SCMTOCPP: a configurable, intelligent back end for SUCHTHAT

Roland Weiss

WSI 97-13

13. Januar 1998

Wilhelm-Schickard-Institut für Informatik
Arbeitsbereich Computeralgebra
Prof. Dr. Rüdiger Loos
Sand 13
D-72076 Tübingen

e-mail: weissr@informatik.uni-tuebingen.de

ISSN 0946-3852

Abstract

Sibylle Schupp introduced a new programming language for creating generic, algebraic libraries called SuchThat in her PhD thesis ([27]). Together with Prof. Dr. Rüdiger Loos she wrote a first compiler that was heavily based on the existing Aldes back end a2c. But the new features of SuchThat forced them to drop this project and restart from scratch.

The new approach uses Flex^{++} and Bison^{++} for providing an adequate front end. In the first step called st2scm, an intermediate representation in Scheme is produced. No significant changes are made to the program logic, it converts SuchThat straightforward to *imperative* Scheme.

The algebraic resolution, which is *the* primary feature of SuchThat, operates on this imperative Scheme representation. scm2scm generates the final Scheme code.

Here enters ScmToCpp the scene. It takes the final Scheme output as its input and converts it to C^{++} code. This paper describes in detail how the necessary transformations are done, what special C^{++} support is needed, and defines the interface between Scheme and ScmToCpp.

Great emphasis is put on a *configurable* back end such that one can easily extend its functionality. This can be done by adding new datatypes and new expressions to the predefined set. We might even think of another programming language like Ada95 or Java as the back end target language.

ScmToCpp should also be *intelligent* in a certain way. It simply means that the back end takes some low-level decisions out of the SuchThat programmer's hand: What kind of parameter passing is used in procedures? What representations are used for numeric datatypes? And so forth.

Why C^{++}? It gains more and more influence as a back end language in algebraic systems. Our compiler ScmToCpp generates efficient and portable C^{++} code. Sebastian Wedeniwski implemented basic algorithms for a multi precision arithmetic in C^{++} for his work Piologie. The algebra group *LidIA* exploits the C^{++} class hierarchy to give their algebra system maximum flexibility. But in the first place, it was Sibylle Schupp's decision to use C^{++} as back end language:

> For three reasons, the target language of SuchThat is C^{++}. First, using C^{++} as a kind of universal assembler makes our programs portable. Secondly, using features of C^{++} like overloading, operators, and templates makes our compiler easier to write. Moreover, as we translate to C^{++} we can apply the Standard Template Library (STL) ([27], p. 52).

But we will also have a look at other languages to abstract from the specific features of C^{++}. The link between SuchThat and C^{++} shouldn't be to close such that a transition to newer and more powerful target languages would be overly hindered.

Contents

Chapter 1

The Legacy of SuchThat

The topic of this chapter is the Scheme interface between the front end and the back end compiler tools of SuchThat. Our goal is the specification of this interface in order to allow smooth communication among the compiler parts and their independent development. We will have to tackle all problems that arise from the decision to use Scheme as intermediate language. These encompass the typical components of a programming language:

- identifiers
- datatypes
- expressions
- statements
- declarations and definitions

Declarations and definitions, the last items in the list, are heavily influenced by the idiosyncrasies of SuchThat. Of course all of them are biased by the legacy of SuchThat, but the declaration concept builds the basis of SuchThat. It's the ascendant of SuchThat's type checking mechanism and the major feature that distinguishes SuchThat from current, non-generic languages.

The syntax of the Scheme interface is presented in EBNF and can be examined as one continuous text in `doc/scm-interface`.

`"doc/scm-interface.txt"` 1 ≡
⟨Scheme interface 3, ... ⟩
◇

In the following discussion we will focus on C^{++}, Java and Scheme. The relevance of C^{++} and Scheme is obvious, C^{++} is the back end target language and Scheme the interface language. We added Java in the comparison in order to see other concepts and because it booms in the field of computer science.

Nuweb This document is an example of literate programming. This means that documentation and source code stay in the same file. We use the tool Nuweb. It uses LaTeX as text formatter. The documentation is created just like a regular LaTeX document. The source code is inserted in *scraps*. They either define files or macros. The scrap a few lines above defines the file `doc/scm-interface.txt`. The file is a collection of macros. The numbers in file and macro references are page numbers in the document. If several scraps appear on one page, they get an additional letter to distinguish them.

1.1 Identifiers

Identifiers are a fundamental lexical element of every programming language. They are used to identify your own functions and variables. Languages vary profoundly in their supported identifier syntax. SuchThat currently has no fixed identifier syntax, therefore we will substantiate the `identifier` token in its grammar ([27], p. 76).

1.1.1 Identifiers in C++

C++ identifiers ([31], p. 478) are very simple. They consist only of letters and digits. The first character must not be a digit. The underscore '_' counts as letter. Letters are case sensitive. Table 1.1 lists all characters in C++ identifiers.

letters	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o p q r s t u v w x y z _
digits	0 1 2 3 4 5 6 7 8 9

Table 1.1: legal characters in C++ identifiers

1.1.2 Identifiers in Java

Java sticks to the rules defined by the C++ standard with one significant extension. All characters in Java are composed of 16-bit Unicode characters. Identifiers are no exception ([11], p. 17). This means that we can use identifiers from our native language in Java programs.

The JDK 1.1 API ([25]) provides the datatype wrapper class `Character` in the language package. This class implements the methods `isJavaLetter` and `isJavaLetterOrDigit` to explicitly test for valid identifier characters.

The second draft of the oncoming C++ standard also extends C++ identifiers for international usage ([14], section 2.10 and annex E).

1.1.3 Identifiers in Scheme

Scheme allows identifiers to be a lot more diversified. Many special characters are added to the syntax, table 1.2 gives an overview. The last three special characters listed in the table must not start an identifier. See [4], p. 32 for a complete specification.

letters	a b c d e f g h i j k l m n o p q r s t u v w x y z
digits	0 1 2 3 4 5 6 7 8 9
special characters	! $ % & * / : < = > ? ~ _ ^ . + -

Table 1.2: legal characters in Scheme identifiers

The special characters are reflected in Scheme's standard naming conventions. For example, the exclamation mark '`!`' indicates a mutation procedure and functions returning a boolean, so-called predicates, end in '`?`'[1].

But Scheme identifiers also have a major drawback. They are case insensitive. `FooName` as well as `fOOnAME` refer to the same identifier.

1.1.4 Identifiers in SuchThat

Identifiers in SuchThat are extremely flexible. This originates from the fact that LaTeX is part of SuchThat's input language ([27], p. 38). Some examples of legal SuchThat identifiers are $\mathbb{X}_1$, $A^\star$, n', $a_{1,2}$ and $\overline{g}$. This very small set of examples illustrates the power SuchThat possesses to

[1] examples: `set-car!, vector-set!, list?, null?`

specify identifiers. We have to examine the underlying LaTeX mechanisms to reduce this (almost) unlimited variety to a manageable amount for our back end.

Command names in LaTeX ([15], p. 9) start with the escape character '\' and are composed only of letters. Curly braces '{ }' enclose mandatory arguments, brackets '[ ]' enclose optional ones. The math mode is entered and left with the dollar sign '$' and makes use of the characters '_' and '^' to ornament symbols and words in subscript, respectively superscript.

We allow all these characters to be present in identifiers passed to the back end, except for the LaTeX braces and brackets: '{ } [ ]'. They are reserved in Scheme ([4], p. 6) for future extensions and can't be used in symbols. We also add all characters used in Scheme. The front end has to map characters prohibited in the Scheme interface to a replacement that complies to the specification. Perhaps the most prominent examples are the apostrophes '`' and '''. They are not part of the interface because they quote expressions in Scheme.

We still have one big problem remaining: Scheme is case insensitive. Both our target language C++ and SUCHTHAT itself are case sensitive. We have to find a way to pass the case information to the back end such that it may reconstruct the original identifier. The solution is simple. The front end lets every capital letter be followed with '#'. This sign has a special meaning in LaTeX as well as in Scheme and therefore can't be used in identifiers normally. The back end will eliminate the '#' sign during code generation. With case sensitive languages in mind the capital letter is reproduced, and that's what we do in SCMTOCPP. If we think of a Scheme back end, some trivial mapping to legal syntax has to take place. Table 1.3 summarizes all decisions.

letters	a b c d e f g h i j k l m n o p q r s t u v w x y z
digits	0 1 2 3 4 5 6 7 8 9
LaTeX characters	\ $ _ ^
Scheme characters	! % & * / : < = > ? ~ . + -
case character	#

Table 1.3: legal characters in the Scheme identifier interface

The comparison in this paragraph has shown that not all characters permitted in our interface are available in C++. Thus a trivial mapping is also necessary in the C++ back end. The mapping is fixed in a table which is read in by SCMTOCPP. The table has an entry for every character which shall be subject to the mapping. An entry consists of the character to be mapped and two alternatives. The alternatives give a short and a long form of the string to which the character will be mapped. Table 1.4 shows a possible configuration. See chapter 6 for a detailed description on how to change the default behavior.

character	short form	long form
$	_DS_	_DollarSign_
\	_BS_	_BackSlash_
^	_H_	_Hat_
%	_P_	_Percent_
~	_T_	_Tilde_
<	_L_	_Less_
>	_G_	_Greater_

Table 1.4: example identifier mapping table

1.1.4.1 Syntax of Scheme Interface for Identifiers

Table 1.3 is captured in the following definitions:

⟨Scheme interface 3⟩ ≡

⟨Scheme interface: identifier alphabet 4a⟩
⟨Scheme interface: identifier syntax 4b⟩
◇
Macro defined by scraps 3, 5, 9, 20, 22a.
Macro referenced in scrap 1.

⟨Scheme interface: identifier alphabet 4a⟩ ≡

```
letter
    : "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j"
    | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t"
    | "u" | "v" | "w" | "x" | "y" | "z"
;
digit
    : "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
;
latex-char
    : "\" | "$" | "_" | "^"
;
scm-char
    : "!" | "%" | "&" | "*" | "/" | ":" | "<" | "=" | ">" | "?"
    | "~" | "." | "+" | "-"
;
case-char
    : "#"
;
```

◇
Macro referenced in scrap 3.

To ease the transition to our target language C++ we prohibit the first character of an identifier to be a digit.

⟨Scheme interface: identifier syntax 4b⟩ ≡

```
identifier
    : { letter | { letter case-char } | latex-char | scm-char }
      { letter | { letter case-char } | latex-char | scm-char | digit }*
;
```

◇
Macro referenced in scrap 3.

1.1.4.2 Examples

We will follow the examples, given at the beginning of the paragraph, through all stages of compilation. These assumptions are made in table 1.5:

1. the front end doesn't simplify the LaTeX expressions
2. the apostrophe '`'`' is mapped to '`~`' by the front end
3. the comma '`,`' is mapped to '`%`' by the front end
4. the curly braces '`{ }`' are mapped to the relation signs '`< >`' by the front end
5. the short forms from table 1.4 are used by ScmToCpp

The left column shows how the identifier looks like to the SuchThat user and the LaTeX column what the SuchThat programmer enters in his editor. In the Scheme interface some transformations have taken place. To be more precise, capital letters and illegal characters are replaced. The last column contains identifiers generated by ScmToCpp.

The examples show quite clearly that the resulting C++ identifiers become almost unreadable. To avoid this effect the front end can simplify the LaTeX representations of the SuchThat identifiers. The redundant dollar signs and braces may be stripped, often used LaTeX commands may be abbreviated and so on. An easy way to simulate this behavior is to set the short forms of the dollar sign and the backslash to empty strings. Then $\mathbb{X}_1$ would become `mathbb_L_X_G__1`, which looks a little bit nicer.

SuchThat	LaTeX	Scheme Interface	C++
$\mathbb{X}_1$	`$\mathbb{X}_1$`	`$\mathbb<x#>_1$`	`_DS__BS_mathbb_L_X_G__1_DS_`
$A^\star$	`$A^\star$`	`$a#^\star$`	`_DS_A_H__BS_star_DS_`
n'	`$n'$`	`$n~$`	`_DS_n_T__DS_`
$a_{1,2}$	`$a_{1,2}$`	`$a_<1%2>$`	`_DS_a__L_1_P_2_G__DS_`
$\overline{g}$	`$\overline{g}$`	`$\overline<g>$`	`_DS__BS_overline_L_g_G__DS_`

Table 1.5: transformation of SuchThat identifiers

1.2 Declarations

The major feature of SuchThat is its declaration part, or to say it in Sibylle Schupp's words:

> Declarations are the heart of SuchThat ... ([27], p. 47)

⟨Scheme interface 5⟩ ≡
```
declaration
    : variable-declaration
    | map-declaration
;
```
⟨Scheme interface: variable declarations 6⟩
⟨Scheme interface: map declarations 7⟩
◇

Macro defined by scraps 3, 5, 9, 20, 22a.
Macro referenced in scrap 1.

1.2.1 Variable and Structure Declarations

They declare the type of parameters and variables, and name new mathematical structures. Let's look at some examples of SuchThat declarations:

```
Global:     Let F be a field; let I be an integral domain.
Algorithm:  ...
Input:      i, j ∈ Integer; r ∈ Real.
Output:     o ∈ F.
```

We see two important things in this trivial example. Declarations often take the form of mathematical structures. This is essential for generic programming, where we want to state an algorithm with its minimal preconditions. But our back end ScmToCpp shouldn't care about them because the front end handles structure declarations with the algebraic resolution. These considerations lead to the conclusion that we only have to deal with variable and parameter declarations.

Next we realize that declarations appear inside a scope. SuchThat differentiates between two scope levels: algorithm and file scope. At file scope only global declarations are permitted, whereas at algorithm scope four kinds of declarations are legal: internal[2], local, input and output declarations. We omit internals from the interface because they are used solely for structural declarations.

1.2.1.1 Syntax of Scheme Interface for Variable Declarations

Scheme doesn't support type declarations as it's a dynamically typed language. But it would be extremely foolish to discard the type informations computed by the front end. So we have to invent a mechanism to pass type information in the Scheme interface. For this purpose we add four new keywords to our interface: `st-global`, `st-local`, `st-input` and `st-output`. Now we can "declare" typed variables. The untyped Scheme gap between SuchThat and C++ [3] is successfully bridged.

[2] they are not mentioned in [27] and were mainly invented to introduce structure declarations that are local to an algorithm and therefore shouldn't litter the global scope

[3] or any other target language

Parameter and variable declarations in the Scheme interface expect a list of pairs. The `car` field holds the identifier and the `cdr` field the type.

⟨Scheme interface: variable declarations 6⟩ ≡

```
variable-declaration
    : "(" st-scope { "(" identifier "." type ")" }+ ")"
;
st-scope
    : "st-global"
    | "st-local"
    | "st-input"
    | "st-output"
;
◇
```

Macro referenced in scrap 5.

Example The simple declarations a few lines above will produce this input for our back end:

```
(st-input (i . i#nteger) (j . i#nteger) (r . r#eal))
(st-output (o . f#))
```

1.2.2 Realization Declarations

The application of realizations is changing defaults ([27], p. 49). This means that a `map` declaration can change a default type or influence the semantics of a function call. For example, if we have implemented a fast procedure `AddInt(Integer i, int j)` that adds a low-level `int` to a multi precision `Integer`, this declaration makes sense:

```
Map operator+ to AddInt;
```

Of course the parameters passed to `AddInt` must be correctly typed by the front end. Here the second realization type fits well. If `Integer` variables are mapped to C^{++} `int`s by default, we might need a type map of this kind:

```
Map Integer to int;
```

But beware! Using the `map` declaration for types is like opening Pandora's box. With this nice feature we jeopardize the whole type checking mechanism of the front end. Appropriate measures have to be taken if we don't want to loose the type-safety of SuchThat.

Renaming declarations in Ada95 ([32], p. 148) are a language feature that goes further than simple preprocessing. The object and subprogram renaming declarations correlate with the mappings just described. What makes Ada95's subprogram renaming declarations noteworthy is that it needs a complete specification when renaming an identifier. The fast addition "mapping" in Ada95 syntax:

```
procedure addInt(i : out LongInteger; j : in Integer) renames "+";
```

1.2.2.1 Syntax of Scheme Interface for Realization Declarations

We support only two forms of realizations. Function mappings exchange function names, which of course can also be operators. What happens is that the identifier denoting the original function name is replaced by the identifier given as the mapping target.

Type mappings do the same for types. The mapping target is not simply an identifier but a type. This gives us the power to map type names to nested types. A simple example shows how to map a `Matrix`, which needs a two-dimensional container as type parameter.

```
;;; type mappings
(st-map (type m#atrix dense_matrix))
(st-map (type mtype (v#ector (v#ector i#nteger))))
;;; declare matrix
(st-global (m# . (m#atrix mtype)))
```

Assuming the standard mappings otherwise, the C++ code for the global declaration is:

```
dense_matrix< vector < vector < int > > > M;
```

⟨Scheme interface: map declarations 7⟩ ≡

```
map-declaration
    : function-map | type-map
;
function-map
    : "(st-map (function" identifier identifier "))"
;
type-map
    : "(st-map (type" identifier type "))"
;
◇
```

Macro referenced in scrap 5.

It might be desirable to have a more fine grained mapping mechanism, but for the time being we have to live with the current, very coarse one.

1.3 Datatypes

The concept of datatypes is fundamental to every programming language. There are many diverse views on datatypes in the computer community and they influence the way a language is employed. Many interesting questions arise: How are types declared and defined? What values are associated with each type? Which operations are possible on different datatypes and how to prevent illegal access of operations to wrong datatypes? At which point should the type check be executed? How are new types constructed?

These questions show that a great deal of work can be put into the elaboration of a sound type system. Brian Meek presented a taxonomy of datatypes ([21]), inside which he distilled the ideas behind datatypes, abstracted from a special language. We will compare his conceptual types with datatypes in real languages. This comparison will lead to the framework of our type system in the Scheme interface.

1.3.1 Datatype Comparison

The comparison is separated in two categories, those for primitive (or fundamental) and aggregate types[4]. The fundamental types are included to provide SuchThat with the ability to express *everyday problems* that don't use mathematical structures. When we speak of aggregate types we really have containers in mind, and that's what Sibylle Schupp intended:

> Internally, all `structure_expressions` are mapped on a C++ data type, mostly a container ([27], p. 49).

1.3.1.1 Fundamental Types

The values of fundamental types can't be reduced any further. This is at least the view of the SuchThat programmer. The target language can take efforts to implement the type efficiently or in its wholeness. An example is the type `Integer`, which can either be a very fast `int` or a multi precision implementation like Sebastian Wedeniwski's Piologie ([34]). We included Aldes ([17]) with the SAC-2 algorithm package ([18]) in the comparison to have there a mathematical concept that comes close to SuchThat. Java is examined together with the JGL ([23]), C++ with the foundation library Tools.h++ ([26]).

It can be argued that strings are usually implemented as a sequence container of characters, but we are more concerned about the utilization of the types than their low-level implementation.

[4]the terminology is taken right from Meek's paper

Meek's taxonomy	C++	Java	Aldes, SAC-2	Scheme
Boolean	bool	boolean	—	Boolean
State	—	—	—	—
Character	char	char, Character	character	Character
—	string	String	—	String
Enumerated	enum	—	—	—
Ordinal	unsigned int	—	—	—
Integer	short, int, long	int, Integer	integer	Integer
Rational	—	—	rational	Rational
Real	float, double	float, Float	real	Real
Complex	complex	—	complex	Complex
Scaled	—	—	—	—
Void	void	void	—	Void
Date-and-Time	RWDate†	Calendar, Date	—	—

Table 1.6: fundamental datatypes comparison (†: Tools.h++)

The same is valid for complex and rational types. Our back end must be prepared to expand this primitive types to more complex data structures in the target language.

