perp$.

```
EXP = EC → C;                    # semantics of expressions
COM = C → C;                     # semantics of commands
```

Now I repeat the examples I gave for direct semantics.

A constant expression passes the constant's value immediately to the destination. The not operator evaluates an expression and passes the negated value to the destination. The plus operator evaluates two expressions, one after the other, and passes their sum to the destination.

```
expression<λec.ec(int)> = number<int>;

expression<λec.exp(λvalue.ec(not value))> =
    "not" expression<exp>;

expression<λec.exp1(λvalue1.exp2(λvalue2.ec(value1 + value2)))> =
    expression<exp1> "+" expression<exp2>;
```

The precise interactions of the lambda-abstractions and arguments are far too complex to unwind in your head. The key to understanding such formulas is to read them imperatively, as instructions. Read the last as "evaluate exp1; call its result value1; evaluate exp2; call its result value2; jump to ec with the sum (value1 + value2)." It is instructive to work out the semantics of a small expression [11, 58].

Commands are easier to understand because they are imperative anyway. Read the if command's semantics as "evaluate exp; if the value is true then execute com and jump to c; else jump to c."

```
command<λc.exp(λvalue.if value then com(c) else c fi)> =
    "if" expression<exp> "then" command<com>;

command<λc.com1(com2 c)> =
    command<com1> ";" command<com2>;
```

As in direct semantics, the while command involves a fixedpoint. Read it as "evaluate the expression; if the value is true then execute com and jump back to the beginning; else jump to c."

```
command<λc.fix λc0.exp(λvalue. if value then com(c0) else c fi)> =
    "while" expression<exp> "do" command<com> ;
```

Continuations can also describe escape commands that jump out of labelled blocks.

```
command = label "begin" command "end" |
          "escape" label

label = ident
```

Define a dynamic environment to hold a continuation for every label, and redefine commands to depend on the environment.

```
DE = NAME → C;                    # dynamic environments
COM = DE → C → C;                 # commands
```

Change the semantic rules to handle environments. Commands that do not need the environment simply pass it along:

```
command<λde c.com1 de(com2 de c)> =
    command<com1> ";" command<com2> ;
```

A labelled block updates the environment; it maps the label to the block's final continuation. The escape command looks up the label in the environment, retrieves the continuation, and jumps to it.

```
command<λde c.com ([name→c]de) c> =
    ident<name> "begin" command<com> "end" ;

command<λde c.de(name)> =
    "escape" ident<name> ;
```

Gordon [11] explains how to describe goto commands and procedure calls.

5.3. Semantics of Variables

In the standard denotational semantics of a programming language, a memory location is considered to be a legitimate data value, and a variable is any expression that yields a memory location when evaluated. The assignment command has the form:

```
exp1 := exp2
```

The semantics signals an error if the value of exp1 is not a location, and dereferences the value of exp2 if it is a location. This smacks of address calculation, and postpones until run-time the syntactic check that exp1 is a variable.

The Pascal grammar describes a variable (section 4.3.3) using two semantic functions: *exp* obtains its value, and *var* assigns it a new value. Given a value and a state, a var produces a new state in which the variable has the new value. In a direct semantics:

```
STATE = NAME → VALUE;
EXP = STATE → VALUE;
VAR = VALUE → STATE → STATE;
```

The exp and var of a simple variable are straightforward:

```
variable<λs.s(name), λvalue s.[name→value]s> = ident<name> ;
```

The exp and var of an array access treat the array like a function, and the subscript like its argument. Exp applies the function to the argument; var updates the function, at the argument, with a new value.

```
variable<λs.exp1(s)(exp2 s),  λvalue s.var1([exp2(s)→value] exp1(s))s> =
    variable<exp1,var1>  "["  expression<exp2>  "]" ;
```

An assignment command evaluates an expression and stores the value in a variable. The direct semantics:

```
command<λs.var(exp s)s> =
    variable<exp1,var>  ": = "  expression<exp>
```

A continuation semantics requires a concrete representation of variable access. This need not be a memory location, but can be a high-level "Variable Access Descriptor," or VAD. Then you can define variable continuations and variables:

```
VAD = [simpleVar[NAME] + subscriptVar[VAD × INT]] ;
VC = VAD → C;                          # variable continuations
VAR = VC → C;                          # variables
```

To complete the continuation semantics, you must define load and store functions using descriptors. I have not tried this on the compiler generator.

5.4. Axiomatic and Operational Semantics

As Madsen [31] suggests, semantic grammars can express semantics in a variety of ways. A grammar specifying both axiomatic and denotational semantics is a "complementary definition" [9].

An axiomatic definition [16, 33] sets up a logical system for proving properties of programs. Properties are written in an assertion language containing logical connectives, quantifiers, arithmetic functions, and other operators. *Backwards predicate transformers* are most amenable to semantic grammars; each command produces a list of *verification conditions* (VCs) and transforms an output assertion into an input assertion. The VCs must be proven as theorems. This proves the correctness of the command relative to its input-output assertions: whenever the input assertion holds before executing the command, the output assertion will hold afterwards.

Every loop in the program must be documented with an invariant assertion; most of the verification conditions are needed to check that the assertions are really invariant. Given a program and its input-output assertions, the method produces the VCs needed to prove the correctness of the program.

To include an axiomatic semantics in a semantic grammar, define the assertion language as a recursive union domain. A simple assertion language of integer expressions and boolean conditions is defined

```
EXP = [ constantE[INT] +
        variableE[NAME] +
        plusE[EXP × EXP] +
        timesE[EXP × EXP]];

COND = [ equalsC[INT × INT] +
         notC[COND] +
         andC[COND × COND] +
         orC[COND × COND]];
```

The semantics of assignment requires a recursive function that substitutes an expression for a variable in an assertion. The rules must translate expressions into the corresponding terms of the assertion language. The command rules must synthesize a list of verification conditions, and synthesize the input assertion, transformed from the (inherited) output assertion. The rule for the compound command, using an *append* function to combine lists of VCs, is

```
command<↓cond ↑cond1 ↑append(vcs1,vcs2)> =
    command<↓cond2 ↑cond1 ↑vcs1> ";"
    command<↓cond ↑cond2 ↑vcs2>;
```

You can write a function to perform local simplifications, such as changing 0+a to a. Appendix E presents the axiomatic definition of a language with if and while commands.

Given such a grammar, the universal translator converts the input program into a list of verification conditions, which it prints as a nest of injection functions. The generated code reconstructs the VCs inside the stack machine.

Similarly, you can specify an operational semantics in a grammar. An operational definition translates a language into instructions for an abstract interpreter, whose behavior describes the semantics. Define the interpreter instructions as a union domain, and write rules that translate programs into lists of instructions. Madsen suggests representing programs as trees, executed by a recursive tree machine. Another representation is intermediate code, such as triples or postfix instructions, for specifying the front end of a production compiler [1].

Chapter 6
The Compiler Generator

The compiler generator consists of three Pascal programs: a grammar analyzer, universal translator, and stack machine.

6.1. Grammar Analyzer

The grammar analyzer is organized like a recursive descent compiler, and performs the following tasks:

- Read a semantic grammar, parsing the domain definitions, expression definitions, and rules.
- Check that the information is consistent.
- Compute LALR(1) parse tables for the syntax part of the grammar.
- Output the language description file, which contains the semantics of each rule and the parse tables.

The language description file contains all the information needed by the universal translator. For each semantic rule, it gives the applied attribute expressions and attribute constraints. It also contains the pseudo attributes, which are generated by with clauses and uniqueName. All expressions are represented in postfix.

The description file contains information that the translator needs to print out formulas and error messages. This includes the names of the domains and union tags, but not the definitions of the domains. Every attribute expression is followed by its location in the rule, for pinpointing semantic errors.

When parsing a rule, the analyzer records all the free variables of applied attributes. These are the *attribute variables* that must be given values by appearing as defined attributes. In a

recursive scan of the defining attribute expressions, the analyzer accumulates constraints and defines the attribute variables. The attribute grammar should not be circular, but there are no other restrictions on how attributes can depend upon each other. The grammar analyzer does not check for potential circularities [27], because that requires exponential time [18]. Instead, the universal translator detects circularities while evaluating attributes.

The analyzer contains an LALR(1) parser generator, based upon Stanford's [14], that processes the syntactic part of the grammar. It checks that the grammar contains no unreachable or useless symbols, computes its LR(0) set of states, and adds LALR(1) lookahead. It resolves shift-reduce conflicts according to the user's resolution part. It generates parse tables, compressed by merging rows whenever possible. Parse tables take up half the space of the language description file; computing them accounts for half the time needed to process a semantic grammar.

6.2. Universal Translator

The universal translator performs several tasks:

- Read a language description file, reconstructing the tables and expressions.
- Read a source program.
- Print a listing of the program's semantic errors.
- Print the semantic function describing the program [44].
- Generate stack machine instructions for the program.

6.2.1. Parsing

The translator's shift-reduce parser builds a directed acyclic graph (DAG) of attribute dependencies during parsing. (A DAG is a tree in which several parent nodes may share the same child node.) Inherited attributes complicate the process. If there were only synthesized attributes, it would be possible to evaluate all of them bottom-up, like constructing a parse tree. This is because the synthesized attributes on the right-hand side of a rule are all defined when the parser reduces by that rule. Inherited attributes, which represent the context of the reduction, may not yet be available. So the parser substitutes dummy nodes for them, and patches the correct value in as soon as it appears.

The following description is adapted from Madsen [31].

A shift-reduce parser uses a stack to record the grammar symbols parsed at a given point. To handle semantics, the stack is augmented with the synthesized and inherited attributes of every symbol. It represents each synthesized attribute as a pointer to a DAG. A symbol's synthesized attributes may depend upon its inherited attributes, which the DAG represents by dummy nodes. The stack represents an inherited attribute as a fixup-list locating its dummy nodes.

The parser reduces by a rule

$$X \rightarrow Y_1 Y_2 \ldots Y_k$$

by popping the right-side symbols and attributes, $Y_1 \ldots Y_k$, and pushing the left-side, X.

The fixup-lists representing left-side inherited attributes are initially empty. They accumulate the locations of dummy nodes during evaluation of the rule's applied attributes: left-side synthesized and right-side inherited. After evaluating a right-side inherited attribute, its fixup-list is scanned to replace its dummy nodes with the correct value.

Each applied attribute is a function

$$f(I_1, \ldots, I_m, S_1, \ldots, S_n)$$

of the rule's defined attributes: left-side Inherited and right-side Synthesized. Evaluation creates a DAG node labelled f, with pointers to the DAGs representing the synthesized attributes, and pointers to dummy nodes representing the inherited attributes. If the applied attribute is simply a copy of a defined attribute, there is usually no need to create a new node.

Rules may contain pseudo attributes, which are created by with clauses and the uniqueName symbol. Pseudo attributes are defined and applied in the same rule. Since other applied attributes may depend on pseudo attributes, the pseudos are evaluated first. Every use of a pseudo attribute refers to the same DAG. This assures that with clauses are only evaluated once, and that every use of a uniqueName attribute gets the same generated name.

The DAG occupies a lot of storage, although no parse tree is constructed. The largest program compiled is a twenty-one page LALR(1) parser generator. Its DAG contains over 15,000 nodes, and swells to over 26,000 during simplification.

6.2.2. Simplification

At first, each DAG node is labelled with a pointer to an attribute function, and its sons are arguments. The simplifier traverses the DAG depth-first, expanding function definitions and applying them to arguments. The expanded function is linked back into the DAG so that shared nodes are only expanded once. Expanded parts of the DAG represent semantic formulas: each node is labelled with an operator, and its sons are the operands.

The DAG contains both semantics and attribute constraints. During expansion, the simplifier checks that the constraints hold and executes all of the compile-time actions in the DAG. The simplified DAG is ready for translation into machine instructions. Example simplifications:

Before	*After*
3-5	-2
left (a,b)	a
tag[a] | tag	a
if true then a else b fi	a
([a→b]f) a	b

An essential but difficult simplification is *beta-reduction*: applying a lambda-expression to its arguments by substituting the arguments for the bound variables. If a bound variable occurs more than once, its argument is replicated. The simplifier performs no beta-reductions that would replicate expressions requiring evaluation at run-time and make the object program less efficient.

Substitution is slow, because it requires copying list structures. The simplifier avoids one copy operation by simplifying during substitution, rather than after substitution in a separate traversal. Whenever possible, the simplifier substitutes for several variables at once to avoid repeated copying. When simplifying **if** x **then** y **else** z **fi**, the simplifier first simplifies x, to see whether it reduces to a constant (**true** or **false**). If so, it need simplify only one of y or z. The case expression uses a similar technique.

Taken together, these improvements cause simplification to resemble symbolic execution of the expression, rather than a sequence of costly macro-expansions. Figure 6-1 is part of the simplified DAG for the Eight Queens program (page 76).

Figure 6-1: Example of a Simplified Semantic Formula

```
\input.
 LET a=
   (LET s=
       LET s0=
          (FIX for.                                                    for k := -7 to 7 do c[k] := true
           \int s.
           IF 7 LT int
           THEN s
           ELSE for
                (int + 1)
                LET s=
                   [1-> ["K"-> INTVAL[int]] s 1] s
                IN LET int=
                      s 1 "K" !INTVAL
                   IN IF (int GE -7) AND (7 GE int)
                      THEN [1->
                            ["C"-> ARRAYVAL[[int-> INTVAL[1]] s 1 "C" ! ARRAYVAL]]
                            s 1]
                           s
                      ELSE BOT
                      FI
           FI)
          -7
          ((FIX for.                                                   for k := 2 to 16 do b[k] := true
            \int s.
            IF 16 LT int
            THEN s
            ELSE for
                 (int + 1)
                 LET s=
                    [1-> ["K"-> INTVAL[int]] s 1] s
                 IN LET int=
                       s 1 "K" !INTVAL
                    IN IF (int GE 2) AND (16 GE int)
                       THEN [1->
                             ["B"-> ARRAYVAL[[int-> INTVAL[1]] s 1 "B" !ARRAYVAL]]
                             s 1]
                            s
                       ELSE BOT
                       FI
           FI)
           2
           ((FIX for.                                                  for k := 1 to 8 do a[k] := true
             \int s. IF 8 LT int THEN s
                     ELSE for(int + 1)(...)
                     FI)
            1
            ([0->
              ["INPUT"-> FILEVAL[LEFT input, RIGHT input, 1, (LEFT input) 1]]
              ["OUTPUT"-> FILEVAL[BOT, 0, 1, BOT]]
              ["$HEAP"-> ARRAYVAL]
              ["$HEAPLIMIT"-> INTVAL[0]]
              BOT]
             BOT)))
       IN LET s=
             ...                                                                Body of procedure Try
             "TRY"                                                                    Try(1,found)
             ([2-> ["I"-> INTVAL[1]] ["Q"-> s0 1 "FOUND"] BOT] s0)
          IN LET s= [2-> s0 2] s IN ([1-> ["FOUND"-> s 2 "Q"] s 1] s)
    IN                                                                          if found then...
       IF s 1 "FOUND" !INTVAL EQ 1 THEN (FIX for. \int s. ...) 1 s ELSE s FI)
   0
   "OUTPUT"
   !FILEVAL
 IN (LEFT a, LEFT RIGHT a)
```

6.2.3. Representation of Bound Variables

If bound variables are represented by identifiers, then substituting an argument for a variable may compute an incorrect result — a free variable of the argument may become bound because of a name conflict. The translator does not use variable names; it numbers bound variables by their depth in the nest of lambda-expressions [6]. Consider the tree structure of an expression; the depth of a variable x is the number of lambdas lying on the path from x up to λx. For instance, the expression

$$\lambda x.f\ x\ (\lambda y.g\ x\ y)$$

has depth numbers

$$\lambda x.f\ x_0\ (\lambda y.g\ x_1\ y_0)$$

When inserting or removing lambdas in front of an expression, the translator must adjust the numbers of the expression's free variables.

Every expression node contains a *free variable index* indicating its deepest variable reference. Indexes are put in incrementally as an expression is built — the index of a node depends only on the indexes of its children. In most cases it is the maximum of the indexes of the children; however, the index of a lambda node is one less than that of its body, because lambda binds the top level free variable.

A closed expression is one with no free variables. The translator can easily identify closed expressions, for they have a free variable index of zero. Many procedures that traverse expressions, such as substitution, perform operations only on the free variables. When they encounter a closed expression, they return immediately, saving a tremendous amount of work (Section 7.6.1).

6.2.4. Error Reporting

The translator only recovers from semantic errors. If it encounters a syntax error, it prints a list of expected symbols and halts. Automatic syntax error recovery is a separate research problem; Graham et al. [13] have made considerable progress.

The simplifier evaluates the DAG depth-first and records all the semantic errors: circularly defined attributes, attributes that equal ⊥, and constraints that are not **true**. The error handler sorts the errors by line number in the source program, reads the program again, and prints the erroneous lines. It names the relevant nonterminal and attribute domain, and

composes a message appropriate for the failed constraint. To prevent one error from triggering many others, it patches the DAG with a dummy value.

Figure 6-2 shows a Pascal program full of errors, and Figure 6-3 shows the error listing from the translator.

Figure 6-2: An Erroneous Pascal Program

```
PROGRAM error;
TYPE
    c = SET OF integer;                 {range too big}
VAR
    v,v: integer;                       {duplicate declaration}
    b: ARRAY [2..16] OF unIdent;        {undeclared identifier}
    c: ARRAY [-7..true] OF boolean;     {type mismatch, duplicate declaration}

    i: integer;    q: boolean;
    rec: RECORD f1,f2: boolean END;
    si: SET OF 0..30;
    sl: SET OF (pascal,lisp,fortran);

BEGIN
sl := ([] + si);                        {set type mismatch}
i[8] := 0;                              {should be array}
i.red := false;                         {should be record}
i↑ := q;                                {should be pointer}
rec.unIdent := 0;                       {no such field}
unIdent(10)                             {undeclared procedure identifier}
END.
```

6.2.5. Code Generation

Since the stack machine is oriented towards execution of lambda-calculus formulas, code generation is straightforward, traversing the simplified DAG depth-first. First the DAG is split into a forest of trees, to prevent a shared tree from being compiled more than once. A shared tree is compiled into a parameter-less subroutine that each of its parents calls.

The code generator treats fixedpoints like recursive function definitions. The fixedpoint's body must be a function or tuple of functions. The code generator creates an entry point for each function, and compiles each use of the fixedpoint's bound variable into a call of the corresponding function.

Although the code generator performs few optimizations, it includes those that Burge [6] recommends. It also emits code to delete a bound variable after its last use, located during the DAG traversal. Some effects of the optimization can be illustrated by an expression transformation. Given the expression

$(\lambda x.A\ B)y$

where A does not use x, it generates code for

Figure 6-3: Sample Listing of Semantic Errors

```
TYPE
    c = SET OF integer;                    {range too big}
                  ↑
Semantic error: SCALARTYPE

VAR
    v,v: integer;                          {duplicate declaration}
      ↑
Semantic error:
Should be UNDEFINED

    b: ARRAY [2..16] OF unIdent;           {undeclared identifier}
                         ↑
Semantic error: IDENTIFIER
Undefined attribute MODE

    c: ARRAY [-7..true] OF boolean;        {type mismatch, duplicate declaration}
    ↑
Semantic error:
Should be UNDEFINED

                    ↑
Semantic error: CONSTANT
Failed check: COMPATIBLE

begin
s1 := ([] + si);                           {set type mismatch}
          ↑
Semantic error: EXPRESSION
Failed check: COMPATIBLE

i[8] := 0;                                 {should be array}
 ↑
Semantic error: COMPONENT
Should be ARRAYTY

i.red := false;                            {should be record}
 ↑
Semantic error: COMPONENT
Should be RECORDTY

i↑ := q;                                   {should be pointer}
  ↑
Semantic error:
Undefined attribute

rec.unIdent := 0;                          {no such field}
         ↑
Semantic error: COMPONENT
Undefined attribute TYPE

unIdent(10)                                {undeclared procedure identifier}
       ↑
Semantic error: IDENTIFIER
Undefined attribute MODE


11 semantic errors in program
```

A((λx.B)y),

removing A from the scope of x. Deleting dead variables eliminates obsolete references to arrays and permits more efficient array compacting, as described below.

The DAG may contain names from several sources: identifiers in the source program, name constants in the semantic grammmar, and the name-generating nonterminal uniqueName. The code generator replaces every distinct name with a distinct integer, so that no names appear in the object code. If the grammar defines the state to be indexed by names, STATE = NAME→VALUE, it will be as efficient as if the state were indexed by integer locations.

6.2.6. Garbage Collection

The simplifier creates a lot of garbage. The translator collects all of it using *reference counting*: it keeps track of how many pointers reference each node and periodically scans the list of allocated nodes, deleting those no longer referenced. While compiling the LALR(1) parser generator mentioned above, the garbage collector reclaims 133,000 nodes.

