Linear Intermediate Representation for Portable Code Generation

Mahadevan Ganapathi
Charles N. Fischer
Stephen J. Scalpone
Keith C. Thompson

Computer Sciences Technical Report #435

September 1981

Table of Contents

Linear Intermediate Representation for Portable Code Generation

Mahadevan Ganapathi
Charles N. Fischer
Stephen J. Scalpone
Keith C. Thompson

Abstract

Design considerations for a machine-independent/language-independent Intermediate Representation (IR), which forms the input language for a portable code generator, are outlined. An attributed Polish-prefix representation is suggested. The advantages and disadvantages of attributed linear IR with respect to tree-structured IRs are discussed. An example of a complete IR specification is provided.

Keywords and Phrases: Code Generation, Intermediate Representation.

CR Categories: 4.12

Computer Sciences Department, University of Wisconsin-Madison

Research supported in part by National Science Foundation Grant MCS78-02570

1. Introduction

In a classical monolithic compiler (i.e., a compiler written for a single programming language to run on a single target machine), an intermediate form of program code is used primarily to allow optimization. Examples of popular intermediate forms are quadruples, triples, tree representation of programs and directed acyclic graphs (DAGs) [Aho 77]. Unfortunately, these conventional intermediate forms are inadequate to serve the needs of compiler portability. The design of an Intermediate Representation (IR) is critical to compiler portability and code generation efficiency. This efficiency issue includes both the efficiency of the code generation algorithm and that of the object code produced by the code generation algorithm. In this paper, an IR is visualized as a demarcation or boundary-line between machine-independent and machine-dependent parts of a compiler. A compiler front-end would translate a programming language to this IR. All language-dependent issues are to be processed by this front-end. For example, in Ada [Ichbiah 79] an operator can have several valid meanings within the same scope. This operator overloading must be disambiguated by the front-end before it generates the IR. Similarly, re-ordering of procedure parameters in Ada must also be done by the front-end. A code generating back-end would translate the IR to target machine code. All machine-dependent issues are to be processed by this back-end. As a consequence of this approach, storage binding is treated as a code generation issue to be addressed by the back-end. However, compiler front-ends may influence this binding decision via the IR. Furthermore, no assumptions are made regarding the basic run-time model of the target architecture (e.g. whether the

architecture is a stack machine or not). Under this compilation model, a compiler can be retargeted to a new machine simply by changing the IR to object code translation phase. Since this phase can be largely automated [Ganapathi 80], retargeting is far less expensive than writing an entirely new compiler.

In our discussion, we distinguish between code-generation languages (FSL [Feldman 64], CGGL [Donegan 79], P-code [Wirth 71], U-code [Sites 79]) and IRs. Code generation languages provide dictions specially suited to describe the generation of target machine code. P-code is a code generation language designed for Pascal and a hypothetical stack machine called the P-machine [Wirth 71, Nelson 79]. U-code is an extension of P-code for purposes of machine-independent Pascal code optimization [Sites 79]. Code generation languages are very closely tied to either a single programming language or a machine architecture and thus they cannot be considered of much use in compiler portability. Furthermore, they intermix the code generation algorithm with machine descriptions whereas IRs help separate machine dependencies from the IR code generation algorithm. An IR is not a definitional language that allows all code generation cases to be defined explicitly. Such cases are assumed to be automatically selected by the code generator from information provided in the machine description. Thus, to implement 'p' programming languages on 'm' architectures, p+m translators are essential when code generation languages are used. On the contrary, using IRs, these 'm' translators can be replaced by 'm' machine descriptions together with a single code generation algorithm [Ganapathi 81]. The BNF for the IR is different from the attribute grammar for the target machine. This separation is essential to isolate the target machine description from the code generation algorithm. A

two-phase single-pass code generation algorithm is employed in [Ganapathi 80]. The implementation is modularly divided into (a) storage-binding phase and (b) instruction selection phase. At the IR level, variables are represented by their names, which are converted to machine addresses before instructions are selected. After the storage binding phase, IR names are transformed to machine addresses, with machine-independent addressing modes that allow multiple levels of indirection and indexing. This expansion is the crux of the mapping between the grammar for the IR and the attribute grammar for the target architecture. The mapping of these machine-independent addressing modes to the actual modes present in the target architecture is done when parsing through the addressing mode productions of the target machine.

Traditionally, IRs have been used during different phases of compilation. They have served as a suitable model for numerous tasks including Global Flow Analysis, Loop Optimizations, Expression Optimizations, Global Register Allocation, Frequency Reduction and Code Generation [Aho 77]. Sometimes, the same IR serves as a model for more than a single compiler phase. For example, trees are used as an IR by almost every phase in the PDP-11 Bliss compiler [Wulf 75]. A multi-model use of an IR implies a multipass compilation scheme. In this paper, we concentrate on an IR that serves as a good model for code generation and machine-dependent optimization under the constraints of compiler portability. For a motivation and justification of IRs as a general requirement in compiler construction, the reader may refer to [Brosgol 80a, 80b, Pebbleman 79]. With the motivation for IR design already established, the next section reviews classical IRs that have appeared in production compilers in the past.

2. Comparison of classical IRs

In this section, we compare and contrast the following forms of IRs:

(1) Tuples comprising quadruples, triples, indirect triples [Gries 71, Aho 77] and N-tuples [Frailey 79],

(2) Abstract Program Trees (APT [Wilcox 71]) and Graph Notations (Dags [Aho 77], TCOL [Brosgol 80a, 80b], AIDA [Dausmann 80]),

(3) Linear Representations such as Reverse Polish (postfix) and Standard Polish (prefix) notations.

Tuples [Aho 77] comprise of operators (only unary and binary operators) and operands including temporary results. Operands may be constants and variables only. They may not be names of functions or subroutines. Subroutine and function calls are implemented by a large number of three-address statements. Quadruples and triples differ in the manner in which temporary results are specified. In quadruples, temporary names are used to specify results whereas in triples, results are specified by referencing other triples. When references to triples are separated from the triples themselves, the notation is called indirect triples. A table is used to carry pointers to triples. With the help of this table, identical triples need be stored only once. N-tuples [Frailey 79] are an extension of three-address code. An operator is allowed to have an arbitrary number of operands. For each application, the arity of the operator is determined by interrogating flags in the symbol table. These flags are essential for processing the intermediate representation in order to generate target code. N-tuples allow compact representation of consecutive operations that have identical operators. Operands may be constants, variables, statement labels and function or subroutine names.

