14 Building a Runnable Program

14.2 Intermediate Forms

In this section we consider three widely used intermediate forms: Diana, GIMPLE, and RTL. Two additional examples, Java byte code and the Common Intermediate Language (CIL) can be found in Chapter 15.

Diana (Descriptive Intermediate Attributed Notation for Ada) is an Ada-specific, high-level tree-based IF developed cooperatively by researchers at the University of Karlsruhe in Germany, Carnegie Mellon University, Intermetrics, Softech, and Tartan Laboratories. It incorporates features from two earlier efforts, named TCOL and AIDA.

GIMPLE and RTL are the intermediate languages of the GNU compiler collection (`gcc`). RTL (Register Transfer Language) is the older of the two. It is a medium-level pseudo-assembly language, and was the basis of almost all language-independent code improvement in `gcc` prior to the introduction of GIMPLE in 2005. GIMPLE, like Diana, is a tree-based form, but not quite as abstract. As of `gcc` v.4, there are approximately 100 code improvement phases based on GIMPLE, and about 70 based on RTL.

14.2.1 Diana

Diana is very complex (the documentation is 200 pages long), but highly regular, and we can at least give the flavor of it here. It is formally described using a preexisting notation called IDL [SS89], which stands for Interface Description Language.[1] IDL is widely used to describe abstract data types in a machine- and implementation-independent way. Using IDL-based tools, one can automatically construct routines to translate concrete instances of an abstract data type to and

1 Unfortunately, the term "IDL" is used both for the general category of interface description languages (of which there are many) and the specific Interface Description Language used by Diana.

from a standard linear textual representation. IDL is perfectly suited for Diana. Other uses include multi-database systems, message passing across distributed networks, and compilation for heterogeneous parallel machines. In addition to providing the interface between the front end and back end of an Ada compiler, Diana frequently serves as the standard representation of fragments of Ada code in a wider program development environment.

Diana structures are defined abstractly as trees, but they are not necessarily represented that way. To guarantee portability across platforms and among the products produced by different vendors, all programs that use Diana must be able to read and write the linear textual format. Vendors are allowed (and in fact encouraged) to extend Diana by adding new attributes to the tree nodes, but a tool that produces Diana conforming to the standard must generate all the standard attributes and must never use the standard attributes for nonstandard purposes. Similarly, a tool that consumes Diana conforming to the standard may exploit information in extra attributes if it is provided, but must be capable of functioning correctly when given only the standard attributes.

Ada compilers construct and decorate the nodes of a Diana tree in separate passes. The Diana manual recommends that the construction pass be driven by an attribute grammar. This pass establishes the lexical and syntactic attributes of tree nodes. Lexical attributes include the spelling of identifier names and the location (file name, line and column number) of constructs. Syntactic attributes are the parent–child links of the tree itself.[2] Subsequent traversal(s) of the tree establish the semantic and code-based attributes of tree nodes. Code-based attributes represent low-level properties such as numeric precision that have been specified in the Ada source.

Symbol table information is represented in Diana as semantic attributes of declarations, rather than as a separate structure. If desired, an *implementation* of Diana can break this information out into a separate structure for convenience, so long as it retains the tree-based abstract interface. Occurrences of names are then linked to their declarations by "cross links" in the tree. A fully attributed Diana structure is therefore in fact a DAG, rather than a tree. The cross links are all among the semantic attributes, so the initial structure (formed of lexical and syntactic attributes) is indeed a tree.

IDL (and thus the Diana definition) employs a tree grammar notation similar to that of Section 4.6. Unlike BNF this notation defines a complete syntax tree, rather than just its fringe (i.e., the yield). To avoid the many "useless" nodes of a typical parse tree, IDL distinguishes between two kinds of symbols, which it calls *classes* and *nodes*. The nodes are the "interesting" symbols—the ones that are in the Diana tree. The classes are the "uninteresting" symbols; they exist to facilitate construction of the grammar. In effect, the distinction between classes and nodes

