Diss. ETH No. 10497

Code-Generation On-the-Fly:

A Key to Portable Software

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH

for the degree of
Doctor of Technical Sciences

presented by
Michael Steffen Oliver Franz
Dipl. Informatik-Ing. ETH
born 1st May 1964
citizen of Germany

accepted on the recommendation of
Prof. Dr. N. Wirth, examiner
Prof. Dr. J. Gutknecht, co-examiner

1994

Acknowledgement

I am deeply indebted to Professor N. Wirth for the privilege of being allowed to work with him. Over the years, he has taught me a wealth of wisdom that will follow me throughout my professional life, and beyond. I admire his unique ability of immediately recognizing the essential, and his courage to eliminate the superfluous even when considered indispensable by others. Often enough, I have attempted to apply his standards of judgement in my own work, and the results have been most gratifying. I thank him most sincerely for taking up the supervision of this thesis and commenting so competently on the ongoing project.

I thank Professor J. Gutknecht for his willingness to be my co-examiner and for advising me on this thesis. He is truly a great pedagogue, and if the contents of this thesis are comprehensible at all, this is also a result of his valuable criticism of an earlier draft, delivered in his unrivalled didactic style.

I am grateful to my colleagues at the Institut für Computersysteme for their openness to new ideas and their enthusiasm in discussing them, even late at night. Together, they have provided a friendly and intellectually most stimulating working environment.

Finally, I thank C. Pfister and S. Ludwig for proof-reading this thesis.

Contents

Abstract 7
Kurzfassung 9
1. Introduction 11
2. Program Representation 13
3. The SDE Encoder and Decoder 29
4. Modular Design of the Implementation 43
5. Benchmark Results 53
6. Portability and Software Components 61
7. Further Applications 73
8. Related Work 79
9. Summary and Conclusion 87
References 89

Abstract

A *technique for representing programs abstractly and independently of the eventual target architecture* is presented that yields a file representation twice as compact as machine code for a CISC processor. It forms the basis of an implementation, in which the process of code generation is deferred until the time of loading. At that point, native code is created on-the-fly by a *code-generating loader*.

The process of loading with dynamic code-generation is so fast that it requires little more time than the input of equivalent native code from a disk storage medium. This is predominantly due to the compactness of the abstract program representation, which allows to counterbalance the additional effort of code-generation by shorter input times. Since processor power is currently rising more rapidly than disk-access times and transfer rates are falling, the proposed technique is likely to become even more competitive as hardware technology evolves.

To users of the implemented system, working with modules in the abstract representation is as convenient as working with native object-files. Both kinds of file-representation coexist in the implemented system; they are completely interchangeable and modules in either representation can import from the other kind. Separate compilation of program modules with type-safe interfaces, and dynamic loading on a per-module basis are both fully supported.

Deferring code-generation until loading time can provide several new capabilities, especially when the intermediate program representation is machine-independent and thereby *portable*. It is argued that the combination of portability with practicality denotes an important step toward a *software-component industry*. Further benefits include a potential for reducing the number of recompilations after changes in source text, and a mechanism to decide at load time whether or not run-time integrity checks should be generated for a library module, eliminating the need to distinguish between development and production library configurations. All of these new possibilities have the potential of lowering the cost of software development and maintenance.

In the long run, fast on-the-fly code-generation may even replace *binary compatibility* for achieving software portability among processors implementing

the same architecture. Already today, different models of a processor family are diverging more and more and it is becoming increasingly difficult to serve all of them equally well with just one version of native code. If code is generated only at the time of loading, however, it can always be custom-tailored toward the specific processor that it will eventually run on.

Kurzfassung

Es wird eine *Technik zur abstrakten und Zielarchitektur-unabhängigen Programmrepräsentation* vorgestellt, die zu einem Dateiformat führt, welches doppelt so kompakt wie der Maschinencode eines CISC-Prozessors ist. Auf der Basis dieser Technik wurde ein System erstellt, in dem die Codegenerierung bis zum Zeitpunkt des Ladens verzögert wird. An jenem Punkt wird dann Maschinencode durch einen *codegenerierenden Lader* erzeugt.

Der Ladeprozess mit dynamischer Codegeneration ist annähernd so schnell wie das Lesen äquivalenten Maschinencodes von einem Plattenspeichermedium. Das ist hauptsächlich der Kompaktheit der abstrakten Programmrepräsentation zu verdanken, welche es gestattet, den Mehraufwand für die Codegenerierung durch einen reduzierten Eingabeaufwand auszugleichen. Vermutlich wird die vorgestellte Technik mit fortschreitender Hardwareentwicklung vergleichsweise sogar noch effizienter werden, da die Leistung von Mikroprozessoren momentan schneller wächst, als Plattenzugriffszeiten und -übertragungsraten abnehmen.

Für die Benutzer des erstellten Systems ist der Komfort bei der Verwendung von Modulen in der abstrakten Repräsentation vergleichbar mit demjenigen für herkömmliche Objektdateien. Die beiden Dateiformate können nebeneinander eingesetzt werden und sind vollständig gegeneinander austauschbar; auch können Module beider Repräsentationsformen einander wahlweise importieren. Die typsichere separate Compilation wird ebenso unterstützt wie das dynamische Laden auf Modulstufe.

Die Verzögerung der Codegeneration auf den Ladezeitpunkt birgt viele neue Möglichkeiten, besonders wenn die Programm-Zwischenrepräsentation maschinenunabhängig und damit *portabel* ist. In der vorliegenden Arbeit wird argumentiert, dass die Kombination von Portabilität und Einfachheit der Verwendung einen wichtigen Schritt hin zu einer *Industrie von Softwarebauteilen* bedeutet. Darüberhinaus wird die Voraussetzung dazu geschaffen, die Anzahl notwendiger Recompilationen nach Änderungen im Quelltext vermindern zu können. Auch wird ein Mechanismus eingeführt, der es gestattet, erst zur Ladezeit entscheiden zu müssen, ob ein Bibliotheksmodul Laufzeittests be-

inhalten soll oder nicht, womit die Notwendigkeit zur Unterscheidung von Entwicklungs- und Produktionskonfigurationen der Programmbibliotheken wegfällt. All diese neuen Möglichkeiten verkörpern ein Kostensenkungspotential für Softwareentwicklung und -unterhalt.

Langfristig ist sogar denkbar, dass schnelle dynamische Codegeneration anstelle von *Binärkompatibilität* eingesetzt werden wird, um Portabilität zwischen Prozessoren zu gewährleisten, die dieselbe Architektur implementieren. Schon heute unterscheiden sich die Modelle einer Prozessorfamilie immer stärker, und es wird immer schwieriger, mit einer einzigen Version von binärem Objektcode auf ihnen allen zufriedenstellende Leistungen zu erbringen. Wird der Maschinencode hingegen erst zur Ladezeit erzeugt, so kann er immer für denjenigen Prozessor massgeschneidert werden, auf dem er schlussendlich ausgeführt werden wird.

1. Introduction

The rapid evolution of hardware technology is constantly influencing software development, for better as well as for worse. On the downside, faster hardware can conceal the complexity and cost of badly-designed programs; Reiser [Rei89] is not far off the mark in observing that sometimes "software gets slower more quickly than hardware gets faster". On the other hand, improvements in hardware, such as larger memories and faster processors, have also provided the means for better software tools.

Often enough, methodical breakthroughs have been indirect consequences of better hardware. For instance, software-engineering techniques such as *information hiding* and *abstract data types* could be developed only after computers became powerful enough to support compilers for modular programming languages. Mechanisms such as *separate compilation* place certain demands on the underlying hardware and so had a chance of proliferation only after sufficiently capable computers were commonplace.

This thesis presents another example of a systematic technological advance that owes its viability to improved hardware. It describes a technique for representing programs abstractly in a format that is twice as compact as object code for a CISC processor. Combined with the speed of current processors and abundance of main storage, the use of an intermediate program representation based on the new technique makes it possible to accelerate the process of code generation to such a degree that it can be performed *on-the-fly* during program loading on ordinary desktop computers, even with a high resulting code quality.

A system has been implemented that permits the convenient use of program modules in this machine-independent intermediate representation, as if they had been compiled to native code. In this thesis, it is argued that such a system presents some completely new possibilities, among which lies the prospect of founding an industry offering *user-serviceable software components* that are immediately functional on more than one kind of target architecture.

Among the themes that recur throughout this thesis are *methods of program representation*, *code generation*, *modular programming*, and *software portability*.

This is a comparatively wide scope for a doctoral dissertation, which is reflected in a large and diverse list of references.

2. Program Representation

For every calculable function, there exists an **algorithm** that computes it [Chu35]. A *program* is an encoded description of such an algorithm that instructs a **universal computing machine** how to compute the corresponding function. A machine is called *universal* exactly if it can compute every function that is representable by an algorithm; Turing [Tur36] has shown that such universal machines exist. *Programming languages* correspond to abstract and relatively complex universal machines, while *digital computers* represent physical realizations of universal machines that are programmable in their characteristic machine languages.

Algorithms can be translated from one program representation into another; the mere existence of an algorithm guarantees that a corresponding program can be constructed for every machine that is universal. This chapter discusses the use of *intermediate languages* as transitional steps in such transformations between program representations, particularly solutions based on *abstract machines*. It then introduces a new program-representation technique called *semantic-dictionary encoding*, which achieves a high information density while facilitating simultaneously the efficient further translation into various machine languages.

Subdivided Compilers

Often, and for a variety of reasons, compilers are subdivided into smaller units. This may be a static separation into **modules**, or a dynamic partition into two or more separate **compilation phases** (or *passes*) that execute one after the other. The initial argument for such a division is often a physical limitation of the host machine that makes the construction of a monolithic compiler impossible. There are, however, other advantages to be gained from splitting a compiler into smaller parts, which make this approach attractive even when it is not mandated by external constraints, and keep it attractive when the constraints are eventually removed by evolving hardware technology.

Most importantly, subdividing a compiler usually reduces its complexity. The individual pieces are smaller and thereby simpler than the whole, and often the combined complexity of all parts taken together is less than that of a corresponding monolithic compiler. The reason for this is that the different parts can usually be decoupled from each other quite well, resulting in narrow interfaces between them. Complexity, on the other hand, is mostly rooted in non-local dependencies.

Another reason for structuring a compiler concerns the ease with which it can be retargeted for another architecture. By disentangling the code-generating functions from the rest of the compiler, the construction of a family of compilers for different target architectures can be simplified considerably. Adapting the compiler for a new target machine then amounts to the construction of a new **code generator**, while the rest of the compiler can remain unchanged. As a matter of fact, this technique has become so common that we tend to speak of "code generators" today as if they were self-contained programs in their own right, disregarding completely that syntactic analysis and code generation were not separated from each other in early compilers.

Intermediate Languages

A compiler transforms a program from one representation into another. When such a compiler consists of several phases, this implies the existence of further intermediate representations carrying the state of the compilation from one phase to the next. These intermediate representations are usually only transient data structures in memory, but in some compilers they have a linear form that can be stored on a data file. In the following, such stand-alone linear representations are referred to as **intermediate languages**.

Intermediate languages are interesting from the aspect of software portability. If an intermediate language is sufficiently simple, it will provide for the easy construction of a series of **interpreters** that can execute the intermediate language directly on a number of different target machines. Interpreters are usually easier to build than code generators, so that this approach is attractive, although it comes at the expense of reduced performance. Implementations that have been based on interpreted intermediate languages in this manner include *BCPL* [Ric71, RW79], *SNOBOL4* [Gri72], and *Pascal* [NAJ76].

Intermediate languages are often also used in the process of **bootstrapping** compilers onto new machines [Hal65]. Suppose that we have a *portable compiler* that translates a source language *SL* into an intermediate language *IL*, and that this compiler is itself written in *SL* and available encoded both in *SL* and in *IL*. Suppose further that we have an interpreter for *IL* that runs on a computer with a machine language *TL*. We may then modify the portable compiler (written in *SL*) to derive a native compiler that translates from *SL* to *TL* directly. By compiling it with the interpretable version of the original compiler we gain a native compiler that is written in *IL*, and hence executable by the interpreter. This compiler can now be used to create a native compiler capable of running directly on the target machine (Figure 2.1).