References from local, temporary variables are not counted. This frees most of the code of the translator from any garbage collection instructions, and reduces the overhead needed to maintain reference counts. A drawback is that the garbage collector can only be called between simplifier calls. Garbage collection consumes about twenty percent of simplification time.

6.3. Stack Machine

The stack machine has the SECD architecture, due to Landin [30]: a *Stack* of pending operands, an *Environment* of bound variables, a *Control* of instructions, and a *Dump* of return addresses and environments.

Instructions that alter control flow:

halt	terminate computation; print value on top of Stack
return	return from function; restore machine state from Dump
jump pc	jump to location pc
falseJump pc	jump to pc if Stack top is **false**

Instructions that push some value onto the Stack:

loadConst value	the given value
loadPos int	the value of the variable at depth int
loadClosure pc	a function closure (explained below)

Instructions that pop several values f, x, y, . . . from the Stack and push some result computed from them:

plus	the sum x + y
lt	the comparison x<y
not	the negation **not** x
alter	the updated function [x→y]f
apply	the result of the call f(x)
pair	the pair (x,y)
left	the component **left** x
inject tag	the injection tag[x]
project tag	the projection x|tag
is tag	the inspection x **is** tag

The Environment is a stack; when a function call f(x) is executed, x is pushed onto the Environment. There are two instructions, related to Burge's *enter* and *exit* [6], for manipulating the Environment:

pushEnv	push the Stack top onto the Environment
popEnv	remove the top element of the Environment

Figure 6-4 shows part of the object code for the Eight Queens program: the command b[i + j] : = false. It illustrates the substitution of integers for names; i is 32, j is 34, b is 29. I have edited it slightly to make it more readable. The instructions **applyInt** and **alterInt** are immediate forms of **apply** and **alter**.

6.3.1. Closures

The **loadClosure** instruction binds an entry point to the current Environment, creating a functional value that may be stored like any other value. The function may be invoked later using the **apply** instruction. These values, called *function closures*, are an important difference between the SECD machine and ordinary computers. Closures free environments from the stack discipline (where they would be like static links) and allow them to persist indefinitely. Reference counting deletes environments that are no longer used.

Figure 6-4: Sample Stack Machine Code

```
Compute subscript of b
        loadPos     0           s           load value of i: state s, level 2
        applyInt    2
        applyInt    32          I
        loadPos     0           s           load value of j: state s, level 2
        applyInt    2
        applyInt    34          J
        plus                                compute i+j
Check against array bounds
        pushEnv                             store sum for repeated use
        loadPos     0           int         load sum
        loadConst   2                       check lower bound of b
        lt
        not
        falseJump   71                      jump if out of bounds
        loadConst   16                      check upper bound of b...
        loadPos     0           int         ...against sum
        lt
        not
        jump        72
71:
        loadConst   0                       load the value false
72:
        falseJump   69                      jump if subscript out of bounds
Update the array b
        loadPos     0           int         load subscript
        loadConst   0                       load false
        loadPos     1           s           load b: state s, level 1
        applyInt    1
        applyInt    29          B
        alter                               update value of b
        loadPos     1           s
        applyInt    1                       load lexical level 1
        alterInt    29          B           update level 1 to have new b
        loadPos     1           s
        alterInt    1                       update s to have new level 1
        popEnv                              discard subscript
        jump        70
69:
        loadConst   bottom                  signal error
        popEnv
70:
```

Any lambda-abstraction in the final DAG can be represented by a closure at runtime. Optimization eliminates many closures that would be invoked immediately after creation.

6.3.2. Input/Output

Input and output are lists of integers. The machine reads a list $k_1, \ldots, k_n$ from the user's input, and pushes

$$([1 \rightarrow k_1] \ldots [n \rightarrow k_n] \perp,\ n)$$

onto the Stack. It then begins execution of the object program. The machine expects to find a similar data structure on the Stack after execution, and prints the list it denotes.

6.3.3. Run Time Errors

The value ⊥ ("bottom") may represent a run-time error. For instance, a subscript out of bounds may set the state to ⊥, which will propagate to the end of the program, producing a final state ⊥. Not every ⊥ indicates an error; ⊥ is also used to initialize mappings.

If ⊥ is the operand of certain instructions, the machine aborts execution. Other instructions return ⊥ as the result, or treat ⊥ as an ordinary value. The machine prints its current state upon halting. To aid debugging of user programs, every value of ⊥ is flagged with the program location where it was generated.

A separate notation for errors is needed, to allow a consistent policy for aborting programs. Debugging is still difficult; more diagnostic tools are needed.

6.3.4. Array Compacting

A denotational definition considers arrays to be functions mapping subscripts to elements. The subscripted assignment A[i]: = v is represented by the function update [i→v]A, a mapping that takes i to v but otherwise is the same as A. After the loop

```
for i := 1 to 5 do A[0] := i*i
```

the value of A is represented:

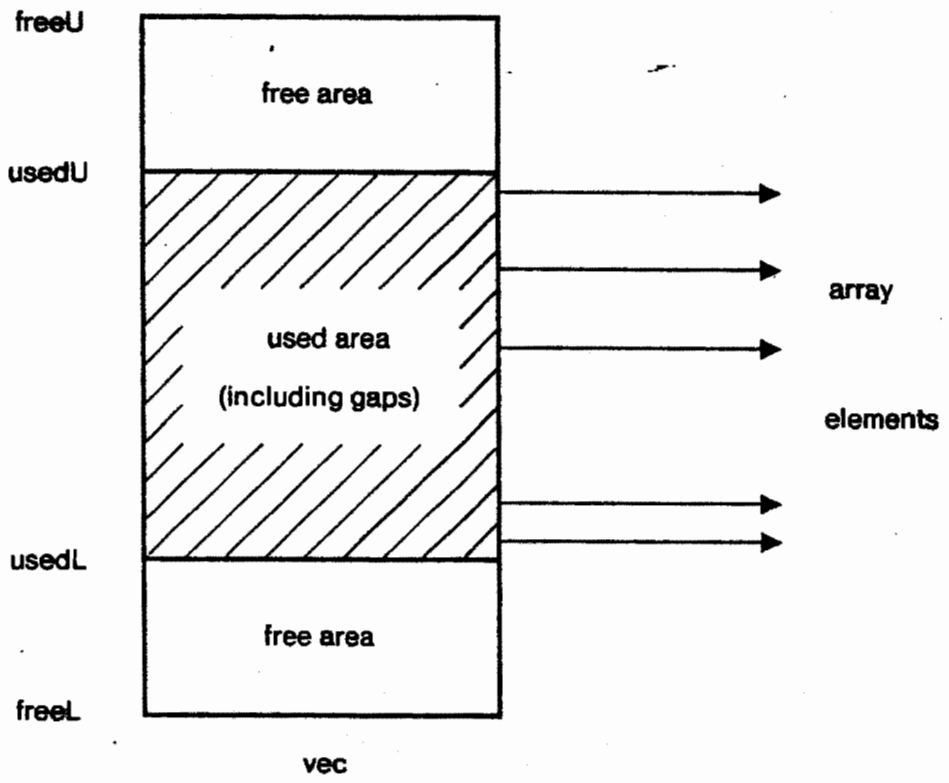
[0→25] [0→16] [0→9] [0→4] [0→1] ⊥

These association lists, or history sequences, waste storage and runtime. States, which are also mappings, suffer the same problem. Efficient execution would be impossible if the machine did not compact association lists into arrays. It compacts arrays indexed by either integers or identifiers, because the translator converts all identifiers into distinct integers. Array compacting is unique to this compiler generator; the literature does not even mention the problem.

All data in the machine are referenced by pointers and may be shared. An association list may be referenced by many pointers, some of them no longer needed but still persisting in the Environment or Dump. The machine must compact lists into arrays without disturbing the value seen by any of the pointers. In effect, the pointers divide a list into segments that must be compacted separately. The machine converts a list of segments into a list of indexable *array blocks*.

An array block is a structure <vec,freeL,freeU,usedL,usedU>, where vec is a vector with bounds freeL and freeU (Figure 6-5). Each vector element vec[i] points to a function value f(i). The vector may contain gaps where vec[i] = nil; the bounds usedL and usedU enclose all of the non-nil data. An array block represents part of a function's association list, not the entire function. A vector gap does not indicate that the function is undefined; its value may appear later in the association list.

Figure 6-5: An Array Block



A segment of an association list has the form $u_1, \ldots, u_n$, where each u is either a simple update [i→x], or an array block representing several simple updates. Each element u_i is referenced only by its predecessor u_{i-1}, except that u_1 may be referenced by many pointers. The machine replaces u_2 through u_n with a single, equivalent array block, and links it into the sequence after u_1. If the segment contains two or more values for the same index, the most recent value is used.

- If the segment contains a block big enough to hold all the data in the segment, then that block is updated to include the other data.

- If no block is big enough to hold all the data, the machine allocates a new one and copies the data into it. So that this expensive step occurs infrequently, the machine includes room for expansion above and below in the new block.

Arrays are the main obstacle to executing denotational formulas. Although the stack machine can execute programs that use arrays extensively, the array compactor is its largest user of time. The algorithm is ad-hoc, and the storage allocator is primitive.*

6.3.5. Tail Recursion

The machine tries to eliminate unnecessary references into association lists, in order to allow the most compacting. The main source of obsolete references is *tail-recursion*, where a function calls another function and then returns. When a function's last action is another function call, the machine does not save the current Environment on the Dump; it will never be needed. The function call is treated like a jump. This optimization is essential because denotational semantics treats every loop and goto command using tail-recursion. Examples in Section 5.2 include both while and escape commands, both direct and continuation semantics.

The most common type of tail recursion is the code sequence **apply;return**, which is easily recognized. Other forms of tail recursion are

apply; jump x; . . .; x:**return**

apply; useless-instruction; **return**

Either peephole optimization or better code generation can convert these to **apply;return**. The compiler generator, redundantly, uses both techniques.

6.3.6. Union Tags

The machine has instructions **inject**, **project**, and **is** for manipulating objects of union domains — inserting, removing, and inspecting tags of union domains. In languages like Pascal, where types are known at compile time, tags provide no useful information at run time. The universal translator has an option to suppress **inject** and **project** instructions, resulting in smaller, faster code. This is allowed only if the code contains no **is** instructions, which require tags at run time.

*There is a fixed set of twenty block sizes, and a common storage area. The machine maintains a free list for each block size. Discarded blocks are not merged into larger ones.

Suppression of tags allows projection errors to go undetected; the expression xtag[x] | ytag denotes ⊥ but evaluates to x. This slackness has not caused problems in my experience with the compiler generator.

Chapter 7
Implementation Issues

A translator is a program, written in some *implementation* language, that translates a *source* language into an *object* language. A compiler generator is a translator whose object language describes another translator. The implementor must chose the source, object, and implementation languages of both translators. For instance: What is the exact form of semantic grammars? In what form are compilers produced? What sort of object code do the compilers generate? This seems like a lot of possibilities, but the algorithms that exist today can only handle a few of them.

The compiler generator evolved over a period of several months. Circumstances favored the choice of certain algorithms, which dictated the rest of the decisions. My goal was that the system be as general as possible, able to process any meaningful input. I also wanted it to be efficient enough to run substantial test programs, so that it could help someone design and evaluate languages.

7.1. Stack Machine

What sort of object code should the compilers generate? A semantic grammar describes how to translate a program into a lambda-calculus formula. Mosses [42] interprets the formula directly, but I translate it into SECD machine instructions. This offers faster execution, is easy to implement, and is a first step towards generating code for an ordinary computer.

Other ideas have since come forth. Neil Jones and David Schmidt [25] suggest translating the formula into a state-transition machine, a finite automaton in which every state has local variables that are updated during transitions from one state to the next. The automaton may be translated into instructions for an ordinary computer. Martin Raskovsky [53] describes how to rewrite a denotational definition into a compiler that generates PDP-10 instructions.

7.1.1. The Concrete SECD Machine

Extending my notation for domains to include finite lists, the abstract definition of the SECD machine may be written [6]:

```
STATE = S X E X C X D;      # complete machine state
VALUE = ...;                # values used in computation
INSTR = ...;                # instruction set

S = list of VALUE;          # Stack of pending operands
E = list of VALUE;          # Environment of bound variables (by depth)
C = list of INSTR;          # Control: the program being executed
D = S X E X C X D;          # Dump: saved state for function return
```

This seems to have little in common with a computer, but it can be made more concrete. The Stack can be implemented as an array with a stack pointer. The Control can be implemented by placing instructions in sequential locations, terminating function bodies with the **return** instruction. A program counter points to the current instruction.

The Dump has the equivalent, non-recursive definition

D = list of (S X E X C).

Since executing a function's body never disturbs the elements initally on the Stack S, there is no need to save S on the dump. This leaves

D = list of (E X C),

where E corresponds to a dynamic link, and C to a return address. The Dump can be implemented with an array and stack pointer.

Functional values, both closures and arrays, cause the major differences between the concrete SECD machine and ordinary computers. Because function closures contain environments, the Environment must be implemented as a linked list. The machine has no notion of updating an array, only of creating a new array based on an existing one. The machine's array compactor attempts to implement this efficiently, updating an array if it is not shared.

7.1.2. Computing Fixedpoints

Burge presents two methods for computing fixedpoints, in a classic trade-off between generality and efficiency. The general method performs a tortuous simulation of the fixedpoint combinator:

fix = λf.(λg.f(g g))(λg.f(g g))

The efficient method only works for functions, compiling them like ordinary recursive functions.

Fixedpoints are mainly used to represent the semantics of while and goto statements; these only define functions. Therefore I use the efficient method, and have not felt limited by its lack of generality. The machine treats tail-recursive calls like jumps to gain even more efficiency.

7.2. Simplification

Mechanically translating a program according to a denotational definition produces a formula that can be greatly simplified, especially if the semantics use continuations. Gordon and Stoy [11, 58] both give sample translations of tiny program fragments, involving dozens of steps.

7.2.1. Simplifying Fixedpoints

It is not obvious how best to represent fixedpoints. Sethi [56] uses circular expressions: a list structure containing cycles. This allows the standard beta-reduction mechanism to simplify fixedpoints automatically, which my simplifier does not. For instance, Sethi's simplifier can resolve label definitions at compile time. Our terminology reflects the difference: I call label environments "dynamic," but Sethi calls them "static."

I represent fixedpoints with an explicit **fix** operator, and avoid circular expressions. This allows me to traverse expressions depth-first without looping, and convert them into postfix for storage on the language description file. The drawback is that the stack machine must look up labels in the dynamic environment at run-time. It may be possible to use Sethi's techniques in the last stage of code generation, eliminating the environment but introducing cycles only in the generated code.

Because goto statements can describe loops, the semantics of the goto uses a fixedpoint.

Stoy takes a goto-program that simulates an if statement, and transforms its semantics into that of the if statement, eliminating the fixedpoint in the process. I developed a way to do the transformation automatically, by repeated use of the identities:

$(x_1, \ldots, x_n)$	$= (x_1, (x_2, \ldots, x_n))$
fix λx.y	= y if x is not free in y
left (**fix** λ(x,y). (u[x,y], v[x,y]))	= **fix** λx. u[x, **fix** λy.v[x,y]]
right (**fix** λ(x,y). (u[x,y], v[x,y]))	= **fix** λy. v[**fix** λx.u[x,y], y]

If a goto-program could be unravelled into if and while statements, then the transformation produced a much simpler output. But it sometimes copied statements that were the target of more than one goto statement. So I removed the transformation from the simplifier.

7.2.2. Beta Reduction

Achieving efficient, powerful beta-reduction was the hardest problem I faced when writing the simplifier. Beta-reduction involves substituting an argument for a bound variable, resulting in replication of the argument if the variable occurs more than once. To prevent formulas from exploding exponentially, the simplifier only performs beta-reductions that replicate only "simple" arguments. The key question: what is simple?

I originally decided that only atomic expressions were simple: variables, numbers, etc. But this did not handle structured bound variables:

(λ(int1,int2). int1 + int2) (3,8)

The bound variable, (int1,int2), was referenced twice; the argument, (3,8), was not atomic. Beta-reduction was prohibited even though no *component* of the bound variable was used more than once. I added a **propagate** command to override the simplicity test, but it proved error-prone. I made the simplifier transform the above expression into

((λint1 int2. int1 + int2) 3) 8

This allowed beta-reductions, but the complete process copied the function several times. Instead of relying on an expensive transformation, I generalized "simple" to include any closed expression, detected using the free variable index. This version of simple allows efficient simplification without exponential blow-up; it facilitates the evaluation of static semantics, which mainly involves closed expressions.

7.3. Cleaning Up Semantic Grammars

My original Pascal grammar [45], intended for human readers only, was unsuitable for machine processing. Of its many informal conventions, the worst was that it represented syntactic repetition using ellipses (. . .) in rules and formulas. I spent weeks developing practical recursive rules for the repeatable constructs, and putting the grammar into concrete form. An easy example is rewriting the rule

commandSequence⟨λc.com$_1$(. . . (com$_n$ c) . . .)⟩ =
    command⟨com$_1$⟩ ";" . . . ";" command⟨com$_n$⟩;

as the two rules

commandSequence⟨com⟩ = command⟨com⟩;

commandSequence⟨λc.com1(com2 c)⟩ =
    commandSequence⟨com1⟩ ";" command⟨com2⟩;

The grammar was circular, to allow a procedure to be referenced before its declaration. It processed declarations in the very environment that the declarations produced. Since Madsen [31] had an algorithm for evaluating circular attribute grammars, and had written a circular grammar for continuations, I attempted to extend his grammar to Pascal. I developed rules for gotos and labels, but could not incorporate procedures into Madsen's framework. Procedures required all semantic functions to be abstracted over continuations, as above; Madsen passed continuations as attributes:

commandSequence⟨↓c ↑c1⟩ = command⟨↓c ↑c1⟩;

commandSequence⟨↓c ↑c1⟩ =
    commandSequence⟨↓c2 ↑c1⟩ ";" command⟨↓c ↑c2⟩;

Faced with this failure, I abandoned circular grammars,* and set out to write a completely non-circular one. I had to develop a new treatment of declarations, introducing dynamic environments.

*Circular grammars still merit study. Combining Madsen's parser with Sethi's simplifier would yield a compiler generator for circular grammars. All we need are interesting grammars to try it on. I and others have pointed out that a circular semantic grammar has a well-defined meaning, expressed as a least fixedpoint [31, 35, 45].

7.4. Parsing Attribute Grammars

My last major decision was what algorithm to use for parsing and attribute evaluation. The one-pass case is easy — if every attribute depends only on attributes to its left in a rule, then all the attributes can be evaluated in a top-down parse of the program. Several compiler generators work this way [29, 38].

One-pass compilation is adequate for Pascal, but not for many other languages. Bochmann [5] gives an algorithm that makes several left-to-right passes over a parse tree, decorating it with evaluated attributes. He shows how to pre-process a grammar and determine how many passes to use and what attributes to evaluate in each pass. The order of evaluation is the same for all programs. Pozefsky [48] elaborates this "multi-pass" approach to include right-to-left passes, special purpose passes, analysis to reduce attribute storage requirements, and methods that require no parse tree.

A multi-pass evaluator performs a complex analysis of attribute dependencies. Yet it can only treat a subset of attribute grammars; Bochmann gives an example in which the number of passes depends on the nesting depth of the source program. I use Madsen's evaluator [31], which works on all attribute grammars. It determines the order of evaluation only after parsing the program. The algorithm is fast and easy to implement. It uses a lot of storage, but so does a multi-pass evaluator, unless complex optimizations are included. The compiler generator owes much of its efficiency, simplicity, and generality to Madsen's evaluator.

The basic structure of the compiler generator was already determined before I chose the attribute evaluator. Both evaluators use a table-driven LL(1) or LR(1) parser. Both imply that there be a grammar analyzer, which writes a language description file, read by a universal translator. They differ only in what processing each component should perform.

7.5. Call by Name vs. Call by Value

For efficiency's sake the stack machine occasionally deviates from the theory of denotational semantics; the main flaw concerns functions. The are two kinds of lambda-abstraction, depending on the value produced when the argument is $\perp$. A call-by-value function must return $\perp$, but a call-by-name function need not. Denotational definitions usually adopt call-by-name.

James Donahue has pointed out to me that the stack machine treats functions inconsistently: sometimes call-by-name, sometimes call-by-value. It has two representations of the value $\perp$: the token **bottom**, or any looping computation. If it applies a function to the token **bottom**, the function may ignore its argument and return anything (call-by-name). If the machine loops while computing the function's argument, then the argument, the function, and the entire computation all have the value $\perp$ (call-by-value). The machine aborts whenever it attempts to perform arithmetic on **bottom**, which is also call-by-value.