2 Terminology here is potentially confusing. We have been using the term "attribute" to refer to annotations appended to the nodes of a parse or syntax tree. Diana uses the term for *all* the information stored in the nodes of a syntax tree. This information includes the references to other nodes that define the structure of the tree.

```
Structure ExpressionTree Root EXP is
    -- ExpressionTree is the name of the abstract data type.
    -- EXP is the start symbol (goal) symbol of the grammar.

    Type Source_Position ;
        -- This is a private (implementation-dependent) type.

    EXP  ::= leaf | tree ;
        -- EXP is a class.  By convention, class names are written
        -- in all upper-case letters.  They are defined with "::="
        -- productions.  Their right-hand-sides must be an alternation
        -- of singletons, each of which is either a class or a node.

    tree  => as_op: OPERATOR,  as_left: EXP,  as_right: EXP ;
    tree  => lx_src: Source_Position ;
    leaf  => lx_name: String ; lx_src: Source_Position ;
        -- tree and leaf are nodes.  They are the symbols actually
        -- contained in an ExpressionTree.  Their attributes (including
        -- substructure) are defined by "=>" productions.  Multiple
        -- productions for the same node are NOT alternatives; they
        -- define additional attributes.  Thus, every tree node has four
        -- attributes: as_op, as_left, as_right, and lx_src.  Every leaf
        -- has two attributes: lx_name and lx_src.  By convention,
        -- Diana uses 'lx_' to preface lexical attributes,
        -- 'as_' to preface abstract syntax attributes,
        -- 'sm_' to preface semantic attributes, and
        -- 'cd_' to preface code attributes.

        -- In a more realistic example, leaf would have a sm_dec
        -- attribute that identified its declaration node, where
        -- additional attributes would describe its type, scope, etc.

    OPERATOR  ::=  plus | minus | times | divide ;
    plus => ;  minus => ;  times => ;  divide => ;
        -- OPERATOR is a class consisting of the standard four binary
        -- operators.  The null productions reflect the fact that an
        -- operator's name tells us all we need to know about it.
        -- We could have made the operator of a tree node a private
        -- type, eliminating the need for the null productions and empty
        -- subtree, but this would have pushed operators out of the
        -- machine-independent part of the notation, which is unacceptable.
End
```

Figure 14.11 Example of the IDL notation used to define Diana.

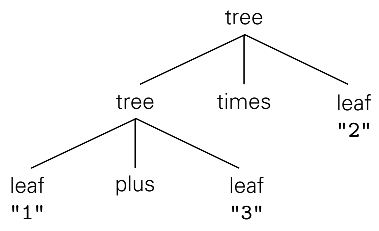

Figure 14.12 Abstract syntax tree for `(1 + 3) * 2`, using the IDL definition of Figure ©14.11. Every node also has an attribute `src` of type `Source_Position`; these are not shown here.

serves the same purpose as the $A : B$ notation introduced for the left-hand sides of productions in Section 4.6 (Figure 14.6).

EXAMPLE 14.19
`ExpressionTree` abstraction in Diana

Figure ©14.11 contains an IDL example adapted from the Diana manual [GWEB83, p. 26]. The `ExpressionTree` abstraction defined here is much simpler than the corresponding portion of Diana, but it serves to illustrate the IDL notation. An `ExpressionTree` for `(1 + 3) * 2` appears in Figure ©14.12. Note that the classes (`EXP` and `OPERATOR`) do not appear in the tree. Only the nodes (`tree` and `leaf`) appear. ■

14.2.2 The `gcc` IFs

Many readers will be familiar with the `gcc` compilers. Distributed as open source by the Free Software Foundation, `gcc` is used very widely in academia, and increasingly in industry as well. The standard distribution includes front ends for C, C++, Objective-C and C++, Ada 95, Fortran, and Java. Front ends for additional languages, including Pascal Modula-2, PL/I, Mercury, and Cobol are separately available. The C compiler is the original, and the one most widely used (`gcc` originally stood for "GNU C compiler"). There are back ends for dozens of processor architectures, including all commercially significant options. There are also GNU implementations, not based on `gcc`, for some two dozen additional languages.

`Gcc` has three main IFs. Most of the (language-specific) front ends employ, internally, some variant of a high-level syntax tree form known as GENERIC. Early phases of machine-independent code improvement use a somewhat lower-level tree form known as GIMPLE (still a high-level IF). Later phases use a linear, medium-level IF known as RTL (register transfer language).

GIMPLE is a recent innovation. Traditionally, all machine-independent code improvement in `gcc` was based on RTL. Over time it became clear that the IF had become an obstacle to further improvements in the compiler, and that a higher-level form was needed. GIMPLE was introduced to meet that need. As of `gcc` v.4, GENERIC is used for semantic analysis and, in a few cases, for certain language-specific code improvement. As its final task, each front end converts the program from GENERIC into GIMPLE. The "middle end" then performs as many as 100 phases of code improvement on the GIMPLE representation, converts to RTL, and performs as many as 70 additional phases before handing the result to the back end for target code generation.

Both GIMPLE and RTL are meant to be kept in memory across compiler phases, rather than being written to a file. Both IFs have a human-readable external format, which the compiler can write and (partially) read, but this format is not needed by the compiler: the internal version is much better suited for automatic manipulation.