C++ and Java can easily be extended to support more datatypes by realizing them as classes. We should acknowledge this fact and enable the back end to be customizable to deal with new, user provided datatypes.

1.3.1.2 Containers (Aggregate Types)

Fundamental types can be grouped and assembled in various ways to create new structures with a meaning implied by the creator. The following overview can only make a rough mapping of the datatype concepts, because they often overlap and have slightly different semantics.

Meek's taxonomy	C++	Java	Aldes, SAC-2	Scheme
Array	array	array	array	vector, array°
Vector	vector	Vector	—	—
Sequence	list, deque, RWslist†, RWdlist†	Deque*, DList*, SList*	list	list
keyed Sequence	map	Dictionary	—	alist, collection°
Record	struct, class	class	—	record°
Set	hash_set	HashSet*	set	—
Bag	hash_multiset, RWBag†	HashSet*	—	—
Table	adapted container	—	matrix	—
—	—	—	polynomial	—

Table 1.7: aggregate datatypes comparison (*: JGL, °: SLIB, †: Tools.h++)

1.3.1.3 SuchThat Default Types

The preceding comparison offered a lot of types for integration into SuchThat. We eliminated those that are of no apparent need for SuchThat. The default types should be supported by every back end implementation.

The types Matrix and Polynomial can be mapped to simple containers, but their importance in algebraic systems make it desirable to have them implemented on their on.

fundamental types	containers
Boolean	Array
Character	Vector
String	List
Integer	Map
Rational	[Matrix]
Real	[Polynomial]
Complex	

Table 1.8: SuchThat default types

The fact that the aggregate types are cut down heavily may be a little surprising. But SuchThat wants to abstract from the concrete realization of the algorithm as well as the datatype. The complexity of the data structures should be hidden from the SuchThat level by our back end.

We will now examine the proposed default types (see table 1.8) accross the languages and define the Scheme interface. Scheme has a runtime type checker and no variable declarations are necessary. This contradicts our approach, which does the type check at compile time. We introduced special declaration keywords to overcome this deficit, the technique is described in section 1.2.

⟨Scheme interface 9⟩ ≡

```
type
    : flat-type
    | array-type
    | container-type
;
⟨Scheme interface: flat type 10⟩
⟨Scheme interface: array type 14⟩
⟨Scheme interface: container-type 19⟩
◇
```

Macro defined by scraps 3, 5, 9, 20, 22a.
Macro referenced in scrap 1.

1.3.2 Fundamental Types

All the fundamental types are part of Scheme. We don't have to deal with special cases. The front end has to convert SuchThat's external representation of the fundamental types to the Scheme representation. Our back end will transfer the literals in the code generation phase to target language syntax.

1.3.3 Generic Types

Right until now we've only spoken about concrete types. But SuchThat is a language to express generic algorithms. Therefore we have to ponder what generic types are to our back end. An algorithm is stated with its minimal preconditions in SuchThat. This means on the one hand that all the algorithm's parameters are declared to be inside a certain domain. The domain should be as simple[5] as possible, for example a ring is preferred to a field. The parameter domain is, from our point of view, a generic type. The SuchThat programmer concretizes this generic type whenever he writes a main program and uses an instantiation of the generic algorithm.

This leads to the perception that fundamental and generic types need no additional information except their name. We call both fundamental and generic types *flat types*. The simple syntax is described below.

[5] simple means that the algebraic structure should presume a minimal set of elements and operations

1.3.3.1 Syntax of Scheme Interface for Flat Types

A fundamental as well as a generic type is represented by an identifier naming it. The type is embedded in type declarations (see section 1.2, where some examples for simple type declarations are to be found).

⟨Scheme interface: flat type 10⟩ ≡

```
flat-type
    : identifier
;
```

◇

Macro referenced in scrap 9.

1.3.4 Containers

Containers differ from fundamental and generic types insofar, as they need extra parameters, the *type parameters*. They tell the container, or to be more precise, the compiler, which datatypes are stored inside the container. In pursuit of the syntax of containers we will exemplary look at random access sequences. The observations made there will guide us to a general view of the container interface.

Ok, let's have a look at sequences that allow random access to their items. There are two very closely related types in most programming languages: arrays and vectors. We will compare their characteristics and usage in C++, Java and Scheme. This comparison should lead us to a definition of their integration in SuchThat. Our interest will focus on the following topics:

- declaration and creation
- memory usage and layout
- definition and item access
- security (bounds checking)
- efficiency

1.3.4.1 Arrays

In many languages arrays are a kind of low-level container. They are usually extremely efficient and immanent to the language definition[6]. The efficiency of arrays is achieved by fixing their size at declaration time. Therefore the compiler can address every item by an easily calculated offset to the array's memory location.

The disadvantage of this approach is evident. There is no way to increase the size of an array if all indices are being used and you want to add another item. In other words, we have hard coded limitations in our program.

1.3.4.2 Arrays in C++

Declarations ([31], p. 531) have the form `T D[n]`, where `T` is the item type, `D` is the array's name and `n` is a constant integer expression greater than 0. `n` can be omitted if the array is used as a function argument, an initializer-list is provided or an external definition is given elsewhere. The items are indexed from 0 to `n`-1.

Multidimensional arrays are declared by simply placing several subscript operators next to one another: `T D`$[\mathtt{n}_1]\ldots[\mathtt{n}_d]$, where d is the array's dimension. Only $\mathtt{n}_1$ may be omitted. A row-wise ordering is applied.

The memory image of a C++ array is a continuous sequence of $\mathtt{n}_1 \cdot \ldots \cdot \mathtt{n}_d$ items of type T and occupies $(\mathtt{n}_1 \cdot \ldots \cdot \mathtt{n}_d)$·`sizeof(T)` bytes of memory. The total memory usage of an array can be determined with the `sizeof` operator. The memory image of array `int A[3][5]` is illustrated in

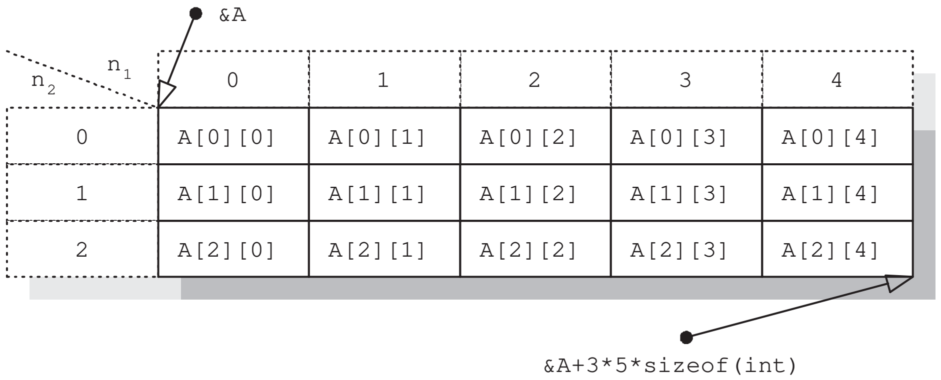

Figure 1.1: memory image of array `int A[3][5]`

figure 1.1. The figure implies some spatial alignment, but `A` is just a linear memory space of 15 `int`s.

The common way to access an item inside an array is to give its indices in a subscript operator. But we can also add an offset to the array's address to reach a specific item. So to get the item at position (2, 4) we can either write `i = A[2][4]` or `i = *(*(A + 2) + 4)`.

No bounds checking of any kind is performed on standard C++ arrays. Access operations on illegal array positions are one of the major error sources in programming.

All these theoretical issues are summed up in a little C++ program:

"src/cpp/ArrayExa.cpp" 11 ≡
```
#include <iostream>
using namespace std;
void main() {
  const int m = 3, n = 5;
  // ----- standard C++ array -----
  // declare
  int A[m][n];
  int B[][m] = { {1, 2, 4}, {8, 16, 32} };
  // access
  A[m-1][n-1] = 4711;
  cout << "A[m-1][n-1]: " << A[m-1][n-1] << endl;
  cout << "*(*(A + (m-1)) + (n-1)): " << *(*(A + (m-1)) + (n-1)) << endl;
  cout << "B[1][2]: " << B[1][2] << endl;
  // space requirements
  cout << "sizeof(A): " << sizeof(A) << endl;
  cout << "sizeof(B): " << sizeof(B) << endl;
}
```
◇

It produces the following output:

```
A[m-1][n-1]: 4711
*(*(A + (m-1)) + (n-1)): 4711
B[1][2]: 32
sizeof(A): 60
sizeof(B): 24
```

1.3.4.3 Arrays in Java

Due to the nature of Java there are some fundamental differences between Java arrays ([7], p. 29) and C++ arrays. First of all, in Java all runtime variables are created dynamically and handled by

[6] Scheme is an exception, arrays are only available when SLIB[5] is installed

the garbage collector. Hence declaration and creation mostly go hand in hand: `T D[] = new T[n]`. As in C++, `n` may be omitted if an initializer-list is given. The simplified Java type declaration syntax allows another form: `T[] D`.

Arrays of multiple dimensions are declared similar to C++: `T D[]...[] = new T[`n_1`]...[`n_d`]`. Not all n_i have to be specified. You can choose a j, such that $1 \leq j \leq d$. Then you only have to specify all n_i with $1 \leq i \leq j$.

As mentioned before all Java user objects are created at runtime. This gives arrays the flexibility to have an irregular shape. We can think of triangular matrices as an example application of this characteristic. Unlike in C++, an array in Java is identified by its stored items *and* its length. Every array has a read-only `length` field. But it's not part of the type as in C++, it's a runtime information. We can traverse an array by writing:

```
int A[];
/* create array A */
for (int i = 0; i < A.length; i++)
  System.out.println("A[i] = " + A[i]);
```

The code shows that items are accessed just as in C++ using the subscript operator. The indexing convention is also the same. But beware: unlike C++, `[]` isn't an operator in Java.

Java doesn't support the `sizeof` operator. That is one consequence of removing pointer arithmetic from the language. So we can't say how Java handles arrays and their memory layout. To access items pointer-like, we have to linearize the array ourselves. A 3x5 `int` array will be created with `int[] A = new int[3*5]`. We then have the power to apply our own ordering to the linear representation of the two-dimensional array. To access item (2, 4) in row-wise ordering, `i = A[2*5 + 4]` is sufficient. The linear treatment of multidimensional arrays yields performance enhancements if functions operate on all items in a specific order, for example functions to copy or to compare matrices. Instead of using d `for`-loops, where d denotes the array's dimension, we only need one. This technique also works in C++.

In the preceding paragraph the `length` field was mentioned. It's used by the Java runtime environment to perform bounds checking on all array access operations. If the access is out of bounds, an exception of type `ArrayIndexOutOfBoundsException` is thrown. Security is one of Java's key features and exception handling is tightly integrated into the core language.

As before, a small example to demonstrate array usage:

"src/java/ArrayExa.java" 12 ≡

```
public class ArrayExa {
  public static void main(String[] args) {
    int m = 3, n = 5;
    // ----- standard Java array -----
    // declare & create
    int A[][];
    A = new int[m][n];
    int[][] B = new int[m][];
    // access
    A[m-1][n-1] = 72;
    System.out.println("A[m-1][n-1]: " + A[m-1][n-1]);
    // space requirements
    System.out.println("A.length: " + A.length);
    System.out.println("A[0].length: " + A[0].length);
    // throw exception with out of bounds access
    A[m][n] = 0;
  }
}
◇
```

Output:

```
A[m-1][n-1]: 72
A.length: 3
```

```
A[0].length: 5
java.lang.ArrayIndexOutOfBoundsException:
        at ArrayExa.main(ArrayExa.java:16)
```

1.3.4.4 Arrays in Scheme

Arrays in Scheme become available with SLIB ([5], p. 5). Add `(require 'array)` to your Scheme program. It tells your interpreter that you need the SLIB array implementation.

There is the type `vector` ([4], p. 26) in the Scheme language, it comes quite close to arrays. But vectors are restricted to usage in one dimension. We will only examine SLIB arrays in this paragraph, because they are more flexible and the basic ideas also apply to vectors. The array implementation is based on the vector type (all items are stored in a vector).

Scheme is a dynamically typed language ([4], p. 2), therefore no variable declaration is necessary. When a variable is defined, its type will be inferred from the given expression. Most types have their own creation procedure, and so do arrays: `(make-array` *init-value* $bound_1 \ldots bound_d$`)`. The function creates an array of dimension[7] d. The expression $bound_i$ is either an integer or a two element list that represents an index range. We can obtain the properties of an array later on with the functions `array-rank`, `array-shape` and `array-dimensions`[8].

According to Scheme's type concept, the items of one array can have different types. We deal with heterogeneous arrays. That's a contradiction with C++ and Java arrays.

The Scheme notation leads to another approach to item access. Items are set with the procedure `array-set!` and referenced with `array-ref`, in compliance with standard Scheme naming conventions. SLIB provides these functions in specialized, more efficient versions[9] for arrays with a rank of 1, 2 or 3. They even allow some nasty tricks to access the items. For instance, if we created a 3x5 array named `A`, the function call `(array-2d-ref A 0 5)` returns item (1, 0). This doesn't work with the general function `array-ref` because bounds checks are done for rows and columns. The specialized functions relax the checks a bit out of performance reasons.

You can't access items outside the array. If you do try to access an item beyond the array's bounds, an error is signalled.

Let's have a look at the mandatory example:

"src/scm/ArrayExa.scm" 13 ≡

```
(require 'array) ;; load SLIB arrays

(define m 3)
(define n 5)
;; creation
(define A (make-array 0 m n))
(define B (make-array #\X '(2 4) '(0 4)))
;; access
(array-2d-set! A 1984 1 n) ;; nasty!
(display "(array-2d-ref A 2 0): ") (display (array-2d-ref A 2 0)) (newline)
(array-set! B #\Y 3 3)
(display "(array-ref B 3 3): ") (display (array-ref B 3 3)) (newline)
;; properties
(display "(array-rank B): ") (display (array-rank B)) (newline)
(display "(array-shape B): ") (display (array-shape B)) (newline)
(display "(array-dimensions B): ") (display (array-dimensions B)) (newline)
;; make Scheme signal an error
(array-ref A 1 n)
◇
```

Output:

[7] SLIB also uses the term *rank* for dimension

[8] This function is incorrectly specified in the manual([5], p. 6). It says: '[...] replaces elements with a 0 minimum with *one greater than* the maximum.' This is not true. An index range with a lower bound of 0 is replaced by the range's upper bound.

[9] `array-1d-ref`, `array-2d-ref`, `array-3d-ref`, `array-1d-set!`, `array-2d-set!` and `array-3d-set!`

```
(array-2d-ref A 2 0): 1984
(array-ref B 3 3): Y
(array-rank B): 2
(array-shape B): ((2 4) (0 4))
(array-dimensions B): ((2 4) 4)
*** ERROR -- array: Bad indices for #(#((rtd) "Array" ... (1 5)
```

1.3.4.5 Arrays in SuchThat

Ok, now that we have examined arrays in different languages, let's fix the Scheme interface for them. Whenever possible, we will stay inside the Scheme standard to minimize customizations on this level.

What's the crucial information about an array? That's very easy to answer, we found the same properties accross all languages:

1. dimension (rank)
2. index range for dimensions
3. item type

The third point only counts for C^{++} and Java. For SuchThat is a strongly typed language, we will designate that a SuchThat array has to be of only one type, no heterogeneous arrays are allowed. The type has to be specified. These decisions also make sense with respect to the back end target language C^{++}.

We require the specification of the index range for every dimension. No omission of any index range as in Java or C^{++} is allowed. The index range may be specified by either giving an integer n_i, which denotes the number of items in dimension i, or by giving a range of two integers n_{i_1} and n_{i_2}. The number of items given by a range are $n_i = (n_{i_2} + 1) - n_{i_1}$.

The back end provides an appropriate mapping to the target language. In our case ScmToCpp we declare arrays by giving the number of items n_i per dimension and access them with indices from 0 to $n_i - 1$.

In SuchThat, it's common to decorate an identifier with extra symbols. An array declaration might look like `Let A[i,j]` $\in$ `T with` $0 \leq$ `i` $\leq m$, $0 \leq$ `j` $\leq n$. We see that the usual array decorations are index variables. They are later referenced in the algorithm body. This observation leads to another requirement:

If the array identifier is decorated with index variables that are referenced in an algorithm body, they must be declared as local variables in the algorithm header. To fulfill this requirement is up to the the front end, either `st2scm` or `scm2scm`.

We can derive the dimension of an array by the number of index ranges given, so no special information about the dimension is necessary.

The bare bones back end can't derive the array's size. It is considered a runtime information that has to be maintained by the programmer. But Java and Scheme both provide means to access this information. The back end can be customized to use specific back end target features. See chapter 6 for a detailed discussion. All these considerations lead to the following definition:

1.3.4.6 Syntax of Scheme Interface for Array Types

⟨Scheme interface: array type 14⟩ ≡

```
array-type
    : "(Array" type { array-index-range }+ ")"
;
array-index-range
    : int-expr
    | "(" int-expr int-expr ")"
;
◇
```

Macro referenced in scrap 9.

We want to avoid clashes with container types and select `Array` as keyword for array types. It's not possible to use a container called `Array`, but this shouldn't be too much a handicap.

Examples The syntax is fixed now and we will give some examples of how to use arrays in SuchThat. We will also show how they look like in the intermediate representation and in our target language C++. We begin with simple array declarations in SuchThat:

`Global:`	`Let` S `be a set.`
`Algorithm:`	`...`
`Input:`	`Let A[i,j] be an array with a[i,j]` $\in S$, $0 \leq$ `i` $\leq m$, $0 \leq$ `j` $\leq n$;
	`Let` $\mathtt{B}_{i,j}$ `be an array with` $\mathtt{b}_{i,j} \in S$, $3 \leq$ `i` $\leq m^{\star}$, $-4 \leq$ `j` $\leq n^{\star}$.

Conforming to our interface specification, parts of the Scheme input to the back end generated by the front end should look like:

```
(st-input (a# . (a#rray s# ((0 m) (0 n)))))
;; also possible: (st-input a# (a#rray s# ((+ m 1) (+ n 1)) ))
(st-input (b# . (a#rray s# ((3 m^\star) (-4 n^\star)))))
(st-local (i . i#nteger))
(st-local (j . i#nteger))
```

Now we get to a simple *real world* example. The algorithm expects a two-dimensional array as input and returns the sum over all its items.

"src/st/ArrayExa.st" 15 ≡

`Global:`	`Let` G `be an additive group.`
`Algorithm:`	n `:= ArraySum2D(`A`,` n_1`,` n_2`)`
`Input:`	`Let` $A_{i,j}$ `be an array with` $a_{i,j} \in G$, $1 \leq i \leq n_1$, $1 \leq j \leq n_2$.
`Output:`	$n \in \mathbb{N}$ `such that` $n = \sum_{i=1}^{n_1} \sum_{j=1}^{n_2} a_{i,j}$.

(1) `[init` n`]`
 n `:= 0;`
(2) `[compute sum]`
 `for` i`=1,...,`n_1 `do`
  `for` j`=1,...,`n_2 `do` n `:=` $n + a_{i,j}$`;`
(3) `stop` □
◇

1.3.4.7 Vectors

Vectors are very closely related to arrays. They have one big advantage over arrays. Vectors are resizeable at runtime and this property is transparent to the programmer. He can add elements to the vector and if its size is exceeded, new memory is allocated. A common allocation strategy is doubling the vector's size whenever the bounds are reached[10]. We will examine vectors as prototype of a container.

1.3.4.8 Vectors in C++

Vectors were introduced to C++ with the acceptance of the STL as part of the standard library.

> A vector is a kind of sequence that supports random access iterators. In addition, it supports (amortized) constant time insert and erase operations at the end; insert and erase in the middle take linear time. Storage management is handled automatically, though hints can be given to improve efficiency ([14], 23.2.4).

