A Semantics-Directed Compiler Generator

Lawrence Paulson
Department of Computer Science
Stanford University

Current address:
Computer Laboratory, University of Cambridge, Cambridge CB2 3QG, U. K.

Work 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.

1. Introduction

Language designers must compromise between their goals and resources, and reconcile conflicting features into a harmonious whole. They cannot try out their ideas on real programs, because of the cost and time required to write compilers. This paper describes research [8] that makes 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 (BNF) 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, and that programs written in high-level languages are not transportable.

This paper introduces a formal notation, the *semantic grammar*, for defining programming languages. Semantic grammars combine denotational semantics and attribute grammars. They 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.

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.

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. The universal translator reads the file and compiles programs into stack machine instructions, reporting semantic errors. The stack machine reads the program's input, executes the instructions, and prints the output.

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.

2. Compiler Generators

A compiler generator is a program that converts a formal description of a programming language into a compiler for that language. The description may take many forms, but usually contains a large amount of program code. Several recent compiler generators accept descriptions in terms of attribute grammars or denotational semantics.

NEATS is a compiler writing system that accepts extended attribute grammars [12]. It provides 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 NEATS attribute evaluator, which I have adopted, is fast and general.

Raiha's [9] Helsinki Language Processor (HLP) has generated compilers, assemblers, and preprocessors for a dozen languages. It constructs a parse tree and evaluates attributes in alternating passes. Attributes are Algol procedures. Raiha is studying optimizations, because HLP compiles Pascal programs at thirteen tokens per second, and consumes 90,000 words when compiling a one-page Euclid program.

SIS, by Peter Mosses, is the first compiler generator that does not need user-coded semantic routines [7]. 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. A six-line program requires several minutes of computer time for both compilation and execution. Despite this inefficiency, SIS proves that compilers can be generated automatically from high-level language descriptions.

Ravi Sethi [11] is experimenting with semantics-directed compilation. His simplifier applies functions to arguments 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. It has processed many of Mosses's example languages, but does not execute programs.

Martin Raskovsky's compiler generator [10] has produced a compiler for a simple language. 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.

My compiler generator is unique in that it accepts a semantic

grammar, a readable notation for denotational semantics. Although its semantic notation is entirely nonprocedural, it is efficient enough to produce compilers for real languages such as Pascal. It can execute programs several pages long, in seconds.

3. Semantic Grammars

A semantic grammar is an attribute grammar that uses the domains and formulas of denotational semantics.

An attribute grammar [5] 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.

Consider the assignment command (statement), with the syntax

command = variable ":=" expression.

An attribute grammar can describe its syntax and static semantics: determining the types of the variable and expression, and checking that they are compatible. The type of a variable depends on the current environment of declarations: a compiler's symbol table. 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. Watt and Madsen's *extended* attribute grammars [12] express the constraint type1 = type2 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 >
```

Denotational semantics [3] uses a powerful theory of recursively defined data structures, lambda-expressions, and fixedpoints of functions. A denotational definition expresses the semantics of a construct in terms of the semantics of its syntactic constitutents. A traditional presentation includes a context-free grammar, and introduces a semantic function for every nonterminal symbol in the grammar, defined by cases on the rules rewriting that nonterminal. In contrast, a semantic grammar embeds the semantic functions in the context-free grammar.

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. Here com is the semantic function for commands:

com[variable:=expression]env s =

var[variable]env (exp[expression]env s) s

Embedding this function in the attribute grammar rule yields the semantic grammar 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 consists of domain definitions, expression definitions, attribute declarations, semantic rules, and a resolution part. The domains and expressions are those of denotational semantics. The symbol **end** terminates the grammar. Comments may appear anywhere; they begin with a number sign (#) and continue to the end of the line.

3.1. Domain Definitions

Domains represent semantic data types, such as mappings, tuples, and tree structures. The domains INT, BOOL, and NAME (representing identifiers in source programs) are built-in. The user can define *product*, *function*, and *union* domains. A special case of union domain resembles an enumerated type of Pascal. If $D_1, \ldots, D_n$ are domains, and d_i denotes any value of D_i, then the following are also domains:

Domain	*Values*
$D_1 \times \ldots \times D_n$	n-tuples $(d_1, \ldots, d_n)$
$D_1 \rightarrow D_2$	functions from D_1 to D_2
$[name_1[D_1] + \ldots + name_n[D_n]]$	$name_i[di]$, i = 1 to n
$[name_1 + \ldots + name_n]$	$name_i$, i = 1 to n

The domain definitions list all the domains used to describe the semantics; there are no syntactic domains. Definitions may be recursive, such as LIST, VAL, and TYPE below.