Abstract syntax trees comprise of leaves and interior nodes. Each interior node represents an operator and each leaf represents an operand. Trees form a hierarchy instead of a linear representation (as in tuples). Pointers are used to associate operands with operators. The arity of the operator determines the number of pointers required for a subtree. Owing to the presence of pointers, trees lend themselves to ease of re-arrangement. However, once the IR is generated by a compiler front-end, the tree structure is not advantageous for code generation. Trees do not promote efficient pattern matching for purposes of target code generation. Graphs are extensions of tree notations. They convey more information about program structure than trees do. A program graph specifies the relation among basic blocks and is thus well suited for machine-independent optimization (a basic block is a linear sequence of code containing no branches).

Polish notation is a contiguous string of characters containing operators and operands. Unlike tuples, there is no necessity for explicit specification of temporary results. Furthermore, Polish notation (especially Polish-prefix) is easy to parse. Unlike infix notation, it is parenthesis free. There are two flavors of Polish notations: Polish prefix and Reverse Polish (postfix notation). In prefix notation, operators are followed by their operands whereas in postfix, operands precede operators. Examples of use of postfix-IR can be found in SNOBOL [Griswold 68] interpreters (the postfix IR is heavily interlaced with subroutine calls).

All of these above forms of IRs are attempts to separate program representation from the underlying data structures that are used to manipulate the representation. All these IRs are logically equivalent. Triples are direct representations of trees in which the last triple represents the root of the tree. Other triples are subtrees with the root of the subtree being the operator of the triple. When references to triples are replaced by temporary names, triples are transformed to quadruples. In effect, triples are quadruples with one level of indirection. Similarly, indirect triples are triples with another level of indirection. N-tuples are extensions of three-address code with considerable flexibility incorporated in the arity of operators. Very similar to the equivalence between tuples and tree-based IRs, postfix and prefix representations can be obtained by traversing trees in post order and preorder respectively. It is straight-forward to derive one intermediate representation from another as illustrated by Elson and Rake [Elson 70]. However, such conversions are not very optimal. These IRs differ considerably in their applicability as a model for various tasks during compilation. Each IR is a good model for a particular compiler phase.

In this paper, we concentrate on the applicability of these IRs as a model to

(1) ease compiler portability.
(2) enable implementation of efficient code generators for a variety of target machines, and
(3) help improve target code quality (both object code size and execution speed).

The considerations used to compare these IRs have been classified into two main categories:

(1) generating the IR (i.e., High-level-language program to IR translation), and

(2) processing the IR (i.e., IR to target code translation).

Generating the IR from a programming language representation

There are two considerations to generating the IR from a high level language representation of a user program. The first criterion is the simplicity and ease with which a compiler front-end could translate a high level representation to the IR. The second criterion is the ability to represent programming language semantics in the IR.

Simplicity and ease of front-end translators:

To translate a HLL (high level language) user program to an IR, front-ends usually employ some automatic parsing technique. At the IR level, some extra symbols are introduced. Examples of such front-end generated symbols are labels and explicitly specified temporary names. In classical compilers, no IR code is generated for declarations (i.e., storage binding). For example, the Pascal declaration statement "k : Integer" or the Fortran array declaration "Dimension A [10]" do not appear in the IR since such information necessary to generate target code for space reservation is extracted from the symbol table. In contrast, when retargetability becomes an important issue, code for declarations should be explicitly specified in the IR. Declarations at the IR level will be used by the code generation phase to bind variables to storage

locations of the target architecture. In general, variables will be represented differently on different machine architectures. Thus, the mapping of variables to storage locations (i.e., operand binding) must be performed by the code generation phase.

The structure of the IR may impose certain restrictions on compiler front-ends and back-ends. Tuples require explicit specification of temporary names. Consequently, temporary binding must be deferred to the code generation phase. Some IRs such as Postfix and Prefix notations do not require explicit temporary name specification. With these notations, the code generation phase automatically performs temporary binding and thus this binding issue is easier than when tuples are used.

Complete representation of Programming language semantics:

An IR should be able to accommodate a variety of programming language structures and constructs. Common operators and control constructs should be expressible in the IR. Trees allow fairly explicit and complete representation of programming language semantics. Quadruples and triples are convenient to represent binary and unary operations only. To use operators with greater arity, other notations must be used. Furthermore, for operations other than simple expressions (i.e., expression evaluation) or assignments, tuples are not easy to use. For example, subscripting and procedure calls typically require a large number of tuples:

```
A := B[i, C[j]]
```

The equivalent quadruple statements would be

```
*, i, di, Ti
index, C, j, Tc
+, Ti, Tc, Tx
index, B, Tx, Tb
:=, Tb, , A
```

Moreover, a sequence of tuples may be subsumable within a single instruction or operand-addressing mode (e.g., using the index addressing mode on the VAX-11/780, the above set of quadruples could be represented as "movl B(ri)[rj]"). In such cases, it is the code generator's responsibility to coalesce a sequence of tuples into a single machine primitive. Another example of verboseness in some IRs is the requirement to specify data type conversions explicitly. Unless tuples carry data-type information, separate operators are needed for similar operations on different data types. Alternately, explicit type conversion operators can be provided (e.g., convert integer to real). This necessity requires the front-end to explicitly convert operands of one data type to another in cases of mixed-mode operation. For example,

* Integer Real

require conversion of either an integer to a real or a real to an integer. After conversion, either an integer or a real multiplication operator is used. Such conversions are useless on tagged architectures [Feustal 73] or on architectures where the data type is part of the operand specifier (as opposed to being part of the machine op-code) [Szewerenko 81].

A related issue is storage space requirement for implementation of the IR and extra information to be carried around during code generation to facilitate

run-time debugging. Quadruples and triples waste some space since occasionally some fields will be unused (empty). If the same temporary value is used more than once, indirect triples can save some space when compared to quadruples and triples.

Symbolic debugging of optimized code has received considerable attention in the literature [Satterthwaite 75, Hennessy 79]. In order to detect incorrect variables in an optimized program and to possibly recover their correct values, necessary program-flow information must be associated with the variables (operands in the IR). Although such information is not needed for code generation, it has to be associated with the target code for purposes of symbolic debugging. The IR should therefore provide a facility to compactly represent such information. As will be seen in a later section, attributes are a good means of representing information needed for symbolic debugging of optimized code. Since some optimization is always done by the code generator, debugging should be done independent of the code generation procedure.

Processing an IR to produce target code

The criteria used in this consideration are (a) the efficiency of the object code produced from the IR (object-code space and execution speed of the target code) and (b) the efficiency of the code generation algorithm itself.

Realization of machine-independent optimization during code generation:

Many machine-independent optimizations are actually realized during code generation. This delay is due to the fact that some of these optimizations anticipate machine-dependent properties. Examples of such optimizations are: register binding of variables and packing of record fields (i.e., placing variables compactly in contiguous areas in machine storage). In such cases, machine-independent optimizers must be able to express their optimization intentions in the IR. Subsequently, the code generating back end should be able to utilize this information in order to produce optimized target code.