One points is especially noteworthy. The remark that insertions happen in *amortized* constant time, hint to the slightly less optimal efficiency with regard to arrays. But this is significant only in very rare situations. Whenever the vector has to be resized, a new memory chunk is allocated

[10] used in the STL and the `java.util.Vetor` class

and *all* vector elements are copied to the new location. Only hardware constraints (virtual and physical memory) limit a vector's size.

Constructors are used to declare and define vectors. The STL provides default and copy constructors with the usual semantics. In addition, a vector can be constructed with a given size and an initial value, which is provided, or obtained by the objects default constructor. The last constructor expects two iterators as arguments and creates a vector initialized with the elements inside the range spanned by the iterators.

Vectors are template classes and therefore always need a type parameter when declared. The C++ compiler generates an instantiation of the template class for every type parameter. Because the declaration demands the type parameter, the compiler can enforce the vector's homogeneity. A heterogeneous vector is achieved with polymorph objects inserted into it. The base class is used as type parameter and any subtype is allowed afterwards to be put into the vector.

Vectors distinguish between size and capacity. The size is the number of elements currently stored in the vector, and the capacity is the number of elements possibly stored without reallocation. This approach is necessary to allow amortized insertion at the end. It would be extremely inefficient to increase the vector's size by the number of elements appended, every time a method is called.

The memory allocation is done by a default allocator, which can be overridden by user written allocators, which is quite uncommon:

> ... the default allocator class supplied with STL implementations is sufficient for most programmers' needs ([22], p. 43).

Vector elements are accessed with the member function `at(size_type n)` or with the subscript operator '`[ ]`', due to the fact that `vector` is a random access container. It also supplies the methods `front` and `back`[11], which all containers support, and their modifier functions, like `push_back(T& x)`. All algorithms are available for vectors that operate on sequences.

The STL has been widely criticized for omitting any bounds checking. The main reason was efficiency. The vector container should be a more versatile array replacement. C++ programmers are known for their "need for speed" and any bounds checks have an impact on runtime efficiency. But quality and robustness are more and more important in todays huge software projects, which led to the insight that the STL as foundation of C++ programming should comply to this criteria, too. The first STL implementation with exception handling is the one by SGI ([28]) and Alexander Stepanow, the main developer of the original STL. The release note of version 2.0 states: "All STL components are now exception safe." Time measurements show that the SGI STL runs 5-10% slower with exception handling turned on.

This short recap on vectors and the STL is summarized in an example:

"src/cpp/VectorExa.cpp" 16 ≡

```
#include <iostream>
#include <vector>
using namespace std;
void main() {
  const m = 5, n = 3;
  // ----- STL vector -----
  // declare
  vector< vector<int> > v(m, vector<int>(n));
  // access
  v[m-1][n-1] = 97;
  v[0].push_back(2408);
  cout << "v[m-1][n-1]: " << v[m-1][n-1] << endl;
  cout << "v.at(0).at(n): " << v.at(0).at(n) << endl;
  // space requirements
  cout << "sizeof(vector<int>): " << sizeof(vector<int>) << endl;
  cout << "v.size(): " << v.size() << endl;
```

[11] `reference front()`, `reference back()`

```
    cout << "v[0].size(): " << v[0].size() << endl;
    cout << "v[0].capacity(): " << v[0].capacity() << endl;
    cout << "v[1].size(): " << v[1].size() << endl;
    cout << "v[1].capacity(): " << v[1].capacity() << endl;
  }
  ◇
```

Output:

```
b[m-1][n-1]: 97
b.at(0).at(n): 2408
sizeof(vector<int>): 16
b.size(): 5
b[0].size(): 4
b[0].capacity(): 6
b[1].size(): 3
b[1].capacity(): 3
```

The only thing that strikes one queer is the constructor of the two-dimensional vector. The intuitive way is to write: `vector< vector<int> > b(m, n);`
At first sight, this looks ok. But we have to be careful. What constructor does the compiler actually call? The only possible is `vector(size_type n, const T& value = T())`[12], which leads to a type mismatch: the second argument must have the type of the template's type parameter, which is `vector<int>`. Now the implicit type conversion of C++ kicks into action. `vector<int>` has a constructor `vector(size_type n)` and therefore an `int` can be cast to it. The solution for this problem is a new keyword in C++: `explicit`. It prohibits the type conversion to take place. And this is really used in the standard, the complete constructor definition says ([14], 23.2.4.1) :

```
explicit vector(size_type n, const T& value = T(),
                const Allocator& = Allocator());
```

Some older compilers, among them gcc v2.72, compile the intuitive form without errors, but newer ones reject it.

1.3.4.9 Vectors in Java

The class `java.util.Vector` defines the interface for Java vectors. They are similar to those in C++ as they serve the same purpose. But the implementation is based on an object oriented model, not on type parametric polymorphism.

A new `Vector` object is created with the provided constructors. The elements stored in a `Vector` are all of type `Object`, the superclass of all objects in Java. There are two consequences of this major discrepancy to C++. Vectors are heterogeneous, and type safety at compile time is nonexistent. We also can't store simple types like `int` or `char` in vectors, they have to wrapped in `Integer()` and `Character()` classes.

When we access elements of a `Vector`, we do this with methods supplied by the class. But to operate on the returned objects, it's necessary to cast them to their type if we use type-specific methods. To add the first two numbers in a `Vector`, we write:

```
x = ((Integer) v.elementAt(0)).intValue()+((Integer) v.elementAt(1)).intValue();
```

The capacity increment can either be a constant, or if this `int` is set to 0, it is doubled whenever a resize of the vector is needed. The user can change this behavior with special constructors.

Methods that access elements inside a vector with an index throw an exception, if the index is not valid. They use the same exception type as arrays, because the implementation uses arrays as low-level container.

The example just doesn't look as nice as in C++, but there are no difficult constructs to understand. It reflects the different approaches in Java and C++: simplicity versus expressive power.

[12]the allocator default argument is omitted for the sake of simplicity

"src/java/VectorExa.java" 18 ≡

```
import java.util.Vector;
public class VectorExa {
  public static void main(String[] args) {
    final int m = 5, n = 3;
    // define
    Vector v = new Vector(m);
    for (int i = 0; i < m; i++) {
      v.addElement(new Vector(n));
      ((Vector) v.elementAt(i)).setSize(n);
    }
    // access
    ((Vector) v.elementAt(m-1)).setElementAt(new Integer(97), n-1);
    ((Vector) v.elementAt(0)).addElement(new Integer(2408));
    System.out.println("((Vector) v.elementAt(m-1)).elementAt(n-1):"
                       + ((Vector) v.elementAt(m-1)).elementAt(n-1));
    System.out.println("((Vector) v.elementAt(0)).elementAt(0): "
                       + ((Vector) v.elementAt(0)).elementAt(n));
    // space requirements
    System.out.println("v.size(): " + v.size());
    System.out.println("((Vector) v.elementAt(0)).size(): "
                       + ((Vector) v.elementAt(0)).size());
    System.out.println("((Vector) v.elementAt(0)).capacity(): "
                       + ((Vector) v.elementAt(0)).capacity());
    System.out.println("((Vector) v.elementAt(1)).size(): "
                       + ((Vector) v.elementAt(1)).size());
    System.out.println("((Vector) v.elementAt(1)).capacity(): "
                       + ((Vector) v.elementAt(1)).capacity());
    // access violation
    ((Vector) v.elementAt(m)).setElementAt(new Integer(97), n);
  }
}
```

◇

Output:

```
((Vector) v.elementAt(m-1)).elementAt(n-1):97
((Vector) v.elementAt(0)).elementAt(0): 2408
v.size(): 5
((Vector) v.elementAt(0)).size(): 4
((Vector) v.elementAt(0)).capacity(): 6
((Vector) v.elementAt(1)).size(): 3
((Vector) v.elementAt(1)).capacity(): 3
java.lang.ArrayIndexOutOfBoundsException: 5 >= 5
        at java.util.Vector.elementAt(Vector.java:250)
        at VectorExa.main(VectorExa.java:28)
```

There is no way to create a multidimensional vector in Java without creating a new class for it. Or we have to initialize every object by hand, as done in the example.

We skip the Scheme section because it has no datatype comparable to vectors and go straight to SuchThat.

1.3.4.10 Containers in SuchThat

It's absolutely not our intention to encourage the SuchThat programmer to make use of containers directly. The structures in SuchThat are mapped on containers and we only provide an interface to which the mapping can occur.

We've seen that the declaration of a container in C++ has a *must*: the type parameters. Vectors, lists and other sequence containers expect one, associative containers[13] want two type parameters: one for the key and one for the value type. Summed up, we need this information about a container:

[13] `map` and `multimap`

1. the container's name
2. type parameters

The type parameters themselves can be containers, we allow recursive types. Of course, arrays and flat types are perfectly nice, too.

This very simple mechanism is doubtful in general. Only growing experience with SuchThat and its compiler tools will show if the simple container mapping will suffice. Maybe the back end will have to be based on a complete C++ class hierarchy that encapsulates the algebraic structures.

1.3.4.11 Syntax of Scheme Interface for Container Types

The container's name is an identifier and the type parameters are types on their own.

⟨Scheme interface: container-type 19⟩ ≡

```
container-type
    : "(" identifier { type }+ ")"
;
◇
```

Macro referenced in scrap 9.

1.3.5 Operations on Datatypes

We now have equipped the Scheme interface with the ability to declare datatypes. But one important discussion was delayed up to the moment. Every datatype is linked with "special functions" that perform operations on it. They can be extremely simple, like accessing elements inside an array, or sophisticated functions like calculating the determinant of a matrix. In our intermediate representation they all look the same, they are given in Scheme's regularized functional notation.

But in the back end target language this is quite a different story. The special functions can be global functions, member functions of classes, or even operator notations. We are left with two opportunities:

1. select a set of special functions and introduce them as keywords in the Scheme interface
2. dynamically provide the code generator with a list of the special functions and customized functions to handle them[14]

The first solution has the advantages that it's very simple to implement and the front end developer can rely on a fixed set of instructions. The second one puts more load on the back end's configurability. But the benefits of a dynamically extendable back end outweigh the more complex implementation. In addition, SuchThat is a young language and only time will show which features and functions are truly needed. We've chosen the second solution to meet these demands. The topic is discussed in more detail in chapter 6.

1.3.6 Datatype Summary

A type can either be *concrete*, which all build-in fundamental SuchThat and container types, with only fundamental types as type parameters, are, or *generic*, which are simple, non-fundamental types. Arrays can be treated like containers in this context. Figure 1.2 visualizes the type dependencies. *Flat type* is a conceptual term. Both flat types, fundamental and generic, can be used in container and array types. They end the possible recursion in container declarations.

All build-in types are named in `type-list`, which reads its content at the compiler startup from the file `type-list.ini`.

[14] these functions may generate code for special functions

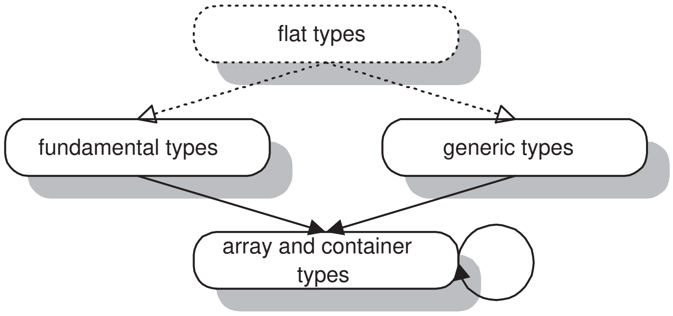

Figure 1.2: dependencies in the type hierarchy

1.4 Expressions

Expressions are constructs that are evaluated according to fixed rules and then return a value. This value is typed, but programming languages differ in the time point of the type check. They vary from checks at compile time, for example C++, and type checks at run time, where Smalltalk or Scheme are well known examples. Mixed forms also exist. A recap of expressions is presented and the Scheme interface defined.

⟨Scheme interface 20⟩ ≡
    ⟨Scheme interface: expressions 21⟩
    ◇

Macro defined by scraps 3, 5, 9, 20, 22a.
Macro referenced in scrap 1.

1.4.1 Expressions in C++ and Java

Expressions in imperative languages are typically made up of these three components:

1. literals, constants and variables
2. binary infix operator expressions
3. unary prefix expressions[15]
4. function calls

C++ and Java constitute no exception. Literals are either characters, strings, integers or floats. Table 1.9 summarizes the notation of literals in C++ and Java.

Expressions are combined with operators. These are either unary or binary. If more than two expressions are combined with operators, special precedence and associativity rules apply, and the expression is evaluated in the order imposed by those rules. The precedence rules may be overruled with parantheses. Java and C++ have a fixed set of operators which don't match completely.[16]

The meaning of operators in Java is well defined for the build-in types and can't be extended any further. This is a major drawback with respect to C++. In C++ overloading of operators for user defined classes is good programming style. It enhances the readability of expressions if used properly. The shift operators gained new appreciation with the I/O library `iostream` of the new C++ standard.

At least for functions Java allows overloading. Function calls look the same in Java and C++. The identifier denoting the function's name is followed by the parameters enclosed in parantheses. But they take different turns semantically. Java enforces call-by-value semantics for build-in types and call-by-reference semantics for classes. There's no way to change this behavior. C++ leaves the decision up to the programmer. The function's signature determines the semantics. Two example expressions sum up the facts:

[15] sometimes even postfix expressions, like the famous C++ notation `i++`

[16] all operators for Java with precedence rules are listed in [7], p. 33; a similar overview for C++ can be found in Lippman's C++ Primer ([16], p. 87)

```
floor(3.1415926 * 2) + 7
!((a + b) / (c - d))
```

1.4.2 Expressions in Scheme

Scheme is a functional language so expressions are written in prefix notation. No evaluation rules for operators are needed and there's made no distinction between operators and function calls. We list the C++ examples in Scheme notation:

```
(+ (floor (* 3.1415926 2)) 7)
(not (/ (+ a b) (- c d)))
```

Scheme is statically scoped whereas SuchThat, C++ and Java use dynamic scoping. This plays no role for our back end ScmToCpp, but if someone wants to implement a Scheme back end, the contradicting scoping techniques must be handled with care. The side effects of dynamic versus static scoping must be examined thoroughly. We present a table with Scheme notations of literals in table 1.9, compared to C++ and Java.

literal type	Scheme	C++ and Java
character	`#\a`	`'a'`
string	`"Hello World!"`	`"Hello World!"`
integer	`1984`	`1984`
rational	`5/3`	–
float	`3.1415926`	`3.1415926`
complex	`3+7i`	–

Table 1.9: notation of literals in Scheme, C++ and Java

1.4.3 Expressions in SuchThat

SuchThat is a typical imperative language and has almost the same characteristics as Java and C++. The set of operators is a union of those from Aldes and LaTeX. Of course this large amount of operators can't be mapped to C++ or Java operators, therefore they will be mapped to functions. The assignment $x = a \oplus b$ is transformed to `(set! x (\oplus a b))`. With the appropriate function mappings set, the C++ code could look like: `x = specialSum(a, b)`.

1.4.4 Syntax of Scheme Interface for Expressions

We use the regularized syntax of Scheme expressions in the interface. This gives us the possibility to give a very compact expression grammar. The literals are adopted straight from [4] and correspond with the types in our datatype comparison.

⟨Scheme interface: expressions 21⟩ ≡

```
expr
    : literal
    | identifier
    | "(" identifier { expr }* ")"
;
literal
    : character-constant
    | string-constant
    | boolean-constant
    | integer-constant
    | rational-constant
    | real-constant
    | complex-constant
;
◇
```

Macro referenced in scrap 20.

1.5 Statements

The statements included in our Scheme interface originate from the imperative SuchThat statement part and standard Scheme statements. Both should be supported by the interface to a reasonable extent. This facilitates the work of the front and the back end. We try to provide an interface that can be the source for an imperative as well as for a functional target language.

SuchThat inherits the imperative statements from Aldes ([17], p. 10, gives the syntax of all Aldes statements). All statements in the interface are listed here and will be described briefly on the next pages.

⟨Scheme interface 22a⟩ ≡
```
statement
    : set!-statement
    | begin-statement
    | if-statement
    | case-statement
    | cond-statement
    | repeat-statement
    | while-statement
    | for-statement
    | for1-statement
    | return-statement
    | define-statement
;
⟨Scheme interface: Scheme statements 22b, ... ⟩
⟨Scheme interface: iteration statements 24a, ... ⟩
⟨Scheme interface: return statement 24d⟩
⟨Scheme interface: let statement 30⟩
⟨Scheme interface: definitions 25⟩
◇
```
Macro defined by scraps 3, 5, 9, 20, 22a.
Macro referenced in scrap 1.

1.5.1 Adopted Scheme Statements

Honoring the fact that Scheme is the language of our interface we adopt some of its statements. We only list them here without further remarks because a description is available with [4].

`define` and `let` have a special meaning in ScmToCpp and will be described on their own.

1.5.1.1 Assignment Operation

Assignments are made with the `set!` statement because of the functional notation in Scheme.

⟨Scheme interface: Scheme statements 22b⟩ ≡
```
set!-statement
    : "(set!" identifier expr ")"
;
◇
```
Macro defined by scraps 22bc, 23abc.
Macro referenced in scrap 22a.

1.5.1.2 Sequencing

In Scheme the order of evaluation is generally undefined. Thus a special statement for sequencing is necessary: `begin`.

⟨Scheme interface: Scheme statements 22c⟩ ≡

```
begin-statement
    : "(begin" { statement }* ")"
;
◇
```

Macro defined by scraps 22bc, 23abc.
Macro referenced in scrap 22a.

1.5.1.3 Conditionals

`if` and `case` are standard conditional constructs in almost every programming language. The `if` statement is allowed with and without alternate branch.

⟨Scheme interface: Scheme statements 23a⟩ ≡

```
if-statement
    : "(if" expr statement [ statement ] ")"
;
◇
```

Macro defined by scraps 22bc, 23abc.
Macro referenced in scrap 22a.

The `case` statements compares values against a given key and enters the first clause with a matching datum.

⟨Scheme interface: Scheme statements 23b⟩ ≡

```
case-statement
    : "(case" expr { case-clause }* [ else-clause ] ")"
;
case-clause
    : "((" { expr }+ ")" { statement }* ")"
;
else-clause
    : "(else" { statement }* ")"
;
◇
```

Macro defined by scraps 22bc, 23abc.
Macro referenced in scrap 22a.

`cond` is a more elaborate `if` statement. It allows a list of conditions to test, extending the `if` semantics of consequent and alternate.

⟨Scheme interface: Scheme statements 23c⟩ ≡

```
cond-statement
    : "(cond"  { cond-clause }* [ else-clause ] ")"
;
cond-clause
    : "(" expr { statement }* ")"
;
◇
```

Macro defined by scraps 22bc, 23abc.
Macro referenced in scrap 22a.

1.5.2 Iteration Statements

Iteration statements are an important building block of imperative languages. We will have a brief glimpse on iteration constructs and define them in the Scheme interface.

1.5.2.1 Iteration in Scheme

Scheme supports iteration mainly with the `do` construct ([4], p. 11). It's got quite a complex syntax and can be used for a wide variety of tasks. They exceed the needs of our back end and no equivalent SuchThat construct is available, therefore we omit it from our interface.

1.5.2.2 Iteration in C++ and Java

C++ and Java have almost the same statement syntax and are treated together in our short examination. Iteration in both languages is supported by three statements: `for`, `while` and `repeat`. They have their counterparts in most languages and are very simple to understand and utilize.

1.5.2.3 Iteration in SuchThat

SuchThat comes along with the same set of iteration constructs as Java and C++. They differ slightly in their syntax but we will equalize them in the interface.