Donahue tells me that call-by-name is essential for handling infinite objects. Unfortunately, a call-by-name interpreter is difficult to implement and expensive to run. The languages I have studied do not require call-by-name. Plotkin [46] discusses the differences between call-by-name and call-by-value with regard to the SECD machine.

The specialized implementation of fixedpoints can cause the machine to loop. If a functional expression f contains a recursive call, you may have to write it $\lambda x.f(x)$, where x is a fresh variable that does not occur in f. This delays the evaluation of f until it is applied to an argument. Theoretically, there should be no difference between $\lambda x.f(x)$ and f, by the rule of η-conversion [58].

It is risky to abuse the theoretical foundations. This project may not be efficient enough to interest engineers; if it is not true to the theory, then it may not interest theorists either. Since the machine may loop in certain cases, its shortcuts make it less general. It can still execute programs for a wide class of languages. The machine never computes the wrong answer unless the user has suppressed union tags.

7.6. Achieving Efficiency

One way to assess an optimization is to see what happens if it is removed. This section presents the compiler generator's time and space requirements with various optimizations disabled.

To explore the differences between the Pascal and Fortran grammars, I consider a program for each. The Pascal program (Figure 7-1) finds a solution to the Eight Queens problem [68]. The Fortran program (Figure 7-2) produces prime numbers without performing any divisions.*

*This program is due to Dijkstra [7], who presented it in a different form.

```
PROGRAM Eq;   {Computes first solution of Eight Queens, 15863724}
VAR k: integer;  found: boolean;
    a: ARRAY [1..8] OF boolean;
    b: ARRAY [2..16] OF boolean;
    c: ARRAY [-7..7] OF boolean;
    x: ARRAY [1..8] OF integer;


PROCEDURE Try(i: integer; VAR q: boolean);
    VAR j: integer;
    BEGIN
    j := 0;
    REPEAT
        j := j+1;  q := false;
        IF a[j] AND b[i+j] AND c[i-j] THEN BEGIN
            x[i] := j;
            a[j] := false;  b[i+j] := false;  c[i-j] := false;
            IF i<8 THEN BEGIN
                Try(i+1,q);
                IF NOT q THEN BEGIN
                    a[j] := true;  b[i+j] := true;   c[i-j] := true
                    END
                END
            ELSE q := true
            END
    UNTIL q OR (j=8)
    END;

BEGIN
FOR k := 1 TO 8 DO a[k] := true;
FOR k := 2 TO 16 DO b[k] := true;
FOR k := -7 TO 7 DO c[k] := true;
Try(1,found);
IF found THEN
    FOR k := 1 TO 8 DO BEGIN output↑ := x[k];  put(output) END
END.
```

Figure 7-1: The Pascal Test Program (283 tokens)

The program looks strange because I have changed Fortran's lexical conventions to suit the compiler generator.

The statistics in this dissertation are meant to be illustrative, not authoritative. They may be impossible to reproduce, because I am always modifying the compiler generator and grammars.

7.6.1. Statistics on the Universal Translator

The most important optimizations of the universal translator involve its simplifier.

Closed Expressions	The simplifier detects closed expressions — it does not attempt to substitute into an expression that references no free variables.
Delay Arms	The simplifier does not simplify the arms of a conditional or case expression until it has simplified the controlling expression. The final result may only incorporate a single arm.

```
{Print a list of prime numbers}
        INTEGER P, N, J, K, R, ORD, SQUARE
        COMMON SQUARE,ORD,MULT(30),P(1000)
:       P(1) = 2
:       J = 1
:       ORD = 1
:       SQUARE = 4
:       READ (2) INPUT

:       DO 30 K = 2, INPUT
: 400   J = J+2
:       CALL UPSQR(J)
:       IF (ORD LE 2) GO TO 30
:       DO 27 N = 2, ORD-1
: 20    IF (MULT(N) GE J) GO TO 25
:       MULT(N) = MULT(N) + P(N)
:       GO TO 20
: 25    R = J - MULT(N)
:       IF (R) 27,400,27
: 27    CONTINUE
: 30    P(K) = J
:       WRITE (3) (P(I), I = 1, INPUT)
:       STOP
        END



        SUBROUTINE UPSQR(J)
        COMMON SQUARE,ORD,MULT(30),P(1000)
: 3     IF (SQUARE GT J) RETURN
:       MULT(ORD) = SQUARE
:       ORD = ORD+1
:       SQUARE = P(ORD) ** 2
:       GO TO 3
        END
```

Figure 7-2: The Fortran Test Program (234 tokens)

Multiple Substitution — The simplifier performs multiple beta-reductions, such as for $(\lambda x\, y.f)\ u\ v$, using a single copy operation.

Trivial Substitution — The simplifier can reduce $(\lambda x.f)y$ to f without copying f, if y is the variable at depth 0. This is valid because bound variables are distinguished by depth number, rather than by name.

I disabled each of these optimizations, one at a time, and compiled the Pascal and Fortran programs. Tables 7-1 and 7-2 show the time and the number of nodes needed to simplify the semantic DAG. (Each DAG node occupies three 36-bit words on the DecSystem-20.) Since both time and space requirements are approximately linear in the size of the source program, I divided the statistics by the number of lexical tokens in the program.

Closed Expressions is the most important optimization, necessary to compile the Fortran program. Many large structures of static semantics, such as symbol tables, are closed expressions.

Table 7-1: Performance Compiling the Eight Queens Program (Pascal)

Optimization Disabled	*Time per token (milliseconds)*	*Storage per token (DAG nodes)*
none	22	4.8
Closed Expressions	89	8.4
Delay Arms	35	27
Multiple Substitution	22	4.4
Trivial Substitution	22	4.9

Table 7-2: Performance Compiling the Prime Number Program (Fortran)

Optimization Disabled	*Time per token*	*Storage per token*
none	35	6.4
Closed Expressions	(aborted due to memory overflow)	
Delay Arms	44	18
Multiple Substitution	56	7.2
Trivial Substitution	33	6.5

Table 7-3: Performance Executing the Eight Queens Program

Optimization Disabled	*Instructions*	*Time (seconds)*	*Storage (nodes)*
none	972	27	169
Array Compactor	972	20	7,116
Obsolete References	962	32	371
Tag Suppression	1,064	29	411

Table 7-4: Performance Executing the Prime Number Program

Optimization Disabled	*Instructions*	*Time*	*Storage*
none	639	37	105
Array Compactor	639	22	3,440
Obsolete References	609	86	5,364
Tag Suppression	660	37	190

Delay Arms saves a substantial amount of space. It is most helpful for evaluating static semantics, where the controlling expression of a conditional or case expression is always a constant, selecting a single arm. Without it, the simplifier creates large structures that are later thrown away.

Multiple Substitution is helpful for Fortran, but not Pascal. Trivial Substitution is useless. Apparently Multiple makes Trivial obsolete — if *both* are disabled, performance drops considerably. There are probably many other interactions among the optimizations.

Compiling Fortran uses more time and space than compiling Pascal, because of the extra beta-reductions needed to unwind Fortran's continuation semantics.

7.6.2. Statistics on the Stack Machine

The stack machine contains optimizations, and the universal translator performs code optimizations that affect the stack machine. Tables 7-3 and 7-4 present the code size, execution time, and storage required to execute the test programs. (Each storage node occupies five words. Not included are the contents of array blocks, which occupy considerably less than half of the total storage.)

Array Compactor	The array compactor converts association lists into addressable structures.
Obsolete References	To prevent obsolete references from impairing the array compactor, the code generator emits **popEnv** instructions to delete dead variables from the environment. Also, the stack machine does not save the environment for a tail-recursive call.
Tag Suppression	At the user's option, the code generator emits no **inject** or **project** instructions, which create and destroy tags of union domains.

If the Array Compactor is disabled, every array forms an association list that grows monotonically until the entire array is garbage-collected. The Array Compactor is essential for running programs in a realistic amount of storage. It slows down execution significantly, even though it allows faster lookup of array elements. Apparently the most important data are usually near the front of the association lists.

If there are many Obsolete References into an association list, the compactor uses extra time but accomplishes little, especially for the prime number program. The list structures are

heavily interlinked — failing to compact one list leaves obsolete references that impair the compacting of many others. The object code is slightly smaller when the popEnv instructions are deleted, but this hardly compensates for the extra data storage required.

Tag Suppression makes the object code slightly smaller, saves a small amount of run-time, and saves a considerable amount of data storage.

7.6.3. Comparison With Other Compiler Generators

Of the many compiler generators that have been implemented and tested, two are most similar to mine: Mosses's SIS [42], which converts denotational definitions into compilers, and Räihä's HLP [49], which processes complex attribute grammars. In some respects, my system is more efficient than these others.

My compiler generator does not produce a parse tree; it makes a single pass over the program and produces a graph of all its semantic dependencies. SIS traverses a parse tree to compute its semantics. Since complex languages require several semantic functions on nonterminals, it may traverse parts of the tree more than once. HLP evaluates attributes in passes [48], traversing the parse tree as many as five times for some languages. A parse tree occupies a great deal of storage, and repeated traversals waste run time. Some traversals only evaluate a few attributes.

HLP copies large structures instead of referring to them by pointers. To avoid copying, it provides *global attributes*, which may be referenced and updated by any descendants of a node. Since global attributes violate the spirit of attribute grammars, the HLP team did not use them in writing a grammar for the language Euclid. That grammar creates a separate copy of the symbol table for every node of the parse tree. The waste of storage precludes compiling large Euclid programs [50].

Interpretation does not execute object programs efficiently. Interpreting a formula requires traversing it, matching operators with operands [6]; this effort is needlessly repeated for each iteration of a loop. SIS's interpreter is especially slow because it implements call-by-name. My system compiles its output formula into stack machine instructions. Compiling the formula determines, once and for all, the proper order of computation and the matching of operators with operands.

SIS is bootstrapped in terms of itself. When generating a compiler, it uses a definition of its own semantics to process the user's semantics. The resulting compiler is represented as a lambda-calculus formula. It compiles a program by applying the compiler, as a function, to the program's parse tree, and interpreting the result. The resulting object code is also a lambda-calculus formula. SIS executes the code by applying it to its input data. We cannot expect efficiency from this elegant embodiment of the algebraic structure of a compiler generator.

Chapter 8
Conclusions

This work, particularly the generation of a Pascal compiler, has been more successful than I could have imagined. The compiler generator's most serious problem is inefficiency, but there are several ways to improve it.

8.1. Defining Languages

I treat well-known languages, as faithfully as possible, to prove that my work applies to real problems. Pascal embodies the major language concepts and has several formal definitions. My Pascal grammar (Appendix D) covers all static and dynamic semantics except goto statements, real numbers, strings, function side effects, procedures passed as parameters, etc.* Most of the deficiencies stem from my attempt to make the semantics as high-level as possible; it avoids both continuations and machine locations.

The grammar includes all types and statements, recursive procedures, and block structure. I have checked most of it, by running test programs on the compiler generator. It is only twenty-one pages long, including comments: two pages of domains, five of functions, and fourteen of rules. The Pascal Report [19] is thirty-two pages.

Fortran, with its low-level state and control structure, and non-recursive subroutines, contrasts well with Pascal. Its grammar uses continuations and locations, dispelling any doubts that my work can only handle "nice" (well-designed) languages. Fortran's grammar is less complete than Pascal's; it lacks real numbers, FORMAT, EXTERNAL, and DATA statements, Hollerith constants, and statement functions. I have altered the lexical syntax as the compiler generator requires. Still, the grammar covers labelled COMMON blocks,

*The initial comments of Appendix D list all of the deficiencies.

EQUIVALENCE statements, DO statements with extended range, assigned and computed GO TO, arithmetic and logical IF, unformatted input/output with implied DO, subroutines, and functions.

Are semantic grammars more readable than other notations? The examples presented so far are idealized; real grammars can get messy. See if you can understand the Pascal grammar, and compare it with the axiomatic and denotational definitions [16, 61].

Appendix F is a semantic grammar that defines semantic grammars, but it describes only their syntax and domain checking, not the language generated by a particular grammar. A language description should not mimic the compiler generator. A suitably abstract description might be a predicate that checks whether a decorated parse-tree belongs to the language.

Writing a grammar for semantic grammars requires a separate, formal definition of semantic grammars. My proposed definition [45] is unsuitable for this purpose. It uses Watt and Madsen's idea that an attribute grammar generates a context-free grammar in which the nonterminals are symbols decorated with attribute values [63]. Checking a parse-tree for conformity with a grammar requires a means of comparing attribute values for equality. But equality is not defined on every domain of denotational semantics: there is no way to compare functions. Mayoh's technique [35] for converting attribute grammars into denotational definitions may lead to a better formal definition of semantic grammars.

8.2. Errors and Debugging

Writing a grammar, like writing a program, requires revising and debugging. The compiler generator recovers well from errors, even those defined by a grammar for a source language. It prints the erroneous line, points to the error, prints a descriptive message, and usually continues processing. On the rare occasions that it aborts, the usual cause is subscript error: overflow of some internal table. Each program's limits appear in the constant definitions at the beginning; they are high enough for the Pascal grammar.

The universal translator only detects two kinds of errors in programs: undefined attributes and violated constraints. It would be useful if auxiliary functions could also report errors. For instance, a function that merges declaration lists could report conflicting declarations. I

propose an error-expression that reports an error and returns a value for continued processing. A similar construct could produce readable messages for run-time errors.

Debugging user programs on the stack machine is difficult. A user program aborts by producing the value $\perp$; the only information reported is the current program counter and machine state, which is often incomprehensible. To locate the error in the source program, you must study the machine code and the semantic formula. A research problem is analyzing the grammar to automatically produce a debugger tailored to the source language.

A grammar may not work as expected, even if it describes correct semantics. It may cause the universal translator to yield inadequately simplified or duplicate formulas. The stack machine may loop, abort, consume large amounts of storage, or run extremely slowly (Sections 7.5, C.3). The best way to avoid bugs is to thoroughly test the grammar on small examples throughout its development.

I encountered few stubborn bugs while developing the Pascal and Fortran grammars. Most led to an obvious correction in the grammar; the grammar analyzer then created a new compiler in less than a minute. This immediate feedback let me work with confidence.

8.3. Improving Efficiency

The compiler generator is efficient enough to run experimental programs, but it is impractical for production runs. This section discusses its performance on several Pascal and Fortran programs (Table 8-1).

The grammar analyzer (Table 8-2) is fast; the language description file is a compact representation of a compiler.

The universal translator (Table 8-3) compiles programs at about eight seconds per page, twenty-five times slower than the regular Pascal compiler. For small programs, startup costs are a major fraction of the total costs. Ignoring startup, space and time requirements are approximately linear in the size of the program. Storage limitations prevent compiling programs longer than twenty pages.

The stack machine (Table 8-4) runs Pascal programs a thousand times slower than the regular Pascal system. This is still fast enough to compute the LR(0) kernels of a small

Table 8-1: Facts About the Test Programs

	Language	*Lines*	*Tokens*	*Function computed*
EQ	Pascal	39	283	Eight Queens solution
LR0	Pascal	438	2,023	LR(0) states
PRIME	Fortran	35	234	Prime numbers
SQRT	Fortran	102	569	Square roots (contrived program)

Table 8-2: Grammar Analyzer Performance

Grammar	*Lines*	*Rules*	*LR states*	*LDF words*	*Seconds*
Pascal	1,465	141	296	14,284	25
Fortran	1,030	114	224	9,115	14

Table 8-3: Universal Translator Performance

Total Costs

	Time (seconds)	*Storage (DAG nodes)*	*Code generated (instructions)*
EQ	8.9	3,592	972
LR0	50	13,165	5,305
PRIME	9.8	2,772	639
SQRT	19	4,975	1,492

Startup Costs

	Reading LDF (seconds)	*Storing semantics (DAG nodes)*
Pascal	3.1	2,200
Fortran	2.3	1,328

Table 8-4: Stack Machine Performance

	Time (seconds)	*Storage (nodes)*	*Speed (instructions/second)*
EQ	22	169	5,030
LR0	30	418	3,220
PRIME	30.6	105	2,590
SQRT	7.8	93	860

grammar. Its speed varies considerably, probably due to the effect of different programs on the array compactor.

The compiler generator is written in Pascal and runs on Stanford's DecSystem-20. It is simple and compact, considering its capabilities. Together with the Pascal grammar, it is an implementation of Pascal that is smaller than the standard Pascal compiler at Stanford:

Component	*Lines*
grammar analyzer	4,506
universal translator	4,097
stack machine	1,403
Pascal grammar	1,465
total	11,471
regular compiler	12,619

8.3.1. Faster Compilation

The universal translator spends most of its time simplifying the semantic DAG. Compiling a seven-page program, it initializes itself in three seconds, parses the program and builds the DAG in seven seconds, and simplifies the DAG in forty-seven seconds. The simplifier uniformly processes both static and dynamic semantics, although the requirements of each differ greatly. Distinguishing static from dynamic semantics would make a more modular, efficient, and predictable translator.

The compiler generator already places a strong emphasis on static semantics, yet more emphasis is needed. Static semantics requires its own notation, separate from that for dynamic semantics. The static notation would be just powerful enough to describe types and environments, and could manipulate quoted (uninterpreted) dynamic formulas. The dynamic notation could be based on any semantic framework: denotational, axiomatic, or operational. Attribute expressions would be written in the static notation, because they must all be evaluated at compile-time.

The difference between static and dynamic semantics is of great practical importance. For instance, a language implementation that checks types at compile-time is simpler and more efficient than one that generates run-time type descriptors. A separate notation for static semantics allows grammars to indicate which actions should be performed at compile-time. It also corrects the formal definition of semantic grammars — the static notation can be simple enough to have equality defined on all domains, since dynamic formulas are treated as text.

The separation replaces the simplifier with two specialized, efficient procedures: an interpreter and an optimizer. The interpreter executes the static semantics. The optimizer transforms its input formula into one that is semantically equivalent, but more useful. The semantic framework determines what optimizations to perform. Denotational semantics requires beta-reduction. Axiomatic semantics requires theorem-proving; such a system could verify programs written in any language.

The simplifier is complex because of its dual role. As an interpreter, it must reduce every static formula to a constant. As an optimizer, it must use whatever heuristics are necessary to produce efficient formulas. The simplifier already distinguishes between compile- and run-time recursion (Section B.1); its most important optimizations concern static semantics (Sections 7.2.2, 7.6.1).

Edinburgh LCF [12], a program verifier for denotational semantics, uses a separate metalanguage for processing lambda-calculus formulas. Formulas are written as quoted constants, and can be combined or built up using constructor functions. The metalanguage is flexible enough to express theorem-proving strategies.

8.3.2. Compiling Large Programs

The universal translator cannot compile programs longer than twenty pages, because it holds the semantics of the entire program in memory. Madsen [31] sketches a way to let the semantics overflow to secondary storage. A simpler, more efficient way to compile large programs may be to translate them one subprogram at a time, as most compilers do.

Every grammar designates a start symbol, so that the translator can recognize when it has parsed a program. If the grammar could designate other symbols, then the translator could recognize subprograms during parsing. (John Hennessy suggested this to me.) It could evaluate the subprogram's semantics, generate code for its synthesized attributes, and release its storage. The problem is designing the semantic interface — making sure the inherited attributes are available and that the rest of the program properly refers to the generated code.

8.3.3. Faster Execution

The stack machine refers to all data through pointers, even though only functions require the indirect representation. Tuples and integers can instead be represented as blocks of words. (The size of each block depends on its domain, but is constant.) The machine's overhead of reference counting and heap allocation would be greatly reduced.

Running Pascal and Fortran, the stack machine never accumulates function closures. Each closure is immediately applied to an argument. If closures were banned,* and the grammar analyzer could certify that a grammar did not require closures, then the stack machine could manage environments without reference counting. This gains efficiency at the price of accepting fewer languages.

If the grammar uses a dynamic environment for labels and procedures, then the machine must look them up every time it executes a goto statement or procedure call. Better code generation would eliminate the dynamic environment during compilation, as Sethi does [56], by unwinding its fixedpoint definition.

The array compactor uses the most time and is the most difficult to improve. It needs a breakthrough — some way of defining an array domain, where each array has fixed bounds and a uniform element type, but where different arrays can have different bounds. It must prevent sharing of arrays, to allow direct updating without the formation of association lists.

8.3.4. Generating Real Machine Code

After implementing the above refinements, the stack machine would be simple enough to eliminate, boosting efficiency tremendously. Ordinary machine code could provide most of its capabilities, with a run-time support package for the remainder, such as the array compactor.

Another way to eliminate the stack machine is to represent semantics operationally instead of denotationally — as a compiler's intermediate code. A semantic grammar can easily specify syntax-directed translation into intermediate code [1]. A final stage would translate the code into optimized hardware instructions.