Efficiency of the code generation algorithm:

To translate the IR to target machine code, the code generator must be efficient. The efficiency of the code generator depends to a great deal on the IR. When using tuples, the entire code has to be scanned (usually, on a basic block basis) before determining the number of temporaries that are active simultaneously. Therefore, temporary binding must be deferred to code generation. Register allocation decisions are related to "next-use" information of variables [Aho 77]. Determination of next uses of every name in a tuple implies a backward scan of a stream of tuples within each basic block. Having computed the next use information, a forward pass through the basic block selects object code. Trees require an outside-in scanning of the leaves and interior nodes and an inside-out generation of target code. In effect, the matching of operators and operands is done in a bottom up fashion whereas parsing (to generate target code) is done top-down. The entire tree is built

before code generation takes place. Hence, tree-based IRs preclude a single pass code generation scheme. Graphs pose even more severe problems. They need cycle elimination in order to prevent infinite loops and pattern-matching graphs is a greater problem than pattern matching trees. In contrast, prefix and postfix notations allow a single pass code generation scheme. Prefix notation implies a left to right scanning. Although top-down parsing is more natural with prefix notation, code generation is more suited to bottom-up parsing. Prefix and postfix notations, being parenthesis free, make expressions easy to decode from either end (i.e., they facilitate top-down and bottom-up parsing respectively). However, it is harder to decode control flow in postfix notation when compared to trees and tuples. In a later section we demonstrate how the presence of attributes in prefix notation alleviate most of the control flow problems posed by postfix notation.

We have compared and contrasted IRs from the point of view of compiler portability and target code generation. These internal forms have much in common (i.e., they mainly comprise of operators and operands and some means of linking the information between operators and operands). They differ mainly in their use in transporting compilers and writing efficient code generators. The choice of an IR has often depended on the whims and prejudices of the compiler writer (i.e., it was a "black art": more of sorcery than science). In this section we have hopefully provided more insight into IR design and in the next section we provide a rationale for it.

3. Design Considerations

Considerations involving the design of a common IR for a family of retargetable compilers are reviewed below:

(1) The level of an IR determines the work to be redone in either transporting a code generator to a new machine or using the same code generator for a new source language. If the level is too high, language dependencies creep in. Similarly if the level is too low, machine dependencies are unavoidable.

(2) From UNCOL [Strong 58, Steel 61] experience, it seems impossible for a single IR to satisfy the requirements of all programming languages. To avoid compromises or inefficiencies, IRs should be flexible. Such a need is addressed in the Janus [Coleman 73] family of abstract machines and the TCOL [Brosgol 80a, 80b] family of IRs. While Janus was developed to study portable compilers, TCOL is more ambitious in using the same family of IRs (e.g., $TCOL_{Ada}$, $TCOL_{Jovial}$) for, among other things, generation of verification conditions, language-oriented editing and code generation.

(3) The IR strongly influences the efficiency of code generation algorithms. Portable code generators commonly match the IR with instruction patterns of the target machine [Ganapathi 81]. Hence pattern matching within the IR should be fast, efficient and (ideally) provably correct.

(4) The IR must facilitate ease of interface between a code generator and machine-independent parts of a compiler. Thus, machine-independent optimizers such as live/dead analyzers can express their intentions of resource allocation in the IR. As a consequence, the IR will also help produce optimized object code.

4. Attributed Prefix Notation

First, we describe the motivation for the choice of a prefix representation instead of infix or postfix representations. Infix notation is ambiguous without parentheses. Prefix notation is preferred to postfix for the following reasons (assuming a single-pass processing of the IR):

(1) Many architectures have non-orthogonal instruction sets. Some op-codes require operands to be in special machine locations (e.g., even-odd register pairs for multiplication and division on the IBM-370 and PDP-11/70, registers for operand movement due to lack of memory to memory operations on the Intel-8086). In postfix notation, an operand is encountered before its operator, and until the operator is seen the other associated operand is not known. The code generator might therefore have to back up in special cases to fix the operands (i.e., move them to valid locations).

(2) The context in which an expression is to be evaluated should be known prior to evaluation of the expression. For example, A < B requires an explicit Boolean result when evaluated in the context "C := A < B" but does not need one in the context "IF A < B THEN C := 1". A similar example is "length (concatenate string1 string2)" where, by knowing the context 'length' prior to evaluation of the resulting string, the actual concatenation can be suppressed by merely adding the lengths of the individual strings.

(3) Even if the IR-context does not require an explicit Boolean result, the instruction set of the target machine may nevertheless require creation of an explicit result. For example, a Boolean OR on the PDP-11/70 or the

VAX-11/780 (bis) will always produce an explicit result. Furthermore, the VAX-11/780 provides a choice between a two-address (bis2) and a three-address (bis3) OR. In a conditional-statement context such as IF A OR B THEN C := 1, bis3 will be used; in an assignment context (e.g., A := A OR B), bis2 may be preferable.

(4) Knowledge of the immediate destination of a result can influence the choice of operands for machine instructions. For example, a new temporary is unnecessary if the left-hand side of an assignment can store temporary results. It is therefore valuable to know the left-hand side of an assignment before encountering the righthand side.

(5) On target architectures that have condition codes, more than rudimentary control-flow must be considered. Operands may be undefined or may have side-effects. In such cases, a postfix implementation may not only be inefficient but also incorrect!

The intermediate representation (IR), we will use, is an attributed Polish-prefix linear notation. It is composed of operators, operands and attributes. The operators and operands form a common structure to the IR family. They are augmented with attributes to carry forward semantic information essential for storage binding (e.g., type, size and scope of a variable) and code optimization (e.g., execution speed and object code space). The attribute domains of operators and operands are determined by the structure of the high level language. At the IR level, variables are represented by their names, which are then bound to machine addresses before instruction selection. The decision of how to address variables is treated as a code generation issue. As an advantage, it is very easy to alter activation record formats on the run time

stack (e.g., changing the offset of the static-chain link from the frame pointer) whereas in conventional compilers, this change may affect several sections of code. The need for a variable arity is found in very common programming language constructs such as array declarations and procedure (subroutine) calls. In such cases, the arity can be expressed as an attribute to the operator:

(1) ALGOL Array ArrayA[L_1:U_1, ... L_n:U_n]

The equivalent IR would be

ArrayA↑2n L_1 U_1 ..... L_n U_n

where n represents the dimension of the array, L_i is the lower bound of $dimension_i$ and U_i is the upper bound of $dimension_i$.

(2) ProcedureA ($parameter_1$, ... $parameter_n$)

The equivalent IR representation would be

CALL ProcedureA↑n $parameter_1$ .... $parameter_n$

where n is the number of parameters of the procedure.