1.5.2.4 Syntax of Scheme Interface for Iteration Statements

The `while` and `repeat` statements are straightforward. They consist of a test expression and the iteration body.

⟨Scheme interface: iteration statements 24a⟩ ≡

```
repeat-statement
    : "(repeat" { statement }* "until" expr ")"
;
while-statement
    : "(while" expr { statement }* ")"
;
◇
```

Macro defined by scraps 24abc.
Macro referenced in scrap 22a.

The `for` statement is supported in two flavours, the specialized `for1` and the general `for`. A very common usage of `for` statements is a counter i with the step width s of $s = 1$. This is captured in the `for1` construct. The expressions are the initial (t_1) and the final (t_2) counter value. Iteration continues as long as $i \leq t_2$.

⟨Scheme interface: iteration statements 24b⟩ ≡

```
for1-statement
    : "(for1" identifier expr expr { statement }* ")"
;
◇
```

Macro defined by scraps 24abc.
Macro referenced in scrap 22a.

The `for` construct in our interface expects the initial (t_1), the second (t_2) and the final (t_3) counter value. Therefore the step width is $s = t_2 - t_1$ and the iteration continues as long as $i \leq t_3$ for $s > 0$ or $i \geq t_3$ for $s < 0$.

⟨Scheme interface: iteration statements 24c⟩ ≡

```
for-statement
    : "(for" identifier expr expr expr { statement }* ")"
;
◇
```

Macro defined by scraps 24abc.
Macro referenced in scrap 22a.

1.5.3 The `return` Statement

Algorithms usually return a value and this may happen at various points in the algorithm description. The `return` statement stops the execution of the algorithm and returns the expression which makes up the statement's operand.

⟨Scheme interface: `return` statement 24d⟩ ≡

```
return-statement
    : "(return" expr ")"
;
◇
```

Macro referenced in scrap 22a.

1.5.4 The `define` Statement

Definitions serve the purpose of either defining new variables or new functions. Function definitions are very important in our back end concept and will be described on their own (see section 1.6).

Variable definitions are composed of an identifier and an expression that holds the value to which the variable should be initialized. Contradictory to the common usage of definitions the identifier is not introduced with the `define` statement. *Declarations* do this job and are the next section's topic.

⟨Scheme interface: definitions 25⟩ ≡
```
definition
    : variable-definition | function-definition
;
variable-definition
    : "(define" identifier expr ")"
;
```
⟨Scheme interface: function definitions 29b⟩
◇

Macro referenced in scrap 22a.

1.6 Function Declarations and Definitions

We give function definitions an extra space for discussion. SuchThat intends to be a language particularly for library construction. The main unit of control is the algorithm. If we want to map the algorithm concept to classical languages the natural choice are functions. In this section we will explore the power of C^{++}, Java and Scheme to express functions.

Both Java and C^{++} are object oriented programming languages. But we will omit almost any of their peculiarities influenced by this property because SuchThat is *not* object oriented. The exploitation of object oriented idioms could have an immense impact on the library design of WLOG, but it's another research topic and won't be discussed here any further.

1.6.1 Functions in C^{++}

Non-template functions are declared by denoting the return value's type, the function name and the parameter types. Identifiers for the parameters, so-called formal arguments, are optional. All parameters may be provided with default values which will be used if they are omitted in a function call. The function header is called *signature*.

The type parameters can be decorated with certain modifiers. Perhaps the two most important ones are reference and `const` declarations. The default semantic of function arguments is call-by-value. This can be overruled with declarations of the form `type & identifier`. All parameters decorated with the ampersand '`&`' are references. The `const` operator is one of the most overloaded keywords in C^{++}. We will give a naive explanation of its use in function declarations. Parameters declared `const` should not be changed within the function, and the compiler does checks at compile time to enforce the `const` declaration. This gives the library designer a powerful facility to protect functions against common misusage. It also prevents the library user from falling into popular pitfalls.

Another important feature of C^{++} is *overloading*, a concept central to all object oriented languages. The new C^{++} draft defines the idea:

> When two or more different declarations are specified for a single name in the same scope, that name is said to be *overloaded* ([14], 13.0.1).

Functions are overloaded by giving different signatures with the same function name. An important exception is stated in the draft:

> Function declarations that differ only in the return type cannot be overloaded ([14], 13.1.2).

This coercion is understandable from the compiler constructor's point of view. It's sometimes impossible to obtain the type of a function call based solely on its return value's type.

```
int sideEffect(int & a, float & b);
float sideEffect(int & a, float & b);
int i; float f;
sideEffect(i, f);
```

The compiler can't decide which function to call. A solution might be to state the expression's type explicitly to avoid ambiguities, just like in a type cast. The front end must deliver appropriate function name mappings, such that the back end target language can resolve the overloading.

Ada95 is, besides SuchThat, a language that supports overloaded functions that differ only in the return type. In order to avoid the ambiguity in the example a few lines above, Ada95 supplies *qualified expressions*:

> A qualified_expression is used to state explicitly the type, and to verify the subtype, of an operand that is either an expression or an aggregate ([32], p. 101).

The example in Ada95 looks like:

```
function sideEffect(a: Integer; b: Real) return Integer;
function sideEffect(a: Integer; b: Real) return Real;
i : Integer; r : Real;
Real'(sideEffect(i, r));        -- qualification needed
```

Function declarations also work for most of C++'s operators. Their declaration must be prefixed with the keyword `operator`. Ok, now we will give some examples of non-generic function declarations.

```
int max(int a, int b, int c = 0); // default argument
float max(float a, float b, float c = 0); // max is overloaded
string& operator+(const string & s, const string & t);
void swap(int & a, int & b);
```

Overloading helps us on the way from SuchThat to C++. But the construct that is closest to generic programming are *templates*. Template techniques have reshaped the C++ library in recent years and are the feature that changed the C++ world to a great extent. Again, we listen to the draft: "A template defines a family of types or functions ([14], 14.0.1)." This statement divides templates in its two main categories: class and function templates. We will explore function templates because they come very close to our generic algorithm concept. All the things said about non-template functions still hold true.

"Function templates define an unbounded set of related functions ([14], 14.0.1)." This is almost what we require from an generic algorithm, with one crucial exception. Function templates put no restrictions on their type parameters. This is the job of the algebraic resolution in the front end. We take for granted that the types presented to our back end are properly checked and we use them in templates without hesitation.

Function templates begin with the keyword `template`, followed by the the *template parameters* enclosed in relation signs '`< >`'. Inside the template function body we can use all of the template parameters just like normal types. The template parameters will be replaced by concrete types at instantiation time. This is done by the compiler, we don't have to tell the actual type parameter when calling a template function.

The syntax and semantics of templates is a very complex field and we refer to section 14 in the C++ standard ([14]) for more detailed information.

A simple example shows how to write a "generic" function in C++. It defines a global template function that performs a bubble sort on elements inside an array.

"src/cpp/SortDemo.cpp" 26 ≡

```
#include <iostream>
using namespace std;
typedef unsigned int size_type;
// global generic function
template<class T>
void bubbleSort(T a[], size_type size) {
  T tmp;
  for (size_type i = 0; i < (size - 1); i++)
    for (size_type j = size - 1; j > i; j--)
      if (a[i] > a[j]) { tmp = a[i]; a[i] = a[j]; a[j] = tmp; }
}
void main() {
  int a[10] = {9, 66, -20, 12, 446, -2987, 0, 1984, -9876, 20};
  bubbleSort(a, 10);
  for (int i = 0; i < 10; i++) cout << a[i] << " ";
  cout << endl;
}
◇
```

Output: `-9876 -2987 -20 0 9 12 20 66 446 1984`

Functions can either be member functions of classes or global. Member functions always have the implicit object `this`, which is a pointer to the object on which the function is used. We will ignore this functions at the moment. Global functions are declared and defined outside class scope and have no implicit object. The STL makes heavy use of global functions and we will map algorithms on global functions, too. More on this in chapter 4.

1.6.2 Functions in Java

Functions in Java are less expressive than in C++. One design goal of Java was simplicity. The template mechanism of C++ can't be called simple and was discarded by Java's designers. Another victim were operators, they can't be overloaded. Last not least Java handles the calling semantic by itself. We already mentioned this in the context of expressions. Java also has no global functions or methods. The closest we get to them is with static methods, so-called *class methods*. They are referred to by their class name.

The only way to write Java programs generic-like is by using the superclass of all other classes, `Object`, as generic type. An example of this idiom is the `Vector` class in the `java.util` package. We will try to write the simple C++ example in Java using `Vector` instead of an array.

```
"src/java/SortDemo.java" 27 ≡
      import java.util.Vector;
      public class SortDemo {
        static void bubbleSort(Vector v) {
          Object tmp;
          for (int i = 0; i < (v.size() - 1); i++)
            for (int j = v.size() - 1; j > i; j--)
              if (v.elementAt(i).hashCode() > v.elementAt(j).hashCode()) {
                tmp = v.elementAt(i);
                v.setElementAt(v.elementAt(j), i);
                v.setElementAt(tmp, j);
              }
        }
        public static void main(String[] args) {
          int [] a = {9, 66, -20, 12, 446, -2987, 0, 1984, -9876, 20};
          Vector v = new Vector(10);
          for (int i = 0; i < a.length; i++)
            v.addElement(new Integer(a[i]));
          bubbleSort(v);
          for (int i = 0; i < v.size(); i++)
            System.out.print(v.elementAt(i) + " ");
```

```
    System.out.println();
  }
}
◇
```

Output: `-9876 -2987 -20 0 9 12 20 66 446 1984`

The function `bubbleSort` operates on `Vector`. Its elements can be of any type, except the build-in types. But wrapper classes like `Integer` do exist for these cases. The method `hashCode`[17] is inherited by every subclass of `Object`.

1.6.3 Functions in Scheme

In Scheme we have a change of view regarding functions. The basic building block for Scheme functions are lambda expressions. In Scheme, functions can return functions and have them as input parameters, these are so-called *higher order functions*. New functions are introduced with the `define` statement. A function that returns the square of its input looks like this (both forms are equivalent, the second one is syntactic sugar provided by most Scheme interpreters):

```
(define square (lambda (n) (* n n)))
(define (sq n) (* n n))
```

In Java, the comparison of elements was solved with the inherited method `hashCode`, which every class can overload. In C++ the same is possible, but it's more common to overload one of the relation operators.

Scheme takes another approach with *predicates*. These are functions that return either `#\t` or `#\f`, and by convention end with a question mark. Well known predicates test the type of a Scheme object[18]. The technique called *higher order programming* expects functions as parameters and returns them. The comparison can simply be done by a predicate which is passed as parameter to the sorting algorithm. The concepts of higher order programming are laid out in [13].

"src/scm/SortDemo.scm" 28 ≡

```
(define more? >)
(define v (vector 9 66 -20 12 446 -2987 0 1984 -9876 20))
(define bubbleSort!
  (lambda (v pred?)
    (do ((i 0 (+ i 1)))
        ((= i (- (vector-length v) 1)))
      (do ((j (- (vector-length v) 1) (- j 1)) (tmp '()))
          ((= i j))
        (if (more? (vector-ref v i) (vector-ref v j))
            (begin (set! tmp (vector-ref v i))
                   (vector-set! v i (vector-ref v j))
                   (vector-set! v j tmp) )) )) ))
(bubbleSort! v more?)
(display v) (newline)
◇
```

Output: `#(-9876 -2987 -20 0 9 12 20 66 446 1984)`

This example is of course horrible functional programming style, but for the sake of conformance we ported the imperative example to Scheme. All type specific function calls could be replaced with function parameters, but it's more important to get an idea of the technique.

The final refinement would be to provide a function, that expects all the type-specific procedures as input and that returns a procedure that needs only the data as function parameters. This concept is called *currying* (see [29], p. 210 for details). In the STL similar idioms are used, the *function objects*. Functions that simulate currying are assembled in the header `functional` of the C++ standard library.

[17] `public int hashCode()`
[18] `list?`, `number?`, `boolean?` ...

1.6.4 Algorithms in SuchThat

The algorithm concept of SuchThat is described in [27]. Input, output and local parameters are declared and all globally declared structures can be used. The sorting example stated in SuchThat:

"src/st/SortDemo.st" 29a ≡

```
Global:    Let S be a set with order <.
Algorithm: bubbleSort(A, n)
Input:     Let A_i be an array with a_i ∈ S, 1 ≤ i ≤ n.
Output:    A with a_i ≤ a_{i+1} ∀ 1 ≤ i ≤ n − 1.
Local:     tmp ∈ S, i, j ∈ Integer.

(1) [bubble sort]
    for i=1,...,n − 1 do
      for j=n,...,i + 1 do
        if a_i > a_j then { tmp = a_i; a_i = a_j; a_j = tmp; }
(2) stop □
◇
```

Prototypes are algorithms without the algorithm description. Algorithms don't state if they are generic or concrete. This is detected by our back end. If one of the parameters is generic, the algorithm is also generic. Operators are distinguished from functions and can be overloaded. As in C++, the keyword `operator` is used. It's up to the back end if these function definitions are really mapped to operator constructs in the target language. The keyword `operator` is seen merely as recommendation.

What do we need to define a function, or at least to declare it? We found the following items:

- the function name
- the parameter list
- the return value type (not in Scheme)
- the function body (only for definitions)

We stick to Scheme syntax and add some special constructs to cover all features of SuchThat. The function name can either be a standard identifier or a list that's made up of a string denoting the function type and an identifier. Currently only two function type strings are supported: `standard` and `operator`. But extensions are possible, for example `generic` or `concrete`.

We support both Scheme function definitions presented in the preceding section and name the input and output parameters[19] there.

After that we tell the back end all the variables that appear in the algorithm and of which type they are. We already discussed how to declare variables in the interface (see section 1.2). The variables already mentioned in the function's parameter list are repeated, but with type information.

The algorithm is completed with its body. The body is either a sequence of statements or a `let` statement (will be described below). An empty body indicates an algorithm prototype.

1.6.4.1 Syntax of Scheme Interface for Function Definitions

The `call/cc` syntax is necessary to enable premature exits of functions with the `return` statement. It is important to note that variable declarations in the function body may not be of global type.

⟨Scheme interface: function definitions 29b⟩ ≡

[19] if an algorithm has only one output parameter, we use it as return value type of the function and don't name it in the parameter list of the Scheme function

```
function-definition
    : type-a-function | type-b-function
;
type-a-function
    : "(define" function-name "(lambda (" { identifier }* ")" function-body "))"
;
type-b-function
    : "(define (" function-name { identifier }* ")" function-body ")"
;
function-body
    :  { variable-declaration }*
       "(call-with-current-continuation (lambda (return)"
       { { statement }* | let-statement } "))"
;
function-name
    : identifier
    | "(" function-type identifier ")"
;
function-type
    : "standard" | "operator"
;
◇
```

Macro referenced in scrap 25.

1.6.4.2 The `let` Statement

Java and C++ allow declarations to appear anywhere in the source code. SuchThat limits this locations to declaration lines that start with `Global:`, `Input:`, `Output:` or `Local:`. Scheme has another way of introducing new identifiers to the namespace. The `let` statement lists all new identifiers with their initial value. Thereafter the `let`-body is evaluated. Because SuchThat has only two scope levels, we abuse Scheme's `let` statement. All output and local variables of an algorithm may be initialized with a `let` that wraps the algorithm body. In Scheme, the local variables are also defined here, but an imperative language treats the `let` statement like a list of assignments of already declared variables.

⟨Scheme interface: `let` statement 30⟩ ≡

```
let-statement
    : "(let (" { "(" identifier expr ")" }* ")" { statement }* ")"
;
◇
```

Macro referenced in scrap 22a.

The `let` statement is only permitted at the start of an algorithm description, see the grammar of function definitions above.

Chapter 2

The Design of ScmToCpp

In this chapter we will describe the design of our back end ScmToCpp. All design decisions are explained with links to established theoretical background in the field of compiler construction. Furthermore we examine the main structure of the implementation.

2.1 Design Decisions

In [35], p. 358, the main criteria in making implementation decisions are mentioned, namely maintainability, performance and portability. The latter is split up into rehostability and retargetability.

2.1.1 Maintainability

The maintainability of a compiler is heavily influenced by its design. A clear, modular design makes it very easy to change specific parts of the compiler, and allows good error detection. The continuing development of SuchThat conditions such a design, because the compiler will be subject to frequent changes until growing experience slows down this progress.

To measure up to this requirements, we decided to implement a classical 2-pass compiler. Pass 1 performs the analysis phase, it creates the parse tree and symbol tables. Pass 2, the synthesis phase, generates the the target code. Both passes are captured in a function, pass 1 in `parse-objects` and pass 2 in `generate-code`.

In order to enforce this structure, we designate a special file structure. The two passes are isolated in different files. The core code of pass 1 is stored in `parse.scm`, pass 2 is stored in `code.scm`. The extensions of pass 2 (see section 6) are stored in `code-ext.scm`. Auxiliary functions went into the file `misc.scm`. These files hold the functions and data that are needed by the main program ScmToCpp, which is stored in the file `scm2cpp.scm`. Figure 2.1 displays the file structure of our compiler.

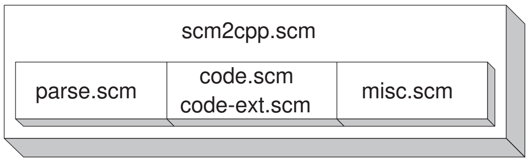

Figure 2.1: the compiler's source code file structure

2.1.2 Performance

A modular design has a negative impact on the performance of the compiler. Every module normally has to traverse the processed data again and a communication channel between the passes is necessary. Several tests showed that a 2-pass compiler is about 10% slower than a 1-pass compiler.

The performance of the compiler isn't at top priority, though. We want to deliver a test bed for SuchThat. In addition, SuchThat's main employment field is algebraic programming, where algorithms don't get very large (compared to C++ system programs).

The two compiler phases share data in global variables. The alternative to this strategy are temporary files. But the immense memory configurations and intelligent virtual memory managers in todays computer systems enable us to discard this slower possibility.

A stable SuchThat development system will lead to more research on better performance of the compiler tools, but we are still far away from this state.

2.1.3 Rehostability

The program should be easily portable to many processor architectures and operating systems. We achieve this by sticking to established formal standards. The back end is written in Scheme, complying to the standard defined by the revised report ([4]). Two main advantages derive from this decision:

- the back end can be ported to any platform that has a Scheme interpreter or compiler
- the `read` function does the lexical analysis and parantheses balance checks

But we also have a major drawback. The `read` function filters the input text and removes the original line formatting. Hence we can't print error and warning messages that specify the exact line number in which the error occurs. On the other hand, we can tolerate this disadvantage, because the input is not handwritten and a general error description is adequate.

The code generated conforms to ANSI C++ ([14]). This gives us the flexibility to compile SuchThat programs on any system with a decent C++ compiler.

2.1.4 Retargetability

An important property of our back end is retargetability. This means that the transition to another back end target language should be facilitated. We support this by two means. The 2-pass compiler design was already mentioned. We single out the target language dependent code in pass 1. We try to make this part as intelligent as possible. It should propagate extensive information about the source code to the code generator through the parse tree.

It should be left to the code generator in pass 2 to map the parse tree to the selected language. Whenever the compiler is retargeted, the programmer should ideally only change code of pass 2.

This is especially eased by a very accurate description of the parse tree, which is the interface between pass 1 and the code generator. We also provide functions to operate on the parse tree and its nodes.

2.1.5 Parsing Technique

Finally we have to choose the parsing technique. Two methods are widely accepted, *LL-* and *LR-parsing* ([36], p. 17, or [35], p. 156). The advantage of LR-parsing is that it covers a broader variety of grammar classes of the input language. It works heavily table oriented and is quite complex. The idea behind it is to read the text symbol by symbol (shift step) and try to replace the resulting sequence with a single nonterminal symbol, if it complies to a syntax rule (reduce step). LR-parsing is also called *bottom-up-parsing* and *shift-reduce-parsing*.