---

*Constant functions such as λint.false do not need a closure representation, because they do not refer to the environment.

Such a compiler generator would require a separate notation for static semantics, because intermediate code only describes dynamic semantics. Even if the generator were not efficient enough for production use, it would help a compiler writer by simulating the production compiler, providing sample object code and error detection.

Recent research [23, 57] uses a notation for denotational semantics that resembles machine instructions. Abandoning denotational semantics solves the problems of reference counting, closures, environments, and arrays at once — if your only goal is fast compilers for traditional languages. But remember that denotational semantics is concise and powerful, can accomodate applicative languages and novel control structures, and is useful for reasoning about programs.

8.4. Implications for Language Design

A programming language should be formally defined even while it is being developed, to reveal its inconsistencies. Unfortunately, most language designers find definitions too difficult to write. The compiler generator allows anyone to debug a formal definition, written as a semantic grammar. As an extra incentive, it offers a free compiler for every definition. Compiling and executing test programs on the compiler generator provides further insights into a language.

Pascal's grammar reveals some trouble spots. Set expressions require special handling because they do not completely determine the set type; likewise, the constant nil can have any pointer type. Enumerated types declare constant identifiers as a side-effect, complicating every rule that refers to types. Using a function's name to designate its return variable requires extra bookkeeping.

A Fortran program can specify a variable's type, dimensions, COMMON block, and storage equivalence in any order, or not at all. These options cause messiness throughout the Fortran grammar, even though it imposes an order on declarations. Other Fortran constructs are so troublesome that the grammar does not handle them at all. A DATA statement affects the initial state, but may appear anywhere in a program. A statement function creates a local environment, but may implicitly declare global variables. A subscripted array variable is syntactically identical to a function call.

I would not condemn a language construct simply because it was difficult to formalize. The fault might lie in the formalism: for instance, denotational semantics can not easily represent tasking. Still, the compiler generator can contribute to the design of consistent, clean, and simple programming languages.

Appendix A
Using the Grammar Analyzer

When started, the grammar analyzer prints:

```
Semantic Grammar Analyzer

Language:
```

If you reply "ln," then the analyzer will look for a semantic grammar with file name ln.SG, write the language description file on ln.LDF, and produce a listing file on ln.LST. The analyzer will optionally list the LALR(1) states or attribute dependencies of the grammar.

A.1. Lexical Conventions for Semantic Grammars

The grammar analyzer uses the following ASCII representations of special symbols in semantic grammars:

symbol	*representation*	*note*
→	->	
λ	\	
|	!	
×	X	x is a reserved word!
⊥	BOT[*domain*]	the domain must be given

Names are strings of up to fifteen letters, digits, and underscores, beginning with a letter. Only variable names may contain digits. The case of letters is not significant.

Reserved words:

and	**attribute**	**bot**	**case**	**define**
div	**domain**	**else**	**end**	**eq**
esac	**false**	**fi**	**fix**	**forward**
ge	**gt**	**if**	**in**	**is**
le	**left**	**let**	**letrec**	**lt**
mod	**ne**	**nonassoc**	**not**	**of**
or	**propagate**	**resolution**	**right**	**rule**
then	**true**	**unspec**	**with**	**x**

Comments begin with # and continue until the end of the line. A separator is a space, end of line, or comment. Separators are required between reserved words, names, and numbers.

A.2. Resolution of Parser Conflicts

To resolve the dangling **else** conflict, include right **"else"** in the resolution part, and use the syntax

```
statement = "if" expression "then" statement "else" statement |
            "if" expression "then" statement | . . .
```

not

```
statement = "if" expression "then" statement elsePart
elsePart = "else" statement | empty
```

To resolve operator precedence, use appropriate resolution declarations and the syntax

```
expression = expression " + " expression | expression "*" expression | . . .
```

not

```
expression = expression operator expression | . . .
operator = " + " | "*" | . . .
```

To use the resolution part properly, you must know exactly how the grammar analyzer resolves conflicts in the parse table. Suppose there is a conflict between *shift* on symbol s and *reduce* by a rule whose rightmost terminal is t. The conflict is unresolvable if the rule has no terminals, or if neither s nor t is mentioned in the resolution part. Otherwise, the grammar analyzer resolves the conflict according to which operator has the higher binding power [1]:

priority	*grouping*	*parse table entry*
s<t	any	reduce
s>t	any	shift
s=t	left	reduce
s=t	right	shift
s=t	nonassoc	error

The analyzer does not resolve reduce-reduce conflicts.

A.3. Terminal Symbols

Terminals in semantic grammars may contain alphanumeric (including underscore) characters or non-alphanumeric characters, but not both. They may not be empty, contain spaces, or begin with a digit. A non-alphanumeric terminal may not be longer than two characters.

A.4. Error Messages

The grammar analyzer does not produce a language description file if there are errors in the semantic grammar. It reports only the first error encountered in a definition or rule. The messages are:

Illegal character *char*
: The semantic grammar contains a non-printing character.

15 characters maximum
: A name is longer than fifteen characters.

No digits allowed here *name*
: Only variable names may contain digits.

Another symbol expected *symbol*
: Syntax error in the input grammar.

Undeclared identifier in expression *name*
: The given name is undefined.

Command improperly terminated *symbol*
: A semicolon is expected after every definition and rule.

Unmatched quotes
: A string constant or terminal symbol runs past the end of a line.

Line too long — truncated .
The input line is too long to process.

Factor expected The current symbol cannot begin a factor.

Function required here *domain*
The current operand must be a function.

Sum domain required here *domain*
The current operand must belong to a sum domain.

Product domain required here *domain*
The current operand must belong to a product domain.

Equality undefined for this domain *domain*
The current operand must allow equality testing — it must not contain a function domain.

No such domain *name*
There is no domain of the listed name. The name of a variable must, after digits are removed, be a domain name, unless the domain of the variable is explicitly given.

Domain mismatch *domain1 domain2*
The two domains are required to be the same.

Illegal redefinition *name*
Attempt to redefine the given name.

Wrong number of attributes
The current nonterminal symbol is supplied the wrong number of attributes.

Illegal in a defining position
The current attribute expression is in a defining position, and may contain only constants, variables, tuples, and injections.

WITH depends on other WITHs
Pseudo-attributes may not depend upon each other. Either a with name or a uniqueName attribute appears in another with clause.

Not a selector of a sum domain *name*
The given name appears where a tag is required.

Wrong selector for case
The tags in a case expression must appear in the same order as in the definition of the union domain.

No such symbol in grammar *symbol*
The given symbol is not a terminal symbol of the grammar.

Illegal terminal symbol
The symbol may not be used as a terminal because it violates the restrictions mentioned above — contains spaces, for example.

No such nonterminal in grammar *name*
The given name is not a nonterminal of the grammar.

No strings generated by nonterminal *name*
The given nonterminal cannot produce any terminal productions.

Unreachable nonterminal *name*
The given nonterminal cannot be produced from the start symbol.

Unresolvable conflict in kernel *kernel number*
The given parsing kernel has a shift-reduce or reduce-reduce conflict, using LALR(1) lookahead.

Unresolved forward reference to *name*
The given name was never defined, but was used as a domain, expression, or attribute variable.

Missing final end token
Every grammar must be terminated by the token **end**.

Appendix B
Using the Universal Translator

When started, the translator prints

```
Universal Translator
Program:
```

It expects a file name of the form pn.ln, where pn is the program name and ln is the language name. It uses pn.ln as the program source file, looks for a language description file named ln.LDF, writes a listing on pn.LST, and writes object code on pn.COD. The listing includes storage and run-time statistics, the simplified semantics of the program, and the generated code.

B.1. Recursive Functions

The universal translator distinguishes two kinds of recursive functions. A function defined using **forward** and **define** declarations is *compile-time recursive*; the simplifier will always apply it to its arguments, expanding its definition. Every instance must disappear from the DAG before code generation, or the translator will signal an error. Use compile-time recursion for type-checking or bookkeeping functions that require evaluation during compilation.

A function defined using **letrec** or **fix** is *run-time recursive*, never expanded during simplification. It will be translated into machine instrutions and called at run-time. Use run-time recursion for the semantics of loops.

B.2. Syntax Errors

If the parser detects a syntax error, it prints a list of expected symbols and stops. (The symbols are the current LALR(1) lookahead; some of them may not really be valid continuations of the input.)

The scanner detects the following lexical errors:

Line longer than 80 characters
The input line is too long to process.

Name longer than 15 characters
The current name is longer than fifteen characters.

End of file in comment
A comment has no closing bracket, but continues until the end of the file.

Illegal character The program contains a non-printing character.

No such symbol in language
The current character has no meaning in the language.

B.3. Semantic Errors

The translator prints information to pinpoint the cause of a semantic error. The first line of a message lists the nonterminal containing the error, blank if the error occurred in a predefined nonterminal or with clause.

Semantic error: nonterminal

If the error is an attribute equal to $\perp$, then the next line lists the name, if any, of the attribute domain:

Undefined attribute domain name

If the error is a circularly defined attribute, then the next line is

Circularly defined attribute domain name

If the error is a failed constraint, then the next message line indicates the constraint's form:

constraint	*message*
x is tag	**Should be tag**
x eq y	**Attribute mismatch**
name(args)	**Failed check: name**
other	no message

Using meaningful names in your semantic grammar will make the error messages more readable.

Appendix C
Using the Stack Machine

The stack machine prompts for its code, input, and output files. After reading the object code and input data, it starts executing the object program. If the program terminates, the machine prints the current state and the value on top of the stack. A value of $\perp$ indicates abortion. The machine also aborts whenever it attempts to perform arithmetic on $\perp$.

C.1. Debugging

The stack machine does not provide a debugger; the state it prints is usually undecipherable. The only useful information printed is the program counter. The listing from the universal translator includes the simplified semantic DAG, as well as the object code. Comparing the code with the DAG, and the DAG with the source program, will locate where the program aborted.

If the machine crashes because the dump has overflowed, then the object program contains a function that loops by calling itself. (Less likely, the program may contain a tail-recursive call that the machine does not recognize.) The return address, repeated in almost every element of the dump, pinpoints the erroneous function call.

C.2. Input and Output

A semantic grammar must be compatible with the stack machine's input/output conventions for programs to run properly. Input and output are sequences of integers; a list $k_1, \ldots, k_n$ is represented as some permutation of:

$$([1 \rightarrow k_1] \ldots [n \rightarrow k_n] \perp, n)$$

The semantics of a program is a mapping from the input file to the output file:

```
INTMAP = INT → INT;          # contents of a file
FILE = INTMAP X INT;         # file including length
PROGRAM = FILE → FILE;       # input/output semantics
```

The Pascal grammar uses a dangerous trick: it defines INTMAP as INT→VALUE instead of INT→INT. Since VALUE is a union domain that includes INT, this works if tagfields are suppressed in the object code. The grammar specifies type-checking to guarantee that the VALUEs in the Pascal files really are INTegers.

The grammar must build the output file entirely from function updates, not lambda-abstractions. Although abstractions define legitimate functional values, the stack machine's output routine cannot evaluate them. Instead of

(λint.int + 10, 3)

the machine requires

[1→11][2→12][3→13] ⊥, 3)

or another permutation, such as

[3→13][1→11][2→12] ⊥, 3)

C.3. A Cautionary Note

A semantic formula may be correct but extremely inefficient to execute.

I represent a Pascal set by its characteristic function: **true** for the members of the set, **false** for the non-members.

SET = INT → BOOL;

The obvious definition of the empty set, λint.**false**, works in practice; the obvious definitions of the set operations, such as union, do not.

My first attempt to define union was

set1 ∪ set2 = λint. set1(int) **or** set2(int)

As a program executed, each set operation created a closure as the set's value. Each closure referenced two others. Over time, each set became a tree of closures. Testing set membership caused evaluation of the entire tree. Furthermore, the closures all contained environments referencing the state. The array compactor bogged down in the hundreds of extra references.

I solved the problem by defining the union operation to construct a new set by inserting the elements of set1 and set2 to the empty set, as function updates.