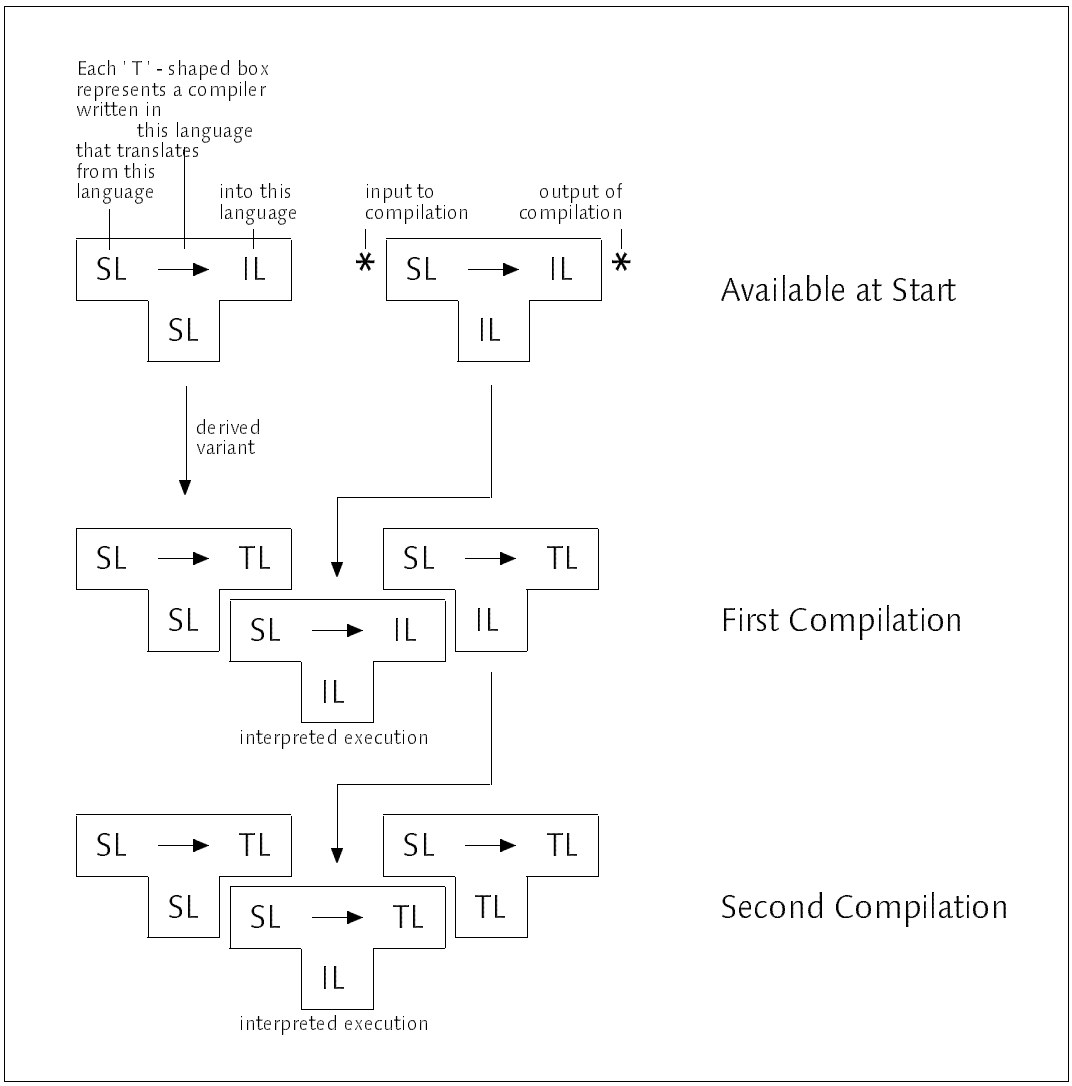

Figure 2.1: Bootstrapping a Compiler

UNCOL

In the 1950's, it became apparent that the ongoing proliferation of programming languages and hardware architectures would continue for some time. This would set off a vast demand for different compilers, as potentially one would require separate compilers for each combination of source language and target architecture.

To counteract the anticipated combinatorial explosion, the idea of a **linguistic switchbox** materialized in 1958 [SWT58, Con58, Ste60, Ste61a]. It was planned to design a *universal intermediate language*, into which programs originating in any problem-oriented language could be translated by an appropriate *generator* (*front-end* in today's terminology), and from which code for any processor architecture could be generated by a suitable *translator* (*back-end*). This would enable the construction of compilers for m languages to be run on n machines by having to write only $m+n$ programs (m front-ends and n back-ends), instead of $m*n$ (Figure 2.2). The intermediate language was to be called UNCOL, for *u*niversal *c*omputer-*o*riented *l*anguage.

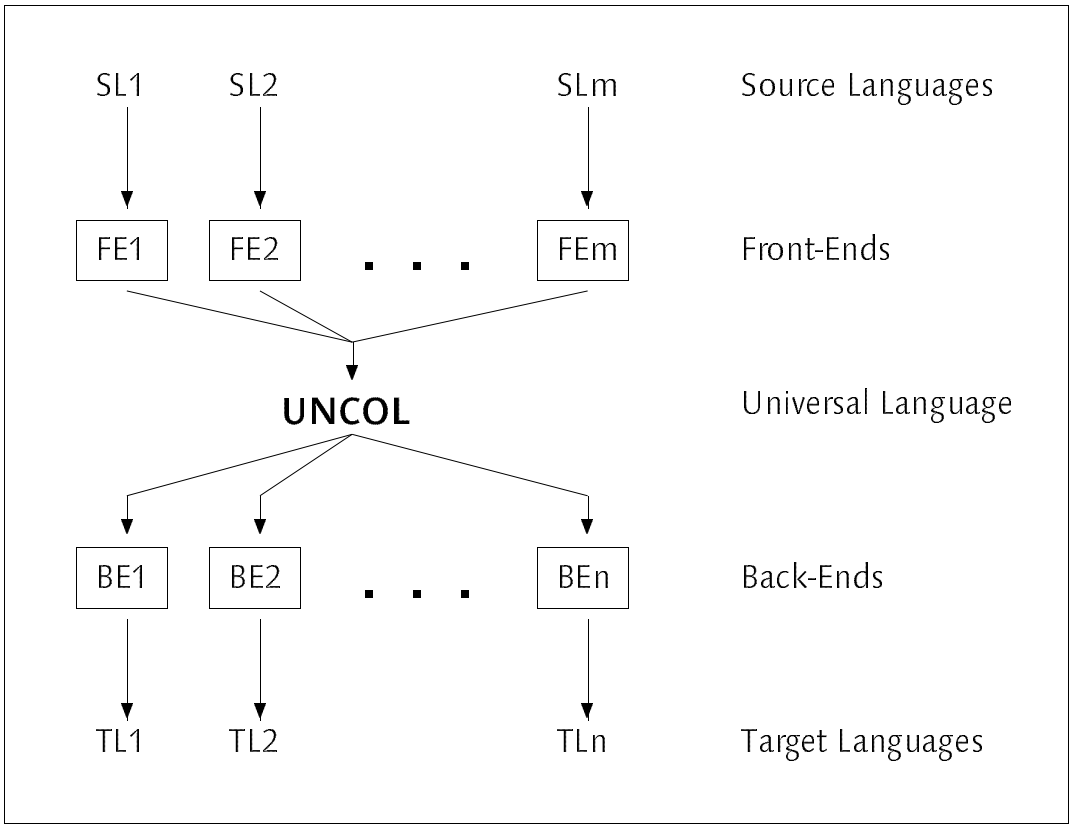

Figure 2.2: The Use of UNCOL as a Linguistic Switchbox

One might of course wonder why such an intermediate language needs to be *designed specifically* for this purpose. If algorithms can be translated from one program representation into another, then in principle the machine language of any sufficiently powerful computer could serve as UNCOL. Unfortunately, however, most translations between languages are so difficult to do that they are not practical. To wit, potential UNCOL candidates need to be amenable to efficient implementation. They should also provide for further developments in source and target languages. Steel [Ste61b] cites the example of a hypothetical UNCOL specified before the invention of *index registers*, which would be very cumbersome to use with programs operating on arrays.

To this date, no proposed UNCOL has been met with universal agreement. However, many **compiler families** have offered varying combinations of front-ends and back-ends by way of common intermediate representations, and thereby kept the spirit of UNCOL alive. Moreover, certain programming languages have been implemented on so many different architectures that they have attained an almost UNCOL-like status. For instance, the programming language *C* [KR78] is nearly pervasive. Atkinson et al. [ADH89] describe a *Cedar* compiler that achieves portability by performing high-level **source-to-source translation**, producing C as its output. The resulting C programs are then passed directly to a C compiler, without further human editing. They serve only as an intermediate representation in this context.

Abstract Machines

A straightforward method for obtaining an intermediate-language representation of a program is to represent it as an instruction sequence for some *fictitious computer*, also called an **abstract machine** [PW69, NPW72, KKM80]. Most intermediate languages, including the very first UNCOL ever proposed [Con58], follow this pattern. Besides abstract machines, intermediate languages have also been based on linearized parse-trees [GF84, DRA93a]. Furthermore, there are compilers that compile via ordinary high-level programming languages by way of source-to-source program translation as mentioned above [ADH89].

Abstract machines reached their heyday in the early 1970's. They were widely popular for a decade, providing software portability to a variety of target architectures while consuming only moderate resources. There were even architectures that started out as abstract machines and, because of their

popularity, were subsequently realized physically [WD79] or implemented in microcode [Hal82]. Some of the better known abstract machines from that era are the following:

- *BCPL*. BCPL [Ric69] is a small, block-structured programming language aimed primarily at systems-programming applications. It has only one data type, the *Rvalue*, to which the language attaches different interpretations depending on the operation applied. Each Rvalue occupies one unit of store on an idealized object computer that is visible at the level of the source language. The implementation of BCPL [Ric71, RW79] is based on an *abstract stack machine*. BCPL's type-less semantic model of an idealized computer allows for portable address arithmetic in source programs and target-machine-independent stack management in the compiler.

- *SNOBOL4*. SNOBOL4 is a member of the *SNOBOL* family of character-string-processing languages [Gri78]. Its original implementation [Gri72] uses a *register-less abstract machine* that offers only memory-to-memory instructions. The basic data unit of this abstract machine is called a *descriptor*. All of SNOBOL4's data types are encoded within such descriptors. However, the internal layout of descriptors varies between different target architectures, allowing for an efficient implementation of the interpreter on different kinds of hardware, but requiring changes in the SNOBOL4 compiler whenever the system is ported.

- *Pascal-P*. The general-purpose programming language Pascal [Wir71] is still widely used today. One of its early implementations, *Pascal-P* [NAJ76], is founded on a hypothetical stack computer similar to the abstract machine used in the implementation of BCPL. Unlike BCPL, however, Pascal differentiates between various data types. In the Pascal-P implementation, the abstract-machine representations of these types are adapted to each individual target architecture, with respect to their storage requirements, and to their alignment. As in the implementation of SNOBOL4, this target-architecture dependence prevents direct portability of the intermediate language and requires parametrization of the compiler, but results in efficient storage allocation and execution of the interpreter.

– *Janus*. Unlike the previous three examples, each of which uses an abstract machine to implement a single source language on a variety of target architectures, *Janus* [CPW74, HW78] comes closer to the original idea of UNCOL by defining the essentials of an intermediate language independently of any source language or target architecture. Rather than providing a single abstract machine, Janus describes the *common basic structure* (stack machine with index register) of a whole *family of abstract machines*, which differ in their instruction sets to accommodate the specific constructs of source languages. Janus has been used successfully in the development of compilers for the programming languages Pascal [Wir71] and Algol-68 [WMP69].