The second method, LL(1)-parsing, is the more intuitive one. The parameter 1 says that we decide by a lookahead of one symbol which recognition algorithm to apply. Every symbol that's decomposed by a recognition algorithm can contain nonterminals on its own. In such cases another recognition algorithm is called recursively. For this reason the method is also called *recursive descent parsing*. Every algorithm is represented by a function in the chosen programming language. The advantages of a recursive descent parser are that a transparent method for obtaining the pushdown automaton exists ([35], p. 123). Furthermore, it's easier written and maintained by hand.

Our input grammar is very simple due to the fact that it's functional, so a lookahead of one symbol is enough. We come to the conclusion that a recursive descent parser satisfies our needs.

2.1.6 Configurability

The title of this paper calls SCMTOCPP *configurable*. So far we only showed how configurability is achieved at the source code level. Configurability at compiler runtime is supported with a simple strategy. Whenever possible *and* reasonable, we store data in initialization files. They are processed by the compiler at startup and influence its runtime characteristics. Chapter 6 explains all available files.

2.2 Overall Structure

Now that we made all necessary design decisions about our compiler, we outline the functioning of our compiler. Figure 2.2 gives a nice overview.

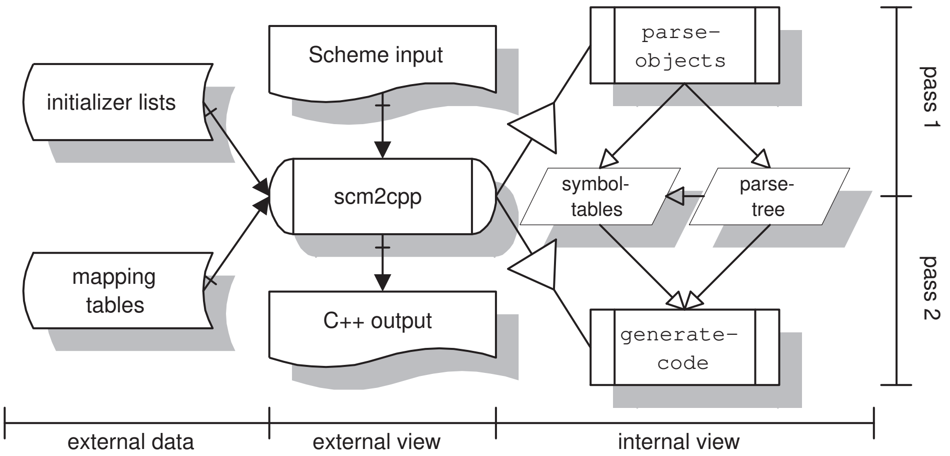

Figure 2.2: general overview

The external view treats our back end like a black box. The Scheme input is fed to SCMTOCPP, called `scm2cpp` on the command line. It writes a C^{++} file, if the compilation was successful, or displays some error messages otherwise.

The behavior of the compiler is influenced by external data, which is split up into two categories: initializer lists and mapping tables. The mapping tables set the default mappings of types and functions (see 1.2 for more details on mapping declarations). The initializer lists are copied to global variables and define the set of operators, types etc.

The internal view shows what happens inside the black box. SCMTOCPP first calls the function `parse-objects` with all objects in the input file. It builds the parse tree, which also includes the symbol tables. If pass 1 ran without errors, `generate-code` evaluates the parse tree and generates the code. In this process the symbol tables are extracted from the parse tree when needed.

Now we plunge right into the implementation details. The main file `scm2cpp.scm` holds the core of the compiler. At the beginning the global variables are defined, succeeded by the main procedure `scm2cpp`.

"src/scm2cpp.scm" 33 ≡

```
;;; scm2cpp.scm - MAIN FILE OF SCM2CPP
⟨global variables 34a, ... ⟩
⟨scm2cpp main procedure 37b⟩
⟨parse tree node identifier list 44a⟩
⟨code emitter list 82a⟩◇
```

The parse tree node identifier list is described in chapter 3.1, the code emitter list in section 4.1. The lists are appended at the end of the file, because they both use quasiquotations. The quasiquoted expressions must already be defined, or the Scheme system signals an error.

2.3 Global Variables

The use of global variables is obvious. They are available in every continuation of the program and set global behavior. Most compiler switches are mapped directly to global variables.

2.3.1 Input and Output

Input and output operates on files by default. The variable `enable-stdin` tells the compiler that it should read the Scheme input from `stdin`. The output goes to `stdout`, if the variable `enable-stout` is set to '`#t`'.

⟨global variables 34a⟩ ≡

```
;;; input and output
(define enable-stdin #f)
(define enable-stdout #f)
◇
```

Macro defined by scraps 34abc, 35abc, 36abc, 37a.
Macro referenced in scrap 33.

2.3.2 Code Layout

The two variables `icount` and `istep` are used for controlling the indentation of the generated C^{++} code. `istep` measures the width of indentation in characters for every parenthesis level. The default value is 2. `icount` represents the current offset to the start of the line. With this simple mechanism we enhance the readability of our generated source code. The flag `ansi-c++` switches between ANSI C^{++} mode and the classic mode. The differences are the names of the header files and the new namespace `std`, in which the standard library is defined.

⟨global variables 34b⟩ ≡

```
;;; global variables of scm2cpp
(define icount 0)
(define istep 2)
(define ansi-c++ #t)
◇
```

Macro defined by scraps 34abc, 35abc, 36abc, 37a.
Macro referenced in scrap 33.

2.3.3 Mapping Declarations

The variables `enable-tm` and `enable-fm` switch type and function mapping on and off. The use of this variables is heavily deprecated. The back end itself utilizes the mapping technique to perform some transformation that are crucial for legal output code. It's better to leave the default mappings untouched and disable the declaration statement `st-map` with the variable `enable-map-decl`. All variables are set to `#t` by default.

⟨global variables 34c⟩ ≡

```
(define enable-tm #t) ; type mapping on
(define enable-fm #t) ; function mapping on
(define enable-map-decl #t) ; declaration mapping with st-map on
◇
```

Macro defined by scraps 34abc, 35abc, 36abc, 37a.
Macro referenced in scrap 33.

2.3.4 Prefix Code

Programs often need some unique initializations. This is done at the beginning of the source with *prefix code*. Typical tasks in prefix code are declaration of used libraries, for example Java uses `import` to read class interfaces. ScmToCpp uses this facility to include header files of C++ libraries. We store the code in the variable `prefix-code`, which is a list of strings[1].

⟨global variables 35a⟩ ≡

```
(define prefix-code '())
◇
```

Macro defined by scraps 34abc, 35abc, 36abc, 37a.
Macro referenced in scrap 33.

2.3.5 Scope Counters

In `recurse-level` the depth of recursion is logged. It's more of academic interest, but comes in handy sometimes. `scope` is set to either `"global"` or `"algorithm"` and always indicates the SuchThat scope level of the compilation.

⟨global variables 35b⟩ ≡

```
(define recurse-level 0)
(define scope "global")
◇
```

Macro defined by scraps 34abc, 35abc, 36abc, 37a.
Macro referenced in scrap 33.

2.3.6 Error Counters and Flags

`error-cnt` and `warn-cnt` hold the number of errors, respectively warnings, that are produced during one compilation pass. Errors lead to an abortion of the compilation, warnings are ignored.

In order to alleviate the problem that we can't print the line of errors and warnings, we save the name of the current algorithm in `error-alg`. The variable is used to locate the error location roughly.

The last variable, `debug-level`, controls the verboseness of error and warning messages. Table 2.1 shows the meaning of `debug-level`'s values[2].

`debug-level`	message verboseness
0	no messages at all
1	brief, fundamental messages
2	detailed messages, with objects
3	gossipy

Table 2.1: meaning of `debug-level`

⟨global variables 35c⟩ ≡

```
(define error-cnt 0)
(define warn-cnt 0)
(define error-alg "")
(define debug-level 2)
◇
```

Macro defined by scraps 34abc, 35abc, 36abc, 37a.
Macro referenced in scrap 33.

[1] the functions `add-prefix-code!` and `write-prefix-code` operate on `prefix-code`

[2] the messages are produced with `error-mes`, `warn-mes` and `debug-mes`

2.3.7 Symbol Tables

A symbol table holds all identifiers of the currently active scope. A common strategy for symbol tables is to make a new copy whenever a new scope is entered. In this copy identifiers can be overwritten without causing harm.

The simple scope semantics of SuchThat presents us another method. We split the symbol table into global, input, output and local symbol tables. The latter ones are fixed for every algorithm. We exploit this property by initializing the algorithm specific symbol tables tables at the definition's entrance. The global symbol table is changed on the fly during code generation.

The structure of a symbol table is described in section 3.1.2.9.

⟨global variables 36a⟩ ≡

```
(define global-symtab '())
(define input-symtab '())
(define output-symtab '())
(define local-symtab '())
◇
```

Macro defined by scraps 34abc, 35abc, 36abc, 37a.
Macro referenced in scrap 33.

2.3.8 Case Sensitive **Scheme**

The boolean variable `enable-case-sensitive` is a flag for what its name suggests: it tells our back end if the input is case sensitive or not. Some Scheme implementations ([10]) are case sensitive. These can skip the case character detection in the identifier transformation (see section 5.1). The default value is `#f`, because that's what the standard says ([4], p. 32).

⟨global variables 36b⟩ ≡

```
(define enable-case-sensitive #f)
◇
```

Macro defined by scraps 34abc, 35abc, 36abc, 37a.
Macro referenced in scrap 33.

2.3.9 Identifier Mapping

Section 6.1 describes how to set up the identifier mapping table. The file `id-map.ini` holds the table. The runtime representation is kept in the global variable `id-map`.

`id-map-long` value	corresponding form
`#f`	short
`#t`	long

Table 2.2: usage of `id-map-form`

The boolean variable `id-map-long` selects, which form in the identifier mapping table is used for character mapping, as depicted in table 2.2. For the sake of readability we choose `#f` as default value.

⟨global variables 36c⟩ ≡

```
(define id-map '())
(define id-map-long #f)
◇
```

Macro defined by scraps 34abc, 35abc, 36abc, 37a.
Macro referenced in scrap 33.

2.3.10 Initializer Lists and Mapping Tables

These global variables are the runtime representations of the initializer lists and the default mapping tables (see chapter 6). They are read in at compiler startup. The root directory of the compiler is stored in `root-dir`. This directory is needed to locate the init files. It should end in a directory delimiter, mostly '\' or '/'.

⟨global variables 37a⟩ ≡

```
(define op-list '())
(define type-list '())
(define type-support '())
(define val-list '())
(define member-funcs '())
; mapping tables
(define func-map '())
(define type-map '())
; root directory
(define root-dir "")
◇
```

Macro defined by scraps 34abc, 35abc, 36abc, 37a.
Macro referenced in scrap 33.

2.4 The Main Procedure `scm2cpp`

Ok, now that we have defined the global variables, we can start the compilation process. The function `scm2cpp` progresses linearly along these steps:

1. evaluate the command line
2. read initialization files
3. parse the input file
4. generate code from the parse tree
5. write generated code to the output file

The structure of `scm2cpp` is an exact copy of the enumeration just made.

⟨scm2cpp main procedure 37b⟩ ≡

```
;;; main procedure scm2cpp
(define scm2cpp
  (lambda ()
    ⟨variable definitions of scm2cpp 37c⟩
    ⟨evaluate command line 38⟩
    ⟨read init files 39f⟩
    ⟨parse input file 40⟩
    ⟨generate code from parse tree 41a⟩
    ⟨write code file 41b⟩))
◇
```

Macro referenced in scrap 33.

The variable `scmfile` holds the name of the Scheme source code, `cppfile` that of the file to create. `parse-tree` holds the parse tree, which is the result of pass 1. It's the input parameter of pass 2, whose result is stored in `target-code`. `inport` and `outport` store the fileports, from which the source file is read, and to which the target code is written.

⟨variable definitions of scm2cpp 37c⟩ ≡

```
(define scmfile "") ; SCM source file
(define cppfile "") ; C++ target file
(define inport '()) ; source port
(define outport '()) ; target port
(define file-content '())
(define argl (cdr (argv))) ; strip command line of own name (scm2cpp)
(define parse-tree '()) ; return value of pass 1
(define target-code "") ; return value of pass 2
◇
```

Macro referenced in scrap 37b.

2.4.1 Evaluate the Command Line

scm2cpp gets its input parameters from the command line. Scheme has a special function argv[3], which returns a list of strings. They correspond with the command line arguments.

The command line switches are examined one by one. If the output file name is not given on the command line and the output goes into a file, we derive it from the input file name. This is done with derive-oname. The syntax of the command line call of scm2cpp is summarized in appendix A.

⟨evaluate command line 38⟩ ≡

```
(if (not (null? argl))
    (begin
      ; parse command line
      (do ((i 0 (+ i 1))) ((= i (length argl)))
        (cond ((equal? (list-ref argl i) "-tm") (set! enable-tm #f))
              ((equal? (list-ref argl i) "+tm") (set! enable-tm #t))
              ((equal? (list-ref argl i) "-fm") (set! enable-fm #f))
              ((equal? (list-ref argl i) "+fm") (set! enable-fm #t))
              ((equal? (list-ref argl i) "-md") (set! enable-map-decl #f))
              ((equal? (list-ref argl i) "+md") (set! enable-map-decl #t))
              ((equal? (list-ref argl i) "-li") (set! id-long-map #f))
              ((equal? (list-ref argl i) "+li") (set! id-long-map #t))
              ((equal? (list-ref argl i) "-cs") (set! enable-case-sensitive #f))
              ((equal? (list-ref argl i) "+cs") (set! enable-case-sensitive #t))
              ((equal? (list-ref argl i) "-so") (set! enable-stdout #f))
              ((equal? (list-ref argl i) "+so") (set! enable-stdout #t))
              ((equal? (list-ref argl i) "-si") (set! enable-stdin #f))
              ((equal? (list-ref argl i) "+si") (set! enable-stdin #t))
              ((equal? (list-ref argl i) "-ac") (set! ansi-c++ #f))
              ((equal? (list-ref argl i) "+ac") (set! ansi-c++ #t))
              ((list-memb? (list-ref argl i) '("-o" "-dl" "-st" "-rd"))
               (if (not (= i (- (length argl) 1)))
                   ⟨switches with argument 39a⟩
                   ⟨error: switch expects argument 39c⟩))
              (else (if (and (= i (- (length argl) 1)) (not enable-stdin))
                        (set! scmfile (list-ref argl (- (length argl) 1)))
                        ⟨warning: unknown command line switch 39e⟩) )))
      ; missing input file name
      (if (and (not enable-stdin) (equal? scmfile ""))
          ⟨error: no input file name specified 39b⟩)
      ; input from stdin -> output to file: stdin.cpp
      (if (and enable-stdin (not enable-stdout) (equal? cppfile ""))
          (set! cppfile "stdin.cpp"))
      ; derive output file name
      (if (and (not enable-stdin) (not enable-stdout))
          (set! cppfile (derive-oname scmfile))) )
    ⟨error: no input file name specified 39b⟩)
◇
```

[3] argv is an extension of standard Scheme, but most newer Scheme implementations support it, like Gambit-C[8]

Macro referenced in scrap 37b.

Some switches require an argument, like the switch `-o`, which sets the output file name. The loop counter `i` is increased manually to jump over the argument.

⟨switches with argument 39a⟩ ≡

```
(begin
  (cond ((equal? (list-ref argl i) "-o") ; output file
          (set! cppfile (list-ref argl (+ i 1))))
        ((equal? (list-ref argl i) "-rd") ; root directory
          (set! root-dir (list-ref argl (+ i 1))))
        ((equal? (list-ref argl i) "-st") ; indentation step width
          (set! istep (string->number (list-ref argl (+ i 1))))
          (if (or (eq? #f istep)
                  (and (number? istep) (< istep 0)))
              ⟨error: bad argument for switch 39d⟩) )
        ((equal? (list-ref argl i) "-dl") ; debug level
          (set! debug-level (string->number (list-ref argl (+ i 1))))
          (if (or (eq? #f debug-level)
                  (and (number? debug-level) (< debug-level 0)))
              ⟨error: bad argument for switch 39d⟩)) )
  (set! i (+ i 1)) )
◇
```

Macro referenced in scrap 38.

Error and Warning Messages

⟨error: no input file name specified 39b⟩ ≡

```
(begin (error-mes "no input file specified")
       (debug-mes 1 " Usage: scm2cpp [option ...] input-file" "" "") )◇
```

Macro referenced in scrap 38.

⟨error: switch expects argument 39c⟩ ≡

```
(begin (error-mes "command line switch expects argument")
       (debug-mes 1 " bad switch: " (list-ref argl i) "") ) ◇
```

Macro referenced in scrap 38.

⟨error: bad argument for switch 39d⟩ ≡

```
(begin (error-mes "bad argument for command line switch")
       (debug-mes 1 " switch: " (list-ref argl i) "")
       (debug-mes 1 " bad argument: " (list-ref argl (+ i 1)) "") )◇
```

Macro referenced in scrap 39a.

⟨warning: unknown command line switch 39e⟩ ≡

```
(begin (warn-mes "unknown command line switch")
       (debug-mes 1 " switch ignored: " (list-ref argl i) "") )◇
```

Macro referenced in scrap 38.

2.4.2 Read Init Files

All files described in chapter 6 are read into global variables at startup. We first check if they can be found, relative to the director `root-dir`.

⟨read init files 39f⟩ ≡

```
(if (and (file-exists? (string-append root-dir "init/id-map.ini"))
         (file-exists? (string-append root-dir "init/op-list.ini"))
         (file-exists? (string-append root-dir "init/type-list.ini"))
         (file-exists? (string-append root-dir "init/val-list.ini"))
         (file-exists? (string-append root-dir "init/type-support.ini"))
         (file-exists? (string-append root-dir "init/member-funcs.ini"))
         (file-exists? (string-append root-dir "init/type-map.ini"))
         (file-exists? (string-append root-dir "init/func-map.ini")) )
    (begin
      (set! id-map (read (open-input-file
                          (string-append root-dir "init/id-map.ini"))))
      (set! op-list (read (open-input-file
                           (string-append root-dir "init/op-list.ini"))))
      (set! type-list (read (open-input-file
                             (string-append root-dir "init/type-list.ini"))))
      (set! val-list (read (open-input-file
                            (string-append root-dir "init/val-list.ini"))))
      (set! type-support (read (open-input-file
                                (string-append root-dir "init/type-support.ini"))))
      (set! member-funcs (read (open-input-file
                                (string-append root-dir "init/member-funcs.ini"))))
      (set! type-map (read (open-input-file
                            (string-append root-dir "init/type-map.ini"))))
      (set! func-map (read (open-input-file
                            (string-append root-dir "init/func-map.ini")))) )
    (error-mes "can't find init files...\n") )
◇
```

Macro referenced in scraps 37b, 141.

2.4.3 Parse Input File

The procedure `parse-objects` expects a list of Scheme objects. Therefore we read the input file object by object and append each to the list `file-content`, until the `eof` object is encountered. `file-content` now contains all objects found in the input file. We pass it to `parse-objects`. This is the top level invocation of our recursive descent parser. The returned value constitutes the complete parse tree of the input file, encapsulated in a *parse tree node* (see section 3.1 for details). We print a short statistic of errors and warnings after completion of each pass.