Data types are not attributes to operators. Instead, they appear as attributes to operands. Thus, "+ A B", where A is an integer and B is a floating-point datum, is a legal IR string. The advantage of this approach is that only one operator is needed for any type of operation with different data types (e.g., both integer and floating point multiplication use a single multiplication operator). Furthermore, on tagged architectures [Feustal 73] and architectures with operand specifiers such as Nebula [Szewerenko 81], such IR operations can be easily mapped to machine instructions without the necessity for operand movement. On architectures where the data-type encoding is part of the op-code (e.g., PDP-11, VAX-11 series of computers), such operations cannot be performed in a single machine instruction (i.e., data types of

operands must be similar). In such cases, the code generator will automatically move an operand to the relevant data type before the operation can be performed. There is no necessity for explicit specification of temporaries. Nevertheless, in certain special cases of programming language constructs such as the C auto-increment statement within a conditional statement:

```
if (*string1++ = *string2++)
```

or multiple assignment statements (a = b = c = d), an explicit temporary must appear in the IR so that the order of evaluation of the statements can be disambiguated. Attributed prefix IR allows specification of explicit temporaries when they are really needed.

The detailed specification for the IR framework is provided in the next section. The symbol '↑' attaches an attribute to an operator or an operand. Variable numbers of attributes may occur and their order of occurrence is not important. Some attributes are essential (e.g., type and size of a datum are needed to perform storage binding) whereas others are optional (e.g., register preference, time and size help in producing optimized target code).

This Attributed Prefix-IR meets the design requirements mentioned earlier as follows:

(1) At the IR level, variables are represented by their names without any assumptions of the addressing scheme used in target architectures. In order to resolve information about variables (e.g., type, size and scope), use of a symbol table by the back-end is implied. The symbol table is intended to be used by the debuggers and the machine-dependent optimizers. Thus, a declaration statement (or its equivalent) is necessary in the IR.

However, by not using such a declaration in the IR (and consequently a symbol table during the code generation phase) symbol table search can be eliminated. In this case, every occurrence of a name in the IR must appear with all its attributes. Furthermore, absence of a symbol table precludes symbolic debugging of target code (Symbolic debugging is a good justification for not deleting the symbol table).

(2) Attributes are used to provide the "flexibility" that is needed in the IR. Thus, they help the IR in accommodating language variability and also provide ease of interface with machine-independent parts of a compiler.

(3) String matching algorithms can be used to perform pattern matching. Typical context-free parsers such as YACC [Johnson 75] use very simple drivers and are efficient (i.e., linear-time algorithms) and well understood. Pattern matching in a tree is not as well understood as string matching. Most tree-matching algorithms rely on heuristics and the results are not provably correct.

(4) Almost all results of machine-independent optimization are expressible as attributes. For example, after the front-end completes Global Analysis of variables, it can determine unique predecessors of basic blocks. The state information of basic blocks is associated with attributes to corresponding labels for the blocks (e.g., label↑{a,b,predecessorlabel}, where a and b are variables that come live into the current basic block). At code generation time, these attributes are used to produce optimized code (i.e., these attributes control pattern matching of the IR with the machine description).

Our IR is conceptually similar to tree-based IRs that have been designed with the goal of compiler portability (APT [Wilcox 71], TCOL [Brosgol 80a, 80b], AIDA [Dausmann 80]). Attributes in the IR are essentially the same as tree linkages. In Attributed Prefix-IR, information is carried in the form of data structures (i.e., Records) whereas in tree-based IRs, information resides in the form of pointers (i.e., tree links). Linkage is then in a trivial way provided by attributes. Records are also more abstract because there is less need to restructure them when compared to tree links. Thus, attributes provide simplicity of usage.

Another distinction between our design and tree-based IRs is that the back-end (i.e., the code generation phase) generates code using string pattern matching techniques as opposed to pattern matching in trees. Conventionally it has been the belief that since parsing inherently bounds tree search it is not well suited for pattern matching in code generation. On the contrary, parsing techniques have been found very elegant and useful [Glanville 78, Ganapathi 80] because in code generation, pattern matching decisions are to be made only over very small sections of the IR. The code generation algorithm is considerably fast and enough scope exists to prove certain formal properties (such as no looping) when generating code. Furthermore, attributes in the IR are used to control this pattern matching in order to produce optimized target code. The efficiency of the object code is thereby considerably improved. Thus, the design of the IR is used to enhance compiler portability while promoting simplicity and efficiency in code generation.

5. IR

This design is presented as two tables and a formal BNF. Implementation of the BNF is described with parse-table sizes and other relevant statistics for both a top-down parser and a bottom-up parser.

Table 1: Operators.

Operator	Arity	Type of operand(s)
negation	1	fixed, floating or integer
not	1	boolean
complement	1	bit vector
addition	2	fixed, floating or integer
subtraction	2	fixed, floating or integer
multiplication	2	fixed, floating or integer
division	2	fixed, floating or integer
remainder	2	fixed, floating or integer
modulo	2	integer
logical and	2	boolean
bitwise and	2	bit vector
concatenation	2	strings
logical or	2	boolean
bitwise or	2	bit vector
logical xor	2	boolean
bitwise or	2	bit vector
equal	2	any type
not equal	2	any type
less than	2	char, fixed, float, integer, pointer, string
less or equal	2	char, fixed, float, integer, pointer, string
greater than	2	char, fixed, float, integer, pointer, string
greater or equal	2	char, fixed, float, integer, pointer, string
indirect	1	pointer
address of	1	variables
size of	1	any type
conversion	2	any type and an expression
array reference	2	array and an index expression
array slice	3	array and a range (two index expressions)
function call	n	function and n-1 expressions
assignment	2	variable assignment
branch	1 or 2	conditional and unconditional branches
case	3	case statement
procedure call	n	procedure name and n-1 parameters

Table 2: Attribute domains and values

Attribute Domains	Attribute Values
Scope control	lexical level static level
Type descriptions	array bit vector boolean character fixed point number floating point number integer record pointer string
Identifiers	type label variable procedure
Type identifiers	type description lexical level
Label identifiers	import export lexical level
Variable identifiers	import export enforce run-time checks in parameter out parameter by reference parameter read only optimize for speed optimize for space optimize for frequency initialization static level lexical level
Procedure identifiers	import export return-value type concurrent lexical level
Arrays	index type low bound high bound base type
Bit vectors and strings	length
Floating point and fixed point numbers	precision range
Integers	range
Records	fixed field variant field
Pointers	base type

BNF

Syntax notation used to describe IR:

(a) Words prefixed with '†' are attributes, for example
    †type

(b) Words prefixed with '.' are statements, such as
    .goto
    or are directives, such as
    .begin

(c) Characters enclosed in quotes are literals, for example
    '*'