The advent of the *personal computer* eventually caused the demise of the abstract machine, since it led to a de-facto standardization of the machine language on the low end of the computer performance spectrum while simultaneously making these computers affordable. In the light of a quasi-standard machine language, however, there was no longer a need to use intermediate languages to achieve software portability, even more so as the interpreted execution of abstract machines was associated with inefficiency.

This picture is changing again only now. Although the standardization of hardware has continued over the past two decades, leaving only about a dozen major architectures to which software needs to be ported, the appeal of traditional *binary compatibility* is waning. This is because different implementations of the same architecture begin to diverge by so much that it is becoming more and more difficult to generate native code that performs well on all processors within a family. As will be shown in a later chapter, the work presented in this thesis offers an alternative, by allowing to generate code quickly at the time of loading. It is thereby possible to deliver optimized instruction sequences to individual processors while still maintaining a user-convenience comparable to that of binary compatibility.

Disadvantages of Abstract Machines

Abstract machines are often defined by forming a conceptual intersection of potential target-machine architectures. Hence, intermediate languages based on abstract machines are by their very nature closer to machine languages than to

high-level programming languages. Unfortunately, this has the undesirable consequence that the abstract-machine representation of an algorithm contains *less structure* than the corresponding source program. Most abstract machines can describe only the algorithmic aspects of a computation, and cannot correctly preserve all of the higher-order information expressed in high-level-language programs, such as block structure and data typing.

For example, consider the following source fragment in the programming language Oberon [Wir88]:

```
VAR
   ch: CHAR;
   lint: LONGINT;
BEGIN
   lint := LONG(ORD(ch));
```

A possible representation as an instruction sequence for an abstract stack computer might be the following:

```
LOAD1    ch        move single-byte CHAR value onto stack
ORD                zero-extend, result is an INTEGER
LONG               sign-extend, result is a LONGINT
STORE4   lint      move four-byte LONGINT value back to memory
```

However, this sequence of operations can be performed by a single instruction on certain processors, such as those of the *National Semiconductor 32000 family* [NS84]:

```
MOVZBD  ch, lint   move, zero-extending byte to double-word
```

Hence, it may be necessary to combine several abstract-machine instructions into one target-machine instruction when generating anything but the most naive code out of an abstract-machine intermediate representation. In these cases, the code generator effectively needs to reconstruct, at considerable expense, information that was more easily accessible in the front-end, but lost in the transition to the intermediate representation.

The only way to solve this particular problem is by bridging the *semantic gap* through the introduction, on the level of the abstract machine, of a *separate move instruction for every combination of source and destination data type*. Still, this is able to rectify only those cases that involve a direct data-format conversion. As target architectures may potentially offer arbitrarily complex operations, e.g.,

combined move-and-shift operations, the merger of several abstract-machine instructions into one single target-machine instruction can never be ruled out completely, no matter how elaborate the abstract machine's design.

Because of their systematic defects, intermediate languages based on abstract machines are not well suited as a basis for the fast generation of high-quality native code. The remainder of this chapter introduces an intermediate representation that is. This representation, which I call **semantic-dictionary encoding** (**SDE**), preserves the full semantic context of source programs while being highly compact.

An Overview of Semantic-Dictionary Encoding

SDE is a dense representation. It encodes a syntactically correct source program by a succession of indices into a **semantic dictionary**, which in turn contains the information necessary for generating native code. The dictionary itself is not part of the SDE representation, but is constructed dynamically during the translation of a source program to SDE form, and reconstructed before (or during) the decoding process. This method bears some resemblance to commonly used data compression schemes [Wel84].

With the exception of wholly constant expressions, which are evaluated during the transformation of a source program to SDE form, SDE preserves all of the information that is available at the level of the source language. Hence, unlike abstract-machine representations, transformation to SDE preserves the *block structure* of programs, as well as the *type* of every expression. Moreover, when used as the input for code-generation, SDE in certain cases provides for *short-cuts* that can increase translation efficiency.

A program in SDE form consists of a **symbol table** (in a compact format, as explained in [Fra93b]) and a series of **dictionary indices**. The symbol table describes the names and the internal structure of various entities that are referenced within the program, such as *variables*, *procedures*, and *data types*. It is used in an initialization phase, in the course of which several *initial entries* are placed into the semantic dictionary. The encoding of a program's actions consists of references to these initial dictionary entries, as well as to other entries added later in the encoding process.

Dictionary entries represent nodes in a directed acyclic graph that describes the semantic actions of a program abstractly. In its most elementary form, a

semantic dictionary is simply such an **abstract syntax-tree** in tabular shape, in which the references between nodes have been replaced by table indices. Each dictionary entry consists of a *class* attribute denoting the semantic action that the entry stands for (e.g., *assignment*, *addition*, etc.), and possibly some references to objects in the symbol table (connected via an *info* attribute in the following diagram) and to other dictionary entries (described by the *links* attribute). Figure 2.3 gives an example of a simple arithmetic expression represented as an abstract syntax tree and as a semantic dictionary.

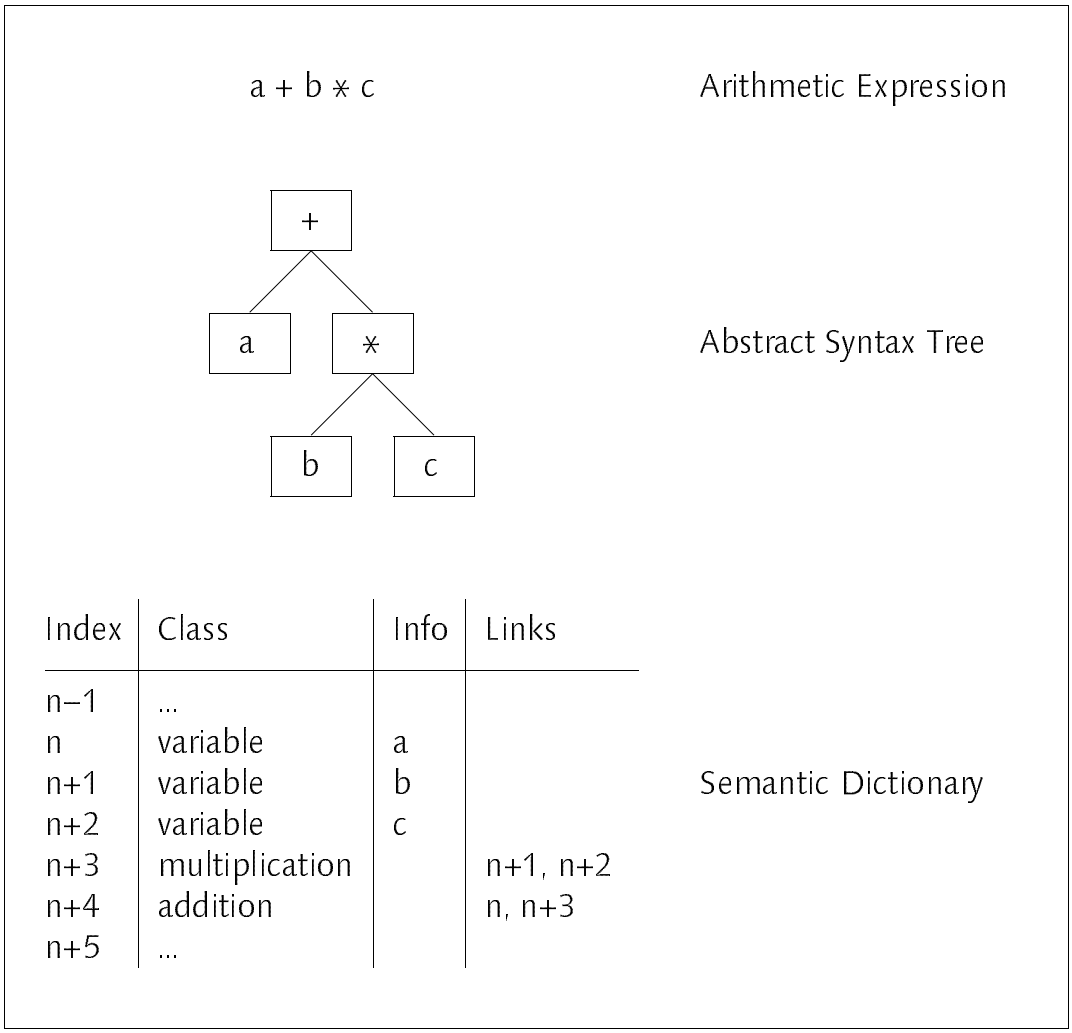

Figure 2.3: Different Representations of an Expression

What differentiates a tabular abstract syntax-tree from a semantic dictionary is that the latter can describe also **generic characteristics** of *potential nodes* that might appear in such a tree. In addition to *complete entries* that directly

correspond to nodes of an abstract syntax tree, semantic dictionaries contain also *generic*, or **template entries**. These templates have the same structure as complete entries, but at least one of their attributes is missing, as recorded in a status flag. In SDE, complex expressions can be represented not only by complete entries, but also by templates in combination with other entries. For example, the expression " a + b " can be represented by the index of an "addition" template followed by the indices of two variable-reference entries.

Now suppose that a template existed in the semantic dictionary for every construct of the source language (*assignment*, *addition*, etc.), and that furthermore the semantic dictionary were initialized in such a way that it contained at least one entry (possibly a template) for every potential use of every object in the symbol table. For example, an object describing a *procedure* in the programming language Oberon [Wir88] requires a minimum of four entries in the dictionary, relating in turn to a *call* of the procedure, *entry* of the procedure, *return* from the procedure, and *addressing* the procedure, which is used in the assignment of the procedure to procedure variables. There are no other operations involving procedures in Oberon.

These preconditions can be fulfilled by initializing the semantic dictionary in a suitable way. They enable us to represent any program by only a symbol table and a succession of dictionary indices. As an example, consider the following module *M* in the programming language Oberon:

```
MODULE M;

   VAR i, j, k: INTEGER;

   PROCEDURE P(x: INTEGER): INTEGER;
   BEGIN  ...
   END P;

BEGIN
   i := P(i);   i := i + j;   j := i + k;   k := i + j;   i := i + j
END M.
```

In order to encode this program by the method of SDE, we first need to initialize the semantic dictionary, which depends on the operations offered by the programming language (Oberon in this case) and on the objects in the symbol table. The symbol table for the example program contains three integer variables (*i*, *j*, and *k*) and a function procedure (*P*). After initialization, the

corresponding semantic dictionary might look like the following (the individual entries' indices are represented by symbolic names so that they can be referenced further along in this chapter, and missing attributes of templates are denoted by a dot).

Index	Class	Meaning	Missing
asgn	assignment	. := .	left, right
plus	addition	. + .	left, right
...	...	...	...
vi	variable	i	–
vj	variable	j	–
vk	variable	k	–
refp	address	P	–
callp	function call	P(.)	left
entp	entry	P-BEGIN	–
retp	return	P-RETURN .	left
...	...	...	...

The instruction sequence that constitutes the body of module *M* may then be represented by the following sequence of 24 dictionary indices:

asgn vi callp vi	*i := P(i)*
asgn vi plus vi vj	*i := i + j*
asgn vj plus vi vk	*j := i + k*
asgn vk plus vi vj	*k := i + j*
asgn vi plus vi vj	*i := i + j*

This is where the second major idea of SDE comes in. What if we were to keep on *adding* entries to the dictionary during the encoding process, based on the expressions being encoded, in the hope that a *similar expression* would occur again later in the encoding process? For example, once we have encoded the assignment " i := P(i) " we might add the following three entries to the dictionary:

Class	Meaning	Missing
function call	P(i)	–
assignment	i := .	right
assignment	i := P(i)	–

Thereafter, if the same assignment " i := P(i) " occurs again in the source text, we can represent it by a single dictionary index. If another assignment of a different expression to the variable *i* is come across, this may be represented using the template "assign to *i*", resulting in a shorter encoding.