Appendix D
Pascal Grammar

```
#P2.SG, Pascal Semantic Grammar "P2"
#
#    missing Pascal features
#       INPUT and OUTPUT are files of integer, not char
#       standard routines PACK,UNPACK,READ,READLN,WRITE,WRITELN,EOLN,PAGE
#       standard procedures should be part of environment, not syntax
#       string constants
#       type REAL and standard functions for reals
#       procedures and functions passed as parameters
#       side-effects in functions
#       variable parameters are passed by value-result, not reference
#       GOTO command, LABEL declarations
#
#    errors not detected
#        subrange bounds
#        disjointness of fields, formals, caselists, enumerated constants
#        pointer type forward declarations
#        program parameters must be declared as variables

DOMAIN

####Values of variables and expressions

VAL = [intVal[INT]
     + arrayVal[ARRAY]
     + setVal[SET]
     + recVal[RECORD]
     + fileVal[FILE]
     + ptrVal[PTR]];

ARRAY = INT -> VAL;                          #array element for each subscript
SET = INT -> BOOL;                           #characteristic functions of sets
RECORD = NAME -> VAL;                        #values of each field
PTR = INT;                                   #indexes into heap
FILE = ARRAY X LEN X POS X VAL;              #file data, length, position, buffer
LEN = INT; POS = INT;
#Records also represent lexical levels -- values of each local variable


#### Environments

ACCESS = [varAc + valAc];                    #variable and value parameters

NAMES = [nilN + consN[NAME X NAMES]];        #name lists (for declarations)

#Lists of declarations for a block
DECLS = [nilD + consD[NAME X MODE X DECLS]];
PTRS = DECLS;                                #pointer type fixups
CONSTS = DECLS;                              #enumerated type constants
PARAMS = DECLS;                              #formal param lists
FIELDS = DECLS;                              #field lists

ACTIVITY = [active + inactive];              #whether a routine is active or not
```

```
PF = [proc + func[TYPE]];                    #whether a routine is a proc or a func

#Meanings of identifiers
MODE = [undefined
      + paramMode[ACCESS X TYPE]             #params and variables
      + constMode[TYPE X VAL]                #constants
      + typeMode[TYPE]                       #types
      + routMode[PARAMS X PF                 #procedures and functions
                 X ACTIVITY]];

LEVEL = INT;                                 #lexical levels
ENVMAP = NAME -> (LEVEL X MODE);
ENV = LEVEL X ENVMAP;                        #full environments

LOCENV = NAME -> MODE;                       #local environments

#Data types
TYPE = [scalarTy[RANGE X SCALID]
      + arrayTy[TYPE X TYPE]
      + setTy[SCALID]
      + recordTy[FIELDS X NAME X LOCENV]
      + fileTy[TYPE]
      + ptrTy[NAME]];

RANGE = INT X INT;                           #subrange of integers
SCALID = NAME;                               #unique id's for scalar types


####Runtime state

#The heap is the array variable "$heap" at level 0
#The last heap pointer used is  "$heapLimit" at level 0
S = LEVEL -> RECORD;                         #states (indexed by display level)
SX = S -> S;                                 #state transforms

DE = NAME -> SX;                             #dynamic environments
DT = DE -> DE -> DE;                         #DE transforms


####Semantic functions

EXP = DE -> S -> VAL;                        #expressions
VAR = DE -> S -> VAL -> SX;                  #variables
VAROPT = [notVar + isVar[VAR]];              #"variable option"
VT = VAR -> VAR;                             #variable transformations
COM = DE -> SX;                              #commands

INTLIST= [nilI + consI[INT X INTLIST]]; #lists of case labels
CASES = INT -> COM;                          #branch of a case command


####Procedure Linkage

#Set up initial state for a procedure
ACTVAL = DE -> S -> RECORD;                  #actual value parameters

#Store final values of params in calling program
ACTVAR = DE -> S -> EXIT;                    #actual variable parameters

#Store part of a lexical level in a state
EXIT = RECORD -> SX;                         #exit functions of final lexical level

#I/O interface is actually INT->INT;  however, this is the same as
#       ARRAY when tagfields are suppressed in runtime code, since
#       the variables "input" and "output" are files of integers.
#       This depends on the type-checking of Pascal, as enforced by
#       the grammar.
INTMAP = ARRAY;
INPUT = INTMAP X INT;                        #input for creating initial state
OUTPUT = INPUT;                              #output from final state
PROG = INPUT -> OUTPUT;                      #programs
```

```
FORWARD

assignCases:    (INTLIST X COM X CASES) -> CASES;
compatible:     (TYPE X TYPE) -> BOOL;
equal:          TYPE -> VAL -> VAL -> BOOL;
eqRecord:       FIELDS -> RECORD -> RECORD -> BOOL;
fixList:        (ENV X PTRS X LOCENV) -> LOCENV;
appendD:        (DECLS X DECLS) -> DECLS;
declNames:      (NAMES X MODE) -> DECLS;
evalDecls:      (DECLS X LOCENV) -> LOCENV;


DEFINE

########################################
# State Operations
########################################

#Function to assign a new value to a variable with given name and level
store = \(level,name) val s. [level -> [name->val] s level] s;


#Execute a command in a new level
run = \level record sx exit s0.
        LET s=sx([level->record]s0) IN
            exit(s level)([level->s0 level] s);


#Semantics of the WITH command in terms of its constituents
withCom = \(level,exp,var,com).
    \de s.run(level+1)(exp de s!recVal)(com de)
           (\record.var de s recVal[record])s;


#Variable transformations applied to varOpts
#   used to form the semantics of variables
vx = \varOpt vt.
    CASE varOpt OF
        notVar. notVar,
        isVar[var]. isVar[vt var]
    ESAC;


#Assign a command to its case labels
assignCases = \(intList,com,cases).
    CASE intList OF
        nilI. cases,
        consI[int,intList1]. [int->com] assignCases(intList1,com,cases)
    ESAC;


#Boolean operations with (1=true, 0=false)
#       numeric representation required because BOOLEAN is an enumerated type

boolVal = \bool. intVal[IF bool THEN 1 ELSE 0 FI];

#select one of two states depending on a condition
cond = \val s1 s2. IF val!intVal EQ 1 THEN s1 ELSE s2 FI;

andInt = \int1 int2. IF int1 EQ 1 THEN int2 ELSE 0 FI;
orInt = \int1 int2. IF int1 EQ 1 THEN 1 ELSE int2 FI;



########################################
#  TYPES
########################################
```

```
#Lookup a type name in 'env' and return the type
#Used to look up pointer type names
typeLookup = \(level,envMap) name.
    LET (level,mode)= envMap(name) IN mode!typeMode;


maxInt = 10000000;              #maximum allowable integer value

#standard types
intType = scalarTy[(-maxInt,maxInt), "$int"];
boolType = scalarTy[(0,1), "$bool"];
charType = scalarTy[(0,127), "$char"];
textType = fileTy[charType];


#Range for sets -- necessary to model their usual interpretation in Pascal
maxElem = 60;  minElem = 0;


#type compatibility rules:
# Name compatibility for records and scalars, others structure compatibility
# There are universal types for the NIL pointer and the empty set []
compatible = \(type1,type2).CASE type1 OF
        scalarTy[range,scalId1]. scalId1 EQ  RIGHT type2!scalarTy,
        arrayTy[type3,type4].
          LET (type5,type6) = type2!arrayTy
              IN compatible(type3,type5) AND compatible(type4,type6),
        setTy[scalId1].
            LET scalId2 = type2!setTy IN
                (scalId1 EQ scalId2) OR
                (scalId1 EQ "$null") OR (scalId2 EQ "$null"),
        recordTy[fields1,name1,locEnv].
            LET (fields2,name2,locEnv) = type2!recordTy IN name1 EQ name2,
        fileTy[type]. compatible(type, type2!fileTy),
        ptrTy[name1].
            LET name2 = type2!ptrTy
            IN (name1 EQ name2) OR (name1 EQ "$nil") OR (name2 EQ "$nil")
      ESAC;


#return a 'common' type of two types
#  currently trivial, but could handle int->real conversions, etc.
common = \(type1,type2).
    IF compatible(type1,type2) THEN
        IF (type1 IS setTy) AND (type1!setTy EQ "$null")
        THEN type2
        ELSE type1      FI
    ELSE     BOT[TYPE]  FI;


isInteger = \type. compatible(intType,type);
isBoolean = \type. compatible(booltype,type);



########################################
# Equality Testing of values
########################################

#The boolean expression x=y is compiled into an expression that does not
#  (after simplification) refer to the type of x and y.
#  "No types at runtime"

#Compare arrays element by element (runtime loop)
eqArray = \(int1,int2) type array1 array2.
    LETREC eqa: INT->BOOL =
        \int.(int GT int2) or
            (equal type (array1 int) (array2 int) AND eqa(int+1))
    IN eqa int1;
```

```
#Compare sets bit by bit
eqSet = \set1 set2.
    LETREC eqs: INT->BOOL =
        \int.(int GT maxElem) or
             ((set1 int EQ set2 int) AND eqs (int+1))
    IN eqs minElem;


#Compare records field by field (loop unwound at compile time)
eqRecord = \fields record1 record2.
    CASE fields OF
      nilD. TRUE,
      consD[name,mode,fields1].
        equal(RIGHT mode!paramMode)(record1 name)(record2 name) AND
        eqRecord fields1 record1 record2
    ESAC;


equal = \type val1 val2.
        CASE type OF
        scalarTy[range,scalId]. val1!intVal EQ val2!intVal,
        arrayTy[type1,type2].
            LET (range,scalId) = type1!scalarTy
            IN eqArray range type2 (val1!arrayVal) (val2!arrayVal),
        setTy[scalId]. eqSet(val1!setVal)(val2!setVal),
        recordTy[fields,name,locEnv].
            eqRecord fields (val1!recVal) (val2!recVal),
        fileTy[type]. BOT[BOOL],
        ptrTy[name]. val1!ptrVal EQ val2!ptrVal
        ESAC;


########################################
# Set Operations
########################################

#These are much less efficient than bitwise operations done by the hardware;
#    they perform one iteration per bit


#Insert a range of elements into a set
insert = \int1 int2 set.
    IF (minElem LE int1) AND (int2 LE maxElem)
    THEN LETREC ins: INT -> SET =
            \int. IF int GT int2 THEN set
                  ELSE [int->TRUE] ins(int+1)  FI
         IN ins(int1)
    ELSE BOT[SET]    FI;


#Test that set1 is a subset of set2
subset = \set1 set2.
    LETREC ss: INT->BOOL =
        \int.(int GT maxElem) OR
             ((NOT set1 int) OR set2 int) AND ss (int+1)
    IN ss minElem;


#The empty set
nullSet = \int.FALSE;


#Set union
#Cannot use   \int.(set1 int) OR (set2 int)    because history sequences of
#  closures would form, each referring to old states.
union = \set1 set2.
    LETREC un: INT->SET->SET =
        \int set. IF int GT maxElem THEN set
                  ELSE un(int+1) :
                       IF (set1 int) OR (set2 int) THEN [int->TRUE]set
                       ELSE set FI
```

```
                FI
    IN un minElem nullSet;


#Set difference
setDiff = \set1 set2.
    LETREC sd: INT->SET->SET =
        \int set. IF int GT maxElem THEN set
                ELSE sd(int+1) :
                        IF (set1 int) AND NOT (set2 int) THEN [int->TRUE]set
                        ELSE set FI
                FI
    IN sd minElem nullSet;


#Set intersection
intersect = \set1 set2.
    LETREC si: INT->SET->SET =
        \int set. IF int GT maxElem THEN set
                ELSE si(int+1) :
                        IF (set1 int) AND (set2 int) THEN [int->TRUE]set
                        ELSE set FI
                FI
    IN si minElem nullSet;



########################################
# Semantics of Array indexing
########################################


arrayExp = \(type,exp,exp1).
    LET (int1,int2) = LEFT type!scalarTy IN
    \de s.
    LET int=exp de s!intVal IN
    IF (int1 LE int) AND (int LE int2) THEN exp1 de s!arrayVal int
    ELSE BOT[VAL] FI;


arrayVar = \(type,exp,exp1,varOpt1).
    LET (int1,int2) = LEFT type!scalarTy IN
    vx varOpt1 : \var.\de s0 val s.LET int=exp de s0!intVal IN
            IF (int1 LE int) AND (int LE int2)
            THEN var de s0 arrayVal[[int->val] exp1 de s!arrayVal] s
            ELSE BOT[S] FI;



########################################
# Declarations
########################################


#Declare a list of names with a mode (for variables, parameters)
declNames = \(names,mode).
    CASE names OF
        nilN. nilD,
        consN[name,names1].
            consD[name, mode, declNames(names1,mode)]
    ESAC;


#Append declarations
appendD = \(decls1,decls2).
    CASE decls1 OF
        nilD. decls2,
        consD[name,mode,decls]. consD[name,mode, appendD(decls,decls2)]
    ESAC;
```

```
#Nest a new LOCENV onto an ENV, making a new block
nest = \(locEnv, (level,envMap)).
    (level+1,
     \name.LET mode = locEnv(name) IN
            IF mode IS undefined THEN envMap(name) ELSE (level+1,mode) FI);


#The empty LOCENV
nullEnv = \name.undefined;


#Working function for fixPtrs
fixList = \(env,ptrs,locEnv).
    CASE ptrs OF
    nilD. locEnv,
    consD[name,mode,ptrs].
        [name->typeMode[typeLookup env (mode!typeMode!ptrTy)]]
        fixList(env,ptrs,locEnv)
    ESAC;


#Bind internal names to referent types of pointers
fixPtrs = \(env,ptrs,locEnv). fixList(nest(locEnv,env), ptrs, locEnv);


#Attach new DECLS onto a LOCENV, making a new LOCENV
evalDecls = \(decls,locEnv).
    CASE decls OF
      nilD. locEnv,
      consD[name,mode,decls1]. [name->mode] evalDecls(decls1,locEnv)
    ESAC;


#Declare a routine to itself, to allow use of the function name as a variable
routDec = \(name, params, pf, (level,envMap)).
    (level, [name -> (level,routMode[params,pf,active])] envMap);


####Standard declarations

#Declare a standard procedure p(VAR x: type)
stdProc = \type.
    routMode[ consD["x", paramMode[varAc,type], nilD], proc, inactive];


#Declare a standard function "f(x: type1): type2"
stdFunc = \(type1,type2).
    routMode[ consD["x", paramMode[valAc,type1], nilD],
              func[type2], inactive];


#Declare a standard function in the dynamic environment
funcDef = \(name, f:INT->INT) de.
    [name -> \s. store(1,name) intVal[ f(s 1 "x" !intVal) ] BOT[S] ] de;


#Set up initial state: heap, files INPUT and OUTPUT
beginProg = \(intMap,int).
        ["input"        -> fileVal[intMap, int, 1, intMap(1)] ]
        ["output"       -> fileVal[BOT[ARRAY], 0, 1, BOT[VAL]]]
        ["$heap"        -> arrayVal[BOT[ARRAY]] ]
        ["$heapLimit"   -> intVal[0]]
        BOT[RECORD];                    #get input into top level

#Extract output from top level of final state
endProg = \record.
    LET (array,len,pos,val) = record "output" ! fileVal IN (array,len);
```

```
ATTRIBUTE

identifier      <ENV.NAME,LEVEL,MODE>;
nameList        <.NAMES>;

constant        <ENV.TYPE,VAL>;                 #Constants
literalConst    <ENV.TYPE,VAL>;
constantDecls   <ENV,LOCENV.LOCENV>;

dataType        <ENV.PTRS,CONSTS,TYPE>;         #Data types
scalarType      <ENV.CONSTS,TYPE>;
arraySpec       <ENV.PTRS,CONSTS,TYPE>;
packed          <.BOOL>;
constList       <TYPE.CONSTS,INT>;
recordSection   <ENV.PTRS,CONSTS,FIELDS>;
fieldList       <ENV.PTRS,CONSTS,FIELDS>;
fixedPart       <ENV.PTRS,CONSTS,FIELDS>;
variantPart     <ENV.PTRS,CONSTS,FIELDS>;
variantList     <ENV,TYPE.PTRS,CONSTS,FIELDS>;
variant         <ENV,TYPE.PTRS,CONSTS,FIELDS>;
tagField        <ENV.NAME,TYPE>;
caseLabelList   <ENV,TYPE.INTLIST>;
typeIdentifier  <ENV.TYPE>;
typeDecls       <ENV,LOCENV.LOCENV>;

component       <ENV.TYPE,EXP,VAROPT>;          #Variables
arrayIndex      <ENV.TYPE,EXP,VAROPT>;
variableDecls   <ENV,LOCENV.LOCENV>;
variableGroup   <ENV,LOCENV.LOCENV,TYPE>;

expression      <ENV.TYPE,EXP>;                 #Expressions
compatExp       <ENV,TYPE.EXP>;
expvar          <ENV.TYPE,EXP,VAROPT>;
actualParamList <ENV,PARAMS.ACTVAL,ACTVAR>;
actualParam     <ENV,MODE.EXP,VAR>;
actualParams    <ENV,PARAMS.ACTVAL,ACTVAR>;
setElements     <ENV.SCALID,EXP>;
elementList     <ENV.SCALID,EXP>;
element         <ENV.SCALID,EXP,EXP>;

command         <ENV.COM>;                      #Commands
compoundCommand <ENV.COM>;
commandSequence <ENV.COM>;
caseListElem    <ENV,TYPE,CASES.CASES>;
caseList        <ENV,TYPE.CASES>;
withTail        <ENV.COM>;

formalParamList <ENV.PARAMS>;                   #Procedures and Functions
formalSection   <ENV.PARAMS>;
formalParams    <ENV.PARAMS>;
formalGroup     <ACCESS,ENV.PARAMS>;
routineDecls    <ENV,ENV,LOCENV.LOCENV,DT>;
routine         <ENV,ENV,LOCENV.LOCENV,DT>;
routineHead     <ENV.PARAMS,PF,NAME>;

constDeclPart   <ENV,LOCENV.LOCENV>;            #Blocks
typeDeclPart    <ENV,LOCENV.LOCENV>;
varDeclPart     <ENV,LOCENV.LOCENV>;
block           <ENV,LOCENV.COM>;
programParams   <.NAMES>;
program         <.PROG>;


RULE program


identifier<(level0,envMap),name,level,mode> = ident<name>
    WITH (level,mode) = envMap(name);
```

```
nameList<consN[name,nilN]> = ident<name>;

nameList<consN[name,names]> = ident<name> "," nameList<names>;


########################################
# Constants
########################################


constant<env,type,val> =
    identifier<env,name,level,constMode[type,val]>;


constant<env,type,val> = literalConst<env,type,val>;


constant<env,intType,intVal[-val!intVal]> =
    "-" constant<env,type,val>          WHERE<isInteger(type)>;


constant<env,intType,val> =
    "+" constant<env,type,val>          WHERE<isInteger(type)>;


literalConst<env,ptrTy["$nil"],ptrVal[0]> = "NIL";

literalConst<env,intType,intVal[int]> = number<int>;


constantDecls<env,locEnv, [name->constMode[type,val]] locEnv1> =
    constantDecls<env,locEnv,locEnv1>
    ident<name>   WHERE<locEnv1 name IS undefined>
    "=" constant<nest(locEnv1,env), type, val> ";" ;


constantDecls<env,locEnv,locEnv> = ;


########################################
# Types
########################################


#type identifiers

dataType<env,nilD,nilD,type> = typeIdentifier<env,type>;

typeIdentifier<env,type> = identifier<env,name,level,typeMode[type]>;


#enumerated types


dataType<env,nilD,consts,scalarTy[(0,int),scalId]> =
    uniqueName<scalId>
    "(" constList<scalarTy[(0,int),scalId], consts, int> ")";


constList<type, consD[name, constMode[type,intVal[1]], nilD],1> =
    ident<name>;

constList<type, consD[name, constMode[type,intVal[int+1]], consts], int+1> =
    constList<type,consts,int> "," ident<name>;


#subrange types

dataType<env,nilD,nilD,scalarTy[(int1,int2), scalId]> =
    constant<env,type1,intVal[int1]> ".." constant<env,type2,intVal[int2]>
```

```
    WHERE<compatible(type1,type2)>
    WITH scalarTy[range,scalId] = type1;


scalarType<env,consts,type> =
    dataType<env,ptrs,consts,type>              WHERE<type IS scalarTy>;


#packing

packed<TRUE> = "PACKED";
packed<FALSE> = ;


#Multi-dimensioned arrays
dataType<env,ptrs,consts,type> =
    packed<bool> "ARRAY" "[" arraySpec<env,ptrs,consts,type>;


arraySpec<env,ptrs,appendD(consts1,consts2),arrayTy[type1,type2]> =
    scalarType<env,consts1,type1> "]" "OF" dataType<env,ptrs,consts2,type2>;


arraySpec<env,ptrs,appendD(consts1,consts2),arrayTy[type1,type2]> =
    scalarType<env,consts1,type1> "," arraySpec<env,ptrs,consts2,type2>;


#sets


dataType<env,nilD,consts,setTy[scalId]> =
    packed<bool>
    "set" "of" scalarType<env,consts,scalarTy[(int1,int2),scalId]>

    WHERE<(int1 GE minElem) AND (int2 LE maxElem)>;


##############
#records

dataType<env,ptrs,consts,recordTy[fields,name,evalDecls(fields,nullEnv)]> =
    packed<bool> "RECORD"
    fieldList<env,ptrs,consts,fields> uniqueName<name> "END";


fieldList<env,ptrs,consts,fields> =
    fixedPart<env,ptrs,consts,fields>;


fieldList<env,ptrs,consts,fields> =
    variantPart<env,ptrs,consts,fields>;


fieldList<env,appendD(ptrs1,ptrs2),appendD(consts1,consts2),
          appendD(fields1,fields2)> =
    fixedPart  <env,ptrs1,consts1,fields1> ";"
    variantPart<env,ptrs2,consts2,fields2>;


#fixed parts


fixedPart<env,ptrs,consts,fields> =
    recordSection<env,ptrs,consts,fields>;


fixedPart<env,appendD(ptrs1,ptrs2),appendD(consts1,consts2),
          appendD(fields1,fields2)> =
    fixedPart    <env,ptrs2,consts2,fields2> ";"
    recordSection<env,ptrs1,consts1,fields1>;
```

```
recordSection<env,ptrs,consts,declNames(names, paramMode[varAc,type])> =
    nameList<names> ":" dataType<env, ptrs, consts, type>;


recordSection<env,nilD,nilD,nilD> = ;


#variant parts


variantPart<env,ptrs,consts, consD[name, paramMode[varAc,type], fields]> =
    "CASE" tagField<env,name,type>      WHERE<type IS scalarTy>
    "OF" variantList<env,type,ptrs,consts,fields>;


variantList<env,type,ptrs,consts,fields> =
    variant<env,type,ptrs,consts,fields>;


variantList<env,type,appendD(ptrs1,ptrs2), appendD(consts1,consts2),
            appendD(fields1,fields2)> =
    variant    <env,type,ptrs1,consts1,fields1> ";"
    variantList<env,type,ptrs2,consts2,fields2>;


variant<env,type,ptrs,consts,fields> =
    caseLabelList<env,type,intList> ":"