(d) Angle brackets enclose syntactic categories, for example
    <type>

(e) Square brackets enclose optional items, for example
    ['*']

(f) Braces enclose a repeated item which may occur 0 or more times, for example
    { <command> }

(g) Parentheses enclose alternative items. Items are separated by a vertical bar, for example
    (.true | .false)

(h) Comments are enclosed between (* and *), for example
    (* this is a comment *)

```
<ir>            ::=   { <command> }
<command>       ::=   <directive>
                ::=   <declaration>
                ::=   <statement>
<directive>     ::=   .begin [ <static level> ] [ <lexical level> ]
                ::=   .end
<declaration>   ::=   ':' Ident †type <type> { <type attrs> }
                ::=   ':' Ident †label { <label attrs> }
                ::=   ':' Ident †variable <type> { <var attrs> }
                ::=   ':' Ident †procedure { <proc attrs> }
<label attrs>   ::=   †import
                ::=   †export
```

```
                ::=   <lexical level>          (* compile time scope *)
<var attrs>     ::=   †readonly
                ::=   †import
                ::=   †export
                ::=   †enforce                 (* run-time checks *)
                ::=   †initialize <expr list>  (* initialization *)
                ::=   †space                   (* optimization *)
                ::=   †speed                   (* optimization *)
                ::=   †usage                   (* optimization *)
                ::=   †reference               (* parameter *)
                ::=   †in                      (* parameter *)
                ::=   †out                     (* parameter *)
                ::=   <static level>           (* run-time scope *)
                ::=   <lexical level>          (* compile time scope *)
<proc attrs>    ::=   †import
                ::=   †export
                ::=   †return <type>
                ::=   †task                    (* concurrent process *)
                ::=   <lexical level>          (* compile time scope *)
<type attrs>    ::=   <lexical level>          (* compile time scope *)
<type>          ::=   †array <bounds> <base type>
                ::=   †bits <length>
                ::=   †boolean
                ::=   †char
                ::=   †fixed [ †delta <expr>  ] <range>
                ::=   †float [ †digits <expr> ] <range>
                ::=   †integer <range>
                ::=   †overlay { <field> } ;
                ::=   †record  { <field> } ;
                ::=   †pointer <base type>
                ::=   †string <string length>
                ::=   <qualified id>
<range>         ::=   <low bound> <high bound>
<low bound>     ::=   <expr>
<high bound>    ::=   <expr>
<bounds>        ::=   <range>
                ::=   †dynamic <type>          (* dynamic array *)
<string length> ::=   †dynamic
                ::=   <length>
<length>        ::=   <expr>
<field>         ::=   †field <type>
<base type>     ::=   <type>
<expr>          ::=   <qualified id>
                ::=   Number
                ::=   Char
                ::=   String
                ::=   <unary op> <expr>
                ::=   <binary op> <expr> <expr>
                ::=   '?' <type>               (* size of *)
                ::=   '!' <type> <expr>        (* convert result *)
```



```
                ::=   '[' <lvalue> <expr> ']' (* array/field ref *)
                ::=   '[' <lvalue> <expr> <expr> ']'  (* array slice ref *)
                ::=   '(' <lvalue> { <expr> } ')'     (* function call *)
<unary op>      ::=   '-'                      (* negation,complement or not *)
<binary op>     ::=   '+'                      (* add and concatenate *)
                ::=   '-'                      (* sub *)
                ::=   '*'                      (* mul *)
                ::=   '/'                      (* div *)
                ::=   ''                       (* rem *)
                ::=   '%'                      (* mod *)
                ::=   '&'                      (* bitwise or boolean and *)
                ::=   '|'                      (* bitwise or boolean or  *)
                ::=   '†'                      (* bitwise or boolean xor *)
                ::=   '='                      (* eq  *)
                ::=   '<>'                     (* neq *)
                ::=   '<='                     (* leq *)
                ::=   '<'                      (* lt  *)
                ::=   '>='                     (* geq *)
                ::=   '>'                      (* gt  *)
                ::=   '@'                      (* indirection *)
                ::=   '#'                      (* address of *)
<addr>          ::=   <expr>                   (* names address or label *)
<lvalue>        ::=   <expr>                   (* ok for lhs of := *)
<statement>     ::=   .assign <lvalue> <expression>   (* assignment *)
                ::=   .branch <addr> [ <conditional> ]       (* goto *)
                ::=   .case <expr> <otherwise> { <case> }    (* case *)
                ::=   .call <addr> { <expr> } (* proc call *)
                ::=   .return [ <expr> ]      (* return *)
<conditional>   ::=   ( †true | †false ) <expr>
<case>          ::=   <selector> <addr>
<otherwise>     ::=   <addr>
<selector>      ::=   <expr>
<static level>  ::=   †static <block id>
<lexical level> ::=   †lexical <block id>
<qualified id>  ::=   Ident
                ::=   '.' Ident <block id>    (* specific lexical level *)
<block id>      ::=   Number
```

Parse-table statistics

The BNF was run through an LALR(1) parser generator and an LL(1) parser generator. The LALR(1) grammar consists of 76 terminals, 51 nonterminals and 141 grammar rules, yielding 190 states. No shift/reduce or reduce/reduce conflicts exist. The LL(1) grammar has 128 productions with 75 terminals and 121 symbols in all. No predict/predict conflicts were reported.

Examples

The following examples show ADA source code and the resulting IR for several program fragments:

```
ADA:    type integer is range -32000 .. 32000;

IR:     : integer †type †integer -32000 32000 -- integer differs from .integer
```

```
ADA:    type LINE is array (1..10) of character;

IR:     : LINE †type †array 1 10 †char
```

```
ADA:    type MATRIX is array (1..10, 1..10) of integer;

IR:     : MATRIX †type †array 1 10 †array 1 10 integer
```

```
ADA:    type DATE is
        record
           DAY  : INTEGER range 1 .. 31;
           MONTH        : INTEGER range 1 .. 12;
           YEAR : INTEGER range 0 .. 4000;
        end record;

IR:     : DATE †record
           †field †integer 1 31 ;        -- DAY
           †field †integer 1 12 ;        -- MONTH
           †field †integer 0 4000;       -- YEAR
```



```
ADA:    type DEVICE is (PRINTER, DISK, DRUM);
        type STATE  is (OPEN, CLOSED);
        type PERIPHERAL(UNIT : DEVICE := DISK) is
           record
              STATUS : STATE;
                case UNIT is
                 when PRINTER =>
                    LINE_COUNT : INTEGER range 1 .. 66;
                 when others =>
                    CYLINDER   : INTEGER;
                    TRACK      : INTEGER;
              end case;
           end record;
```

```
IR:     : DEVICE †type †integer 0 2
        : STATE  †type †integer 0 1
        : PERIPHERAL †record
           †field DEVICE          -- This is the tag field UNIT
           †field STATE ;         -- STATUS
           †field †overlay
              †field †integer 1 66 ;     -- LINE_COUNT
              †field †record
                 †field integer ;-- CYLINDER
                 †field integer ;-- TRACK
              ;
           ;
        ;
```