```
domain
LIST = [nil + cons[INT × LIST]];   #lists of integers
VAL = [intV[INT] +
      arrayV[INT → VAL] ];         #data values
ENV = NAME → TYPE;                 #environments
TYPE = [intTy + arrayTy[TYPE] ];   #types
S = NAME → VAL;                    #states
EXP = S → VAL;                     #expressions
COM = S → S;                       #commands
VAR = VAL → COM;                   #variables
```

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

$cons[int_1, \ldots, cons[int_n, nil] \ldots]$

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.

3.2. Expression Definitions

Expressions may contain integer and boolean constants, the bottom element ⊥, and variables. If d, e, f, e_1, . . ., e_n are expressions, and v, v_1, . . ., v_n are variables, then the following are also expressions:

⊥	the bottom element of any domain
0, 1, 2, . . .	integer constants
false, true	boolean constants
"a", "x98", . . .	name constants
d+e, d−e, . . .	integer operators
d **and** e, d **or** e, . . .	boolean operators
d **eq** e, d **ne** e, . . .	relational operators
if d **then** e **else** f **fi**	conditional expression
(e_1, . . ., e_n)	n-tuple
left e	extract left element of a tuple
λv.e	lambda-abstraction of e over v
λ(v_1, . . ., v_n).e	abstraction over a tuple of variables
fix λv.e	fixedpoint of the function λv.e
f e	application of function f to arg e
let v=d **in** e	local definition of v to be d
[d→e]f	function f updated at d

If $name_1$, . . ., $name_n$ are the "tags" of a union domain, then the following are expressions:

$name_i$[e_i]	injection creating a union value
e | $name_i$	projection of e onto domain D_i
e **is** $name_i$	test that the tag of e is $name_i$

Every expression belongs to a unique domain, and the grammar analyzer checks that operators are only applied to operands of the proper domain, detecting many user errors.

The expression definitions list the expressions used to describe the semantics; most grammars define functions to check types or combine declarations. Definitions may be recursive. 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 and the **case** expression.

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

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

abort = λs.⊥;
```

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>;
expression<ENV.TYPE,EXP>;
variable<ENV.TYPE,VAR>;
command<ENV.COM>;
```

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

number<.INT>	represents an integer number, a string of digits
name<.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

3.4. 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 paper 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> ;
```

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. If the defined attribute is an expression, then the value it sees must have the same form, or the program contains an error [12]. This pattern-matching is implemented as a list of constraints on the rule.

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.

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
```

There is no way to specify the lexical conventions of a language; the 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

3.5. 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.

4. Example Grammar

Here is a semantic grammar for a tiny language lacking both side-effects and jump commands. Integer expressions may contain arithmetic operators. The condition of an **if** or **while** command may contain boolean connectives and integer relational operators. There are integer variables and unbounded integer arrays. Input and output are each a single integer, transmitted via the pre-declared variables *input* and *output*. The compiler generator has executed a prime number program using this grammar.

Although this grammar is trivial compared to Pascal's, it illustrates many of the same concepts. The grammar defines *static* semantics: the compiler functions of type-checking and symbol table management. It defines a domain TYPE to hold the two types, **integer** and **array**, and a domain ENV to hold the types of variables in the program. In the Pascal grammar, TYPE is a recursive tree structure, and ENV holds the meanings of constants, types, and procedures, as well as variables. Watt [13] illustrates the basic techniques.

The grammar also defines *dynamic* semantics: the compiler function of code generation. Denotational semantics provides two frameworks for control flow: a *direct* semantics can describe "structured" commands; a *continuation* semantics can describe any flowchart. This grammar uses direct semantics — an expression only computes a value, and a command only changes the values of variables. The Pascal grammar also uses direct semantics, but the Fortran grammar requires continuations because of Fortran's GO TO statements.

Gordon [3] explains how to represent procedures, parameter passing, expressions, and other dynamic concepts. A grammar can also define axiomatic or operational semantics, as Madsen [6] discusses. If the grammar defines axiomatic semantics, then the compiler generator translates a program into a list of *verification conditions* — assertions that, if proven, certify that the program is correct.