Encoding module M in this manner results in the following additional entries being placed in the dictionary (assuming that, after initialization, the dictionary comprised *n-1* entries):

Index	Class	Meaning	Missing
...	...	...	...
n	function call	P(i)	–
n+1	assignment	i := .	right
n+2	assignment	i := P(i)	–
n+3	addition	i + .	right
n+4	addition	. + j	left
n+5	addition	i + j	–
n+6	assignment	i := i + j	–
n+7	addition	. + k	left
n+8	addition	i + k	–
n+9	assignment	j := .	right
n+10	assignment	j := i + k	–
n+11	assignment	k := .	right
n+12	assignment	k := i + j	–
...	...	...	...

The body of module *M* can then be encoded as follows:

asgn	vi	callp vi	*i := P(i)*
n+1		plus vi vj	*i := i + j*
asgn	vj	n+3 vk	*j := i + k*
asgn	vk	n+5	*k := i + j*
n+6			*i := i + j*

Instead of the previous 24 dictionary indices, this encoding requires only 16 of them, although the individual indices themselves may be larger since the dictionary has more entries. Still, the second encoding is usually much more space-economical and has further advantages, as will be shown in the following paragraph.

Increasing the Speed of Code Generation

The decoding of a program in SDE form is similar to the encoding operation. At first, the dictionary is initialized to a state identical to that at the onset of encoding. Since the symbol table is part of the SDE file-representation, the decoder has all required information available to perform this task.

Thereafter, the decoder repeatedly reads dictionary indices from the file, looking up each corresponding entry in the semantic dictionary. Whenever a complete entry is found in this manner, its meaning is encoded directly in the dictionary and the decoder can proceed to process the next index on the input stream. If a template is retrieved instead, it is copied, further entries corresponding to its undefined attributes are input in turn, and the modified copy is added to the dictionary as a new complete entry. Moreover, additional templates are sometimes added to the semantic dictionary according to some fixed heuristics, in the hope that a corresponding branch of the program's syntax tree will show up further along. A more detailed discussion of these heuristics follows in the next chapter.

In addition to providing a dense program representation, SDE is able to supply certain information that is not explicitly available in source text, namely about **multiple textual occurrences of identical subexpressions** (including, incidentally, *common designators*). This can sometimes be exploited during code generation, resulting in an increase in the speed of code output.

Consider again the (second, more compact) SDE-representation of module *M* above. Now suppose that we want to generate object code for a simple stack machine directly from this SDE form. Let us assume that we have processed the first two statements of *M*'s body already, yielding the following instruction sequence starting at address *a*:

```
a       LOAD    i       load i onto stack
a+1     BRANCH  P       call procedure P with argument i
a+2     STORE   i       assign result to i
a+3     LOAD    i       load i
a+4     LOAD    j       load j
a+5     ADD             add i to j
a+6     STORE   i       assign their sum to i
```

Let us further assume that we have kept a note in the decoder's semantic dictionary, describing which of the generated instructions correspond to what dictionary entry. For example, we might simply have recorded the program

counter value twice for every entry in the semantic dictionary, both before and after generating code for the entry:

Index	Class	Meaning	Begin	End	Invar
...	...	...	...	...	...
n	function call	P(i)	a	a+1	No
n+1	assignment	i := P(i)	a	a+2	No
...	...	...	...	...	...
n+5	addition	i + j	a+3	a+5	Yes
n+6	assignment	i := i + j	a+3	a+6	Yes
...	...	...	...	...	...

A setup such as this allows us to bypass the usual code generation process under certain circumstances, replacing it with a simple *copy operation* of instructions already generated. For example, when we encounter the second reference to entry *n+6* in the SDE-representation of module *M*, we know that we have already compiled the corresponding statement " i := i + j " earlier on. In this case, we may simply re-issue the sequence of instructions generated at that earlier time, which can be found in the object code between the addresses *a+3* and *a+6*, as recorded in the dictionary.

Of course, this method cannot be applied in every case and on all kinds of machine. First of all, code-copying is possible only for **position invariant** instruction sequences. For example, a subexpression that includes a call of a local function by way of a *relative branch* is not position invariant because the branch distance is different each time that the function is called. It happens that information about position invariance can also be recorded conveniently in the dictionary, as shown in the example above.

Secondly, the instruction sequences obtained by code-copying may not be optimal for more complex processors with many registers and multistage instruction-pipelines. For these machines, it may be necessary to employ specific optimization techniques. However, since SDE preserves all information that is available in the source text, arbitrary optimization levels are possible at the time of code generation, albeit without the shortcuts provided by code-copying. One then simply treats the semantic dictionary as an abstract syntax-tree in tabular form.

The interesting part is that code-copying is beneficial especially on machines that are otherwise slow, i.e., CISC processors with few registers. Consider module *M* again, in which the subexpression " i := i + j " appears three

times. On a simple machine that has only a single accumulator or an expression stack, the identical instruction sequence *will have to be used* in each occurrence of the subexpression. Code-copying can accelerate the code-generation process in this case. The optimal solution for a modern RISC processor, on the other hand, might well consist of three distinct instruction sequences for the three instances of the subexpression, due to the use of register variables and the effects of instruction pipelining. However, such processors will generally be much faster, counterbalancing the increased code-generation effort, so that an acceptable speed of code generation can still be achieved even if optimization is necessary.

3. The SDE Encoder and Decoder

The previous chapter has introduced the technique of *semantic-dictionary encoding* (SDE). On the basis of this encoding technique, a system has been implemented, in which the code generator is no longer part of the compiler, but has been incorporated into the module loader. The implemented system features an **Oberon-to-SDE compiler** that transforms programs written in the programming language Oberon [Wir88] into the SDE file-representation, and a **code-generating loader** that reads SDE-files and uses the information contained in them for generating code on-the-fly for *Motorola 68020* processors [Mot87].

This chapter discusses the encoding and decoding mechanisms and relates the motivation behind key design decisions. Most importantly, it explains some of the heuristics used in the management of semantic dictionaries.

Making use of Scoping

Many programming languages limit the scope of certain objects appearing within a program, for example by restricting the visibility of a procedure's local variables to the extent of the procedure body. These **scoping rules** can be exploited for reducing the size of the semantic dictionary. Smallness of the semantic dictionary is beneficial because a program in SDE form is composed of *dictionary indices*. The larger the dictionary, the more bits are required on average for the representation of these indices.

It is therefore desirable to remove from the semantic dictionary continuously all entries which (directly or indirectly) relate to objects that have become inaccessible by the scoping rules. However, the encoder and the decoder of the SDE format need to synchronize their dictionary operations. Any removal of an entry from the encoder's dictionary must, therefore, be communicated to the decoder, and this can happen only via the encoded program itself.

We also note that there are objects that are visible throughout a module, such as *constants* and *global variables*. Furthermore, the dictionary contains also

the *operation templates* that are used for encoding the semantic actions of a program. These must never be removed from the dictionary.

An easy and efficient solution to dictionary management, taking into account these differences between temporary and permanent entries, is to let the dictionary grow *in two directions* simultaneously, as illustrated in Figure 3.1. Entries with unlimited scope are added at one end of the dictionary, which grows unboundedly. All other nodes are added at the opposite end of the dictionary, which is managed as a stack. Every time a new scope is opened, the dictionary's current extent in this second direction is marked. It is reset to the same point when the scope is closed again. The decoder of the SDE format can keep track of these mark and restore operations by monitoring its input stream for the occurrence of dictionary indices denoting *procedure entry* and *return*.

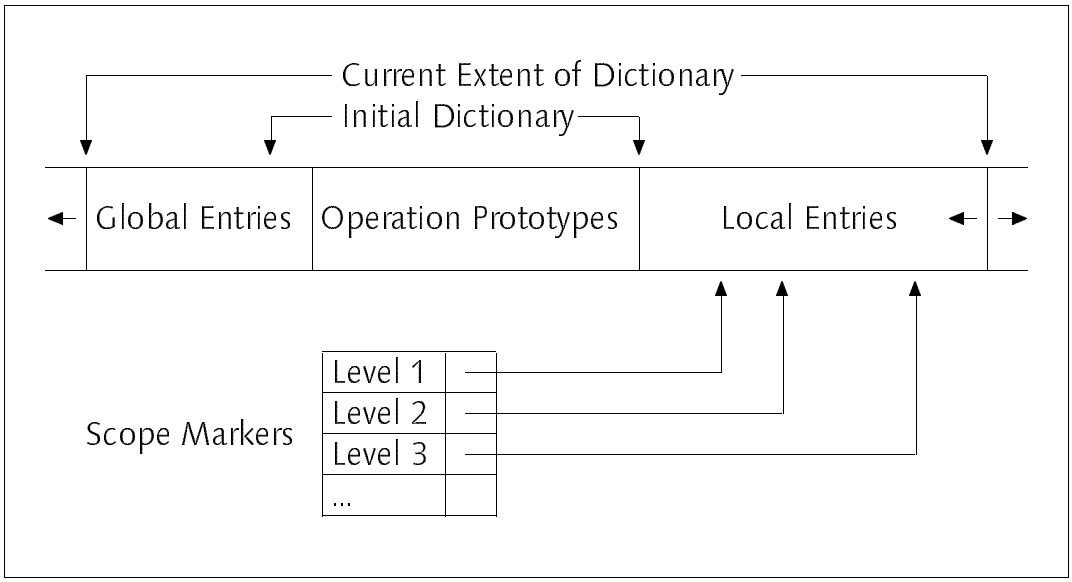

Figure 3.1: Two Directions of Dictionary Growth

Representing Indices on the File

In the implemented system, dictionary indices are represented on SDE-files in a **variable-length data format** based on a *stop-bit*, as has been described in [Fra93b]. This variable-length data format requires less space for encoding small absolute values than it needs for encoding large absolute values.

Such a compact format is ideally suited for representing dictionary indices. However, the information density of the resulting encoding depends significantly on the magnitudes of the individual indices and, therefore, on the

distance of entries from the origin of the dictionary. For a compact encoding, it is essential to *map the index vector* onto the dictionary in such a way that often-occurring dictionary entries come to lie within the range that can be addressed by a minimum number of bits.

Experiments have revealed that the most common dictionary references in SDE-encoded programs relate to templates describing the primitive operations of the source language, and to expressions involving local variables. A dictionary addressing scheme as indicated in Figure 3.2 has, therefore, been adopted. Since most procedures start at lexical level zero and are relatively short (i.e., they generate few new dictionary entries), a high proportion of the dictionary indices appearing in typical SDE representations will then actually fall within the range of indices that can be represented space-efficiently.

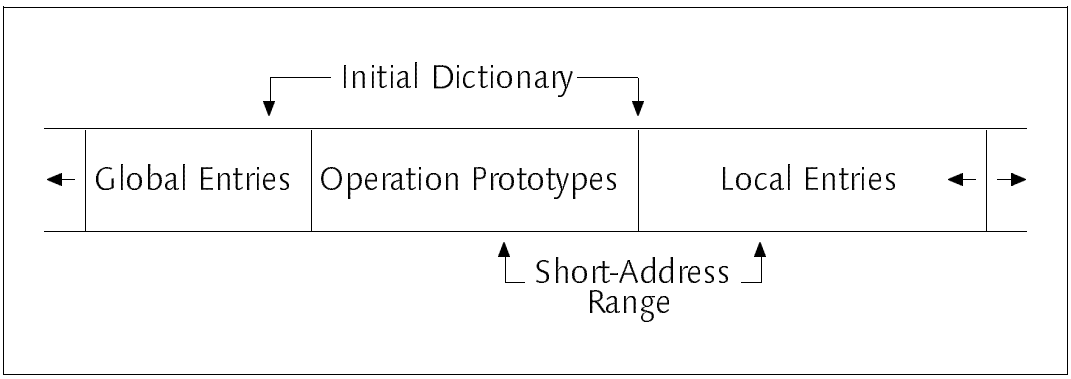

Figure 3.2: Relative Position of the Short Dictionary Addressing Range

Encoding Oberon Source-Programs into SDE

The **Oberon-to-SDE compiler** incorporates a fairly conventional compiler front-end [Cre91]. It parses each Oberon source text by recursive descent and builds a symbol table as well as an **abstract syntax tree**, i.e., a directed acyclic graph that describes the semantic actions of the program. If no error is detected during parsing, control passes to the **encoder**, which creates an SDE-file representing the contents of the source program in a compact and machine-independent manner.