GIMPLE

The GIMPLE code generated by a gcc front end is essentially a distillation of GENERIC, with many of the most complex (and often language-specific) features "lowered" into a smaller, common set of tree node types. As a simple example, consider the gcd program of Example 1.20:

EXAMPLE 14.20
GCD program in GIMPLE

```
int main () {
    int i, j;
    int i = getint(), j = getint();
    while (i != j) {
        if (i > j) i = i - j;
        else j = j - i;
    }
    putint(i);
}
```

Figure 14.13 illustrates the "high GIMPLE" produced by the C front end of gcc 4.0 when given this program as input. If we compare this GIMPLE code to Figure 14.2, which loosely[3] resembles GENERIC, we see at least two significant differences. First, temporary variables have been introduced to hold the values obtained from getint (GIMPLE declines to write the result of a function call directly to an in-memory variable). Second, the while loop has been recast with explicit gotos.

Over the course of its many phases, the gcc middle end will make many additional changes to this code, not only to improve its quality, but also to further lower its level of abstraction. The if statement inside the loop, for example, will see its then and else parts converted into simple gotos, which will jump to separate statements. This "flattening" of the tree makes it easier to translate into RTL.

Perhaps the most significant transformation of GIMPLE is the conversion to *static single assignment (SSA) form*. We will study SSA in more detail in

3 Unlike the informal notation of Figure 14.2, GENERIC and GIMPLE make no distinction between syntax tree nodes and symbol table nodes. In effect, the symbol table is merged into the syntax tree.

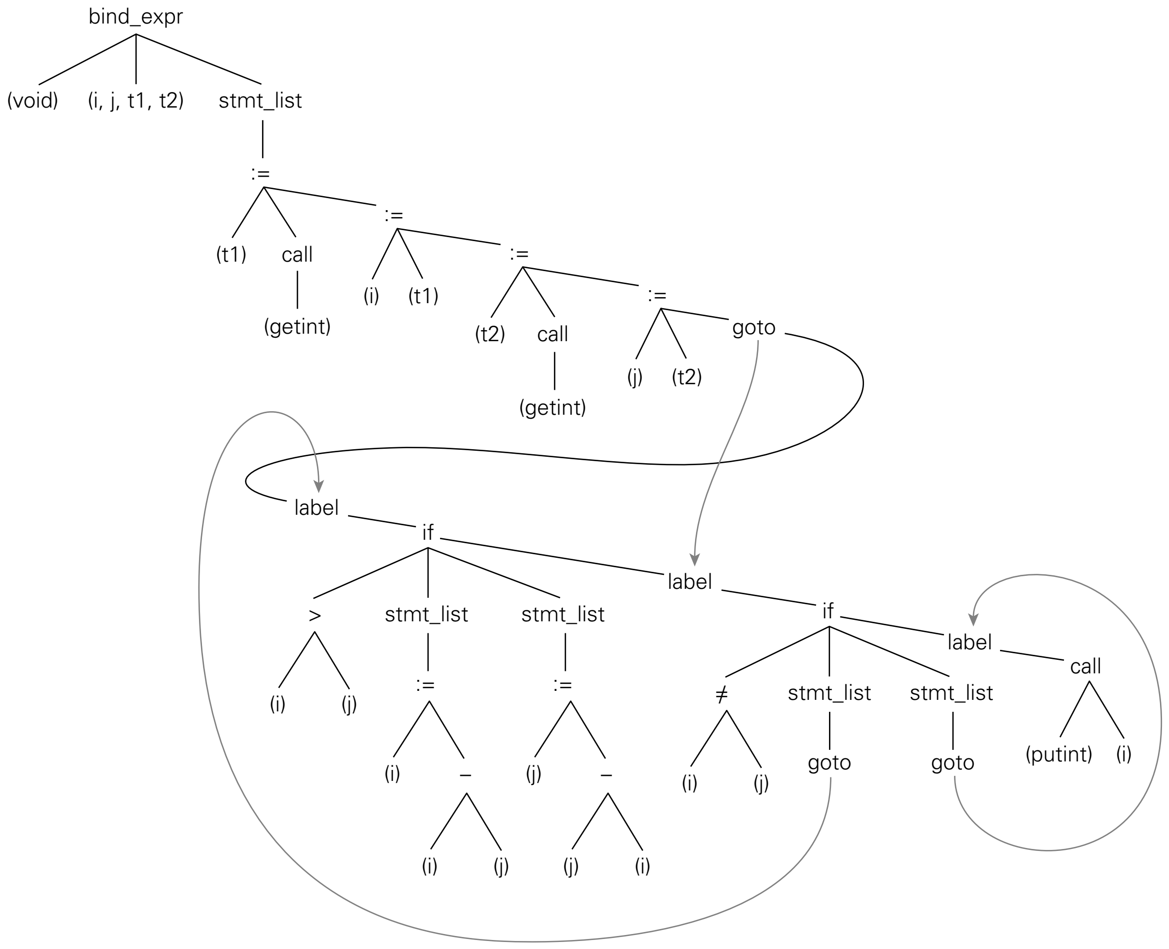

Figure 14.13 **Simplified GIMPLE for the `gcd` program.** Only structural nodes are shown: references to nodes that constitute symbol table information are indicated by parenthesized names. The node for function **`main`** would contain a pointer to the bind_expr (block) node at the root of the tree shown here.