```
domain
VAL = [intV[INT] +
   arrayV[INT → INT] ];     #values: integers and arrays
ENV = NAME → TYPE;          #environment:types of variables
TYPE = [intTy + arrayTy];   #types
S = NAME → VAL;             #states: values of variables
EXP = S → INT;              #integer expressions
COND = S → BOOL;            #boolean conditions
COM = S → S:                #commands (statements)

INTFILE = INT → INT;        #integer mappings for I/O
DATA = INTFILE × INT;       #I/O interface for stack machine

define
#Input/output interface functions
beginProg = λ(intFile,int). ["input" → intV[intFile(1)]] ⊥;
endProg = λs. ( [1 → s("output") ]intV] ⊥, 1);

attribute
identifier<ENV.NAME,TYPE>;
expression<ENV.EXP>;
condition<ENV.COND>;
command<ENV.COM>;
declaration<.ENV>;
program<.DATA→DATA>;

rule program

#Look up the name in the environment; return the type
identifier<env,name,env(name)> = name<name>;

#### Expressions

expression<env,exp>    = "(" expression<env,exp> ")";

#Integer constants
expression<env, λs.int> = number<int>;

#Integer variables
expression<env, λs.s(name)]intV> = identifier<env,name,intTy>;

#Subscripted variables
expression<env, λs.s(name)]arrayV (exp s)> =
  identifier<env,name,arrayTy> "[" expression<env,exp> "]";

#Arithmetic operators
expression<env, λs.exp1(s) + exp2(s)> =
    expression<env,exp1> "+" expression<env,exp2>;
expression<env, λs.exp1(s) − exp2(s)> =
    expression<env,exp1> "−" expression<env,exp2>;
expression<env, λs.exp1(s) * exp2(s)> =
    expression<env,exp1> "*" expression<env,exp2>;
expression<env, λs.exp1(s) div exp2(s)> =
    expression<env,exp1> "/" expression<env,exp2>;

#### Conditions for if and while commands

condition<env,cond> = "(" condition<env,cond> ")";

#Comparisons of integers
condition<env, λs.exp1(s) lt exp2(s)> =
    expression<env,exp1> "<" expression<env,exp2>;
condition<env, λs.exp1(s) gt exp2(s)> =
    expression<env,exp1> ">" expression<env,exp2>;
condition<env, λs.exp1(s) eq exp2(s)> =
    expression<env,exp1> "=" expression<env,exp2>;

#Boolean connectives
condition<env, λs. not cond(s)> =
    "not" condition<env,cond>;
condition<env, λs.cond1(s) and cond2(s)> =
    condition<env,cond1> "and" condition<env,cond2>;
```

```
condition<env, λs.cond1(s) or cond2(s)> =
    condition<env,cond1> "or" condition<env,cond2>;

#### Commands

#Assignment to integer variable
command<env, λs.[name→intV[exp(s)] ] s> =
  identifier<env,name,intTy> ":=" expression<env,exp>;

#Assignment to array element
command<env,
   λs.[name→arrayV[[exp1(s)→exp2(s)]s name|arrayV]]s> =
  identifier<env,name,arrayTy> "[" expression<env,exp1> "]"
  ":=" expression<env,exp2>;

#Compound commands
command<env,λs.com2 (com1 s)> =
  command<env,com1> ";" command<env,com2>;

#If commands
command<env,λs.if cond(s) then com(s) else s fi> =
  "if" condition<env,cond> "then" command<env,com> "fi";

#While commands
command<env,
    fixλcom.λs.if cond(s) then com(com1 s) else s fi> =
  "while" condition<env,cond> "do" command<env,com1> "od";

#### Declarations

#Empty declaration
declaration<["input"→intTy] ["output"→intTy] ⊥> = ;

#Integer variable declarations
declaration<[name→intTy]env> =
    "int" name<name> ";" declaration<env>;

#Array variable declarations
declaration<[name→arrayTy]env> =
    "array" name<name> ";" declaration<env>;

program<λdata.endProg (com (beginProg data))> =
  "begin" declaration<env> command<env,com> "end";

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

5. Grammar Analyzer

The grammar analyzer reads a semantic grammar and converts it into a language description file. The 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 [1].
- 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 circularity.

The analyzer contains an LALR(1) parser generator 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.

6. Universal Translator

The universal translator can compile a program written in any language, given the proper language description file. It performs several steps:

- 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.
- Generate stack machine instructions for the program.