ADA:

```
type VECTOR is array(0..10) of REAL;
function DOT_PRODUCT(X, Y : VECTOR) return REAL is
   SUM : REAL := 0.0;
begin
   for J in X'RANGE loop
      SUM := SUM + X(J)*Y(J);
   end loop;
   return SUM;
end DOT_PRODUCT;
```

IR:

```
: VECTOR ↑type ↑array real                -- real is predefined
: DOT_PRODUCT ↑procedure ↑return real
.begin ↑static 1 ↑lexical 1
   : X ↑variable VECTOR ↑enforce ↑in -- in parameter of type VECTOR
   : Y ↑variable VECTOR ↑enforce ↑in -- checking should be enforced
   : SUM ↑variable integer ↑enforce ↑speed -- optimize for speed
   := SUM 0.0                             -- executable
   : J ↑variable ↑integer 0 10 ↑enforce ↑speed
   := J 0
   : Loop ↑label
      := SUM + SUM * [X J] [Y J] -- SUM := SUM + (X(J) * Y(J))
      .branch End ↑true = J 10 -- if J = 10 then goto end
      := J + J 1
      .branch Loop -- unconditional branch
   : End ↑label
   .return SUM
.end
```


6. Summary and Conclusions

A variety of intermediate representations have been compared and contrasted in this paper. All these representations serve as a good model for some phase of compilation. Quadruples and triples have been conventionally useful for program optimization. They possess the deficiency of requiring explicit temporary specification. The number of temporaries required for evaluation is often a machine-dependent issue. Furthermore, N-ary operations such as array indexing and procedure calls must be realized as multiple unary and binary operations. An extension of triples called N-tuples alleviates this problem but it continues to inherit the drawbacks of explicit temporary specification. Trees allow a more complete representation of programming language semantics but they do not promote efficient pattern matching for purposes of target code generation. They often contain redundant information in parse-tree form. Graphs are not a good model for code generation. Cycles must be eliminated and pattern matching poses a greater problem. Prefix and postfix notations by themselves are inadequate for complete representation of programming language semantics. However, the incorporation of attributes in the IR provides a fairly adequate representation of semantics and the prefix nature of the IR retains code generator efficiency. Furthermore, attributes are very useful in guiding the code generator to generate efficient target code.

In our opinion, a linear prefix intermediate representation augmented with attributes seems to be very promising when considering compiler retargetability and code generation efficiency. It has served to isolate machine dependencies

to a separate package. Apart from augmenting a prefix-IR with attributes we have also elevated the level of the IR to represent a suitable interface between language-dependent and machine-dependent issues (i.e., the IR helps to "cleanly" serve as a demarcation or boundary-line between the language-dependent and machine-dependent aspects of compilers). As a consequence, storage binding falls out as part of the issue of code generation to be done when processing the IR to produce target machine code. The motivation for such a view of the IR is an attempt to isolate all machine-dependent aspects of compiler code generation to a single software package. In contrast, tree-based IRs (e.g., TCOL) possess a multi-pass structure for code generation in which the trees are queried every phase.

The following advantages of using an attributed prefix representation seem evident:

(1) Attributes are very useful for storage binding.

(2) Attributes provide the "flexibility" needed for representing popular programming languages. Moreover, in order to cater to the idiosynchrasies of particular architectures (e.g. object-based addressing on the iAPX-432), even more variable accessing modes could be added as attributes without any real difficulty.

(3) Attributes serve as a suitable interface to help machine-independent global optimizers convey information to the code generator. Thus, they are useful for optimization of target code.

(4) Attributes are more abstract than trees in the sense that there is less need to restructure them.

(5) In table-driven approaches [Ganapathi 81], the code generation problem is divided into parts: register allocation, storage binding and instruction selection. These phases interact very strongly. Attributes are a powerful and unique mechanism to bind these interactions cohesively.

Almost all of the advantages of a flexible IR design for compiler portability and efficient code generation are provided in one single concept -- attributes. This design therefore provides ease of understanding (i.e., a "low confrontation" level to the compiler designer) and simplicity of use. It provides a better interface between machine-independent and machine-dependent parts of compilers and significantly simplifies retargeting all aspects of code generation to new machines.

It is fairly easy to produce this IR from a compiler front-end. Some experiments were conducted in which a Modula front-end produced this IR. The results suggest that the front-end need not take more than a few seconds to produce this IR for reasonably large programs. Moreover, it is very easy to interface the IR with other language front-ends. On the other hand, it is also fairly easy to process this IR to produce target machine code. The IRs produced in the experiments were targeted to the PDP-11/70, VAX-11/780 and Intel 8086 using code generators that were automatically produced from an attributed grammar description of the target machines [Ganapathi 80].

In conclusion, we have presented:

(1) the design of an IR to ease compiler portability

(2) the use of attributes in the IR to interface machine-independent aspects of a compiler with the machine-dependent parts.

(3) the use of attributes in the IR to produce optimized object code.

Attributed linear IRs prove to be as useful as tree-oriented IRs and in some ways better (e.g., code generator efficiency). Experience with code generators suggests this point. We hope that the design philosophies in attributed prefix IR has (a) served to simplify and (b) provided more insight into restructuring the organization of portable compilers.

8. Bibliography

[Aho 77] A.V. Aho and J.D. Ullman, "Principles of Compiler Design", Addison-Wesley publishing Co., 1977.

[Brosgol 80a] B.M. Brosgol, J.M. Newcomer, D.A. Lamb, D.R. Levine, M.S. Van Deusen, W.A. Wulf, "$TCOL_{Ada}$: Revised Report on an Intermediate Representation for the Preliminary Ada Language", Technical Report CMU-CS-80-105, Carnegie-Mellon University, February 1980.

[Brosgol 80b] B.M. Brosgol, "$TCOL_{Ada}$ and the 'Middle End' of the PQCC Ada Compiler", Proceedings of SIGPLAN Symposium on the Ada Programming Language", December 1980.

[Coleman 73] S.S. Coleman, P.C. Poole and W.M. Waite, "The Mobile Programming System: Janus", National Tech. Infor. Center PB220322, U.S. Dept. of Commerce, Springfield, Va., 1973.

[Dausmann 80] M. Dausmann, S. Drossopolou, G. Goos, G. Persch, G. Winterstein, "AIDA - An Informal Introduction", University of Karlsruhe, Federal Republic of Germany, February 1980.