    "(" fieldList<env,ptrs,consts,fields> ")";


variant<env,type,nilD,nilD,nilD> = ;


tagField<env,name,type> =       ident<name> ":" typeIdentifier<env,type>;


tagField<env,"$noName",type> =  typeIdentifier<env,type>;


caseLabelList<env,type,consI[val|intVal,nilI]> =
    constant<env,type1,val>             WHERE<compatible(type,type1)>;


caseLabelList<env,type,consI[val|intVal,intList]> =
    constant<env,type1,val>             WHERE<compatible(type,type1)>
    ","  caseLabelList<env,type,intList>;


#files

dataType<env,ptrs,consts,fileTy[type]> =
    packed<bool> "FILE" "OF" dataType<env,ptrs,consts,type>;


#Pointer types -- generate fixuplist of forward references to types

dataType<env,
         consD[name1, typeMode[ptrTy[name2]], nilD],
         nilD, ptrTy[name1]> =
    uniqueName<name1>
    "↑"  ident<name2>;


#type declarations

typeDecls<env,locEnv,fixPtrs(env,ptrs,locEnv1)> =
    ident<name>   WHERE<locEnv(name) IS undefined>
    "=" dataType<nest(locEnv,env), ptrs, consts, type> ";"
    typeDecls<env, [name->typeMode[type]]evalDecls(consts,locEnv), locEnv1>;
```

```
typeDecls<env,locEnv,locEnv> = ;


########################################
# Variables and Components
########################################


#To avoid syntactic ambiguity, Components cover all uses of identifiers in
#    expressions:
#       variables and parameters,
#       constants,
#       function return variables,
#       calls of functions of zero arguments.

component<env,type,exp,varOpt> = identifier<env,name,level,mode>
    WITH (type,exp,varOpt) =

    CASE mode OF
    undefined. BOT[TYPE X EXP X VAROPT],

    paramMode[access,type1].
        (type1,
         \de s.s(level)name,
         isVar[\de s1 val.store(level,name)val ]),

    constMode[type1,val]. (type1, \de s.val, notVar),

    typeMode[type]. BOT[TYPE X EXP X VAROPT],

    routMode[params,pf,activity].
        IF activity IS active THEN
             (pf!func,
              \de s.s(level+1)name,
              isVar[\de s1 val.store(level+1,name)val ])
        ELSE IF params IS nilD THEN
             (pf!func,
              \de s.de name ([level+1 -> BOT[RECORD]] s) (level+1) name,
              notVar)
             ELSE BOT[TYPE X EXP X VAROPT]
             FI
        FI
    ESAC;


#array access

component<env,type,exp,varOpt> =
    arrayIndex<env,type,exp,varOpt> "]";


arrayIndex<env, type2, arrayExp(type1,exp,exp1),
          arrayVar(type1,exp,exp1,varOpt1)> =
    component<env,arrayTy[type1,type2],exp1,varOpt1>
   "[" compatExp<env,type1,exp>;


arrayIndex<env, type2, arrayExp(type1,exp,exp1),
          arrayVar(type1,exp,exp1,varOpt1)> =
    arrayIndex<env,arrayTy[type1,type2],exp1,varOpt1>
   "," compatExp<env,type1,exp>;


#record field access

component<env, RIGHT locEnv name!paramMode,
          \de s.exp de s !recVal name, varOpt1> =
    component<env,recordTy[fields,name0,locEnv],exp,varOpt> "." ident<name>
    WITH varOpt1 =
        vx varOpt : \var.\de s0 val s.var de s0
```

```
                recVal[ [name->val] exp de s !recVal ] s;

#pointer dereferencing / file buffer access

component<env,type1,exp1,varOpt1> =
    component<env,type,exp,varOpt> "↑"
    WITH (type1,exp1,varOpt1) =
        IF type IS ptrTy THEN
            (typeLookup env (type!ptrTy),
             \de s.LET ptr = exp de s !ptrVal IN
             IF ptr NE 0 THEN s 0 "$heap" ! arrayVal ptr ELSE BOT[VAL] FI,
             isVar[\de s val s.LET ptr = exp de s !ptrVal IN
                    IF ptr NE 0 THEN
                     store(0,"$heap")
                     arrayVal[ [ptr->val] (s 0 "$heap" ! arrayVal)]
                     s
                    ELSE BOT[S] FI])
        ELSE
        IF type IS fileTy THEN
            (type!fileTy,
             \de s. LET (array,len,pos,val) = exp de s!fileVal IN val,
             vx varOpt : \var,\de s0 val s.
                          LET (array,len,pos,val0) = exp de s!fileVal
                          IN var de s0 fileVal[array,len,pos,val] s)
        ELSE BOT[TYPE X EXP X VAROPT] FI FI;


#variable declarations


variableDecls<env,locEnv,locEnv2> =
    variableDecls<env,locEnv,locEnv1>
    variableGroup<nest(locEnv1,env), locEnv1, locEnv2, type>;


variableDecls<env,locEnv,locEnv> = ;


variableGroup<env,locEnv,
              [name->paramMode[varAc,type]]
                fixPtrs(env,ptrs,evalDecls(consts,locEnv)),
              type> =
    ident<name>                 WHERE<locEnv(name) IS undefined>
    ":" dataType<env,ptrs,consts,type> ";";


variableGroup<env,locEnv,locEnv1,type> =
    ident<name>                 WHERE<locEnv(name) IS undefined>
    "," variableGroup
        <env, [name->paramMode[varAc,type]]locEnv, locEnv1, type>;



########################################
# Expressions
########################################

# To avoid syntactic ambiguity because actual parameters can be either
#     variables or expressions, the symbol ExpVar handles the semantics
#     of both.  VarOpt holds the variable semantics, if any.

expression<env,type,exp> = expVar<env,type,exp,varOpt>;


#Expression that must have a particular (inherited) Type
compatExp<env,type,exp> =
    expression<env,type1,exp>           WHERE<compatible(type,type1)>;


expVar<env,type,exp,varOpt> = component<env,type,exp,varOpt>;
```

```
expVar<env,type,\de s.val,notVar> = literalConst<env,type,val>;

expVar<env,type,exp,notVar> = "(" expression<env,type,exp> ")";


##############
#function call


expVar<env,type,
       \de s.de name([level+1 -> actVal de s] s) (level+1) name,
       notVar> =
    identifier<env,name,level,routMode[params,func[type],activity]>
    "(" actualParams<env,params,actVal,actVar> ")";


#### Parameter passing for procedures and functions
#   Variable parameters are passed by value-result.

actualParamList<env,params,actVal,actVar> =
    "(" actualParams<env,params,actVal,actVar> ")";


actualParamList<env, nilD, \de s.BOT[RECORD], \de s0 record s.s> = ;



actualParams<env,consD[name,mode,nilD],
             \de s.[name->exp de s] BOT[RECORD],
             \de s0 record.var de s0 (record name)> =
    actualParam<env,mode,exp,var>;


actualParams<env,consD[name,mode,params],
             \de s.[name->exp de s] (actVal de s),
             \de s0 record s.var de s0 (record name)(actVar de s0 record s)> =
    actualParam<env,mode,exp,var> ","
    actualParams<env,params,actVal,actVar>;


actualParam<env, paramMode[access,type1], exp, var> =
    expVar<env,type2,exp,varOpt>
    WHERE<compatible(type1,type2)>
    WITH var = CASE access OF
                   varAc. varOpt!isVar,        #VAR param -- store final value
                   valAc. \de s0 val s.s       #value param -- do nothing
               ESAC;



#standard functions

expVar<env,type, \de s. intVal[exp de s!intVal + 1], notVar> =
    "SUCC" "(" expression<env,type,exp> ")"
    WHERE<type IS scalarTy>;


expVar<env,type, \de s. intVal[exp de s!intVal - 1], notVar> =
    "PRED" "(" expression<env,type,exp> ")"
    WHERE<type IS scalarTy>;


expVar<env,boolType,
       \de s. LET (array,len,pos,val) = exp de s!fileVal
              IN boolVal(pos GT len),
       notVar> =
    "EOF" "(" expression<env,fileTy[type],exp> ")";


###############
```

```
#set expression


expVar<env,setTy[scalId],exp,notVar> =
    "[" setElements<env,scalId,exp> "]";


setElements<env, "$null", \de s.setVal[nullSet]> = ;            #Empty set


setElements<env;scalId,exp> =    elementList<env,scalId,exp>;


elementList<env,scalId,
          \de s.setVal[insert(exp1 de s!intVal)(exp2 de s!intVal)
                          (exp de s!setVal)]> =
    elementList<env,scalId,exp> "," element<env,scalId,exp1,exp2>;


elementList<env,scalId,
          \de s.setVal[insert(exp1 de s!intVal)(exp2 de s!intVal)nullSet]> =
    element<env,scalId,exp1,exp2>;


element<env,scalId,exp,exp> =                              #Single element
    expression<env,scalarTy[range,scalId],exp>;


element<env,scalId,exp1,exp2> =                            #Range of elements
    expression<env,scalarTy[range1,scalId],exp1>  ".."
    expression<env,scalarTy[range2,scalId],exp2>;


################
#Unary operators


expVar<env, boolType, \de s.intVal[1-exp de s!intVal], notVar> =
    "NOT"  expression<env,type,exp>     WHERE<isBoolean(type)>;


expVar<env, intType,  \de s.intVal[-exp de s!intVal], notVar> =
    "-"  expression<env,type,exp>       WHERE<isInteger(type)>;


expVar<env,intType,exp,notVar> =
    "+"  expression<env,type,exp>       WHERE<isInteger(type)>;


################
#Infix operators

#arithmetic operators (some overloaded with set operations)


expVar<env,common(type1,type2),exp,notVar> =
    expression<env,type1,exp1>   "+"   expression<env,type2,exp2>
    WITH exp=
        IF type1 IS setTy THEN
            \de s.setVal[union(exp1 de s!setVal)(exp2 de s!setVal)]
        ELSE
        IF isInteger(type1) THEN
            \de s.intVal[exp1 de s !intVal + exp2 de s !intVal]
        ELSE BOT[EXP] FI FI;


expVar<env,common(type1,type2),exp,notVar> =
    expression<env,type1,exp1>   "-"   expression<env,type2,exp2>

    WITH exp=
```

```
        IF type1 IS setTy THEN
            \de s.setVal[setDiff(exp1 de s!setVal)(exp2 de s!setVal)]
        ELSE
        IF isInteger(type1) THEN
            \de s.intVal[exp1 de s !intVal - exp2 de s !intVal]
        ELSE BOT[EXP] FI FI;


expVar<env,common(type1,type2),exp,notVar> =
    expression<env,type1,exp1>   "*"   expression<env,type2,exp2>

    WITH exp=
        IF type1 IS setTy THEN
            \de s.setVal[intersect(exp1 de s!setVal)(exp2 de s!setVal)]
        ELSE
        IF isInteger(type1) THEN
            \de s.intVal[exp1 de s !intVal = exp2 de s !intVal]
        ELSE BOT[EXP] FI FI;


expVar<env,intType,
       \de s.intVal[exp1 de s!intVal DIV exp2 de s!intVal], notVar> =
    expression<env,type1,exp1>  "DIV"  expression<env,type2,exp2>
    WHERE<isInteger(type1)>            WHERE<isInteger(type2)>;


expVar<env,intType,
       \de s.intVal[exp1 de s!intVal MOD exp2 de s!intVal], notVar> =
    expression<env,type1,exp1>  "MOD"  expression<env,type2,exp2>
    WHERE<isInteger(type1)>            WHERE<isInteger(type2)>;


#Boolean connectives


expVar<env, boolType,
       \de s. intVal[andInt(exp1 de s!intVal)(exp2 de s!intVal)], notVar> =
    expression<env,type1,exp1> "AND" expression<env,type2,exp2>
    WHERE<isBoolean(type1)>            WHERE<isBoolean(type2)>;


expVar<env, boolType,
       \de s. intVal[orInt(exp1 de s!intVal)(exp2 de s!intVal)], notVar> =
    expression<env,type1,exp1> "OR" expression<env,type2,exp2>
    WHERE<isBoolean(type1)>            WHERE<isBoolean(type2)>;


#relations (some overloaded with subset relation)

expVar<env,boolType,
       \de s.boolVal(exp1 de s!intVal LT exp2 de s!intVal), notVar> =
    expression<env,type1,exp1>   "<"   expression<env,type2,exp2>

    WHERE<compatible(type1,type2)>     WHERE<type1 IS scalarTy>;


expVar<env,boolType,
       \de s.boolVal(exp1 de s!intVal GT exp2 de s!intVal), notVar> =
    expression<env,type1,exp1>   ">"   expression<env,type2,exp2>

    WHERE<compatible(type1,type2)>     WHERE<type1 IS scalarTy>;


expVar<env,boolType,exp,notVar> =
    expression<env,type1,exp1>   "<="   expression<env,type2,exp2>
    WHERE<compatible(type1,type2)>

    WITH exp=
```

```
        IF type1 IS setTy THEN
            \de s. boolVal(subset(exp1 de s!setVal)(exp2 de s!setVal))
        ELSE
        IF type1 IS scalarTy THEN
            \de s. boolVal(exp1 de s !intVal LE exp2 de s !intVal)
        ELSE BOT[EXP] FI FI;


expVar<env,boolType,exp,notVar> =
    expression<env,type1,exp1>   ">="   expression<env,type2,exp2>
    WHERE<compatible(type1,type2)>

    WITH exp=
        IF type1 IS setTy THEN
            \de s. boolVal(subset(exp2 de s!setVal)(exp1 de s!setVal))
        ELSE
        IF type1 IS scalarTy THEN
            \de s. boolVal(exp1 de s !intVal GE exp2 de s !intVal)
        ELSE BOT[EXP] FI FI;


expVar<env,boolType,
       \de s.boolVal : equal type1(exp1 de s)(exp2 de s), notVar> =
    expression<env,type1,exp1>   "="   expression<env,type2,exp2>
    WHERE<compatible(type1,type2)>;


expVar<env,boolType,
       \de s.boolVal : NOT equal type1(exp1 de s)(exp2 de s), notVar> =
    expression<env,type1,exp1>   "<>"   expression<env,type2,exp2>
    WHERE<compatible(type1,type2)>;


expVar<env, boolType,
       \de s.boolVal : exp2 de s!setVal : exp1 de s!intVal, notVar> =
    expression<env,scalarTy[range,scalId],exp1>
    "IN"   expression<env,setTy[scalId],exp2>;


########################################
# Commands
########################################


#assignment command
command<env,\de s.var de s(exp de s)s> =
    component<env,type,exp1,isVar[var]> ":=" compatExp<env,type,exp>;


#procedure call
command<env,\de s0.run(level+1)(actVal de s0)(de name)(actVar de s0)s0> =
    identifier<env,name,level,routMode[params,proc,activity]>
    actualParamList<env,params,actVal,actVar>;


#standard procedures


command<env, \de s. LET (array,len,pos,val)= exp de s!fileVal
                     IN var de s fileVal[array,len,1,array 1] s> =
    "RESET" "(" component<env,fileTy[type],exp,isVar[var]> ")";


command<env, \de s. var de s fileVal[BOT[ARRAY],0,1,BOT[VAL]] s> =
    "REWRITE" "(" component<env,fileTy[type],exp,isVar[var]> ")";
```

```
command<env, \de s. LET (array,len,pos,val)= exp de s!fileVal
                      IN IF pos GT len THEN BOT[S]
                         ELSE var de s fileVal[array,len,pos+1,array(pos+1)] s
                         FI> =
    "GET" "(" component<env,fileTy[type],exp,isVar[var]> ")";


command<env,
          \de s. LET (array,len,pos,val)= exp de s!fileVal
                 IN IF pos GT len
                    THEN var de s
                         fileVal[[pos->val]array,len+1,pos+1,BOT[VAL]] s
                    ELSE BOT[S]
                    FI> =
    "PUT" "(" component<env,fileTy[type],exp,isVar[var]> ")";


#Empty command
command<env,\de s.s> = ;


#Compound command


command<env,com> = compoundCommand<env,com>;


compoundCommand<env,com> = "BEGIN" commandSequence<env,com> "END";


commandSequence<env,com> = command<env,com>;


commandSequence<env,\de s.com2 de (com1 de s)> =
    commandSequence<env,com1> ";" command<env,com2>;


#Conditional command


command<env,\de s. cond(exp de s)(com1 de s)(com2 de s)> =
    "IF" compatExp<env,boolType,exp> "THEN" command<env,com1>
    "ELSE" command<env,com2>;


command<env,\de s. cond(exp de s)(com de s)s> =
    "IF" compatExp<env,boolType,exp> "THEN" command<env,com>;


#Case command


command<env,\de s.cases (exp de s!intVal)de s> =
    "CASE" expression<env,type,exp>              WHERE<type IS scalarTy>
    "OF" caseList<env,type,cases>  "END";


caseList<env,type,cases> =
    caseListElem<env,type,\int.BOT[COM],cases>;


caseList<env,type,cases> =
    caseListElem<env,type,cases1,cases> ";"
    caseList    <env,type,cases1>;


caseListElem<env,type,cases,assignCases(intList,com,cases)> =
```

```
   caseLabelList<env,type,intList> ":" command<env,com>;

caseListElem<env,type,cases,cases> = ;

#Repeat and while commands

command<env,\de.FIX \sx.\s.cond(exp de s)(sx(com de s))s> =
    "WHILE" compatExp<env,boolType,exp> "DO" command<env,com>;

command<env,\de.FIX \sx.\s0.LET s=com de s0 IN cond(exp de s)s(sx s)> =
    "REPEAT" commandSequence<env,com> "UNTIL" compatExp<env,boolType,exp>;

#For command -- TO and DOWNTO

command<env,\de s0.LET int2 = exp2 de s0 !intVal IN
        LETREC for:INT->SX = \int s.
           IF int GT int2 THEN s
           ELSE for(int+1)(com de (store(level,name) intVal[int] s)) FI
        IN for (exp1 de s0 !intVal) s0> =
    "FOR" identifier<env,name,level,paramMode[varAc,type]>
    WHERE<type IS scalarTy>
    ":="
    compatExp<env,type,exp1> "TO" compatExp<env,type,exp2>
    "DO" command<env,com>;

command<env,\de s0.LET int2 = exp2 de s0 !intVal IN
        LETREC for:INT->SX = \int s.
           IF int LT int2 THEN s
           ELSE for(int-1)(com de (store(level,name) intVal[int] s)) FI
        IN for (exp1 de s0 !intVal) s0> =
    "FOR" identifier<env,name,level,paramMode[varAc,type]>
    WHERE<type IS scalarTy>
    ":="
    compatExp<env,type,exp1> "DOWNTO" compatExp<env,type,exp2>
    "DO" command<env,com>;

#with command

command<env,com> =  "WITH" withTail<env,com>;

withTail<env, withCom(level,exp,var,com)> =
    component<env, recordTy[fields,name,locEnv], exp, isVar[var]>
    "DO"  command<nest(locEnv,env), com>
    WITH level = LEFT env;

withTail<env, withCom(level,exp,var,com)> =
    component<env, recordTy[fields,name,locEnv], exp, isVar[var]>
    ","   withTail<nest(locEnv,env), com>
    WITH level = LEFT env;

##########################################
# Procedures and Functions
##########################################

#Forward declarations are not needed because the BODIES of the routines
```

```
#       are evaluated in the FINAL environment for the block.  This does
#       not make the grammar circular because environments do not contain
#       the run-time semantics of routines, but only information from the
#       ROUTINE HEAD, which is evaluated using the CURRENT enviroment.
#       The run-time semantics are stored in the dynamic environment 'de'.


routineDecls<env1,env,locEnv,locEnv2,\de de0.dt1 de(dt2 de de0)> =
    routineDecls<env1,env,locEnv,locEnv1,dt1>
    routine<env1,env,locEnv1,locEnv2,dt2>;


routineDecls<env1,env,locEnv,locEnv,\de de0.de0> = ;


routine<env1,env,locEnv, [name->routMode[params,pf,inactive]] locEnv,
        \de de0.[name->com de]de0> =
    routineHead<nest(locEnv,env), params, pf, name>
    WHERE<locEnv(name) IS undefined>
    ";"
    block<routDec(name,params,pf,env1), evalDecls(params,nullEnv),~com>
    ";";


routineHead<env,params,proc,name> =
    "PROCEDURE" ident<name> formalParamList<env,params>;


routineHead<env,params,func[type],name> =
    "FUNCTION" ident<name>  formalParamList<env,params>
    ":" typeIdentifier<env,type>;


formalParamList<env,params> = "(" formalParams<env,params> ")";

formalParamList<env,nilD> = ;

formalParams<env,params> = formalSection<env,params>;

formalParams<env,appendD(params1,params2)> =
    formalParams<env,params1> ";" formalSection<env,params2>;


formalSection<env,params> =
    formalGroup<valAc,env,params>;


formalSection<env,params> =
    "VAR" formalGroup<varAc,env,params>;


formalGroup<access,env,declNames(names,paramMode[access,type])> =
    nameList<names> ":" typeIdentifier<env,type>;



########################################
# Declarations
########################################


constDeclPart<env,locEnv,locEnv1> = "CONST" constantDecls<env,locEnv,locEnv1>;

constDeclPart<env,locEnv,locEnv> = ;


typeDeclPart<env,locEnv,locEnv1> = "TYPE" typeDecls<env,locEnv,locEnv1>;

typeDeclPart<env,locEnv,locEnv> = ;
```

```
varDeclPart<env,locEnv,locEnv1> = "VAR" variableDecls<env,locEnv,locEnv1>;

varDeclPart<env,locEnv,locEnv> = ;


block<env,locEnv1,\de0.com : FIX \de.dt de de0> =
    constDeclPart<env,locEnv1,locEnv2>
    typeDeclPart<env,locEnv2,locEnv3>
    varDeclPart<env,locEnv3,locEnv4>
    routineDecls<env1,env,locEnv4,locEnv,dt>
    compoundCommand<env1,com>
    WITH env1 = nest(locEnv,env);


programParams<nilN> = ;

programParams<names> = "(" nameList<names> ")";


program<\input.endProg (com de ([0->beginProg input] BOT[S]) 0)> =
    "program" ident<name> programParams<names> ";"
    block<env,nullEnv,com> "."

    WITH de = funcDef("abs", \int. IF int GE 0 THEN int ELSE -int FI) :
              funcDef("sqr", \int. int*int) :
              funcDef("odd", \int. IF (int MOD 2) EQ 0 THEN 0 ELSE 1 FI) :
              funcDef("ord", \int. int) :
              funcDef("chr", \int. int) :
              ["new"->
                    \s.LET ptr = 1 + (s 0 "$heapLimit" ! intVal)
                       IN store(1,"x") ptrVal[ptr] :
                          store(0,"$heapLimit") intVal[ptr] s]
              ["dispose"-> \s.store(1,"x") ptrVal[0] s ]
               \name.BOT[SX]

    WITH env = nest
               (["integer"->    typeMode[intType]]
                ["boolean"->    typeMode[boolType]]
                ["char"->       typeMode[charType]]
                ["text"->       typeMode[textType]]
                ["true"->       constMode[boolType,intVal[1]]]
                ["false"->      constMode[boolType,intVal[0]]]
                ["input"->      paramMode[varAc,fileTy[intType]]]
                ["output"->     paramMode[varAc,fileTy[intType]]]
                ["abs"->        stdFunc(intType,intType)]
                ["sqr"->        stdFunc(intType,intType)]
                ["odd"->        stdFunc(intType,boolType)]
                ["ord"->        stdFunc(charType,intType)]
                ["chr"->        stdFunc(intType,charType)]
                ["new"->        stdProc(ptrTy["$nil"])]
                ["dispose"->    stdProc(ptrTy["$nil"])]
                nullEnv,
                (-1, \name.(0,BOT[MODE])));


RESOLUTION
    nonassoc  "NOT";
    left  "*"  "DIV"  "MOD"  "AND";
    left  "+"  "-"  "OR";
    nonassoc  "<"  ">"  "<="  ">="  "="  "<>"  "IN";
    right  "ELSE";
END
```