Section 16.4.1. Briefly, the SSA conversion introduces extra variable names into the program in such a way that nothing is ever written in more than one place. If there are 10 assignments to variable foo in the source code, there will be (at least) ten separate variables $\text{foo}_1, \ldots, \text{foo}_{10}$ in SSA. When control paths merge (e.g., after an **`if... then... else`**), versions of a variable arriving on different paths are combined, using a hypothetical "phi function" to create yet another version ($\text{foo}_{11} := \phi(\text{foo}_1, \text{foo}_2)$). As in functional programming languages, the single-assignment character of SSA means that expressions are *referentially transparent*—independent of evaluation order. Referential transparency significantly simplifies many forms of code improvement.

RTL

RTL is loosely based on the S-expressions of Lisp. Each RTL expression consists of an operator or expression type and a sequence of operands. In its external form, these are represented by a parenthesized list in which the element immediately inside the left parenthesis is the operator. Each such list is then embedded in a wrapper that points to predecessor and successor expressions in linear order. Internally, RTL expressions are represented by C structs and pointers. This pointer-rich structure constitutes the interface among the compiler's many back-end phases. There are several dozen expression types, including constants, references to values in memory or registers, arithmetic and logical operations, comparisons, bit-field manipulations, type conversions, and stores to memory or registers.

The body of a subroutine consists of a sequence of RTL expressions. Each expression in the sequence is called an `insn` (instruction). Each `insn` begins with one of six special codes:

insn: an "ordinary" RTL expression.

jump_insn: an expression that may transfer control to a label.

call_insn: an expression that may make a subroutine call.

code_label: a possible target of a jump.

barrier: an indication that the previous `insn` always jumps away. Control will never "fall through" to here.

note: a pure annotation. There are nine different kinds of these, to identify the tops and bottoms of loops, scopes, subroutines, and so on.

The sequence is not always completely linear; `insns` are sometimes collected into pairs or triples that correspond to target machine instructions with delay slots. Over a dozen different kinds of (non-*note*) annotations can be attached to an individual `insn`, to identify side effects, specify target machine instructions or registers, keep track of the points at which values are defined and used, automatically increment or decrement registers that are used to iterate over an array, and so on. `Insns` may also refer to various dynamically allocated structures, including the symbol table.

EXAMPLE 14.21
An RTL `insn` sequence

A simplified `insn` sequence for the code fragment `d := (a + b) * c` appears in Figure ◎14.14. The three leading numbers in each `insn` represent the `insn`'s unique id and those of its predecessor and successor, respectively. The `:SI` *mode specifier* on a memory or register reference indicates access to a single (4-byte) integer. Fields for the various `insn` annotations are not shown. ■

In order to generate target code, the back end matches `insns` against patterns stored in a semiformal description of the target machine. Both this description and the routines that manipulate the machine-dependent parts of an `insn` are segregated into a relatively small number of separately compiled files. As a result, much of the compiler back end is machine independent, and need not actually be modified when porting to a new machine.

```
(insn 8 6 10 (set (reg:SI 2)
        (mem:SI (symbol_ref:SI ("a")))))

(insn 10 8 12 (set (reg:SI 3)
        (mem:SI (symbol_ref:SI ("b")))))

(insn 12 10 14 (set (reg:SI 2)
        (plus:SI (reg:SI 2)
            (reg:SI 3))))

(insn 14 12 15 (set (reg:SI 3)
        (mem:SI (symbol_ref:SI ("c")))))

(insn 15 14 17 (set (reg:SI 2)
        (mult:SI (reg:SI 2)
            (reg:SI 3))))

(insn 17 15 19 (set (mem:SI (symbol_ref:SI ("d")))
        (reg:SI 2)))
```

Figure 14.14 Simplified textual version of the RTL for `d := (a + b) * c`.

CHECK YOUR UNDERSTANDING

24. Characterize Diana, GIMPLE, RTL, Java byte code, and Common Intermediate Language as high-, medium-, or low-level intermediate forms.
25. What is an *interface description language*?
26. Give a brief description of Diana.
27. Explain the distinction between *attributes* and *nodes* in Diana.
28. Name three languages (other than C) for which there exist `gcc` front ends.
29. What is the internal IF of `gcc`'s front ends?
30. Give brief descriptions of GIMPLE and RTL. How do they differ? Why was GIMPLE introduced?