6.1. 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 [6].

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

$$X \rightarrow Y_1 Y_2 \ldots Y_n$$

by popping the right-side symbols and attributes, $Y_1 \ldots Y_n$, 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 an 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.

The rule's constraints, produced by the grammar analyzer, are evaluated like applied attributes and accumulated, producing a list of all the constraints in the program.

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 consumes 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. 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. 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.

6.3. 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 [2]. 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. This saves the simplifier a tremendous amount of work.

If a bound variable occurs more than once, then beta-reduction replicates its argument. The simplifier performs no beta-reductions that would replicate expressions requiring evaluation at run-time, which would make the object program less efficient. The simplifier only replicates "simple" arguments. The key question: what is simple?

The safest answer is that only atomic expressions are simple — variables, numbers, etc. But this does not handle structured bound variables:

$$(\lambda(int1,int2).\ int1 + int2)\ (3,8)$$

The bound variable, (int1,int2), is referenced twice; the argument, (3,8), is not atomic. Beta-reduction is prohibited even though no component of the bound variable is used more than once. One solution is making the simplifier transform the above expression into:

$$((\lambda int1\ int2.\ int1 + int2)\ 3)\ 8$$

This allows beta-reductions, but the complete process copies the

function several times. Instead of relying on an expensive transformation, I generalize "simple" to include any closed expression; these can be detected using the free variable index. Experience shows that this version of simple allows efficient simplification without exponential blow-up, although it is not obvious why.

6.4. 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.

The simplifier evaluates the DAG depth-first and records all the semantic errors: 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. Sample listing from the translator:

```
VAR v,v: integer;                {v is declared twice}
       ↑
Semantic error:
Should be UNDEFINED

x: array[1..n] of integer;       {n is not declared}
             ↑
Semantic error: IDENTIFIER
Undefined attribute MODE

i[8] := 0;                       {i is not an array}
 ↑
Semantic error: COMPONENT
Should be ARRAYTY
```

6.5. Code Generation

Since the stack machine is oriented towards execution of lambda-calculus formulas, code generation is straightforward, using a depth-first traversal of the simplified DAG. 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.

Burge [2] 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:

$$\mathbf{fix} = \lambda f.(\lambda g.f(g\ g))(\lambda g.f(g\ g))$$

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 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.

The code generator performs a few optimizations, besides those Burge recommends. For instance, it emits code to delete a bound variable after its last use, located during the DAG traversal. Given the expression

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

where A does not use x, it generates code for

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

It also optimizes cases that cannot be illustrated by expression transformations. 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.

6.6. 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 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.

7. Stack Machine

The stack machine executes the object code produced by the universal translator. It has the SECD architecture [2]: a *stack* of pending operands, an *environment* of bound variables, a *control* of instructions, and a *dump* of return addresses and environments.

Control instructions:

halt	stop program; print top of stack
return	return from function; restore state
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	the function compiled at pc

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

plus	the sum x+y
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

7.1. 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

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

onto the stack. The machine expects to find a similar data structure on the stack after executing the object program, and prints the list it denotes.

7.2. Run Time Errors

The value $\bot$ ("bottom") represents run-time errors. For instance, a subscript out of bounds may set the state to $\bot$, which will propagate to the end of the program, producing a final state $\bot$. The machine should halt immediately upon seeing $\bot$, for prompt error detection, but not every $\bot$ indicates an error: $\bot$ is also used for initialization.

If $\bot$ is the operand of an instruction, the machine usually halts, but some instructions return $\bot$ as the result, or treat $\bot$ as an ordinary value. The machine prints its current state upon halting. To aid debugging of user programs, every value of $\bot$ is flagged with the program location where it was generated.

7.3. 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.

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.

7.4. 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 which 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:

$$[0 \rightarrow 25]\ [0 \rightarrow 16]\ [0 \rightarrow 9]\ [0 \rightarrow 4]\ [0 \rightarrow 1]\ \bot$$

These association lists, or history sequences, waste storage and runtime. States, which are also mappings, suffer the same problem. Efficient execution is impossible unless the machine compacts association lists into arrays.

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 arrays.

The machine tries to eliminate unnecessary references into 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 any loop or goto command causes tail-recursion.

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**

The compiler generator uses peephole optimization and careful code generation to convert these to **apply;return**.