⟨parse input file 40⟩ ≡

```
;;; - PASS 1 -
(if (= 0 error-cnt)
    (if enable-stdin
        (set! inport (current-input-port))
        (if (file-exists? scmfile)
            (set! inport (open-input-file scmfile))
            (error-mes "can't open input file...\n")) ))
(cond ((= 0 error-cnt)
       (set! warn-cnt 0)
       ;;; put all scm objects in the file into file-content
       (do ((obj (read inport) (read inport))) ((eof-object? obj))
         (set! file-content (append file-content (list obj))))
       ; top level invocation of parse-objects
       (set! parse-tree (parse-objects file-content))
       ; errors and warnings in PASS 1
       (debug-mes 2 "Errors while parsing: " error-cnt "")
       (debug-mes 3 "Warnings while parsing: " warn-cnt "") ))
◇
```

Macro referenced in scrap 37b.

2.4.4 Generate Code from Parse Tree

The parse tree node resides in `parse-tree`, and we pass it to `generate-code`, which returns the compiled code in a string. We add the header file `iostream` to the list of include files and set the namespace `std` of the C++ library as default, if ANSI C++ output is selected. This is done by changing the prefix code properly.

⟨generate code from parse tree 41a⟩ ≡

```
;;; - PASS 2 -
;(display parse-tree) (newline)
(cond ((= 0 error-cnt)
       (set! warn-cnt 0) ; clear warning counter
       (set! target-code (generate-code parse-tree))
       (if ansi-c++
           (begin (add-prefix-code! "#include <iostream>")
                  (add-prefix-code! "using namespace std;") )
           (add-prefix-code! "#include <iostream.h>"))
       (debug-mes 2 "Errors during code generation: " error-cnt "")
       (debug-mes 3 "Warnings during code generation: " warn-cnt "") ))
```
◇

Macro referenced in scrap 37b.

2.4.5 Write Output File

First, the derived output file name is used to open the output file. The returned fileport is assigned to the variable `file-port`. Then a short header is produced.

At the start of the file all needed initializer and preprocessing instructions are prepended, using the function `write-prefix-code`, to the the generated code from pass 2.

Last not least a final message is appended to the output file and the fileport is closed.

⟨write code file 41b⟩ ≡

```
(cond ((= 0 error-cnt)
       ; open outport for output and write header
       (if (eq? enable-stdout #f)
           (set! outport (open-output-file cppfile))
           (set! outport (current-output-port)) )
       (display "// File created by scm2cpp" outport) (newline outport)
       (display (string-append "// " scmfile " -> " cppfile) outport)
       (newline outport) (newline outport)
       ; write prefix code
       (display (write-prefix-code) outport)
       ; write code
       (display target-code outport)
       (newline outport)
       ; write trailer and close file
       (display "// --- EOF by scm2cpp ---" outport) (newline outport)
       (if (not enable-stdout) (close-output-port outport)) )
      (else (display "errors occurred, suspending compilation...\n")) )
```
◇

Macro referenced in scrap 37b.

2.5 Main Program

The main program calls the procedure `scm2cpp`. This procedure gets the input parameters from the command line via `argv`. The elapsed time of the compilation process is printed from debug level 2 on.

`"src/main.scm"` 41c ≡

```
(define elapsed-time 0)
(set! elapsed-time (runtime))
(debug-mes 2 "scm2cpp v1.0" "" "")
(scm2cpp)
(set! elapsed-time (- (runtime) elapsed-time))
(if (= 0 error-cnt) (debug-mes 2 "compilation took " elapsed-time " seconds"))
```
◇

Chapter 3

Pass 1: Parsing

The parsing is done in pass 1 of our compiler. It's initiated by `scm2cpp` with the top level invocation of `parse-objects`. The function is described, together with all the functions it calls recursively. The functions are classified in four conceptual categories: statement, expression, declaration and type parsing. These items make up the file `parse.scm`. The *parser list* is added at the end such that all the quasiquoted functions are already defined (see section 3.3).

`"src/parse.scm"` 43 ≡

⟨parse-objects procedure 53b⟩
⟨statement parsing 55, ... ⟩
⟨expression parsing 71b⟩
⟨declaration parsing 74b, ... ⟩
⟨type parsing 77c⟩
⟨parser list 54b⟩
◇

But first we look at the data structures on which all the functions operate.

3.1 The Parse Tree

The `parse-objects` function returns the parse tree representation of the Scheme input. In this section we will specify the parse tree's underlying data structures[1].

3.1.1 A Parse Tree Node

Every parse tree node is represented by a Scheme pair. The `car` field contains the *node identifier*, which is stored as a string. The node identifier indicates the type of the node. The `cdr` field of the node contains the *node value*. The data stored in the node value depends on the node's type. Figure 3.1 shows the skeleton of a parse tree node.

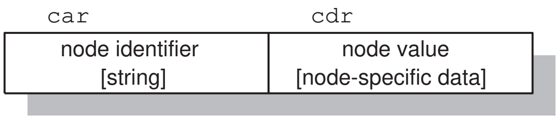

Figure 3.1: general structure of a parse tree node

[1] functions that operate on parse tree nodes are detailed in section 5.2

3.1.2 Description of the Parse Tree Nodes

We will now describe the node value's structure for every node type, as well as their node identifier string. The nodes are grouped into several categories. Some of them follow the subdivision of Scheme expressions in the standard's chapters four and five (see [4]), some are introduced to meet the special needs of SuchThat.

⟨parse tree node identifier list 44a⟩ ≡

```
;;; list of all node identifiers and their corresponding string
(define pt-nodeids `(⟨node identifiers: parse tree 44b⟩
                     ⟨node identifiers: identifiers 45a⟩
                     ⟨node identifiers: definitions 45b⟩
                     ⟨node identifiers: statements 46, ... ⟩
                     ⟨node identifiers: literals 49⟩
                     ⟨node identifiers: expressions 50a⟩
                     ⟨node identifiers: declarations 50b⟩
                     ⟨node identifiers: symbol table 52b⟩
                     ⟨node identifiers: types 52a⟩
                     ))
◇
```

Macro referenced in scrap 33.

The node identifiers are employed by the code generator. Depending on the node type, a certain function is called. We create node types by a symbolic name, because it would be cumbersome to use the identifier string. In order to avoid inconsistencies, the corresponding elements reside in the list `pt-nodeids`. The first element is the node type's symbolic name, followed by its identifier string. The last element is a quasiquoted Scheme function, which generates code in pass 2 for this node type.

3.1.2.1 Parse Tree Recursion

We write a recursive descent parser, therefore we need a parse tree node to express the recursion. The node is generated by the `parse-objects` function. It simply puts all parsed objects in a n-element list, which is copied into the node's value field.

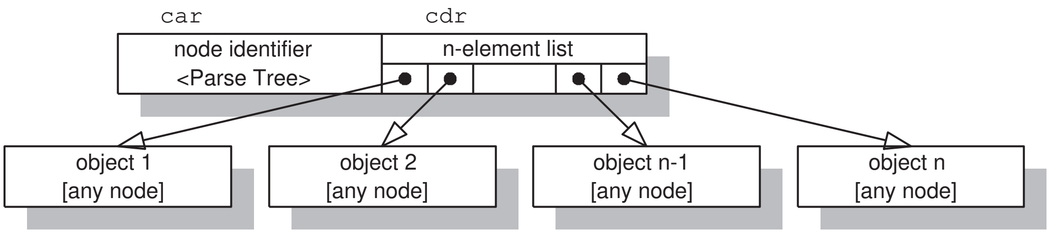

Figure 3.2: structure of a parse tree recursion node

⟨node identifiers: parse tree 44b⟩ ≡

```
; parse tree
(pt-id "PTN Parse Tree" ,emit-pt)◇
```

Macro referenced in scrap 44a.

3.1.2.2 Identifiers

We discussed identifiers in section 1.1. The function `id-transform` converts the original identifier to one appropriate for the back end target language. This string is the first element of the node value. The second element is the original symbol, which was passed to `id-transform`. Figure 3.3 shows such a parse tree node. We divide identifiers into variable, function, operator and type identifiers.

⟨node identifiers: identifiers 45a⟩ ≡

```
; identifiers
(vname-id "PTN Identifier: Variable"  ,emit-vname)
(fname-id "PTN Identifier: Function" ,emit-fname)
(opname-id "PTN Identifier: Operator" ,emit-opname)
(typename-id "PTN Identifier: Type" ,emit-typename)◇
```

Macro referenced in scrap 44a.

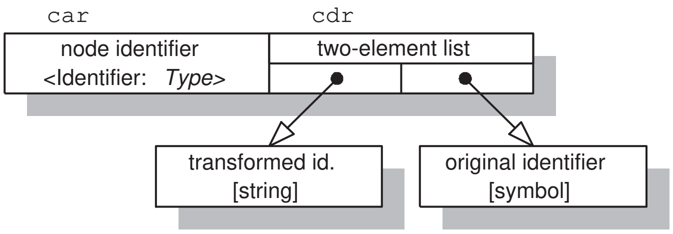

Figure 3.3: structure of a parse tree node for identifiers

3.1.2.3 Definitions

Definitions can be split into two groups:

1. variable definitions
2. function and operator definitions

Variable definitions collapse into assignments, because all variables are already typed by the front end. It's sufficient to store the variable identifier's name and the defining Scheme expression. Therefore the node value consists of a two element list. The first element is the identifier's parse tree node, and the second one is the node for the Scheme expression that defines the variable's initial value, as shown in figure 3.4.

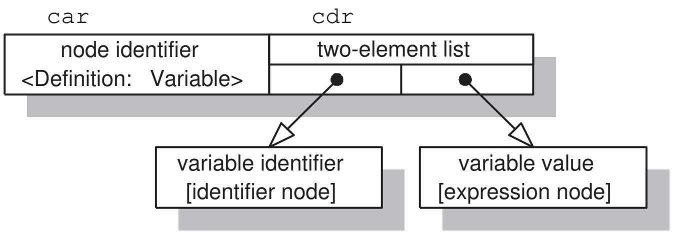

Figure 3.4: structure of a parse tree node for variable definitions

Function definitions are more complex, we need a six-element list as node value to store all information. The first element is the function's name, represented by an identifier node. We discriminate between "standard" and operator functions. In the second through fourth field the declarations of input, output and local variables are stored in a symbol table. A declaration associates each identifier with its type, which is the work of `scm2scm`. The fifth element holds the function parameters in a list of variable identifier nodes. Finally, the sixth element gets the body of the function definition. This is usually a `let` statement which wraps the real function, but we don't force the usage of this construct.

⟨node identifiers: definitions 45b⟩ ≡

```
; definitions
(var-id "PTN Definition: Variable" ,emit-var)
(func-id "PTN Definition: Function" ,emit-func)
(op-id "PTN Definition: Operator" ,emit-op)◇
```

Macro referenced in scrap 44a.

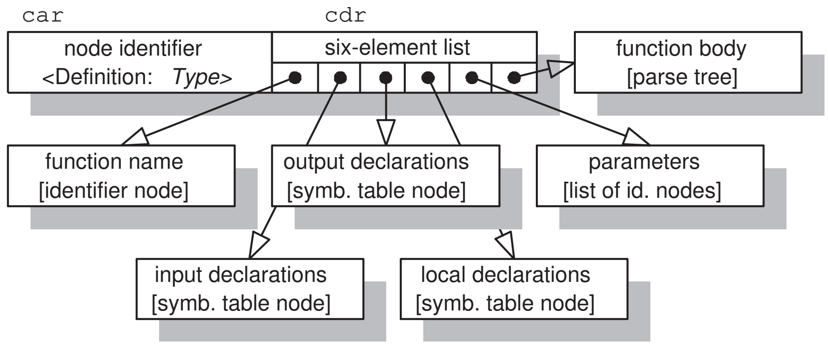

Figure 3.5: structure of a parse tree node for function definitions

3.1.2.4 Statements

Every statement has its own node identifier. Their node values differ profoundly, reflecting the statements' semantics. We treat *real* Scheme statements exactly the same way as our own SuchThat macros (see appendix C) that were introduced to simulate imperative SuchThat statements.

These are the Scheme standard statements that found their way into our back end:

⟨node identifiers: statements 46⟩ ≡

```
;;; R4RS statements
(set!-id "PTN Statement: set!" ,emit-set!)
(let-id "PTN Statement: let" ,emit-let)
(begin-id "PTN Statement: begin" ,emit-begin)
(if-id "PTN Statement: if" ,emit-if)
(case-id "PTN Statement: case" ,emit-case)
(cond-id "PTN Statement: cond" ,emit-cond)
;(do-id "PTN Statement: do" ,emit-do)◇
```

Macro defined by scraps 46, 48.
Macro referenced in scrap 44a.

The `set!` statement looks just like a variable definition. We store the target identifier's name as well as the expression it's set to. Figure 3.6 depicts the node structure.

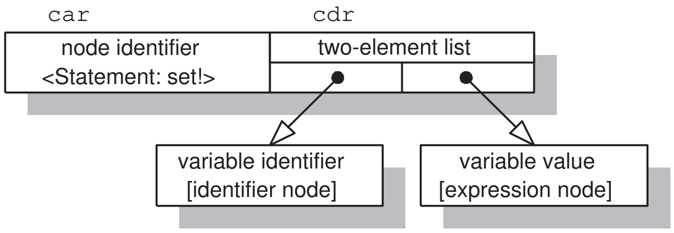

Figure 3.6: structure of a parse tree node for the `set!` statement

The `let` statement has a special purpose in our intermediate language. It's used to declare and initialize local variables of algorithm scope. The node value is a two-element list. The first element is a n-element list, which holds `set!` nodes for all local variables and the value they are initialized to. The body of the `let` statement is stored in the second element. Figure 3.7 sheds some light upon the node structure.

The `if` statement consists of the test expression, the consequent that is executed if the test expression evaluates to `true`, and the alternate that is chosen if the result is `false`. We always store a three-element list in the node value, even if the alternate is not present. The first element is an expression node for the `if` condition, the second and third element are parse trees for the consequent and alternate bodies.

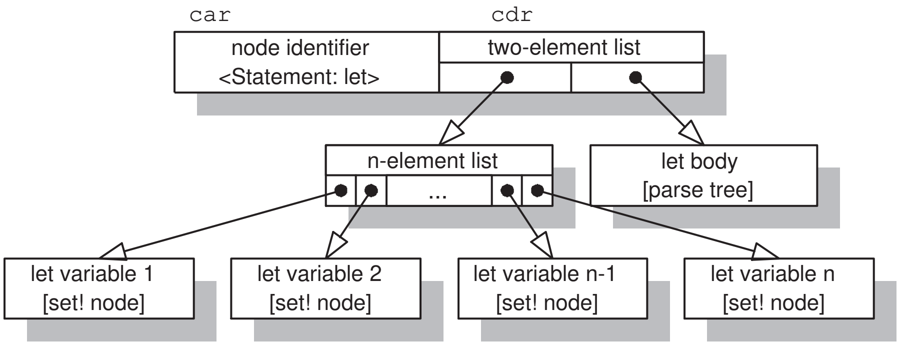

Figure 3.7: structure of a parse tree node for the `let` statement

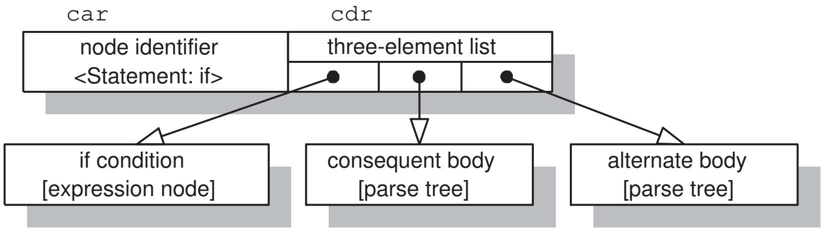

Figure 3.8: structure of a parse tree node for the `if` statement

Another conditional is the `case` statement. A key is presented and then a list of special clauses is examined in order. A clause is composed of a list of datums, which are compared to the key, and a sequence of expressions which are evaluated in order, if a datum in the preceding list matched the key. The clauses are stored in pairs. The `car` field is a list of the key expressions, the `cdr` field holds the body of the alternate. The datum of the last list may be the special keyword `else`, which preludes a default branch. We just store the parse tree of the default branch in the node value's last list element. If no default branch is present, an empty list is stored instead. This structure is mimicked in the parse tree node, summed up by figure 3.9.

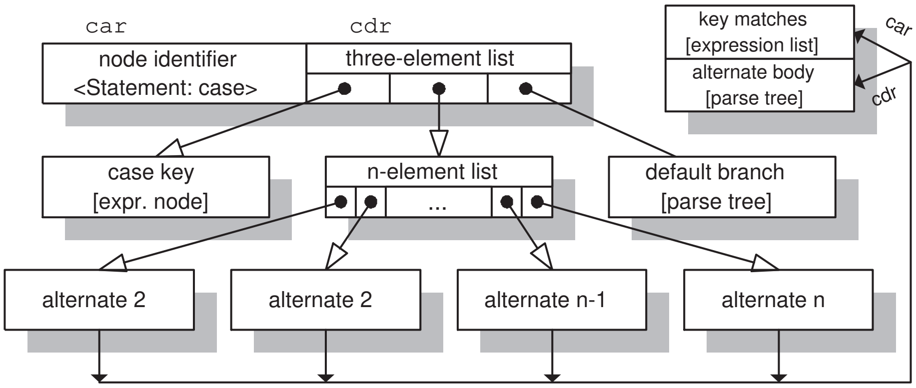

Figure 3.9: structure of a parse tree node for the `case` statement

The behavior of the `cond` statement is almost the same as that of the `case` statement. The difference is that the expressions of a clause are evaluated, if its test is true. No matching with a key is done, therefore the key expression misses entirely. The slightly different structure is recapitulated in figure 3.10.

The `begin` statement simply stores its node identifier in the node's `car` field and a parse tree of the expressions from its body in the `cdr` field.

Now we describe the node structure of the SuchThat macros, they complement the regular

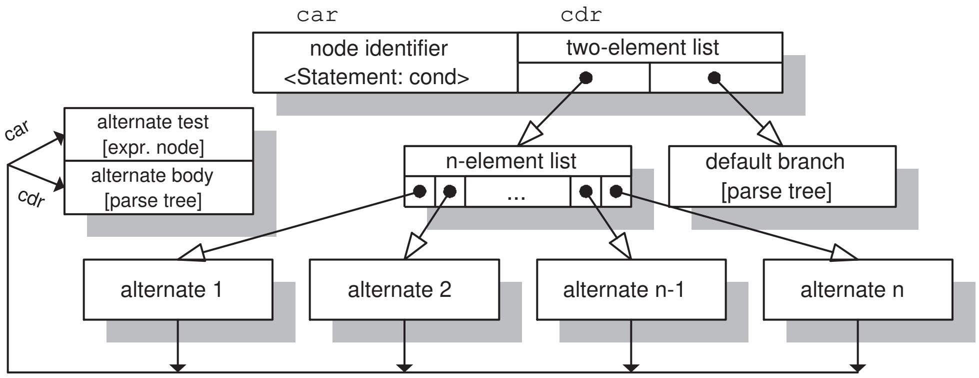

Figure 3.10: structure of a parse tree node for the `cond` statement

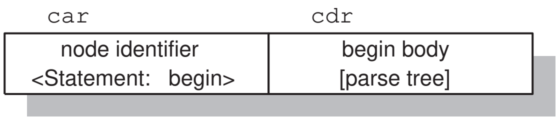

Figure 3.11: structure of a parse tree node for the `begin` statement

Scheme statements.

⟨node identifiers: statements 48⟩ ≡

```
;;; SuchThat macros
(return-id "PTN Statement: return" ,emit-return)
(while-id "PTN Statement: while" ,emit-while)
(repeat-id "PTN Statement: repeat" ,emit-repeat)
(for1-id "PTN Statement: for1" ,emit-for1)
(for-id "PTN Statement: for" ,emit-for)◇
```

Macro defined by scraps 46, 48.
Macro referenced in scrap 44a.