In doing so, the encoder *traverses* the abstract syntax tree. For each node reached during this traversal, it searches the current semantic dictionary for an entry with a high **conformance** to the node. Conformance describes how well a dictionary entry matches a node of the tree. An entry is **fully conforming**, or

isomorphic, to a node of the abstract syntax tree, if both of them describe the same semantic action, and if they both either have no descendants at all, or if all of their corresponding descendants are isomorphic to each other. For example, in Figure 3.3, the dictionary entry with index *n+2* is isomorphic to the node labelled *"t"*, and the entry with index *n* is isomorphic to node *"u"*.

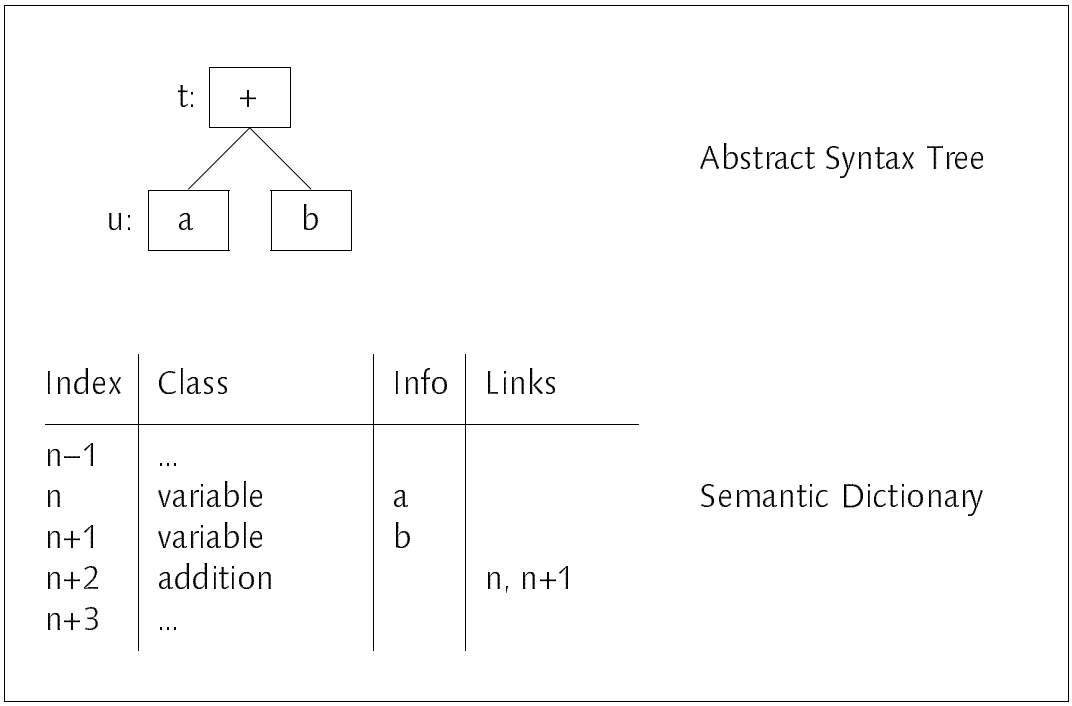

Figure 3.3: Dictionary Entry "n+2" *is Isomorphic to Node* "t".

Only complete entries (i.e., entries with no undefined descendants) can conform fully to syntax-tree nodes. Conformance for templates is defined in a more modest manner. A template conforms to a node of the abstract syntax-tree, if it describes the same semantic action and all of its *defined descendants* are isomorphic to the corresponding descendants of the node. For example, in Figure 3.4 below, the templates with indices *x*, *x+1*, and *x+2* all conform to the node labelled *"t"*.

The search for a conforming dictionary entry succeeds always, because through the initialization of the dictionary we guarantee that there is at least one minimally conforming template in the dictionary for every possible node in the abstract syntax tree. After finding a conforming dictionary entry, the encoder writes its index to the SDE-file and then traverses recursively all of the sub-trees of the abstract syntax-tree that correspond to undefined descendants of the

conforming entry. In cases in which there are several alternative entries that all conform to a node in the abstract syntax tree, as in the example of Figure 3.4, the encoder can maximize the information density of the SDE-file by selecting always the entry with the largest number of defined descendants.

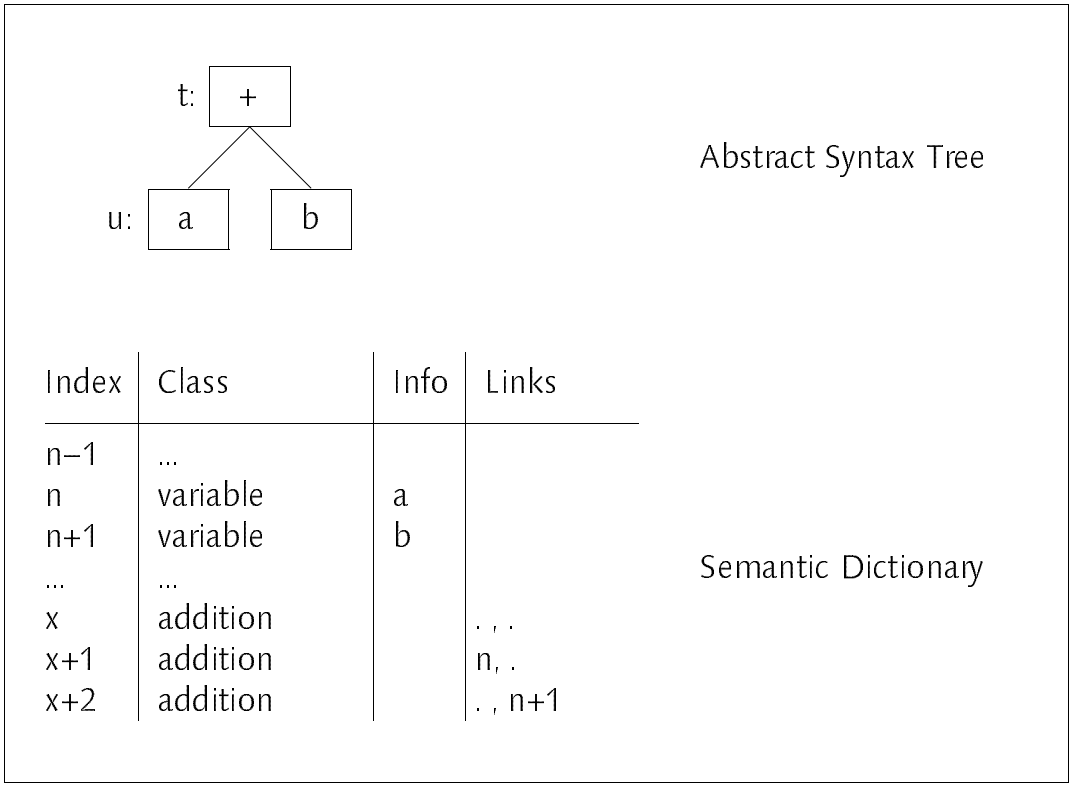

Figure 3.4: Dictionary Entries "x", "x+1", *and* "x+2" *all Conform to Node* "t".

The representation used in SDE-files is heavily biased towards the ease of *decoding*. The process of *encoding* requires many searches in the dictionary and is, therefore, inherently less efficient than decoding. In principle, it would be quite possible to perform all of these required searches by traversing the whole dictionary linearly each time until an adequate entry had been found. However, this would make the semantic-dictionary encoding of large programs unpractical.

Instead of searching the whole dictionary over and over, the current implementation uses an overlay sorting structure to reduce the number of dictionary entries that need to be inspected. Dictionary entries are grouped by their semantic action in this sorting structure, and all entries related to a certain procedure-scope are removed as soon as that scope has been closed.

Note that for correctness of the algorithm, it would suffice to reset the

dictionary *counter* at the end of each scope, while the entries themselves would not have to be removed from the dictionary. Due to the nature of the scoping rules, it would be impossible for any dictionary entry ever to be returned as the result of a search after the corresponding scope had been closed.

Heuristics of Dictionary Management

There are several time and space trade-offs to be considered in the algorithm that manages the semantic dictionary. This becomes most important when considering the strategy by which new templates are added. In general, adding a greater number of templates leads to a denser program representation on SDE-files, but it also inflates the size of the dictionary and thereby the memory requirements of the encoding and decoding steps. It may also increase the time of decoding, due to the overhead of having to allocate a larger dictionary and having to perform more initializations. The following discussion should exemplify the sort of questions that need to be answered in the design of dictionary-management heuristics. As before, an expression syntax is used, in which a dot stands for a missing link of a template entry.

Consider the expression " a + b ". Since *addition* is a fundamental operation of the Oberon language, a template for the generic expression " . + . " is guaranteed to exist in every Oberon-specific semantic dictionary after its initialization. Similarly, if some variables " a " and " b " appear in the symbol table of a certain program, then the corresponding semantic dictionary will be initialized automatically to contain appropriate references. The expression " a + b " occurring at the very beginning of a program will, therefore, be translated into a sequence of three dictionary indices, standing for " . + . ", " a ", and " b ".

At this point, the designer of the dictionary-management heuristics has *a choice* which of the related entries " . + b ", " a + . ", and " a + b " should be added to the dictionary. Clearly, the strategy that yields the most compact SDE-files, at the expense of table space, is to add all three variants. For the moment, let us assume that all three variants were added. Now consider what happens if, further down in the same program, the expression " a + x " is come across. This can be encoded based on the entry " a + . " already in the semantic dictionary. But then what? Again there is a choice of further entries that could be added to the dictionary. Clearly, adding " a + x " is beneficial,

but what about " . + x "?

The problem lies in the fact that " . + x " may very well be in the semantic dictionary already, for example, resulting from a previous encoding of " y + x ". However, determining whether an entry is in the dictionary requires a search, which is expensive and, therefore, not acceptable at the the time of decoding. Hence, there is a choice of either leaving the entry in question out of the dictionary altogether, or to blindly enter it anyway, possibly for a second time and at a waste of table space.

The question of which strategy is better is still very much open. Different heuristics of template-generation have been experimented with, leading to the observation that the effect varies with the style in which the original source program has been written, depending, among other things, on the number of common subexpressions that have been factored out by the programmer explicitly. A possible solution would of course be to try several variants during the encoding process, finally using the strategy best suited for each particular source program. The corresponding heuristic to be applied during decoding could then be identified by a tag in the SDE-file. To keep it simple, the current implementation uses a straightforward strategy generating relatively few templates.

Factorization of Procedure Calls

When we examine existing Oberon programs, we observe that the parameter lists of procedures are usually arranged in such a way that parameters become "more variable" to the right side of a parameter list. For example, the procedure

WriteBytes(VAR r: Files.Rider; VAR x: ARRAY OF CHAR; n: LONGINT)

is likely to be called several times for any fixed rider *r* with varying further parameters. Moreover, the actual parameter substituted for *n* is still more likely to vary than the one assigned to the formal parameter *x*.

This arrangement is partly due to the extensive use of abstract data types, which naturally appear as the first parameters of the procedures operating on them. However, there also seems to exist a deeper, unwritten, but generally observed rule among programmers to structure parameter lists in the way mentioned. The dictionary-management heuristics used in the encoding of procedure calls has been based on the hypothesis that it is natural to place "less

variable" parameters before "more variable" ones.

Consequently, the implemented system supports the factorization of common parameter lists that partially match from the left side. For example, after encoding the procedure call " P(x, y, z) ", a corresponding complete entry is added to the semantic dictionary, as are the two templates " P(x, ., .) ", and " P(x, y, .) ". A subsequent procedure call " P(x, y, a) " can then be represented space-efficiently by only two dictionary indices, standing for " P(x, y, .) " and " a ".