During execution of a program, the machine compacts the same list many times. The machine updates an existing array, instead of allocating a new one, as long as the new list elements fit within its bounds. When it allocates a new array, it allows room for expansion above and below. The machine merges two segments into one if the reference separating them disappears.

The array compacting algorithm is complex and slow, but satisfactory. The stack machine has executed prime number and Eight Queens programs that use arrays extensively.

7.5. 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.

8. Conclusions

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 covers all static and dynamic semantics except goto statements, real numbers, strings, aliasing, 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 nineteen pages long (1400 lines), including comments: one page of domains, five of functions, and thirteen of rules.

Fortran, with its low-level state and control structure, and non-recursive subroutines, contrasts well with Pascal — its grammar uses continuations and locations. Fortran's grammar is less complete than Pascal's, but still covers labelled COMMON blocks, EQUIVALENCE statements, DO statements with extended range, assigned and computed GO TO, unformatted input/output with implied DO, subroutines, and functions.

8.1. 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: defining too

many domains, terminal symbols, rules.

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 program's machine code and semantic formula.

8.2. Efficiency

The compiler generator is efficient enough to run experimental programs, but it is impractical for production runs. Consider the following approximate statistics:

The grammar analyzer is 4400 lines of Pascal. It processes the Pascal grammar in twenty-three seconds, producing a language description file of 14,000 thirty-six-bit words. Half of this is parse tables, the rest attributes and definitions.

The universal translator is 3900 lines of Pascal. Using the Pascal grammar, it compiles programs at eight seconds per page, which is twenty-five times slower than the regular Pascal compiler. Storage limitations prevent it from compiling programs longer than twenty pages.

The stack machine is 1300 lines of Pascal. Its speed varies considerably, depending on the grammar and user program, averaging 4000 instructions per second. It runs Pascal programs a thousand times slower than the regular Pascal system.

I have run dozens of programs in several languages. The longest is an LR(0) parser generator, a seven-page program that uses nearly every feature of Pascal. I have also run prime number generators, Eight Queens solvers, and binary search tree sorters. The compiler generator 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.

My dissertation [8] suggests several ideas to improve the efficiency.

- Separate the semantic language into independent notations for static and dynamic semantics. Replace the simplifier with two specialized, more efficient routines: an *evaluator* for static semantics, and an *optimizer* for dynamic semantics.
- Compile programs one procedure at a time, so that programs of any size can be compiled. Allow a grammar to designate certain nonterminal symbols, such as *procedure*, to be compilation units.
- Eliminate the stack machine. A real machine, with a run-time support package, can perform all of its functions much more efficiently.

8.3. Implications for Language Design

A programming language should be formally defined even while it is being developed, because a formal definition highlights the language's 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 define. The fault might lie in the formalism: for instance, denotational semantics can not represent tasking. Still, semantic grammars, along with the compiler generator, can contribute to the design of consistent, clean, and simple programming languages.

Acknowledgment. I would like to thank my advisor, John Hennessy, for supervising and supporting this research.

References

[1] Alfred V. Aho, Jeffrey D. Ullman.
Principles of Compiler Design.
Addison-Wesley, 1978.

[2] W. H. Burge.
Recursive Programming Techniques.
Addison-Wesley, 1976.

[3] Michael Gordon.
The Denotational Description of Programming Languages: An Introduction.
Springer-Verlag, 1979.

[4] Neil D. Jones (editor).
Semantics-Directed Compiler Generation.
Springer-Verlag, 1980.

[5] D. E. Knuth.
Semantics of Context-Free Languages.
Mathematical Systems Theory 2:127 – 145, February, 1968.

[6] 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 [4].

[7] Peter D. Mosses.
Mathematical Semantics and Compiler Generation.
PhD thesis, Oxford University, 1975.

[8] Lawrence Paulson.
A Compiler Generator for Semantic Grammars.
PhD thesis, Stanford University, 1982.
Forthcoming.

[9] Kari-Jouko Raiha.
Experiences with the Compiler Writing System HLP.
Pages 350 – 362 of Jones [4].

[10] 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.

[11] 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.

[12] David A. Watt, Ole L. Madsen.
Extended Attribute Grammars.
Technical Report DAIMI PB-105, Computer Science Department, Aarhus University, Denmark, November, 1979.

[13] David A. Watt.
An Extended Attribute Grammar for Pascal.
SIGPLAN Notices 14:60 – 74, February, 1979.