[Donegan 79] M.K. Donegan et al., "A Code Generator Generator Language", Proceedings of the SIGPLAN Symposium on Compiler Construction, Denver, Colorado, August 6-10, 1979.

[Elson 70] M. Elson and S.T. Rake, "Code Generation Technique for Large Language Compilers", I.B.M. Systems Journal Vol. 9 No. 3 pp. 166-188, 1970.

[Feldman 64] J. Feldman, "A Formal Semantics for Computer Oriented Languages", PhD thesis, Computer Science Department, Carnegie Mellon University, 1964.

[Feustal 73] E.A. Feustal, "On the Advantages of Tagged Architecture", IEEETC, Vol. C-22 No. 7, July 1973.

[Frailey 79] D.J. Frailey, "An Intermediate Language for Source and Target Independent Code Optimization", Proceedings of the SIGPLAN Symposium on Compiler Construction, Denver, Colorado, August 6-10, 1979.

[Ganapathi 80] M. Ganapathi, "Retargetable Code Generation and Optimization using Attribute Grammars", PhD thesis and Tech. Report #406, Computer Sciences Department, University of Wisconsin - Madison, 1980.

[Ganapathi 81] M. Ganapathi and C.N. Fischer, "A Review of Automatic Code Generation Techniques", Tech. Report #407, Computer Sciences Department, University of Wisconsin - Madison, 1981.

[Glanville 78] R.S. Glanville and S.L. Graham, "A New Method for Compiler Code Generation", Conf. Record Fifth ACM Symp. Principles of Programming Languages, Jan. 1978.

[Gries 71] D. Gries, "Compiler Construction for Digital Computers", John Wiley & Sons, 1971.

[Griswold 68] R.E. Griswold, J.F. Poage and I.P. Polonsky, "The SNOBOL4 Programming Language", Bell Telephone Labs, Prentice-Hall Inc., Englewood Cliffs, New Jersey 1968.

[Hennessy 79] J.L. Hennessy, "Symbolic Debugging of Optimized Code", Technical Report #175, Computer Systems Lab, Stanford Electronics Labs, Dept. of Electrical Engineering, Stanford University, SEL 79-026, 1979.

[Ichbiah 79] J. Ichbiah et al., "Preliminary Ada Reference Manual", SIGPLAN Notices, Vol. 14 No. 6, June 1979.

[Johnson 75] S.C. Johnson "YACC - Yet Another Compiler Compiler", C.S. Tech Report #32, Bell Telephone Laboratories, Murray Hill, New Jersey, 1975.

[Nelson 79] P.A. Nelson, "A Comparison of Pascal Intermediate Languages", Proceedings of the SIGPLAN Symposium on Compiler Construction, Denver, Colorado, August 6-10, 1979.

[Pebbleman 79] Pebbleman Report, Revised, U.S. Department of Defence, January 1979.

[Satterthwaite 75] E.H. Satterthwaite, "Source Language Debugging Tools", PhD dissertation, Stanford University, STAN-CS-75-494, 1975.

[Sites 79] R.L. Sites and D.R. Perkins, "Machine-Independent Pascal Code Optimization", Proceedings of the SIGPLAN Symposium on Compiler Construction, Denver, Colorado, August 6-10, 1979.

[Steel 61] T.B. Steel, Jr., "A First Version of UNCOL", Proceedings WJCC, 19, pp. 371-378, 1961.

[Strong 58] J. Strong et al., "The Problem of Programming Communication with Changing Machines: A Proposed Solution", CACM Vol.1 No. 8 pp. 12-18, 1958.

[Szewerenko 81] L. Szewerenko, W.B.Dietz and F.E. Ward, "Nebula: A New Architecture and Its Relationship to Computer Hardware", IEEE Computer Vol. 14 No. 2, February 1981.

[Wilcox 71] T.R. Wilcox, "Generating Machine Code for High Level Programming Languages", Tech. Report 71-103, PhD thesis, Dept. of Computer Sciences, Cornell University, 1971.

[Wirth 71] N. Wirth, "The Design of the Pascal Compiler", Software-Practice and Experience 1:4, 309-333, 1971.

[Wulf 75] W. Wulf, R.K. Johnsson, C.B. Weinstock, S.O. Hobbs and C.M. Geschke, "The Design of an Optimizing Compiler", American-Elsevier, 1975.

8. Appendix A: LALR(1) grammar (YACC)

```
%token
        LT
        LE
        NE
        GT
        GE
        Assign
        Begin
        Branch
        Call
        Case
        End
        Return
        IDENT
        CHAR
        NUMBER
        STRING
        _array
        _bits
        _boolean
        _char
        _delta
        _digits
        _dynamic
        _enforce
        _export
        _false
        _field
        _fixed
        _float
        _import
        _in
        _initialize
        _integer
        _label
        _lexical
        _out
        _overlay
        _pointer
        _procedure
        _readonly
        _record
        _reference
        _return
        _space
        _speed
        _static
```



```
        _string
        _task
        _true
        _type
        _usage
        _variable
%%
ir              :   commands

commands        :   commands command
                |   command

command         :   directive
                |   declaration
                |   statement

directive       :   Begin optStatic optLexical
                |   End

optStatic       :   staticLevel

optLexical      :   lexicalLevel

declaration     :   ':' IDENT attributes

attributes      :   _type type typeAttrs
                |   _label labelAttrs
                |   _variable type varAttrs
                |   _procedure procAttrs

labelAttrs      :   labelattrs

labelattrs      :   labelattrs labelattr
                |   labelattr

labelattr       :   _import
                |   _export
                |   lexicalLevel

varAttrs        :   varattrs

varattrs        :   varattrs varattr
                |   varattr

varattr         :   _import
```

```
                    | _export
                    | _enforce
                    | _speed
                    | _space
                    | _usage
                    | _reference
                    | _in
                    | _initialize exprlist
                    | _out
                    | staticLevel
                    | lexicalLevel
                    :
procAttrs           : procattrs
                    :
procattrs           : procattrs procattr
                    | procattr
                    :
procattr            : _import
                    | _export
                    | _return type
                    | _task
                    | lexicalLevel
                    :
typeAttrs           : typeattrs
                    :
typeattrs           : typeattrs typeattr
                    | typeattr
                    :
typeattr            : lexicalLevel
                    :
type                : _array bounds basetype
                    | _bits length
                    | _boolean
                    | _char
                    | _fixed delta range
                    | _float digits range
                    | _integer range
                    | _overlay fields
                    | _record fields
                    | _pointer basetype
                    | _string stringlength
                    | qualifiedId
                    :
bounds              : _dynamic type
                    | range
                    :
range               : lowbound highbound
                    :
```



```
lowbound            : expr
                    :
highbound           : expr
                    :
basetype            : type
                    :
stringlength        : _dynamic
                    | length
                    :
length              : expr;
                    :
delta               :
                    | _delta expr
                    :
digits              :
                    | _digits expr
                    :
fields              : ';'
                    | fieldlist ';'
                    :
fieldlist           : fieldlist field
                    | field
                    :
field               : _field type
                    :
exprlist            :
                    | exprs
                    :
exprs               : exprs expr
                    | expr
                    :
expr                : qualifiedId
                    | NUMBER
                    | CHAR
                    | STRING
                    | unaryop expr
                    | binaryop expr expr
                    | '?' type
                    | '!' type expr
                    | '[' lvalue expr ']'
                    | '[' lvalue expr expr ']'
                    | '(' lvalue exprlist ')'
                    :
unaryop             : '~'
                    | '#'
                    | '@'
                    :
binaryop            : '+'
                    | '-'
                    | '*'
```

```
                |   '/'
                |   '\'
                |   '%'
                |   '&'
                |   '|'
                |   '+'
                |   '='
                |   NE
                |   LT
                |   LE
                |   GT
                |   GE
                ;
lvalue          :   expr
                ;
addr            :   expr
                ;
statement       :   Assign lvalue expr
                |   Branch addr conditional
                |   Case expr otherwise caselist
                |   Call addr exprlist
                |   Return expr
                ;
conditional     :
                |   _true expr
                |   _false expr
                ;
otherwise       :   addr
                ;
caselist        :
                |   cases
                ;
cases           :   cases case
                |   case
                ;
case            :   selector addr
                ;
selector        :   expr
                ;
staticLevel     :   _static blockId
                ;
lexicalLevel    :   _lexical blockId
                ;
qualifiedId     :   IDENT
                |   '.' IDENT blockId
                ;
blockId         :   NUMBER
                ;
```