This strategy represents a trade-off between information density and size of the dictionary, while it is amenable to efficient implementation. An adequate mapping onto a semantic-dictionary representation can be found quite simply by viewing each procedure call as a *binary tree of partial evaluations*, as shown in Figure 3.5.

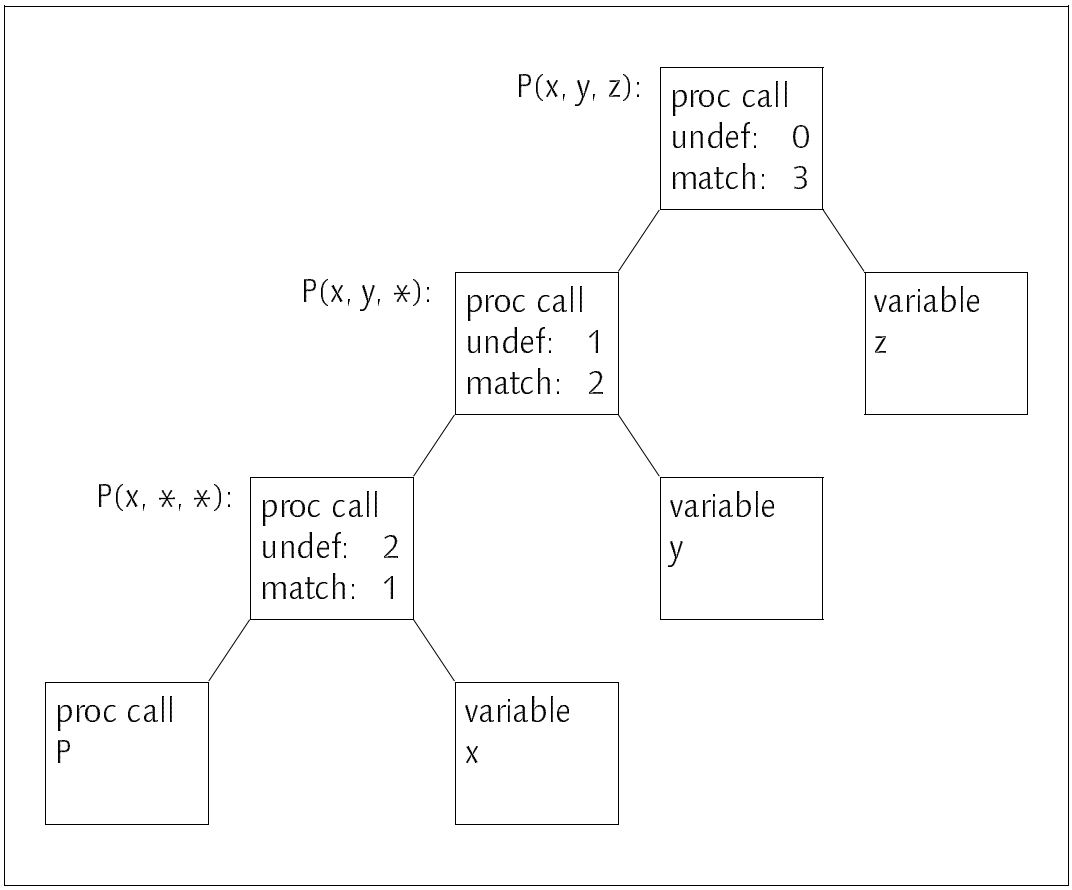

Figure 3.5: Tree View of a Procedure Call.

An alternative strategy would have been to generate even further dictionary entries describing every possible permutation of *wild-cards* in parameter lists. For example, after encoding the procedure call " P(x, y, z) " above, one might have also included the following entries, in addition to the ones mentioned previously: " P(., y, .) ", " P(., ., z) ", " P(., y, z) ", and " P(x, ., z) ".

Preparing the Dictionary for Decoding

In the implemented system, SDE-files are decoded at the time of loading. This is done by an SDE-decoding code-generating loader that is based on the code generator of the MacOberon system [Fra90a, Fra90b, Fra91, BCF92, Fra93a, Fra93b]. It makes use of the idea of *code-copying*, as outlined in the previous chapter, for which a preprocessed syntax tree, as it results from semantic-dictionary encoding and in which source-level common subexpressions and common designators have been factorized, offers an ideal basis. The strategy of initializing the semantic dictionary with entries based on the objects occurring in the symbol table is able to further enhance the efficiency of code generation.

Consider an object of the *variable* category in the symbol table. No further discrimination is usually made in the compiler front-end between different kinds of variables; for example, with respect to whether they are global to a module or local to a procedure. This distinction is left to the code generator, which for each variable reference has to determine what particular kind of variable it is handling, i.e., which addressing mode is to be employed.

Within an SDE-file, however, all references to variables are based on entries that are placed into the dictionary in an initialization step. The important insight is that the *meanings* of the individual entries in the decoder's dictionary need not directly match those of the corresponding entries used in the encoding step, as illustrated in Figure 3.6. Hence, it is possible to simplify the task of code generation by performing a certain amount of **preprocessing** at the time of dictionary initialization, discriminating more finely between different variants of symbol-table objects on the decoder's side than is necessary in the encoder.

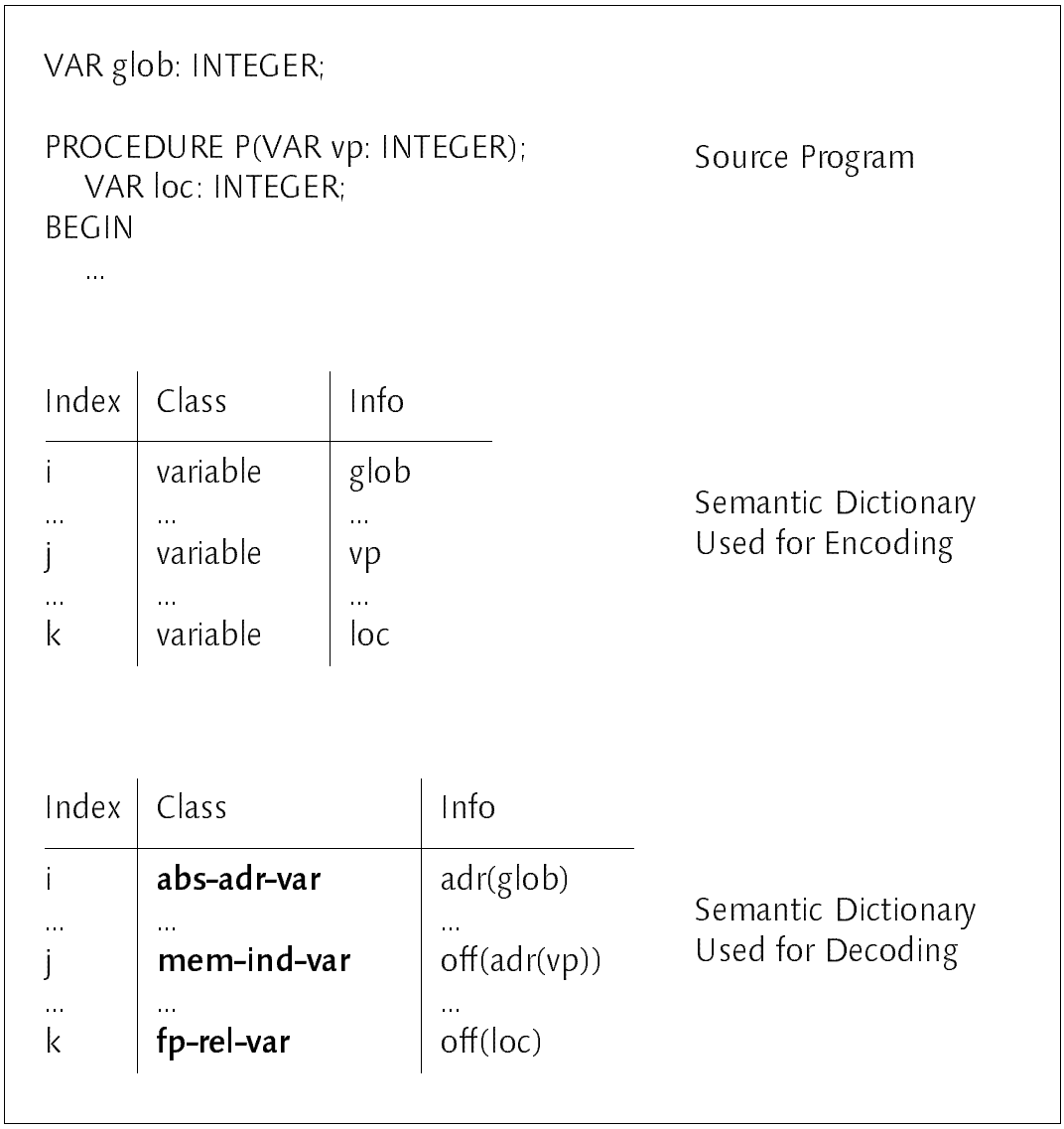

Figure 3.6: Different Semantic Dictionaries for Encoding and Decoding.

The implemented code-generating loader for the MC68020 [Mot87] performs some of this preprocessing. For example, it differentiates between global variables (absolute addressing), direct local variables (frame-pointer relative addressing) and indirect local variables (memory indirect addressing, used for variable parameters). Consequently, three different kinds of dictionary entry are used to represent these different kinds of variables, and all information required for address generation is gathered in the semantic dictionary already when it is initialized. Since in typical programs most variables are referenced more than once, this strategy usually accelerates the code-generation process.

Decoding SDE-Files

After all the previous explanations, the design of the *SDE-decoding code-generating loader* then becomes quite straightforward. First, the global symbol table is read from the SDE-file. Next, the semantic dictionary is initialized as discussed before. Then, dictionary indices are read in succession and processed as is outlined below:

```
< read an index idx and look up the corresponding entry E := dict[idx] >

IF   E.copy-safe   THEN
   < duplicate the code between E.beg and E.end at the current PC location >
ELSE
   IF   < E is a template >   THEN
      < input undefined descendants of E recursively >
      < possibly create further templates heuristically >
      < create a new complete entry E from template and descendants >
   END;

   E.beg := PC;
   < generate code for E, updating the E.copy-safe flag appropriately >
   E.end := PC
END;
```

The **copy-safe** property needs to be kept track of for every complete dictionary entry. It indicates whether code for the corresponding expression can be produced simply by duplicating an instruction sequence generated earlier. Templates are never copy-safe.

A dictionary entry is copy-safe if there are *no pending fix-ups* within the resulting code fragment, and if the code is *position-independent*. In the implemented code-generating loader for the Motorola 68020, these criteria are fulfilled by all full entries with the exception of those describing expressions that contain calls of local functions. These calls are translated into relative branches, which have a different branch distance at every occurrence.

In fact, code-copying was the main reason why it was decided not to translate calls to external procedures into relative branches, which would have been easy since all addresses in external modules are known when generating code on-the-fly. This was the single instance in which speed of execution was traded for speed of code generation. However, the performance of the two call

instructions does not differ by much, while external calls occur often in typical modules so that one can benefit from code-copying frequently.

Miscellaneous Implementation Details

There is a multitude of smaller design decisions in any project such as the one described here. Many of these decisions may be rather ad-hoc, while the motivations behind others may long be lost before they can be documented. The following list is by no means complete, but illustrates some of the finer points that are not mentioned elsewhere in this thesis.

- *Symbol Table.* From what has been said before, the reader might have obtained the impression that the complete symbol table is stored contiguously at the beginning of the SDE-file. This is not entirely correct. In fact, only the global scope is stored at the front of the SDE-file, while the remaining scopes follow the dictionary indices that represent the corresponding procedure entries. This provides for a more efficient storage management in the decoder, allowing a *stack mechanism* to be used not only for the dictionary, but also for the symbol table.

- *Constants.* In the currently implemented system, unnamed literal constants are actually not part of the symbol table, but inserted literally into the stream of dictionary indices, following the index of a "constant" template that implicitly indicates the type of the constant. For each constant represented in this way, a complete entry is constructed and added to the global end of the dictionary, so that further occurrences of the same constant can be referenced directly by an index. The constant *NIL* corresponds to a reserved word of the Oberon language and is placed into the dictionary during its initialization, as are the pre-defined constants *TRUE* and *FALSE*.