Appendix E
Example of Axiomatic Semantics

Using a semantic grammar for the axiomatic semantics of a simple language, the compiler generator has produced the verification conditions of a division program. The language has integer and array variables, and the basic control structures, but no procedures. All expressions are integer; assertions and conditions controlling if or while commands may contain boolean connectives and integer comparisons. Every program must be documented with an input and output assertion; every loop must be documented with an invariant assertion.

```
# Semantic Grammar for program verification

DOMAIN

VAL = [intV[INT] + arrV[INT -> INT] ];  #values: integers and arrays
ENV = NAME -> TYPE;                     #environments: types of variables
TYPE = [intTy + arrayTy];               #types

#Expressions for assertions
EXP =   [ constE[INT]                   #constants
        + varE[NAME]                    #variables
        + plusE[EXP X EXP]              #integer operators
        + minusE[EXP X EXP]
        + timesE[EXP X EXP]
        + divE[EXP X EXP]
        + indexE[EXP X EXP]             #array operators
        + updateE[EXP X EXP]];


#Assertions (conditions)
COND =  [ ltC[EXP X EXP]                #integer comparisons
        + gtC[EXP X EXP]
        + eqC[EXP X EXP]
        + notC[COND]                    #boolean operators
        + andC[COND X COND]
        + orC[COND X COND]];


VCS = [nilV + consV[COND X VCS]];       #lists of verification conditions


FORWARD
substC  : (EXP X NAME X COND) -> COND;
substE  : (EXP X NAME X EXP) -> EXP;
appendV : (VCS X VCS) -> VCS;


DEFINE
```

```
#Substitute an expression for a variable within a condition
substC = \(exp,name,cond).
    CASE cond OF
        ltC[exp1,exp2]. ltC[substE(exp,name,exp1), substE(exp,name,exp2)],
        gtC[exp1,exp2]. gtC[substE(exp,name,exp1), substE(exp,name,exp2)],
        eqC[exp1,exp2]. eqC[substE(exp,name,exp1), substE(exp,name,exp2)],
        notC[cond].     notC[substC(exp,name,cond)],
        andC[cond1,cond2].
            andC[substC(exp,name,cond1), substC(exp,name,cond2)],
        orC[cond1,cond2].
            orC[substC(exp,name,cond1), substC(exp,name,cond2)]
    ESAC;


#Substitute an expression (exp) for a variable within an expression (exp0)
substE = \(exp,name,exp0).
    CASE exp0 OF
        constE[int].    exp0,
        varE[name1].    IF name EQ name1 THEN exp ELSE exp0 FI,
        plusE[exp1,exp2].
            plusE[substE(exp,name,exp1), substE(exp,name,exp2)],
        minusE[exp1,exp2].
            minusE[substE(exp,name,exp1), substE(exp,name,exp2)],
        timesE[exp1,exp2].
            timesE[substE(exp,name,exp1), substE(exp,name,exp2)],
        divE[exp1,exp2].
            divE[substE(exp,name,exp1), substE(exp,name,exp2)],
        indexE[exp1,exp2].
            indexE[substE(exp,name,exp1), substE(exp,name,exp2)],
        updateE[exp1,exp2].
            updateE[substE(exp,name,exp1), substE(exp,name,exp2)]
    ESAC;


#Append two lists of verification conditions
appendV = \(vcs1,vcs2).
    CASE vcs1 OF
        nilV. vcs2,
        consV[cond,vcs]. consV[cond, appendV(vcs,vcs2)]
    ESAC;


#Implication, defined in terms of OR and NOT
implC = \(cond1,cond2). orC[notC[cond1], cond2];


ATTRIBUTE
identifier<ENV.NAME,TYPE>;
expression<ENV.EXP>;
condition<ENV.COND>;
command<ENV,COND.COND,VCS>;
declaration<.ENV>;
program<.VCS>;


RULE program


identifier<env,name,env(name)> = ident<name>;


#### Expressions

expression<env, varE[name]>
    = identifier<env,name,intTy>;                    #integer variable

expression<env, indexE[varE[name], exp]> =           #subscripted variable
    identifier<env,name,arrayTy> "[" expression<env,exp> "]";
```

```
expression<env,exp> =
    "(" expression<env,exp> ")";                    #parenthesized expression

expression<env, constE[int]> = number<int>;         #integer constant

expression<env, plusE[exp1,exp2]>  =
    expression<env,exp1>  "+"  expression<env,exp2>;

expression<env, minusE[exp1,exp2]> =
    expression<env,exp1>  "-"  expression<env,exp2>;

expression<env, timesE[exp1,exp2]> =
    expression<env,exp1>  "*"  expression<env,exp2>;

expression<env, divE[exp1,exp2]>   =
    expression<env,exp1>  "/"  expression<env,exp2>;


#### Conditions (for IF and WHILE commands)

condition<env,cond> = "(" condition<env,cond> ")";

condition<env, ltC[exp1,exp2]> =
    expression<env,exp1>   "<"   expression<env,exp2>;

condition<env, gtC[exp1,exp2]> =
    expression<env,exp1>   ">"   expression<env,exp2>;

condition<env, eqC[exp1,exp2]> =
    expression<env,exp1>   "="   expression<env,exp2>;

condition<env, notC[cond]> =
    "NOT" condition<env,cond>;

condition<env, andC[cond1,cond2]> =
    condition<env,cond1> "AND" condition<env,cond2>;

condition<env, orC[cond1,cond2]> =
    condition<env,cond1> "OR"  condition<env,cond2>;


#### Commands  (statements)

command<env, cond, substC(exp,name,cond), nilV> =
    identifier<env,name,intTy> ":=" expression<env,exp>;


command<env, cond, substC(updateE[varE[name],exp1], name, cond), nilV> =
    identifier<env,name,arrayTy>
    "[" expression<env,exp1> "]"  ":=" expression<env,exp2>;


command<env,cond,cond1,appendV(vcs1,vcs2)> =
    command<env,cond2,cond1,vcs1> ";" command<env,cond,cond2,vcs2>;


command<env,cond, andC[implC(cond0,cond1), implC(notC[cond0],cond)], vcs> =
    "IF" condition<env,cond0> "THEN" command<env,cond,cond1,vcs> "FI";


command<env, cond, cond2,
     consV[andC[ implC(andC[cond1,cond2],cond3),
                 implC(andC[notC[cond1],cond2], cond) ], vcs]> =
    "WHILE" condition<env,cond1> "invariant" condition<env,cond2>
    "DO" command<env,cond2,cond3,vcs> "OD";


#### Declarations
```

```
declaration<["input"->intTy] ["output"->intTy] BOT[ENV]> = ;

declaration<[name->intTy]env>   =
    "INT"   ident<name>  ";"  declaration<env>;

declaration<[name->arrayTy]env> =
    "ARRAY" ident<name>  ";"  declaration<env>;

program<consV[implC(cond1,cond), vcs]> =
    "BEGIN" declaration<env>
    "BEFORE" condition<env,cond1>
    "AFTER" condition<env,cond2> ";"
      command<env,cond2,cond,vcs>
    "END";


RESOLUTION
    NONASSOC  "NOT";
    LEFT   "*"   "/"   "AND";
    LEFT   "+"   "-"   "OR";
    NONASSOC  "<"   ">"   "=";
    RIGHT ";";
END
```

Given this grammar, the universal translator converts an input program into a list of verification conditions. Consider a program that performs integer division.

```
{Program to divide a by d, producing quotient q and remainder r;
 Algorithm is repeated subtraction.}
BEGIN
INT a;  INT d;  INT q;  INT r;
BEFORE  NOT a<0
AFTER   (NOT r<0) AND r < d AND a = d*q+r;

r := a;  q := 0;
WHILE NOT r<d   INVARIANT (NOT r<0) AND a = d*q + r
    DO  r := r - d;  q := q + 1  OD
END
```

The universal translator prints the simplified DAG, which represents the verification conditions. They are all true, proving partial correctness of the program. The verification conditions must be simplified by hand; for practical program verification, the compiler generator should provide a standard assertion language and theorem prover.

The semantic DAG:

```
CONSV[ORC[NOTC[NOTC[LTC[VARE["A"], CONSTE[0]]]],
         ANDC[NOTC[LTC[VARE["A"], CONSTE[0]]],
              EQC[VARE["A"], PLUSE[TIMESE[VARE["D"], CONSTE[0]], VARE["A"]]]]],
      CONSV[ANDC[ORC[NOTC[ANDC[NOTC[LTC[VARE["R"], VARE["D"]]],
                               ANDC[NOTC[LTC[VARE["R"], CONSTE[0]]],
                                    EQC[VARE["A"],
                                        PLUSE[TIMESE[VARE["D"], VARE["Q"]],
                                              VARE["R"]]]]]]],
                 ANDC[NOTC[LTC[MINUSE[VARE["R"], VARE["D"]], CONSTE[0]]],
```

```
                EQC[VARE["A"],
                    PLUSE[TIMESE[VARE["D"],
                                 PLUSE[VARE["Q"], CONSTE[1]]],
                          MINUSE[VARE["R"], VARE["D"]]]]],
          ORC[NOTC[ANDC[NOTC[NOTC[LTC[VARE["R"], VARE["D"]]]],
                        ANDC[NOTC[LTC[VARE["R"], CONSTE[0]]],
                             EQC[VARE["A"],
                                 PLUSE[TIMESE[VARE["D"], VARE["Q"]],
                                       VARE["R"]]]]]],
              ANDC[ANDC[NOTC[LTC[VARE["R"], CONSTE[0]]],
                        LTC[VARE["R"], VARE["D"]]],
                   EQC[VARE["A"],
                       PLUSE[TIMESE[VARE["D"], VARE["Q"]], VARE["R"]]]]]],
    NILV]]
```

Unsimplified verification conditions:

$$\neg(\neg(a<0)) \vee (\neg(a<0) \wedge (a = d*0 + a))$$

$$[\neg((\neg((r<d)) \wedge (\neg((r<0)) \wedge (a = (d*q + r)))))$$
$$\vee$$
$$(\neg(((r-d)<0)) \wedge (a = ((d*(q+1)) + (r-d))))]$$
$$\wedge$$
$$[\neg((\neg(\neg((r<d))) \wedge (\neg((r<0)) \wedge (a = (d*q + r)))))$$
$$\vee$$
$$((\neg((r<0)) \wedge (r<d)) \wedge (a = (d*q + r)))]$$

Simplified verification conditions:

$$a \geq 0 \Rightarrow (a \geq 0 \wedge a = a)$$

$$[(r \geq d \wedge r \geq 0 \wedge a = (d*q + r)) \Rightarrow$$
$$(r \geq d \wedge a = (d*(q+1) + r - d))]$$
$$\wedge$$
$$[((r<d \wedge r \geq 0 \wedge a = (d*q + r))) \Rightarrow$$
$$(r \geq 0 \wedge r<d \wedge a = (d*q + r))]$$

Appendix F
Semantic Grammars Grammar

```
# SG.SG   Semantic grammar for semantic grammars
# Describes a large subset of the notation accepted by the Grammary Analyzer.
# Checks domain compatibility, but does not check that the grammar is
#       syntactically or semantically valid.
#
# Uncorrectable problems due to limitations of the lexical scanner:
#    Trailing digits of variable names are not stripped off;
#       domain definitions like DOM1=DOM are required
#       (note: the Grammar Analyzer does not allow domain names to contain
#       digits)
#    Quoted strings are restricted to identifiers;  likewise, terminals are
#       restricted to identifiers or strings of special characters.
#       Note that keywords defined in this grammar are not considered
#       identifiers, so "IF", "TRUE", etc. may not be terminal symbols.

DOMAIN

# Domain representing a domain

DOM = [ unspecD[NAME] +
        nameD[NAME] +
        productD[DOM X DOM] +
        functionD[DOM X DOM] +
        unionD[NAME X SUMMANDS]];

# Summands of a union domain
SUMMANDS = [nilS + consS[TAG X DOM X SUMMANDS]];
TAG = NAME;                                     #Tagfields
UNION = DOM;                                    #Union domains

#Modes of identifiers
MODE = [undefined +
        domainMode[DOM] +
        tagMode[DOM X UNION] +                  #tag field
        forwardMode[DOM] +                      #FORWARD-declared
        expMode[DOM] +                          #DEFINE expression
        varMode[DOM] +                          #bound variable
        attributeMode[DOM] +
        symbolMode[ATTRS]];                     #nonterminal symbol


#Attribute information for a nonterminal symbol
FLOW = [synthesized + inherited];
ATTRS = [nilA + consA[DOM X FLOW X ATTRS]];

#Environments
ENV = NAME -> MODE;
FINALENV = ENV;                         #Final environment after domain defs
VARENV = ENV;                           #Environment of bound variables

NAMESET = NAME -> BOOL;                 #Sets of names for equalValid

FORWARD
```

```
noname          : (ENV X DOM) -> DOM;
addUnion        : (SUMMANDS X ENV X UNION) -> ENV;
compatible      : (ENV X DOM X DOM) -> BOOL;
equalValid      : (ENV X NAMESET X DOM) -> BOOL;
eqValidS        : (ENV X NAMESET X SUMMANDS) -> BOOL;


DEFINE

#Predefined domains
voidDom = unspecD["VOID"];              #Void summands of unions
intDom = unspecD["INT"];
boolDom = unspecD["BOOL"];
nameDom = unspecD["NAME"];


#Skip past references to named domains, get actual definition
noname = \(env,dom).
        IF dom IS nameD THEN noname(env,env(dom!nameD)!domainMode)
        ELSE dom FI;


#Add a union domain's tags to the environment
addUnion = \(summands,env,union).
    CASE summands OF
        nilS. env,
        consS[tag,dom,summands].
            addUnion(summands, [tag->tagMode[dom,union]]env, union)
    ESAC;


#Empty environment; contains standard definitions
nullEnv = ["VOID"->       domainMode[voidDom]]
          ["INT" ->       domainMode[intDom]]
          ["BOOL"->       domainMode[boolDom]]
          ["NAME"->       domainMode[nameDom]]
          ["where"->      symbolmode[consA[boolDom,inherited,nilA]]]
          ["ident"->      symbolmode[consA[nameDom,synthesized,nilA]]]
          ["uniqueName"->symbolmode[consA[nameDom,synthesized,nilA]]]
          ["number"->     symbolmode[consA[intDom,synthesized,nilA]]]
          \name.undefined;


#Check that two domains are compatible
compatible = \(env,dom1,dom2).
    IF dom1 EQ dom2 THEN TRUE
    ELSE IF dom2 IS nameD THEN
        compatible(env,dom1, env(dom2!nameD)!domainMode)
    ELSE
        CASE dom1 OF
            unspecD[name1]. name1 EQ (dom2!unspecD),
            nameD[name1]. compatible(env, env(dom1!nameD)!domainMode, dom2),
            productD[dom3,dom4].
                LET (dom5,dom6) = dom2!productD IN
                    compatible(env,dom3,dom5) AND
                    compatible(env,dom4,dom6),
            functionD[dom3,dom4].
                LET (dom5,dom6) = dom2!functionD IN
                    compatible(env,dom3,dom5) AND
                    compatible(env,dom4,dom6),
            unionD[name1,summands1].
                LET (name2,summands2) = dom2!unionD IN
                    name1 EQ name2
        ESAC
    FI FI;


#Test whether equality is defined for this domain
#Equality is defined for all non-functional domains
#The nameSet prevents looping over recursive domains --
```

```
#  it contains the named domains already visited
equalValid = \(env,nameSet,dom).
    CASE dom OF
        unSpecD[name]. TRUE,
        nameD[name].
            IF nameSet(name) THEN TRUE
            ELSE equalValid(env, [name->TRUE]nameSet, env(name)!domainMode)
            FI,
        productD[dom1,dom2].
            equalValid(env,nameSet,dom1) AND equalValid(env,nameSet,dom2),
        functionD[dom1,dom2]. FALSE,
        unionD[name,summands]. eqValidS(env,nameSet,summands)
    ESAC;


#Test whether equality is defined on a union domain
#   -- apply equalValid to its summands
eqValidS = \(env,nameSet,summands).
    CASE summands OF
        nilS. TRUE,
        consS[tag,dom,summands].
            equalValid(env,nameSet,dom) AND eqValidS(env,nameSet,summands)
    ESAC;


ATTRIBUTE

newName         <ENV.NAME>;
identifier      <ENV.NAME,MODE>;

Domain          <ENV.DOM>;                          #Domains
def             <ENV.DOM>;
summandList     <ENV.SUMMANDS>;
summand         <ENV.TAG,DOM>;

expression      <ENV,VARENV.DOM>;                   #Expressions
term            <ENV,VARENV.DOM>;
factor          <ENV,VARENV.DOM>;
caseBoundVars   <ENV,VARENV.DOM,VARENV>;
caseArm         <ENV,VARENV,TAG.DOM>;
caseList        <ENV,VARENV,SUMMANDS.DOM>;
lambdaTail      <ENV,VARENV.DOM>;
tuple           <ENV,VARENV.DOM>;
integerOp       <.>;
booleanOp       <.>;
relationOp      <.>;
equalsOp        <.>;
prefixOp        <DOM.DOM>;
boundVars       <ENV,VARENV.DOM,VARENV>;
boundV          <ENV,VARENV.DOM,VARENV>;

nonterminal     <ENV.>;                             #Grammar rules
attr            <ENV.DOM>;
attributes      <ENV,ATTRS.>;
rightSide       <ENV.>;
withClause      <ENV.>;
terminal        <.>;
delimeter       <.>;
delimChar       <.>;

domainPart      <.ENV>;                             #Declarations
domainDefs      <FINALENV.ENV>;
domDefin        <FINALENV,ENV.ENV>;
forwardPart     <FINALENV,ENV.ENV>;
forwardDecls    <FINALENV,ENV.ENV>;
definePart      <ENV.ENV>;
definitions     <ENV.ENV>;
attributePart   <ENV.ENV>;
attributeDecls  <ENV.ENV>;
attributeList   <ENV,FLOW,ATTRS.ATTRS>;
rulePart        <ENV.>;
```

```
ruleList        <ENV.>;
resolutionPart  <.>;
resolutions     <.>;
assocKind       <.>;
terminalList    <.>;
grammar         <.ENV>;


RULE grammar


newName<env,name> = ident<name>  WHERE<env(name) IS undefined>;

identifier<env,name,env(name)> = ident<name>;


#############################################################
#### DOMAINS
#############################################################

#Domains are evaluated in an environment that includes the domain definitions,
#to allow forward references to domains.  See declarations section.


#Check that the name is defined to be a domain
sDomain<env,nameD[name]> = identifier<env,name,domainMode[dom]>;


sDomain<env,dom> = "(" sDomain<env,dom> ")";


sDomain<env,productD[dom1,dom2]> =
    sDomain<env,dom1> "X" sDomain<env,dom2>;


sDomain<env,functionD[dom1,dom2]> =
    sDomain<env,dom1> "->" sDomain<env,dom2>;


#Summand lists for union domains
#This grammar only allows unions at the top level of domain definitions

summandList<env,consS[tag,dom,nilS]> =
    summand<env,tag,dom>;

summandList<env,consS[tag,dom,summands]> =
    summand<env,tag,dom> "+" summandList<env,summands>;


summand<env,tag,dom> =                          #Summand with domain
    ident<tag> "[" sDomain<env,dom> "]";


summand<env,tag,voidDom> =                      #Void summand
    ident<tag>;


#############################################################
#### EXPRESSIONS
#############################################################

expression<env,varEnv,dom> = term<env,varEnv,dom>;


#### Infix operators


expression<env,varEnv,boolDom> =
    term<env,varEnv,union1> "IS" identifier<env,tag, tagMode[dom,union2]>
    WHERE<compatible(env,union1,union2)>;
```

```
expression<env,varEnv,noname(env,dom)> =
    term<env,varEnv,functionD[dom1,dom]> ":" expression<env,varEnv,dom2>
                                        WHERE<compatible(env,dom1,dom2)>;

#Integer operators
expression<env,varEnv,intDom> =
    term<env,varEnv,intDom> integerOp<> expression<env,varEnv,intDom>;

integerOp<> = "+";
integerOp<> = "-";
integerOp<> = "*";
integerOp<> = "DIV";
integerOp<> = "MOD";


#Boolean operatos
expression<env,varEnv,boolDom> =
    term<env,varEnv,boolDom> booleanOp<> expression<env,varEnv,boolDom>;

booleanOp<> = "AND";
booleanOp<> = "OR";


#Integer relations
expression<env,varEnv,boolDom> =
    term<env,varEnv,intDom> relationOp<> expression<env,varEnv,intDom>;

relationOp<> = "LT";
relationOp<> = "LE";
relationOp<> = "GT";
relationOp<> = "GE";


#Equality testing
expression<env,varEnv,boolDom> =
    term<env,varEnv,dom1> equalsOp<> expression<env,varEnv,dom2>
                WHERE<compatible(env,dom1,dom2)>
                WHERE<equalValid(env, \name.FALSE, dom1)>;

equalsOp<> = "EQ";
equalsOp<> = "NE";


# Prefix operators

expression<env,varEnv,noname(env,dom1)> =
    prefixOp<dom,dom1>  expression<env,varEnv,dom>;

prefixOp<productD[dom1,dom2], dom1> = "LEFT";

prefixOp<productD[dom1,dom2], dom2> = "RIGHT";

prefixOp<boolDom,boolDom> = "NOT";

prefixOp<intDom,intDom> = "-";


#Function update
expression<env,varEnv,dom> =
    "[" expression<env,varEnv,dom1>  "->"  expression<env,varEnv,dom2> "]"
    expression<env,varEnv,dom>
                WHERE<dom IS functionD>
                WHERE<compatible(env,dom, functionD[dom1',dom2])>;


#Conditional expression
expression<env,varEnv,dom2> =
    "IF" expression<env,varEnv,boolDom>
```

```
    "THEN" expression<env,varEnv,dom1>
    "ELSE" expression<env,varEnv,dom2>  "FI"
              WHERE<compatible(env,dom1,dom2)>;


####Bound variables for CASE, lambda, LET, LETREC

boundVars<env,varEnv,dom,varEnv1> = boundV<env,varEnv,dom,varEnv1>;

#Tuple of bound variables
boundVars<env,varEnv,productD[dom1,dom2], varEnv2> =
    boundV<env,varEnv,dom1,varEnv1> ","
    boundVars<env,varEnv1,dom2,varEnv2>;

boundV<env,varEnv,dom,varEnv1> = "(" boundVars<env,varEnv,dom,varEnv1> ")";

#Implicit domain from variable name
boundV<env,varEnv, dom, [name->varMode[dom]]varEnv> =
    identifier<env,name,domainMode[dom]>;

#Explicit domain
boundV<env,varEnv, dom, [name->varMode[dom]]varEnv> =
    ident<name>  ":"  sDomain<env,dom>;


#### CASE expression

expression<env,varEnv,dom> =
    "CASE" expression<env,varEnv,unionD[name,summands]> "OF"
        caseList<env,varEnv,summands,dom>  "ESAC" ;


caseList<env,varEnv, consS[tag,dom9,summands], dom1> =
    caseArm<env,varEnv,tag,dom1>  ","
    caseList<env,varEnv,summands,dom2>
    WHERE<compatible(env,dom1,dom2)>;


caseList<env,varEnv, consS[tag,dom9,nilS], dom> =
    caseArm<env,varEnv,tag,dom>;


caseArm<env,varEnv,tag,dom> =
    identifier<env,tag, tagMode[dom1,union]>
    caseBoundVars<env,varEnv,dom2,varEnv1>  "."
    expression<env,varEnv1,dom> ;


caseBoundVars<env,varEnv,voidDom,varEnv> = ;

caseBoundVars<env,varEnv,dom,varEnv1> =
    "[" boundVars<env,varEnv,dom,varEnv1> "]" ;


#Lambda-abstraction

expression<env,varEnv,dom> =
    "\" lambdaTail<env,varEnv,dom>;

lambdaTail<env,varEnv,functionD[dom1,dom2]> =
    boundVars<env,varEnv,dom1,varEnv1>
    lambdaTail<env,varEnv1,dom2>;

lambdaTail<env,varEnv,functionD[dom1,dom2]> =
    boundVars<env,varEnv,dom1,varEnv1> "." expression<env,varEnv1,dom2>;


#Fixedpoint expression
expression<env,varEnv,dom2> =
```

```
    "FIX" "\" boundVars<env,varEnv,dom1,varEnv1> "."
     expression<env,varEnv1,dom2>
                WHERE<compatible(env,dom1,dom2)>;

####Local definitions LET and LETREC

expression<env,varEnv,dom> =
    "LET" boundVars<env,varEnv,dom1,varEnv1>
    "=" expression<env,varEnv,dom2>     WHERE<compatible(env,dom1,dom2)>
    "IN" expression<env,varEnv1,dom>;


expression<env,varEnv,dom> =
    "LETREC" boundVars<env,varEnv,dom1,varEnv1>
    "=" expression<env,varEnv1,dom2>    WHERE<compatible(env,dom1,dom2)>
                                        WHERE<dom2 IS functionD>
    "IN" expression<env,varEnv1,dom>;


#### Terms

term<env,varEnv,noname(env,dom)> = factor<env,varEnv,dom>;


#Function application
term<env,varEnv, noname(env,dom)> =
    term<env,varEnv,functionD[dom1,dom]>  factor<env,varEnv,dom2>

    WHERE<compatible(env,dom1,dom2)>;


#Projection from a union domain
term<env,varEnv, noname(env,dom)> =
    term<env,varEnv,union1> "!"  identifier<env,tag, tagMode[dom,union2]>

    WHERE<compatible(env,union1,union2)>;


#### Factors


factor<env,varEnv,dom> = "("  tuple<env,varEnv,dom>  ")" ;


tuple<env,varEnv,dom> = expression<env,varEnv,dom>;

tuple<env,varEnv,productD[dom1,dom2]> =
    expression<env,varEnv,dom1> ","  tuple<env,varEnv,dom2>;


#Constants

factor<env,varEnv,boolDom> = "TRUE";

factor<env,varEnv,boolDom> = "FALSE";

factor<env,varEnv,intDom> = number<int>;

factor<env,varEnv,nameDom> = """ ident<name> """;


#Identifiers in expressions can be:
#   void tag injections
#   named expressions (FORWARD or DEFINE)
#   bound variables
#   attribute variables
factor<env,varEnv,dom> = identifier<varEnv,name,mode>
    WITH dom =
        CASE mode OF
```

```
        undefined. BOT[DOM],
        domainMode[dom]. BOT[DOM],
        tagMode[dom,union]. IF dom EQ voidDom THEN union ELSE BOT[DOM] FI,
        forwardMode[dom]. dom,
        expMode[dom]. dom,
        varMode[dom]. dom,
        attributeMode[dom]. dom,
        symbolMode[attrs]. BOT[DOM]
        ESAC;


#Injection to a union domain
factor<env,varEnv,union> =
    identifier<env,tag,tagMode[dom1,union]>
                                        WHERE<compatible(env,dom1,dom)>
    "["  tuple<env,varEnv,dom>  "]" ;


#The bottom element of a domain
factor<env,varEnv,dom> = "BOT" "[" sDomain<env,dom> "]";



#############################################################
#####  RULES
#############################################################


#Right sides of rules

rightSide<env> = ;

rightSide<env> = rightSide<env>  nonterminal<env>;

rightSide<env> = rightSide<env>  terminal<>;


nonterminal<env> =
    identifier<env, name, symbolMode[attrs]>
    "<"  attributes<env,attrs>  ">" ;


#Lists of attribute expressions

attributes<env,nilA> = ;

attributes<env, consA[dom1,flow,nilA]> =
    attr<env,dom2>      WHERE<compatible(env,dom1,dom2)>;


attributes<env, consA[dom1,flow,attrs]> =
    attr<env,dom2>      WHERE<compatible(env,dom1,dom2)>
    ","  attributes<env,attrs> ;


#Attribute expression
#    Its variable environment contains all possible attribute names
attr<env,dom> =
    expression<env,varEnv,dom>
    WITH varEnv =
        \name. LET mode = env(name) IN
                IF mode IS domainMode THEN attributeMode[mode!domainMode]
                ELSE mode FI;


withClause<env> = ;

withClause<env> =
    withClause<env>
    "WITH"  attr<env,dom1>  "="  attr<env,dom2>
```

```
                WHERE<compatible(env,dom1,dom2)>;


terminal<> =    """ ident<name> """;

terminal<> =    """ delimeter<> """;

delimeter<> = delimChar<>;

delimeter<> = delimeter<>  delimChar<>;

delimChar<> = "!" ;    delimChar<> = "#" ;    delimChar<> = "%" ;
delimChar<> = "$" ;    delimChar<> = "↑" ;    delimChar<> = "&" ;
delimChar<> = "*" ;    delimChar<> = "(" ;    delimChar<> = ")" ;
delimChar<> = "+" ;    delimChar<> = "-" ;    delimChar<> = "=" ;
delimChar<> = "[" ;    delimChar<> = "]" ;    delimChar<> = ";" ;
delimChar<> = ":" ;    delimChar<> = "'" ;    delimChar<> = "<" ;
delimChar<> = ">" ;    delimChar<> = "," ;    delimChar<> = "." ;
delimChar<> = "/" ;    delimChar<> = "?" ;


#############################################################
#####  DECLARATIONS
#############################################################

# Domain definitions
#    In the domain definition part, a named domain may be referenced before
#    its definition.  To check that each named domain is eventually defined,
#    the final environment is passed back through the domain definitions as
#    finalEnv.  Each named domain checks that the name is defined in the
#    finalEnv.

domainPart<nullEnv> = ;

domainPart<env> = "DOMAIN"  domainDefs<env,env>;


domainDefs<finalEnv,nullEnv> = ;

domainDefs<finalEnv,env1> =
    domainDefs<finalEnv,env>  domDefin<finalEnv,env,env1>  ";" ;


#UNSPEC domain
domDefin<finalEnv, env, [name->domainMode[unspecD[name]]]env> =
    newName<env,name>  "="  "UNSPEC";


#Union domain
domDefin<finalEnv, env,
        [name->domainMode[union]]
           addUnion(summands,env,union)> =
    newName<env,name> "=" "[" summandList<finalEnv,summands> "]"
    WITH union = unionD[name,summands];


#Ordinary domain
domDefin<finalEnv, env, [name->domainMode[dom]]env> =
    newName<env,name> "=" sDomain<finalEnv,dom>;


#### Forward declarations

# The FINALENV checks that all Forward-declared names are eventually defined.
# FinalEnv contains the environment after the DEFINE part.

forwardPart<finalEnv,env,env> = ;

forwardPart<finalEnv,env1,env2> = "FORWARD" forwardDecls<finalEnv,env1,env2>;
```

```
forwardDecls<finalEnv,env,env> = ;

forwardDecls<finalEnv,env, [name->forwardMode[dom]] env2> =
    forwardDecls<finalEnv,env,env2>
    identifier<finalEnv,name,expMode[dom99]>
            WHERE<env2(name) IS undefined>
            WHERE<compatible(env,dom,dom99)>
    ":"  sDomain<env2,dom>  ";" ;


#### Expression definitions

definePart<env,env> = ;

definePart<env1,env2> =  "DEFINE"  definitions<env1,env2>;


definitions<env,env> = ;


definitions<env1, [name->expMode[dom]] env2> =
    definitions<env1,env2>
    ident<name> "=" def<env2,dom>  ";"

    #May redeclare FORWARD name
    WHERE<LET mode = env2(name) IN
            (mode IS undefined) OR (mode IS forwardMode)>;


def<env,dom> = expression<env,env,dom>;

def<env,dom> = "UNSPEC"  ":"  sDomain<env,dom>;


#### Attribute declarations

attributePart<env,env> = ;

attributePart<env1,env2> =   "ATTRIBUTE"  attributeDecls<env1,env2>;


attributeDecls<env,env> = ;


attributeDecls<env, [name -> symbolMode[attrs2]] env2> =
    attributeDecls<env,env2>
    newName<env2,name>
    "<"  attributeList<env,inherited,attrs,attrs2>  "."
    attributeList<env,synthesized,nilA,attrs>  ">"
    ";" ;


#Lists of attribute domains
attributeList<env,flow,attrs,attrs> = ;

attributeList<env,flow,attrs, consA[dom,flow,attrs]> =
    sDomain<env,dom>;

attributeList<env,flow,attrs1, consA[dom,flow,attrs2]> =
    sDomain<env,dom>  ","  attributeList<env,flow,attrs1,attrs2>;


#### Semantic Rules

rulePart<env> =
    "RULE" identifier<env,name,symbolMode[attrs]>
    ruleList<env>;
```

```
ruleList<env> = ;

ruleList<env> =
    ruleList<env>
    nonterminal<env>  "="  rightSide<env>  withClause<env>  ";" ;


#### Resolution Part


resolutionPart<> = ;

resolutionPart<> =
    "RESOLUTION"   resolutions<> ;


resolutions<> = ;

resolutions<> =   resolutions<>  assocKind<>  terminalList<>  ";"  ;

assocKind<> = "LEFT";
assocKind<> = "RIGHT";
assocKind<> = "NONASSOC";

terminalList<> = terminal<>;

terminalList<> =  terminalList<>  terminal<> ;


grammar<env> =
    domainPart<env1>
    forwardPart<env3,env1,env2>
    definePart<env2,env3>
    attributePart<env3,env>
    rulePart<env>
    resolutionPart<>
    "END" ;


RESOLUTION
    RIGHT  "X"  "->" ;

END
```