9. Appendix B: LL(1) grammar (FMQ)

```
BNF for IR
*fmq
bnf
noerrortables
text
nobinary
statistics
nos
noe
*terminals
!
SHARP
%
&
(
)
*
+
,
-
/
:
;
LT
LE
NE
=
GT
GE
?
@
[
\
]
^
|
~
Ident
Char
Number
String
^array
^bits
^boolean
^char
^delta
^digits
```

```
†dynamic
†enforce
†export
†false
†field
†fixed
†float
†import
†in
†integer
†initialize
†label
†lexical
†out
†overlay
†pointer
†procedure
†readonly
†record
†reference
†return
†space
†speed
†static
†string
†task
†true
†type
†usage
†variable
*productions
<ir>              ::=    <commands>
<commands>        ::=    <command> <commands>
                  ::=
<command>         ::=    <directive>
                  ::=    <declaration>
                  ::=    <statement>
<directive>       ::=    .begin <? static level> <? lexical level>
                  ::=    .end
<? static level>  ::=    <static level>
                  ::=
<? lexical level> ::=    <lexical level>
                  ::=
<declaration>     ::=    : Ident <attribute list>
<attribute list>  ::=    †type <type> <type attrs>
                  ::=    †label <label attrs>
                  ::=    †variable <var attrs>
                  ::=    †procedure <proc attrs>
<label attrs>     ::=    <label attr> <label attrs>
                  ::=
```



```
<label attr>      ::=    †import
                  ::=    †export
                  ::=    <lexical level>
<var attrs>       ::=    <var attr> <var attrs>
                  ::=
<var attr>        ::=    †readonly
                  ::=    †import
                  ::=    †export
                  ::=    †enforce
                  ::=    †speed
                  ::=    †space
                  ::=    †usage
                  ::=    †reference
                  ::=    †in
                  ::=    †out
                  ::=    †initialize <expr list>
                  ::=    <static level>
                  ::=    <lexical level>
<proc attrs>      ::=    <proc attr> <proc attrs>
                  ::=
<proc attr>       ::=    †import
                  ::=    †export
                  ::=    †return <type>
                  ::=    †task
                  ::=    <lexical level>
<type attrs>      ::=    <type attr> <type attrs>
                  ::=
<type attr>       ::=    <lexical level>
<type>            ::=    †array <bounds> <base type>
                  ::=    †bits <length>
                  ::=    †boolean
                  ::=    †char
                  ::=    †fixed <delta> <range>
                  ::=    †float <digits> <range>
                  ::=    †integer <range>
                  ::=    †overlay <fields>
                  ::=    †record <fields>
                  ::=    †pointer <base type>
                  ::=    †string <string length>
                  ::=    <qualified id>
<base type>       ::=    <type>
<bounds>          ::=    †dynamic <type>
                  ::=    <range>
<low bound>       ::=    <expr>
<high bound>      ::=    <expr>
<string length>   ::=    †dynamic
                  ::=    <length>
<length>          ::=    <expr>
<delta>           ::=    †delta <expr>
                  ::=
```

```
<digits>          ::=     †digits <expr>
                  ::=
<range>           ::=     <low bound> <high bound>
<fields>          ::=     <field> <fields> ;
                  ::=     ;
<field>           ::=     †field <type>
<expr list>       ::=     <expr> <expr list>
                  ::=
<expr>            ::=     <qualified id>
                  ::=     Number
                  ::=     Char
                  ::=     String
                  ::=     <unary op> <expr>
                  ::=     <binary op> <expr>
                  ::=     ? <type>
                  ::=     ! <type> <expr>
                  ::=     [ <lvalue> <expr> <? slice> ]
                  ::=     ( <lvalue> <expr list> )
<? slice>         ::=     <expr>
                  ::=
<unary op>        ::=     ~
                  ::=     @
                  ::=     SHARP
<binary op>       ::=     %
                  ::=     &
                  ::=     *
                  ::=     +
                  ::=     -
                  ::=     /
                  ::=     LT
                  ::=     LE
                  ::=     NE
                  ::=     =
                  ::=     GT
                  ::=     GE
                  ::=     \
                  ::=     ↑
                  ::=     |
<lvalue>          ::=     <expr>
<statement>       ::=     .assign <lvalue> <expr>
                  ::=     .branch <addr> <? condition>
                  ::=     .case <expr> <otherwise> <cases>
                  ::=     .call <addr> <expr list>
                  ::=     .return <expr>
<addr>            ::=     <expr>
<? condition>     ::=     ↑true <expr>
                  ::=     ↑false <expr>
                  ::=
<otherwise>       ::=     <addr>
<cases>           ::=     <case> <cases>
```



```
                  ::=
<case>            ::=     <selector> <addr>
<selector>        ::=     <expr>
<static level>    ::=     ↑static <block id>
<lexical level>   ::=     ↑lexical <block id>
<qualified id>    ::=     Ident
                  ::=     . Ident <block id>
<block id>        ::=     Number
*end
```