- *Storage Allocation Primitives.* In Oberon, calls to the standard procedures *NEW* and *SYSTEM.NEW* are usually translated into dedicated supervisor calls, so that only the appropriate trap vector number need be statically known to the compiler, but not the routine that implements it. In the absence of memory protection on the target machine, it makes no sense

to utilize the expensive supervisor call mechanism when the actual address of the Kernel routine that implements the supervisor instruction is obvious at the time of code generation. Hence, the current implementation maps these two storage-allocation primitives onto ordinary procedure calls, which not only shortens the instruction sequence, but also accelerates the call.

4. Modular Design of the Implementation

The previous chapters have introduced the technique of *semantic-dictionary encoding* and have presented an *Oberon-to-SDE compiler* and a *code-generating loader*. Up to this point, however, it has been disregarded completely that the implemented system has a **modular structure**. This chapter gives an overview of this modular architecture and points out some alternatives to the chosen approach.

Modular Architecture

The implemented system is based on the *Oberon System* [WG89, WG92], specifically its *MacOberon* implementation [Fra90a, Fra90b, Fra93a] for *Apple Macintosh* computers [App85]. The Oberon System supports **separate compilation** with static interface checking of programs written in the programming language Oberon [Wir88]. It also offers **dynamic loading** of individual modules, meaning that separately compiled modules can be linked into an executing computing session at any time, provided that their interfaces are consistent with the interfaces of the modules that have been loaded already. In MacOberon, dynamic loading is accomplished by a **linking loader**, which modifies the code image retrieved from an object file in such a way that all references to other modules are replaced by absolute addresses.

The new implementation adds a *second loader* to MacOberon, namely a **code-generating loader** that processes files containing semantic-dictionary-encoded programs (SDE-files). It has been possible to integrate this code-generating loader transparently into the existing MacOberon environment, in the sense that SDE-files and native object-files are completely interchangeable and can, therefore, import from each other arbitrarily. Depending on a *tag* in the file (first two bytes), either the native linking loader or the code-generating loader is used to set up the module in memory and prepare it for execution.

The code-generating loader has not been incorporated into the core of MacOberon, but constitutes a self-contained module package at the application

level. The existing module loader was modified slightly, introducing a *procedure variable* into which an **alternate load procedure** can be installed. Once initialized, this procedure variable is up-called whenever an abnormal tag is detected in an object file, and so initiates the passage of control, from the linking loader that is part of the system core, to the code-generating loader external to it. The installation of such an alternate load procedure is considered a privileged operation, analogous to the modification of an interrupt vector.

Besides the obvious advantages during the design and testing phases, this architecture permits us to view machine-independence quite naturally as an *enhancement of an existing system*. It also hints at the possibility of adding successive external code-generating loaders as time progresses and standards for machine-independent object-file formats emerge. Several such formats could in fact be supported simultaneously, among which one would distinguish by different file-tags.

Discussion: On System Architecture

In the implemented system, the code-generating loader is installed by an explicit call to *Modules.This*. However, it would be entirely possible to issue such a call during the initialization of module *Modules* itself, leading to a three-stage bootstrap of successively more powerful loading capabilities:

1. Boot Loader
2. Native Linking Loader
3. Code-Generating Loader

The only task of the boot loader is to install the native linking loader and the core operating system routines required by it, such as file-system-access capabilities. The native linking loader in turn sets up the code-generating loader from a pre-compiled object file. All higher modules, even such essential ones as the display system, may then be represented by SDE-files, as long as the algorithms contained in them can be formulated machine-independently.

One immediately senses the possible implications of this. Entirely feasible, and implemented in an experimental system variant, is a mode of operation in which external inputs during the bootstrapping process modify the default behaviour of the system. For example, depressing the key labelled "x" on the keyboard during startup could turn on *array subscript checking* for all library

modules being loaded during this phase, while it would normally be turned off. Likewise, symbolic-debugging information could be generated only when needed, and need not take up memory space in configurations that are used solely for running finished applications.

It is also notable that, after an initial bootstrap, one could have done away with machine-specific object-files altogether by including a code-generating loader in the boot-file and providing a mechanism for generating new boot-files. In fact, with some simple modifications for supporting the generation of *relocation information*, the code-generating loader itself could be employed for the generation of new boot-files. The contents of a boot-file are very similar to those of the module area in the system heap at run-time, except that the former contains also a list of references that need to be relocated.

A similar approach has been taken in the Smalltalk-80 system [Gol84], in which there are only target-machine-independent source-files and an *image file* containing the **snapshot** of a compiled system state. The image file needs to contain an interpreter or compiler, so that the portable source-files can be used at all. After modifying the run-time environment, a new image file can be generated that incorporates all of the changes applied to the previous image.

Unfortunately, however, systems in which all machine-dependent parts are encapsulated within a boot-file have a drawback, which is that a change in any of the modules contained in the boot-file requires the generation of a whole new boot-file. It was exactly this inflexibility that modular architectures were trying to avoid in the first place. For this reason, the idea has not been pursued any further for the time being.

Execution Frequency Hierarchy Considerations

Programs are usually *compiled* far less frequently than they are *executed*. Therefore, it is worthwhile to invest some effort into compilation, because the benefit will be repeated. The same holds for program *loading* versus *execution*. While a program is normally loaded more often than it is compiled, the individual machine instructions are executed even more often, many times on average per load operation. Unfortunately, we need to shift workload unfavourably when we employ a loader that performs code generation. This can be justified only if important advantages are gained in return, or if the resulting code quality is increased. After all, it is execution speed that matters ultimately.

From the outset, three desired properties were therefore put forward that a system incorporating a code-generating loader should possess in comparison to a system employing a traditional compiler and linking loader:

1. run-time performance should be at least as good
2. loading time should not be much worse
3. source-encoding time should be tolerable

These requirements could be met. Benchmarks in the next chapter will show that it was not only possible to construct a fast code-generating loader for *Motorola 68020* processors [Mot87], but that the native code produced by it is of high quality. Indeed, although its method of code-generation is quite straightforward, the code-generating loader is sometimes able to yield object code that would require a much more sophisticated code-generation strategy in a regular compiler. This is mainly due to the fact that the absolute addresses of all imported objects are known at loading time, which enables the code-generating loader to optimize access to them; for example, by using short displacements instead of long ones.

The time required by the code-generating loader for creating native code on-the-fly turned out to be quite acceptable, too. Loading with code-generation takes only slightly longer than loading of a native object-file with linking; the difference is barely noticeable in practice. A partial reason for this is that the Oberon system has a modular structure, in which many functions are shared between different application packages. In traditional systems, these common functions would be replicated in many applications and statically linked to each of these applications.

As a consequence of Oberon's modular design, each new application adds just little code to an already running system. Hence, at most times during normal operation, the code-generating loader need only process the moderate number of modules that are unique to an application, while other modules that are also required will already be in memory from past activations of other applications.

Interchangeability of Object Files

SDE-files and their native-object-file counterparts are completely interchange-

able in the implemented system. This flexibility doesn't come as readily as it may seem. Not only does it require that the code-generating loader is able to interpret *native symbol-files* and the corresponding entry-tables in memory, but also that information can be communicated from the new load architecture to the old one, in formats that are *backward-compatible* with the native compiler and loader.

In the present case, the more difficult part of this problem had an almost trivial solution, since the two varieties of "object" files in the implemented system share a *common representation of symbol-table information*. As documented in a paper by the author [Fra93b], a **portable symbol-file format** had been introduced into MacOberon some time ago, leading to improvements in performance completely unrelated to portability. But now, there were further benefits from the previous investment into machine-independence:

SDE-files use exactly the same symbol-table encoding as do the symbol files of the native compiler. Furthermore, the symbol table is embedded within each SDE-file in such a way that its front-most part would constitute also a valid symbol file (Figure 4.1). Consequently, the native MacOberon compiler cannot distinguish SDE-files from its regular symbol-files. It is able to compile modules that import libraries stored in SDE form, as it can extract the required interface information directly from SDE-files.

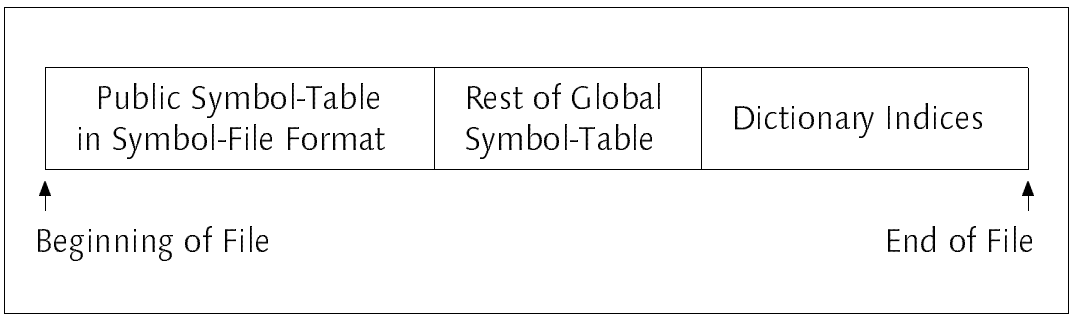

Figure 4.1: Embedding of Symbol File within SDE-File

Apart from the native compiler, the *native loader* need also be able to handle modules that import machine-independent libraries. This is achieved by enabling the code-generating loader to construct on-the-fly not only object code, but also all of the remaining data structures that are usually part of a native object-file, such as *entry tables*. Once that a module has been loaded from an SDE-file, it loses all aspects of machine-independence, and can

henceforth be maintained (enumerated, unloaded, etc.) by the regular MacOberon module manager.

Discussion: On Interfacing Without a Common Symbol File Format

Without a common symbol-table encoding, the task of interconnecting machine-independent modules in the SDE format with others in a regular object-file format would have been more difficult. Indeed, since SDE-files are independent of the target architecture, we might want to use them on more than one kind of machine, on which many different native symbol-file formats could be in concurrent use. In the following discussion, the adjective *native* refers to the formats used on the eventual target machine, while *portable* denotes the machine-independent SDE format.

Enabling *native clients* to import *portable libraries* is quite straightforward, although it may not be so simple to implement. It requires that each local compiler on every target machine can understand the portable symbol-table encoding in addition to its own, and that each code-generating loader can provide entry tables matching this symbol information, in the format required by the corresponding native loader. For some target machines, fulfilling the first requirement may entail a significant amount of programming, because the portable symbol-table encoding contains no machine-dependent information such as type-sizes and offsets. Consequently, local compilers need to be able to calculate these values on-the-fly when reading the symbol table from an SDE-file.

The reverse import relationship, however, is even more problematic. Consider a portable module that imports module *Files*. Implementations of *Files* are inherently non-portable, so that they need to be compiled by a native compiler always. Nevertheless, a portable client module should be linkable to the local *Files* module of an arbitrary target machine, assuming that the different *Files* modules all have the same interface. But how do we *ensure* that all of these interfaces on different machines are indeed compatible with the portable client, and how do we encode references to them in the portable object-file?

A possible solution to this problem, which has actually been implemented in a precursor version of the present system, starts by defining a second, portable interface for each of the native library modules in question. These portable interfaces serve as **place-holders** for a portable subset of the actual interfaces found on different machines. They define capabilities that are

expected to be present on every machine, so that native interfaces are in fact allowed to be a superset of their portable counterparts. Portable symbol-files are then generated from these "place-holder" interfaces, and used whenever portable clients are compiled.

For each target machine, one then constructs a tool called a *localizer*. The localizer connects each portable "place-holder" interface with the corresponding native one, reading both the native and the portable symbol file. If the native version of the module includes every feature specified in the portable interface, the localizer creates a *mapping file* that will tell the code-generating loader which local entry number is associated with which feature in the portable interface. This mapping mechanism is illustrated in Figure 4.2. Each mapping file contains also the keys of both related symbol files, so that the validity of a mapping can be verified by comparing module keys.