References

[1] Alfred V. Aho, Jeffrey D. Ullman.
Principles of Compiler Design.
Addison-Wesley, 1978.

[2] American National Standards Institute.
Clarification of Fortran Standards: Second Report.
Communications of the ACM , October, 1971.

[3] American Standards Association.
Fortran vs. Basic Fortran.
Communications of the ACM , October, 1964.

(4) F. L. Bauer, J. Eickel (editors).
Compiler Construction: An Advanced Course.
Springer-Verlag, 1976.

[5] Gregor V. Bochmann.
Semantic Evaluation from Left to Right.
Communications of the ACM :55 – 62, February, 1976.

[6] W. H. Burge.
Recursive Programming Techniques.
Addison-Wesley, 1976.

[7] O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare.
Structured Programming.
Academic Press, 1972.

[8] Department of Defense.
STEELMAN Requirements for High Order Computer Programming Languages
1978.

[9] James E. Donahue.
Complementary Definitions of Programming Language Semantics.
Springer-Verlag, 1976.

[10] J. H. Eriksen, B. B. Jensen, B. B. Kristensen, O. L. Madsen.
The BOBS-System.
Technical Report DAIMI PB-71, Computer Science Department, Aarhus University, Denmark, February, 1979.

[11] Michael Gordon.
The Denotational Description of Programming Languages: An Introduction.
Springer-Verlag, 1979.

[12] Michael Gordon, R. Milner, C. Wadsworth.
Edinburgh LCF.
Springer-Verlag, 1979.

[13] Susan L. Graham, Charles B. Haley, William N. Joy.
Practical LR Error Recovery.
SIGPLAN Notices 14:168 – 175, August, 1979.
Proceedings of the SIGPLAN Symposium on Compiler Construction.

[14] John L. Hennessy.
The Stanford Pascal Parser Generator.
Technical Report, Stanford University, 1981.
In preparation.

[15] C. A. R. Hoare.
Hints on Programming Language Design.
Technical Report CS-403, Computer Science Department, Stanford University, October, 1973.

[16] C. A. R. Hoare, N. Wirth.
An Axiomatic Definition of the Programming Language Pascal.
Acta Informatica :335 – 355, 1973.

[17] J. D. Ichbiah et al.
Preliminary Ada Reference Manual.
SIGPLAN Notices , June, 1979.

[18] Mehdi Jazayeri, William F. Ogden, William C. Rounds.
The Intrinsically Exponential Complexity of the Circularity Problem for Attribute Grammars.
Communications of the ACM :697 – 706, December, 1975.

[19] Kathleen Jensen, Niklaus Wirth.
Pascal User Manual and Report.
Springer-Verlag, 1975.

[20] P. Jesperson, M. Madsen, H. Riis.
NEATS, New Extended Attribute Translation System.
Technical Report, Computer Science Department, Aarhus University, Denmark, 1978.
Cited in Watt and Madsen [63].

[21] Stephen C. Johnson.
YACC: Yet Another Compiler-Compiler.
Technical Report CSTR 32, Bell Laboratories, Murray Hill, New Jersey, July, 1975.

[22] Neil D. Jones (editor).
Semantics-Directed Compiler Generation.
Springer-Verlag, 1980.

[23] Neil D. Jones, Henning Christiansen.
Control Flow Treatment in a Simple Semantics-Directed Compiler Generator.
Technical Report PB-137, Aarhus University, Denmark, 1981.

[24] Neil D. Jones, Michael Madsen.
Attribute-Influenced LR Parsing.
Pages 393 – 407 of Jones [22], 1980.

[25] Neil D. Jones, David A. Schmidt.
Compiler Generation from Denotational Semantics.
Pages 70 – 93 of Jones [22], 1980.

[26] D. E. Knuth.
The Remaining Trouble Spots in Algol 60.
Communications of the ACM 10:611 – 618, October, 1967.

[27] D. E. Knuth.
Semantics of Context-Free Languages.
Mathematical Systems Theory 2:127 – 145, February, 1968.
Corrected in 1971 volume, pages 95 – 96.

[28] C. H. A. Koster.
Affix Grammars.
In J. E. L. Peck (editor), *Algol 68 Implementation*, pages 95 – 109. North-Holland, 1971.

[29] C. H. A. Koster.
Using the CDL Compiler-Compiler.
Pages 366 – 426 of Bauer and Eickel [4], 1976.

[30] P. J. Landin.
The Mechanical Evaluation of Expressions.
Computer Journal 6:308 – 320, April, 1964.

[31] Ole L. Madsen.
On Defining Semantics by Means of Extended Attribute Grammars.
Technical Report DAIMI PB-109, Computer Science Department, Aarhus University, Denmark, January, 1980.
Pages 259 – 299 of Jones [22].

[32] Ole L. Madsen.
Towards a Practical and General Translator Writing System.
Technical Report, Computer Science Department, Aarhus University, Denmark, March, 1980.

[33] Zohar Manna.
Mathematical Theory of Computation.
McGraw-Hill, 1974.

[34] M. Marcotty, H. F. Ledgard, G. V. Bochmann.
A Sampler of Formal Definitions.
Computing Surveys :191 – 276, June, 1976.

[35] Brian H. Mayoh.
Attribute Grammars and Mathematical Semantics.
Technical Report DAIMI PB-90, Computer Science Department, Aarhus University, Denmark, August, 1978.

[36] John McCarthy.
Towards a Mathematical Science of Computation.
In C. M. Popplewell (editor), *Information Processing 62*, pages 21 – 28. North-Holland, 1963.

[37] R. E. Milne, C. Strachey.
A Theory of Programming Language Semantics.
John Wiley, 1976.
Published in Great Britain by Chapman and Hall.

[38] D. R. Milton, L. W. Kirchhoff, B. R. Rowland.
An ALL(1) Compiler Generator.
SIGPLAN Notices 14:152 – 157, August, 1979.
Proceedings of the SIGPLAN Symposium on Compiler Construction.

[39] Peter D. Mosses.
The Mathematical Semantics of Algol 60.
Technical Report PRG-012, Programming Research Group, Oxford University, 1974.

[40] Peter D. Mosses.
SIS: Semantics Implementation System. Reference Manual and User Guide.
Technical Report DAIMI MD-33, Computer Science Department, Aarhus University, Denmark, 1979.

[41] Peter D. Mosses.
SIS: Semantics Implementation System. Tested Examples.
Technical Report DAIMI MD-30, Computer Science Department, Aarhus University, Denmark, 1979.

[42] Peter D. Mosses.
Mathematical Semantics and Compiler Generation.
PhD thesis, Oxford University, 1975.

[43] P. Naur, J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samuelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, M. Woodger.
Revised Report on the Algorithmic Language Algol 60.
Communications of the ACM 6:1 – 17, January, 1963.

[44] Derek C. Oppen.
Pretty Printing.
ACM Transactions on Programming Languages and Systems :465 – 483, October, 1980.

[45] Lawrence Paulson, John Hennessy.
Semantic Grammars: A Formalism for Defining the Syntax and Semantics of Programming Languages.
Technical Report, Stanford University, 1980.
Submitted to the *Journal of the ACM.*

[46] G. D. Plotkin.
Call-by-name, Call-by-value, and the λ-Calculus.
Theoretical Computer Science :125 – 159, 1975.

[47] Peter C. Poole.
Portable and Adaptable Compilers.
Pages 427 – 497 of Bauer and Eickel [4], 1976.

[48] D. P. Pozefsky.
Building Efficient Pass-Oriented Attribute Grammar Evaluators.
PhD thesis, University of North Carolina at Chapel Hill, 1979.

[49] Kari-Jouko Räihä, Mikko Saarinen, Eljas Soisalon-Soininen, Martti Tienari.
The Compiler Writing System HLP (Helsinki Language Processor).
Technical Report A-1978-2, Department of Computer Science, University of Helsinki, Finland, 1978.

[50] Kari-Jouko Räihä.
Experiences with the Compiler Writing System HLP.
Pages 350 – 362 of Jones [22], 1980.

[51] Kari-Jouko Räihä.
Bibliography on Attribute Grammars.
SIGPLAN Notices 15:35 – 44, March, 1980.

[52] Martin Raskovsky.
Step by Step Generation of a Compiler for Flow Diagram Language with Jumps.
Technical Report CSM-42, Department of Computer Science, University of Essex, June, 1981.

[53] Martin Raskovsky, Phil Collier.
From Standard to Implementation Denotational Semantics.
Pages 94 – 139 of Jones [22], 1980.

[54] A. H. J. Sale.
Pascal Compatibility Report.
Technical Report R77-5, Department of Information Science, University of Tasmania, November, 1977.

[55] R. W. Scheifler.
A Denotational Semantics of CLU.
Technical Report TR-201, Laboratory for Computer Science, M.I.T., 1978.

[56] Ravi Sethi.
Circular Expressions: Elimination of Static Environments.
In S. Even, O. Kariv (editors), *Eighth International Colloquium on Automata, Languages and Programming*, pages 378 – 392. Springer-Verlag, 1981.

[57] Ravi Sethi.
Control Flow Aspects of Semantics Directed Compiling.
Technical Report, Bell Laboratories, Murray Hill, New Jersy 07974, 1981.

[58] Joseph E. Stoy.
Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.
M. I. T. Press, 1978.

[59] R. D. Tennent.
Mathematical Semantics of SNOBOL4.
In *Symposium on Principles of Programming Languages*, pages 95 – 107. Association for Computing Machinery, 1973.

[60] R. D. Tennent.
The Denotational Semantics of Programming Languages.
Communications of the ACM :437 – 453, August, 1976.

[61] R. D. Tennent.
A Denotational Definition of the Programming Language Pascal.
Technical Report 77-47, Department of Computing and Information Science, Queen's University, Kingston, Ontario, 1977.

[62] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, R. G. Fisker.
Revised Report on the Algorithmic Language Algol 68.
Springer-Verlag, 1976.

[63] David A. Watt, Ole L. Madsen.
Extended Attribute Grammars.
Technical Report DAIMI PB-105, Computer Science Department, Aarhus University, Denmark, November, 1979.

[64] David A. Watt.
An Extended Attribute Grammar for Pascal.
SIGPLAN Notices 14:60 – 74, February, 1979.

[65] David A. Watt.
Rule Splitting and Attribute-Directed Parsing.
Pages 363 – 392 of Jones [22], 1980.

[66] J. Welsh, W. J. Sneeringer, C. A. R. Hoare.
Ambiguities and Insecurities in Pascal.
Software Practice and Experience 7:685 – 696, November-December, 1977.

[67] Wayne T. Wilner.
Declarative Semantic Definition.
PhD thesis, Stanford University, 1971.
Computer Science Report STAN-CS-233-71.

[68] Niklaus Wirth.
Algorithms + Data Structures = Programs.
Prentice-Hall, 1976.