The `while` statement's node value consists of a two-element list: the conditional expression and the `while` body, as shown in figure 3.12.

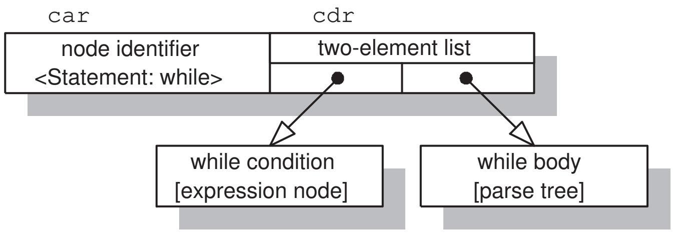

Figure 3.12: structure of a parse tree node for the `while` macro

The `repeat` node looks almost the same as the `while` node, but the conditional expression and the body have changed positions. The first element is the `repeat` body, the second element is the conditional expression. Figure 3.13 illustrates the node structure.

Both `for1` and `for` are very similar. The only difference is that the `for1` macro assumes the second value to be one greater than the initial variable value. This leads to a step width of +1. The more general `for` statement computes the step width by subtracting the initial value from the second value.

We get the following node value structure. It's a four-element list for the the `for1` statement and a five-element list for the `for` statement. The first two elements are the variable identifier and

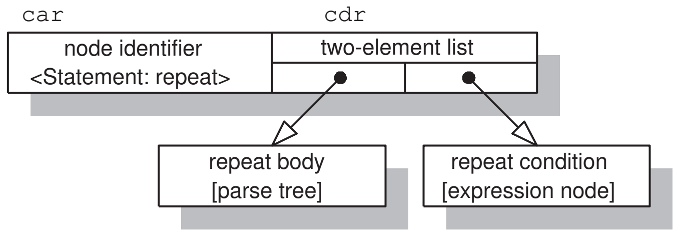

Figure 3.13: structure of a parse tree node for the `repeat` macro

its initial value. The `for` node gets an additional entry for the variable's second value. Now the final variable value is stored in an expression node. The list is finished by the parse tree of the statement's body. The structure is summarized for both statements in figure 3.14.

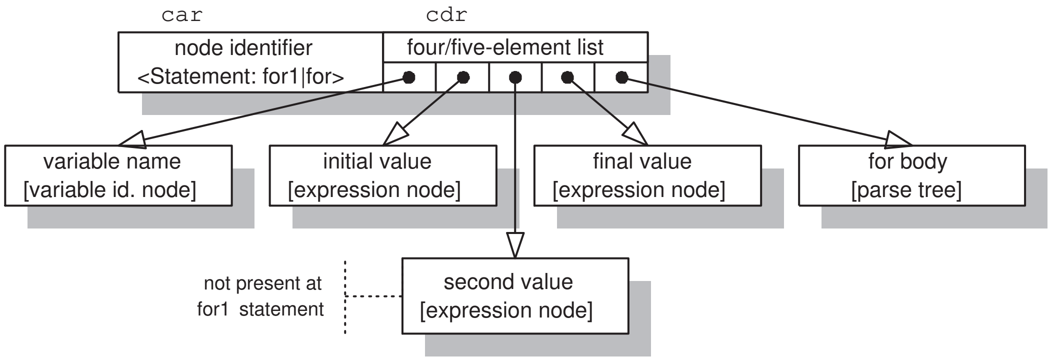

Figure 3.14: structure of a parse tree node for the `for1/for` macros

The `return` macro is very simple, the node value is the node for the Scheme expression that should be returned.

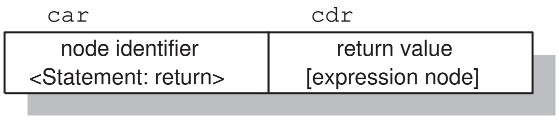

Figure 3.15: structure of a parse tree node for the `return` macro

3.1.2.5 Literals

Nodes for literals are very simple. The node value consists of the literal's external Scheme representation stored in a string. Figure 3.16 shows a parse tree node for literals.

For most numerical types and strings the code generator just copies this string to the output. But some types have to be fine tuned to fit the syntax of the back end language. The back end target language specific code has to deal with these exceptions.

We decided to provide a separate node identifier for every literal type, in order to take work out of the code generator's hands.

⟨node identifiers: literals 49⟩ ≡

```
;;; literals
(bool-id "PTN Literal: Boolean" ,emit-bool)
(char-id "PTN Literal: Character" ,emit-char)
(string-id "PTN Literal: String" ,emit-string)
(int-id "PTN Literal: Integer" ,emit-int)
```

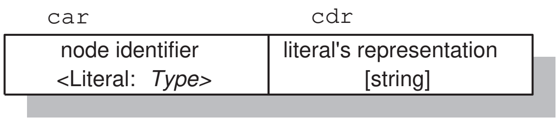

Figure 3.16: structure of a parse tree node for literals

```
(rational-id "PTN Literal: Rational" ,emit-rational)
(real-id "PTN Literal: Real" ,emit-real)
(complex-id "PTN Literal: Complex" ,emit-complex)◇
```

Macro referenced in scrap 44a.

3.1.2.6 Expressions

⟨node identifiers: expressions 50a⟩ ≡

```
;;; expressions
(expr-id "PTN Expression" ,emit-expr)◇
```

Macro referenced in scrap 44a.

Now we have a look at the representation of expressions. The Scheme report ([4], p. 7) gives a concise summary of the expression flavors. We represent a subset of Scheme expressions:

1. literals
2. variables
3. procedure/function calls

Literals and variables are flat expressions, whereas function calls, which also include expressions connected with operators, are nested expressions. This fact is easily worked into our node structure. The first node value is either a literal or an identifier. If it's a nested expression, the second value is a list of the operator's or function's operands, which are expressions themselves. Flat expressions just put an empty list there.

Nested expressions are introduced by an operator or function name identifier, flat ones by a variable name identifier or a literal.

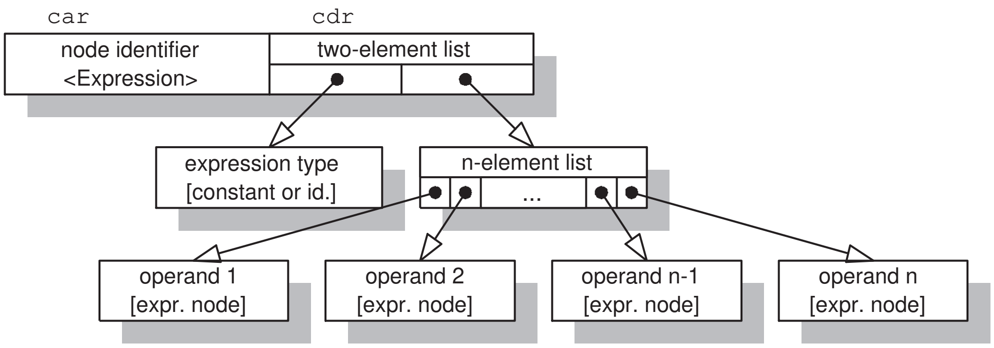

Figure 3.17: structure of a parse tree node for an expression

3.1.2.7 Declarations

⟨node identifiers: declarations 50b⟩ ≡

```
;;; declarations
(globaldecl-id "PTN Declaration: Global" ,update-global)
(funcmap-id "PTN Declaration: Function Map" ,update-funcmap)
(typemap-id "PTN Declaration: Type Map" ,update-typemap)
◇
```

Macro referenced in scrap 44a.

Global declarations introduce new variables to the global namespace. All the variables with their types are stored in a symbol table node, which is the node value of a global declaration, show in figure 3.18.

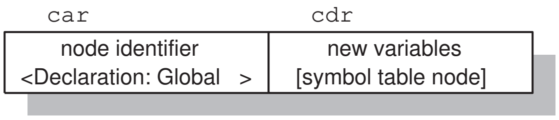

Figure 3.18: structure of a parse tree node for a global declaration

Mapping declarations are split in two groups: type and function mappings. They both have a two-element list as node value. The list of function mappings is made of two identifier nodes. Type mappings want a type node as the second list element. Both are shown in figure 3.19. The dotted square shows the difference between a type mapping node and a function mapping node (dotted: type map).

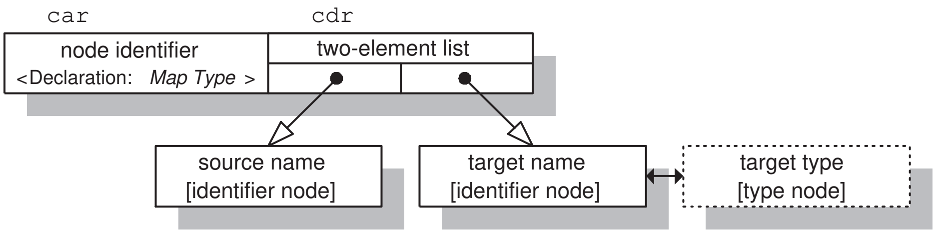

Figure 3.19: structure of a parse tree node for mapping declarations

Mapping can be turned off entirely with a command line switch. In this case, the declaration's node value is the empty list. The manipulator function for mappings must pay attention to this peculiarity.

3.1.2.8 Types

Types are also encoded in parse tree nodes. The first element is the type name and the second one is a n-element list. The list is used to store the type parameters of the encoded type. Lists of fundamental types collapse into empty lists.

In the last field of the node value a string is stored that keeps special information about the type. Table 3.1 shows the encoding. The unused characters are reserved for future extensions.

character	information
`f`	fundamental type
`g`	generic type
`a`	array type
`c`	container type
`r`	call-by-reference
`v`	call-by-value

Table 3.1: type information encoding in string

Arrays get a special treatment, because the array's index range has to be handled also. The dotted parts of figure 3.20 show the data structure for arrays which substitutes the normal one. Instead of the type name, we store the array's type in the first slot of the node value list. The

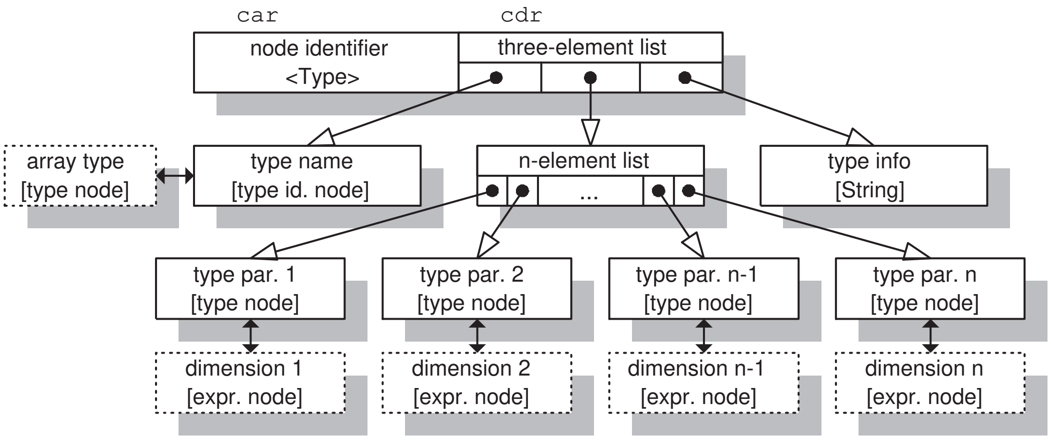

Figure 3.20: structure of a parse tree node for types

n-element list in the second slot is used to hold the expressions for the index ranges. If the index range is given by lower and upper bounds, even pairs of expressions are needed[2].

⟨node identifiers: types 52a⟩ ≡
```
; types
(type-id "PTN Type" ,emit-type)◇
```
Macro referenced in scrap 44a.

3.1.2.9 Symbol Tables

The different kinds of symbol tables are stored in the same parse tree node. They can only appear in the function definition node and are therefore distinguished by their position in this node. The node value of a symbol table is a list which contains two-element lists with these members:

1. the variable's name (original symbol and transformed identifier string)
2. the variable's type

Figure 3.21 shows a symbol table node and the symbol table entry's structure.

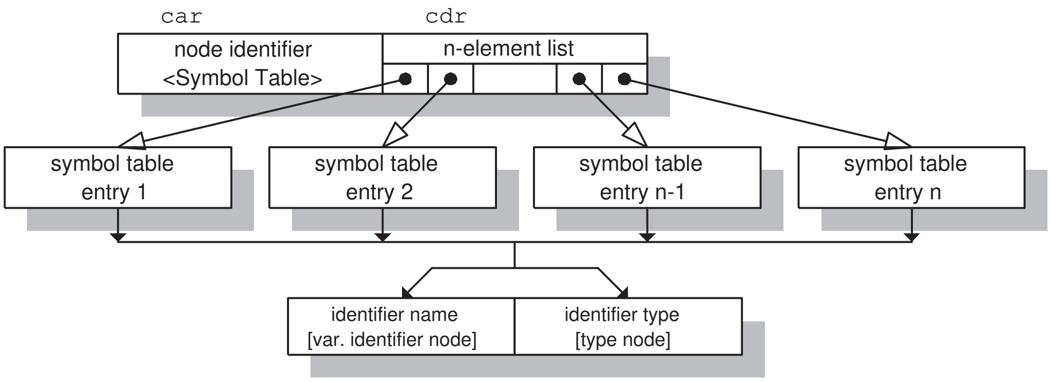

Figure 3.21: structure of a parse tree node for symbol tables

⟨node identifiers: symbol table 52b⟩ ≡
```
; symbol table
(symtab-id "PTN Symbol Table" ,emit-symtab)
◇
```
Macro referenced in scrap 44a.

[2]not yet implemented

3.2 The `parse-objects` Function

As `scm2cpp` wraps the whole program execution, `parse-objects` is a wrapper for the parsing process. It expects a single parameter *objs*, which should be a list of Scheme objects, and returns a parse tree node.

> **Formal description:** (`parse-objects` *objs*)
>
> *objs* – a list of Scheme objects that comply to the specification of the input grammar, given in chapter 1
>
> return value: a parse tree node whose `car` field is the node identifier for a parse tree, and whose `cdr` field is a list of parse trees, one for every Scheme object in *objs*

At the beginning of the function the needed variables are defined. The most important is `pt-node`, in which the return value will be stored. It is a parse tree recursion node itself.

⟨parse-objects variable definitions 53a⟩ ≡

```
(define pt-node (make-ptn 'pt-id)) ; holds return value
(define obj '()) ; holds object to compile
(define func #f) ; holds pair with Scheme keyword and parser function
◇
```

Macro referenced in scrap 53b.

We iterate over all objects in the list and the adequate recognition algorithm is invoked. The first command increases `recurse-level` by 1, and the last decreases it by 1.

⟨parse-objects procedure 53b⟩ ≡

```
;;; parses list of Scheme objects
(define parse-objects
  (lambda (objs)
    ⟨parse-objects variable definitions 53a⟩
    (if (list? objs)
        (begin
          (set! recurse-level (+ recurse-level 1))
          (debug-mes 3 "recurse level: " recurse-level "")
          ; iterate over all objects in list
          (do ( (i 0 (+ i 1)) ) ((= i (length objs)))
            (set! obj (list-ref objs i))
            ⟨invoke recognition algorithm 53c⟩)
          (set! recurse-level (- recurse-level 1))
          pt-node)
        ⟨error: wrong parse-objects argument 54a⟩) ))
◇
```

Macro referenced in scrap 43.

The `car` field of every object in *objs* is compared against the `car` fields of the parser list. In this list all known statements and their corresponding recognition algorithms (represented by their procedure names) are stored as pairs.

⟨invoke recognition algorithm 53c⟩ ≡

```
(if (pair? obj) (set! func (assoc (car obj) parser-list)) (set! func #f))
(cond ((pair? func) ; call function in parser-list
       (add-ptn-val! pt-node ((cdr func) (cdr obj))) )
      (else ; otherwise parse Scheme expression
       (add-ptn-val! pt-node (parse-expr obj)) ))
◇
```

Macro referenced in scrap 53b.

The comparison is performed by the Scheme standard procedure `assoc`, which operates on association lists (see [4], p. 17 for more details). Whenever a match is found, the parser function in the parser list is called. These functions will be described in the following sections. The returned parse tree node is appended to the current `cdr` field of the parse tree recursion node `pt-node`.

If no matching `car` field is found, `assoc` returns `#f`. In this case the expression resolver `parse-expr` is started. The technique of first testing all known statements and otherwise retreating to expression parsing is often used in hand crafted recursive descent parsers ([9], p. 220). See section 3.5 for a discussion of expression parsing.

The test for SuchThat declarations is done in the same way as for statements. The difference between statements and declarations is that the latter won't generate any code in pass 2. They usually change global variables, for example the `update-global` function modifies the global symbol table.

Error and Warning Messages We issue an error message if the argument to `parse-objects` is not a list:

⟨error: wrong parse-objects argument 54a⟩ ≡

```
(begin (error-mes "wrong argument type for parse-objects, list expected")
       (debug-mes 2 " wrong argument: " objs "") )◇
```

Macro referenced in scrap 53b.

3.3 The Parser List

In the preceding section we mentioned the parser list. The list contains only pairs, a so-called *alist*. In this alist we keep the recognition algorithm of every keyword that is part of our Scheme interface. The parser list contains parser functions for statements as well as for SuchThat declarations.

The `car` field of a pair holds the keyword's symbol. In the `cdr` field we find the quasiquoted recognition algorithm, a so-called *parser function*. The quasiquote is necessary, because the `cdr` field is used as function caller. Or to be more precise, we store the function's lambda expression by using the quasiqote syntax.

⟨parser list 54b⟩ ≡

```
;;; list with all legal statements, used by parse-objects
(define parser-list `(⟨list of statements 54c⟩
                      ⟨list of declarations 74a⟩
                      ))
◇
```

Macro referenced in scrap 43.

3.3.1 A Parser Function

The skeleton of a parser function is always the same. It has one input parameter, which is the `cdr` field of the Scheme expression for which the function is written. The `car` field is, of course, the name of the construct to parse. This is the same symbol that is used as `car` field in the parser list.

A parser function must return a parse tree node. Therefore most parser functions first define a node and initalize its identifier field with `make-ptn`. Then they fill the value field according to the node's specification. If a new parser function should be added, heed the given guidelines.

3.4 Statement Parsing

The parser functions for statements make up the majority of pass 1:

⟨list of statements 54c⟩ ≡

```
(define . ,parse-define)
(let . ,parse-let)
(set! . ,parse-set!)
(begin . ,parse-begin)
(if . ,parse-if)
(case . ,parse-case)
(cond . ,parse-cond)
; SuchThat macros
(while . ,parse-while)
(repeat . ,parse-repeat)
(for . ,parse-for)
(for1 . ,parse-for1)
(return . ,parse-return)
◇
```

Macro referenced in scrap 54b.

Those of you familiar with Scheme might raise an eyebrow when they study the statements. There sure is no such thing like a `return` or `while` statement in Scheme.

As SuchThat inherited its statement part from Aldes, and because Aldes is an imperative language, S. Schupp and R. Loos decided to introduce some Scheme macros to simulate the Aldes statements in the intermediate language. Several reasons make this step plausible:

- ScmToCpp can recognize the original SuchThat program much easier
- the SuchThat to Scheme transformation is simplified
- easy mapping of the Scheme to the C++ language (or any other imperative language)

The SuchThat macros can be found in the file `stmac.scm`. The documented source for the macros is in appendix C.

3.4.1 The `parse-define` Function

⟨statement parsing 55⟩ ≡
⟨parse-define function 56a⟩◇

Macro defined by scraps 55, 60e, 62b, 63ad, 64b, 66a, 67ad, 68c, 69d, 70d.
Macro referenced in scrap 43.