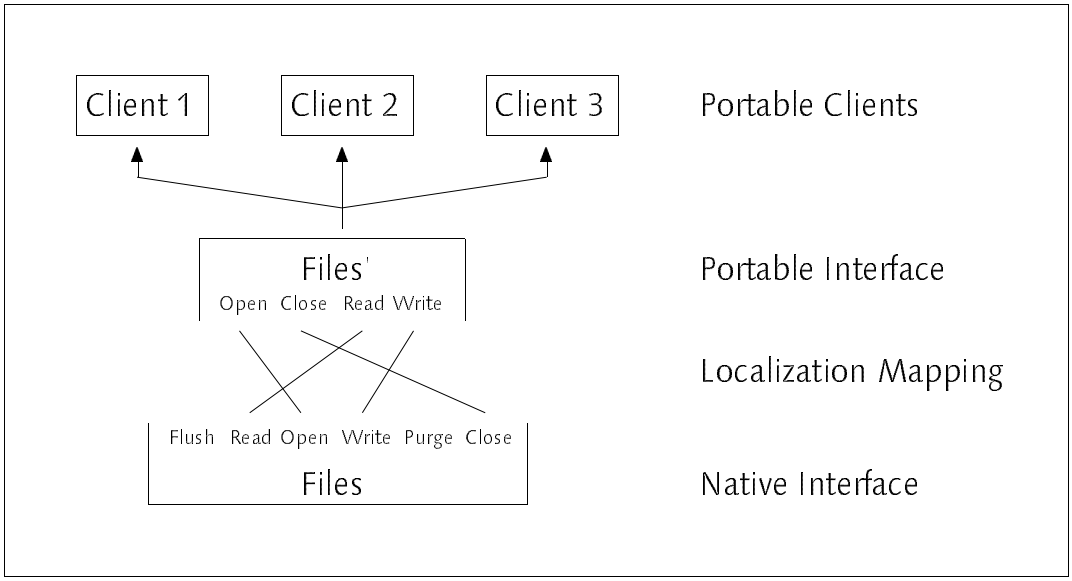

Figure 4.2: Localization of an Interface

Levels of Symbolic Information

As mentioned in [Fra93b], machine-independence of symbol files mandates that the size of each data type (as well as offsets and other machine-dependent data) is calculated only at the time that the symbol file is read, because it may differ from machine to machine. This has consequences when parts of a data structure can be hidden, as is possible in Oberon. Since it is impossible to calculate the total size of the invisible components beforehand, portable

symbol-files need to contain a *full structural description* (the *names* of the hidden parts are obviously not needed) of every exported feature, including all non-exported parts.

This has the undesirable effect that a portable symbol-file (and thereby its key) may change when certain modifications are made in the *hidden part* of the corresponding module, at least as long as a conventional combination of compiler, linker, and loader is used. When employing a code-generating loader instead, this situation changes. Upon closer analysis of a symbol file's information content, we find that ordinary compilers require more information than code-generating loaders:

- Code-generating loaders need only know about *exported names and structure*. They never require any addresses, offsets, or sizes, nor do they need information about invisible parts of an exported object. This is because all inter-module references are handled strictly symbolically and are not resolved until the loading phase, which includes code generation.

- A compiler generating *native code* requires the same symbolic information, plus knowledge about all *invisible fields and structures* to which references exist from exported types. This is because the native compiler has to calculate *offsets*, on which the addressing modes of some instructions may depend; certain offsets may also appear literally in the object code.

It would therefore in principle be possible to use a two-tier structure for storing the symbol table, in which the second layer reveals some finer grained details of the implementation. This mechanism has not been implemented because it adds complexity while applying only to a side-issue of the dissertation project, but the following elaboration on the idea of two-tier symbol-files suggests a possible evolution of the current work. Consider the following structure of a symbol file:

Interface Part
  re-exported modules
  exported constants
  exported types (not mentioning invisible fields)
  exported global variables
  exported procedures

Offset Calculation Part
more imported modules (containing the types of invisible fields)
invisible types of invisible fields
invisible fields (of types mentioned in Interface Part)

With this organization, all client modules that are represented by SDE-files are impervious to changes in the hidden part of a library module, because they depended only on the *interface part* of the library's symbol file. Whenever a module is compiled under such a scheme, both parts of the symbol file are updated, regardless of the type of "object" file generated. However, changes in the *offset calculation part* can affect only those clients that have been compiled into directly executable object code.

In systems supporting two-level symbol-files, it is therefore advantageous to use as few native object-files as possible, in order to reduce intermodule dependencies. Moreover, the penalty for using native object-files is greater at a higher level in the hierarchy than at a low level, because interface invalidations propagate upwards. It has always been good software engineering practice to isolate machine dependencies in low-level modules and strive for portability higher up in the module hierarchy. Having a technical argument might further increase the acceptance of these rules.

The interface part of the symbol file may even be changed to include **per-feature keys** instead of a single key for the whole interface. This would further reduce the interdependency of the portable modules in the system, as it would make possible a **link-by-name** facility. A client module can be connected to a library if all of the requirements of the client can be fulfilled by features of the library. Instead of a crude comparison of whole interfaces, we could therefore compare *exported objects* on one side with *imported objects* on the other. Changes in a module's interface would then only invalidate those portable clients that made direct use of the features that were modified, but left all other portable clients unperturbed. By appending a separate version key to every name, we would be able to use a simple string comparison to determine whether the requirements of one module conform to the features of another, without having to repeat the compiler's structural analysis.

Note that some key changes could be avoided also if the native compiler were allowed to change the ordering of fields in record types (i.e., moving the visible fields to the lowermost offsets). However, the native compiler requires also the *size* of these types. Adding or removing data fields therefore necessarily invalidates clients represented by native object-files.

5. Benchmark Results

This chapter provides some data on the performance of the implemented system and discusses these results. Four different Oberon application-packages have been used in the following benchmarks. *Filler* is a program that draws the Hilbert and Sierpinski varieties of space-filling curves onto the screen. *Hex* is a byte-level file editor of medium sophistication. *Draw* is an extensible object-oriented graphics editor [WG92], of which the three main modules and three extension modules are included in the survey. *Edit* (formerly called *Write*) is a relatively sophisticated extensible document processing system [Szy92], five modules of which are studied here.

Memory and File-Store Requirements

Figure 5.1 gives an impression of the compactness of SDE-files, the influence of the presence of run-time integrity checks on code size, and the memory requirements of the code-generating loader. Its first three columns compare the sizes (in bytes) of native MacOberon object-files with those of SDE-files. Two different values are given for the former, reflecting different levels of run-time integrity checking. SDE-files contain enough information to generate code with full integrity checking, but the user need only decide at loading time whether or not he requires code that includes these run-time checks, and which ones should be included.

The column labelled *"Native –"* in the table below displays the size of native object-files that include reference information for the MacOberon post-mortem debugger, but no extra code for nil-checking, index-checking, type-checking, nor for the initialization of local pointers to NIL. Conversely, the column labelled *"Native +"* gives the sizes of native object-files that include not only reference information, but also code for the aforementioned run-time checks. Neither kind of object file includes symbolic information for interactive debugging, which can be added by enabling a further option.

The fourth column of Figure 5.1 shows the maximum size to which the

semantic dictionary grows during compilation and code-generation. The code-generating loader requires about 80 times this number of bytes as temporary storage while loading a module.

Module	Native –	Native +	SDE	Dictionary
Filler	2900	3236	1232	1003
Hex	10810	12678	4480	1308
Graphics	12692	15935	5281	1382
GraphicFrames	10045	12186	4273	1549
Draw	6033	7133	2193	1498
Rectangles	3255	4297	1378	1271
Curves	5810	7340	2168	1287
Splines	4611	5659	1955	1566
TextFrames	30687	37491	11667	1892
TextPrinter	11503	12537	4494	1486
ParcElems	15259	17540	5273	1437
Edit	11670	13431	5157	1659
EditTools	18756	20898	7397	1488
	144031	170361	56948	

Figure 5.1: Object-File Size (Bytes) and Dictionary Size (Entries)

This data shows that on average, SDE-files are about 2.5 times more compact than native MacOberon object-files not containing run-time integrity checks, and about 3 times more compact than native files incorporating these checks. It is also noteworthy that the maximum size, to which the semantic dictionary grows during encoding and decoding, is not proportional to the overall size of a module, but only roughly proportional to the length of the longest procedure in a module. This seems to level out at a relatively small value in typical modules, much smaller than anticipated originally.

Figure 5.2 indicates how much information is output into memory by the code-generating loader. The first column repeats the sizes of SDE-files from the previous table. The second and third columns give two different values for the size of the object code generated on-the-fly, depending on whether or not run-time integrity checks (in the same combinations as before) are emitted. The remaining columns list the sizes of dynamically-generated constant data

(including type descriptors), reference data for the Oberon post-mortem debugger, and link data. The latter comprises entry tables generated on-the-fly, so that native client modules can later be connected to dynamically generated library modules.

Module	File	Code –	Code +	Const	Ref	Link
Filler	1232	2352	2682	139	244	8
Hex	4480	9224	11020	267	1094	12
Graphics	5281	9036	12140	799	1452	204
GraphicFrames	4273	8268	10354	433	824	76
Draw	2193	4754	5752	236	451	60
Rectangles	1378	2644	3596	104	292	20
Curves	2168	5042	6462	100	439	24
Splines	1955	3812	4806	137	437	20
TextFrames	11667	25878	32536	629	2843	156
TextPrinter	4494	9176	10180	478	1367	40
ParcElems	5273	12944	15180	493	1085	52
Edit	5157	9368	10940	555	1098	72
EditTools	7397	14844	16886	723	2009	116
	56948	117342	142534	5093	13635	860

Figure 5.2: Sizes of SDE-Files and of Dynamically-Generated Data (Bytes)

It is notable that SDE-files encode programs more than twice as densely as object code for the MC68020 architecture [Mot87]. If code that includes run-time checking is considered instead, the factor becomes even 2.5. This is in spite of the fact that SDE-files can additionally also serve as symbol files and contain reference information as well.

Performance

Unless stated otherwise, the following benchmarks were all carried out under *MacOberon Version 4.03* on a *Macintosh Quadra 840AV* computer (*MC68040* Processor running at 40MHz, equipped with a graphics accelerator card), using version *7.1.1* of the Macintosh System Software. All results are given in real

time, i.e. the actual delay that a user experiences sitting in front of a computer, with a resolution of 1/60th of a second. Each benchmark was executed after a cold start, so that no files were left in the operating system's file cache between a compilation and subsequent loading, and the best of three measurements is given in each case. Every attempt has been made to present the system as a regular user would experience it in everyday use.

Figure 5.3 presents the times (in milliseconds) required for compilation and for module loading. The first two columns compare the native compilation times of the MacOberon compiler with the times that the Oberon-to-SDE compiler requires for generating an SDE-file out of an Oberon source program. The remaining two columns report the loading times for both varieties of object file, including disk access, and, in the case of SDE-files, also including on-the-fly generation of native code. On average, 10% of the object code emitted by the code-generating loader can be generated by code-copying. All timings apply to the situation in which no run-time integrity checking is used.

Module	Compile	Encode	Ld Native	Ld SDE
Filler	583	633	83	100
Hex	666	900	100	116
Graphics	766	1083	116	166
GraphicFrames	650	950	66	116
Draw	516	683	66	100
Rectangles	400	500	50	83
Curves	450	650	50	116
Splines	433	583	66	83
TextFrames	1416	2183	216	316
TextPrinter	666	950	100	133
ParcElems	750	1150	116	200
Edit	716	1166	100	166
EditTools	900	1483	150	233
	8912	12914	1279	1928

Figure 5.3: Compilation versus SDE-Encoding and Module Loading Times (ms)

As can be seen from these timings, Oberon-to-SDE encoding on average takes about 1.5 times as long as normal compilation. This factor could certainly be further reduced by using