Formal description: (`parse-define` *obj*)

obj – a Scheme object complying to one of the following three forms:

1. variable definition
   (*id scm-expr*)
2. function definition, type a
   (*fname* `(lambda (`*par-list*`)` *st-decls* `(call/cc (lambda (return)` *body*`))))`
3. function definition, type b
   `((`*fname par-list*`)` `(`*st-decls* `(call/cc (lambda (return)` *body*`))))`

fname – either an identifier or a 2-element list of the form (*function-type id*)
par-list – a sequence of identifiers
st-decls – a sequence of SuchThat declarations
<u>return value:</u> a parse tree node for the define statement (see figures 3.4 and 3.5)

The `parse-define` function is perhaps the most important one. We decide at the start, which of the possible three cases we have to deal with. The function definitions are distinguished from the variable definitions by either the first element in *obj* being a list or the second element starting with the keyword `lambda`. One of these two conditions identifies a function definition.

⟨parse-define function 56a⟩ ≡

```
;;; parse define statement
(define parse-define
  (lambda (obj)
    ; use variable definition as default
    (define pt-node (make-ptn 'var-id)) ; return value
    ; check syntax
    (cond ((and (list? obj)
                (>= (length obj) 2))
           ; decide which definition type
           (if (or (list? (car obj))
                   (and (list? (cadr obj))
                        (eq? (car (cadr obj)) 'lambda)))
               ⟨function definition parsing 56c⟩
               ⟨variable definition parsing 56b⟩
            ))
          (else ⟨error: wrong define argument 59d⟩)) ))
◇
```

Macro referenced in scrap 55.

Parsing a variable definition works analog to parsing the `set!` statement. The only difference to a `set!` node is the node identifier. The defined identifier is inserted into the node value, right before the defining expression.

⟨variable definition parsing 56b⟩ ≡

```
(begin
  ; first element: var. id
  (add-ptn-val! pt-node (get-ptn (make-ptn 'vname-id)
                                 `(,(id-transform (car obj)) ,(car obj))))
  ; second element: expression
  (add-ptn-val! pt-node (parse-expr (cadr obj)))
  pt-node)
◇
```

Macro referenced in scrap 56a.

We get started with function definition parsing by binding some local variables to initially empty values and setting the global variable `scope` to `"algorithm"`. It's set back to `"global"` at the end of the `let` block. SuchThat supports only two scope levels, algorithm and file (global) scope ([27], p. 46). This is checked before any parsing starts.

Then we honor the two syntactical possibilities in Scheme for function definitions by getting the needed data out of the slightly different list structures such that further computation can be uniform. Having accomplished this, the first thing we do is obtaining the function type. Thereafter the declarations are processed. Finally we store all the data in the node.

⟨function definition parsing 56c⟩ ≡

```
(let ((id "") (body '()) (par-list '()) (decl-list '()) )
  (if (equal? scope "algorithm")
      ⟨error: algorithm scope clash 59c⟩
      (begin
        (set! scope "algorithm")
        (set! error-alg "")
        (set! input-symtab '())
        (set! output-symtab '())
        (set! local-symtab '())
        ⟨unify different function syntax 57a⟩
        ⟨obtain function type 58a⟩
        ⟨process declarations 59a⟩
        ; save symbol tables
        (add-ptn-val! pt-node (get-ptn (make-ptn 'symtab-id) input-symtab))
        (add-ptn-val! pt-node (get-ptn (make-ptn 'symtab-id) output-symtab))
```

```
        (add-ptn-val! pt-node (get-ptn (make-ptn 'symtab-id) local-symtab))
        ; parameter list
        (do ((i 0 (+ i 1)) (plist '()))
            ((= i (length par-list)) (add-ptn-val! pt-node plist))
          (set! plist (append plist (list ⟨id-node of parameter 59b⟩))) )
        ; algorithm body
        (add-ptn-val! pt-node (parse-objects body)) ))
  (set! scope "global")
  pt-node )
◇
```

Macro referenced in scrap 56a.

Before we can unify the two function types, we have to check their syntax thoroughly. This is done by examining the object structure and testing for keywords like `lambda` and `return`. Then we access *obj* very selectively and retrieve the parts we need.

⟨unify different function syntax 57a⟩ ≡

```
(if (and (list? (cadr obj))
         (eq? (car (cadr obj)) 'lambda))
    ; type a function
    (if ; syntax check
     (and (>= (length (cadr obj)) 3) ; all mandatory elements present?
          (eq? (car (cadr obj)) 'lambda)
          (list? (cadr (cadr obj))) ; parameter list needed
          ⟨check call/cc syntax 57b⟩)
     (begin
       ;(display "type a") (newline)
       (set! id (car obj))
       (set! par-list (cadr (cadr obj)))
       (set! body
             (cdr (cdr (cadr
                        (list-ref (cadr obj) (- (length (cadr obj)) 1))))))
       (set! decl-list (list-tail (reverse (list-tail (cadr obj) 2)) 1)) )
     ⟨error: wrong define syntax 59e⟩)
    ; type b function
    (begin
      (set! obj (list (car obj) (cdr obj)))
      (if ; syntax check
       (and (>= (length (car obj)) 1) ; at least function name exists
            (list? (cadr obj))
            (>= (length (cadr obj)) 1) ; at least body exists
            ⟨check call/cc syntax 57b⟩)
       (begin
         ;(display "type b") (newline)
         (set! id (car (car obj)))
         (set! par-list (cdr (car obj)))
         (set! body
               (cdr (cdr (cadr
                          (list-ref (cadr obj) (- (length (cadr obj)) 1))))))
         (set! decl-list (list-tail (reverse (cadr obj)) 1)) )
       ⟨error: wrong define syntax 59e⟩) ) )
◇
```

Macro referenced in scrap 56c.

The position of the `call/cc` part is exactly the same for both function types, therefore we can save some code duplication. All function definitions must comply to the `call/cc` syntax, even if they don't use the `return` statement.

⟨check call/cc syntax 57b⟩ ≡

```
(list? (list-ref (cadr obj) (- (length (cadr obj)) 1)))
(= (length (list-ref (cadr obj) (- (length (cadr obj)) 1))) 2)
(eq? (car (list-ref (cadr obj) (- (length (cadr obj)) 1)))
     'call-with-current-continuation)
(list? (cadr (list-ref (cadr obj) (- (length (cadr obj)) 1))))
(>= (length (cadr (list-ref (cadr obj) (- (length (cadr obj)) 1)))) 2)
(eq? (car (cadr (list-ref (cadr obj) (- (length (cadr obj)) 1)))) 'lambda)
(equal? (cadr (cadr (list-ref (cadr obj) (- (length (cadr obj)) 1)))) '(return))
◇
```

Macro referenced in scrap 57a.

We look at the identifier to obtain the function type. A Scheme symbol stands for a standard function. Other types are declared by using a two-element list instead of an identifier. This list consists of a *type-keyword* and a Scheme symbol. Right now, only two type-keywords are allowed:

1. `standard`
2. `operator`

⟨obtain function type 58a⟩ ≡

```
(if (symbol? id) ; standard function
    ⟨standard function identifier 58b⟩
    (if (and (list? id)
             (= (length id) 2)
             (symbol? (cadr id))
             (symbol? (car id)))
        ; examine id
        (cond ((eq? (car id) 'standard) ; standard function
               (begin (set! id (cadr id))
                      ⟨standard function identifier 58b⟩ ))
              ((eq? (car id) 'operator) ; operator function
               (begin (set! id (cadr id))
                      ⟨operator function identifier 58c⟩ )))
        ⟨error: wrong function identifier 60a⟩))
◇
```

Macro referenced in scrap 56c.

The standard function identifier is treated like an ordinary function identifier and thus subject to `id-transform`. We save the algorithm name for error localization.

⟨standard function identifier 58b⟩ ≡

```
(begin
  ; node identifier
  (set! pt-node (make-ptn 'func-id))
  ; first element: id
  (set! error-alg (id-transform id))
  (add-ptn-val! pt-node (get-ptn (make-ptn 'fname-id) `(,error-alg ,id))) )
◇
```

Macro referenced in scrap 58a.

For a discussion of operators see section 1.4. As mentioned there, operators stand somehow outside the regular identifier syntax. We support this peculiarity with the operator list, see 6.5. If an operator appears in this list, it won't be transformed.

⟨operator function identifier 58c⟩ ≡

```
(begin
  ; node identifier
  (set! pt-node (make-ptn 'op-id))
  ; first element: id
  (set! error-alg (if (list-memb? id op-list)
                      (symbol->string id) (id-transform id)))
  (add-ptn-val! pt-node (get-ptn (make-ptn 'opname-id) `(,error-alg ,id))) )
◇
```

Macro referenced in scrap 58a.

Now that all parts of the function are nicely separated, we have a look at the declarations. We iterate over the whole declaration list and examine each entry. According to their declaration tag they are inserted into the input, output or local symbol table. Ill-formed or unknown declarations are signalled.

⟨process declarations 59a⟩ ≡
```
(do ( (i 0 (+ i 1)) (tmp '()))
    ((= (length decl-list) i)) ; end of iteration
  (set! tmp (list-ref decl-list i)) ; get current element
  (if (and (list? tmp) (> (length tmp) 1))
      (cond ((eq? (car tmp) 'st-input)
             (set! input-symtab (make-symtab-entry input-symtab (cdr tmp))) )
            ((eq? (car tmp) 'st-output)
             (set! output-symtab (make-symtab-entry output-symtab (cdr tmp))) )
            ((eq? (car tmp) 'st-local)
             (set! local-symtab (make-symtab-entry local-symtab (cdr tmp))) )
            ((eq? (car tmp) 'st-global)
             ⟨error: illegal global declaration 60d⟩ )
            (else ⟨error: unknown declaration 60b⟩) )
      ⟨error: wrong declaration structure 60c⟩) )
◇
```
Macro referenced in scrap 56c.

In this little code fragment an variable identifier node is constructed, which will be put into the parameter list.

⟨id-node of parameter 59b⟩ ≡
```
(get-ptn (make-ptn 'vname-id)
         (list (id-transform (list-ref par-list i)) (list-ref par-list i)))
◇
```
Macro referenced in scrap 56c.

Error and Warning Messages

⟨error: algorithm scope clash 59c⟩ ≡
```
(begin
  (error-mes "wrong algorith scope, no nested algorithms allowed")
  (debug-mes 2 " wrong argument: " obj "") )◇
```
Macro referenced in scrap 56c.

At least two elements have to be in the list passed to `parse-define`.

⟨error: wrong define argument 59d⟩ ≡
```
(begin
  (error-mes "wrong argument for define statement, at least two-element list needed")
  (debug-mes 2 " wrong argument: " obj "") )◇
```
Macro referenced in scrap 56a.

If the heavy syntax checks fail we emit this simple message, hinting to the `call/cc` construct needed by the back end.

⟨error: wrong define syntax 59e⟩ ≡
```
(begin
  (error-mes "wrong syntax of define statement, call/cc syntax mandatory")
  (debug-mes 2 " wrong argument: " obj "") )◇
```
Macro referenced in scrap 57a.

The function identifier can either be a Scheme symbol or a two-element list, otherwise this message is printed.

⟨error: wrong function identifier 60a⟩ ≡
```
(begin
  (error-mes "wrong function identifier: symbol or two-element list needed")
  (debug-mes 2 " wrong argument: " obj "") )◇
```
Macro referenced in scrap 58a.

⟨error: unknown declaration 60b⟩ ≡
```
(begin
  (error-mes "unknown declaration type in algorithm definition")
  (debug-mes 2 " wrong declaration tag: " (car tmp) "") )◇
```
Macro referenced in scrap 59a.

⟨error: wrong declaration structure 60c⟩ ≡
```
(begin
  (error-mes "wrong declaration structure, at least two element list expected")
  (debug-mes 2 " wrong argument: " tmp "") )◇
```
Macro referenced in scrap 59a.

⟨error: illegal global declaration 60d⟩ ≡
```
(begin
  (error-mes "global declaration inside algorithm")
  (debug-mes 2 " wrong declaration: " tmp "") )◇
```
Macro referenced in scrap 59a.

3.4.2 The `parse-let` Function

⟨statement parsing 60e⟩ ≡
⟨parse-let function 60f⟩◇

Macro defined by scraps 55, 60e, 62b, 63ad, 64b, 66a, 67ad, 68c, 69d, 70d.
Macro referenced in scrap 43.

Formal description: `(parse-let` *obj*`)`
obj – a Scheme object complying to the following form: `((`*init-list*`)` obj_1 … `)`
init-list – must only contain two-element lists of the form: `(`*id scm-expr*`)`
obj_i – arbitrary Scheme objects
return value: a parse tree node for the let statement (see figure 3.7)

As mentioned before, the `let` statement has only one special purpose in the intermediate Scheme representation. Therefore we know that it can only occur at algorithm scope. This will be checked. If `scope` is not `"algorithm"`, an error message is printed, otherwise the parse tree node is computed.

⟨parse-let function 60f⟩ ≡
```
;;; parse let statement
(define parse-let
  (lambda (obj)
    (define pt-node (make-ptn 'let-id)) ; return value
    (define vars '())
    (if (equal? scope "algorithm")
        ⟨let parsing 61a⟩
        ⟨error: wrong let scope 61c⟩) ))
◇
```
Macro referenced in scrap 60e.

First of all we check if *obj* complies to the formal description. Then all local variable initializations are extracted from *init-list*, which in our case is to be found in `(car obj)`. The n-element list of `set!` nodes is stored in `vars`. Next the body of the `let` statement is parsed, which is achieved by recursively calling `parse-objects`. Finally we return the calculated parse tree node.

⟨let parsing 61a⟩ ≡
```
(begin
  ; test for match with expected syntax
  (if (or (not (pair? obj))
          (not (list? (car obj))) )
      ⟨error: wrong let argument 61d⟩
      (begin
        ; iterate over all local variables
        (do ((i 0 (+ i 1)))
            ((= i (length (car obj))) )
          ; check if variable form is ok
          (if (or (not (list? (list-ref (car obj) i)))
                  (not (= 2 (length (list-ref (car obj) i)))) )
              ⟨error: wrong let list element 61e⟩
              ⟨unfold local variable 61b⟩) )
        ; put list as first element in node value
        (add-ptn-val! pt-node vars)
        ; warning for empty let body
        (if (= 0 (length (cdr obj)))
            ⟨warning: empty let body 62a⟩ )
        ; compile body of let statement
        (add-ptn-val! pt-node (parse-objects (cdr obj)))
        ; return value
        pt-node )))
◇
```
Macro referenced in scrap 60f.

Ok, now that the checks are over we create the parse tree nodes for the local variables. These are `set!` nodes and are stored in the list `vars`. We use `parse-set!` to create the `set!` nodes.

⟨unfold local variable 61b⟩ ≡
```
(set! vars (append vars (list (parse-set!
                                (list
                                 (car (list-ref (car obj) i)) ; identifier
                                 (cadr (list-ref (car obj) i)) ))))) ; expr.
◇
```
Macro referenced in scrap 61a.

Error and Warning Messages The error message consists of the wrong `recurse-level` and the rejected Scheme object. As the `let` statement is already stripped from the object, the `cons` construct is needed to restore the complete object.

⟨error: wrong let scope 61c⟩ ≡
```
(begin (error-mes "wrong scope for let statement (global)")
       (debug-mes 2 " recurse-level = " recurse-level "")
       (debug-mes 2 " while processing: " (cons 'let (list obj)) "") )◇
```
Macro referenced in scrap 60f.

Here we just print the object that doesn't comply to the required form with an appropriate error message.

⟨error: wrong let argument 61d⟩ ≡
```
(begin (error-mes "wrong argument type for let statement, special pair expected")
       (debug-mes 2 " wrong argument: " obj "") )◇
```
Macro referenced in scrap 61a.

A list element doesn't comply to the specification and therefore we print this error message with the wrong list element.

⟨error: wrong let list element 61e⟩ ≡

```
(begin (error-mes "wrong list element for let statement")
       (debug-mes 2 " wrong list element: " (list-ref (car obj) (- i 1)) "") )◇
```

Macro referenced in scrap 61a.

A warning is produced for an empty body:

⟨warning: empty let body 62a⟩ ≡

```
(begin (warn-mes "body of let statement is empty")
       (debug-mes 2 " while processing: " (cons 'leti (list obj)) "") )◇
```

Macro referenced in scrap 61a.

3.4.3 The `parse-set!` Function

⟨statement parsing 62b⟩ ≡

⟨parse-set! function 62c⟩◇

Macro defined by scraps 55, 60e, 62b, 63ad, 64b, 66a, 67ad, 68c, 69d, 70d.
Macro referenced in scrap 43.

Formal description: `(parse-set!` *obj*`)`
obj – a Scheme object complying to the following form: (*id scm-expr*)
return value: a parse tree node for the `set!` statement (see figure 3.6)

The semantics of the `set!` expression in Scheme are clear. The second parameter, an expression, is evaluated and the resulting value is stored in the location to which the first parameter, a variable, is bound. First we check, if *obj* complies to the specification.

⟨parse-set! function 62c⟩ ≡

```
;;; parses set! statement
(define parse-set!
  (lambda (obj)
    (define pt-node (make-ptn 'set!-id)) ; return value
    (if (and (list? obj)
             (= 2 (length obj)))
        ⟨set! parsing 62d⟩
        ⟨error: wrong set! argument 62e⟩) ))
◇
```

Macro referenced in scrap 62b.

Now we fill the value field of the parse tree node with the identifier and the associated expression.

⟨set! parsing 62d⟩ ≡

```
(begin
  ; first element: id
  (add-ptn-val! pt-node (get-ptn (make-ptn 'vname-id)
                                 `(,(id-transform (car obj)) ,(car obj))))
  ; second element: expression
  (add-ptn-val! pt-node (parse-expr (cadr obj)))
  pt-node)
◇
```

Macro referenced in scrap 62c.

Error and Warning Messages We spill out an error message if *obj* passed to the `parse-set!` function doesn't match the requirements.

⟨error: wrong set! argument 62e⟩ ≡

```
(begin
  (error-mes "wrong argument type for set! statement, two element list expected")
  (debug-mes 2 "wrong argument: " obj "") )◇
```

Macro referenced in scrap 62c.

3.4.4 The `parse-begin` Function

⟨statement parsing 63a⟩ ≡

⟨parse-begin function 63b⟩◇

Macro defined by scraps 55, 60e, 62b, 63ad, 64b, 66a, 67ad, 68c, 69d, 70d.
Macro referenced in scrap 43.

Formal description: `(parse-begin` *obj*`)`
obj – a list of arbitrary Scheme expressions: `(`$scm\text{-}expr_1$ `... )`
return value: a parse tree node for the begin statement (see figure 3.11)

In Scheme the order of evaluation of an expression list is usually undefined. The `begin` statement is introduced to enable sequencing in Scheme. All expressions passed to `begin` are evaluated in order, $scm\text{-}expr_1$ is the first one, $scm\text{-}expr_2$ the second one and so on. This behaviour is already implemented in `parse-objects`, see section 3.2. Thus all we have to do is pass the expression list to `parse-objects` and construct a begin node with the return value.

⟨parse-begin function 63b⟩ ≡

```
;;; parse begin statement
(define parse-begin
  (lambda (obj)
    (if (= 0 (length obj))
        ⟨warning: empty begin list 63c⟩ )
    ; return value
    (get-ptn (make-ptn 'begin-id)
             (parse-objects obj)) ))
◇
```

Macro referenced in scrap 63a.

Error and Warning Messages A warning is printed if an empty list is passed to `parse-begin`:

⟨warning: empty begin list 63c⟩ ≡

```
(warn-mes "expression list of begin statement empty")
◇
```

Macro referenced in scrap 63b.

3.4.5 The `parse-if` Function

⟨statement parsing 63d⟩ ≡

⟨parse-if function 63e⟩◇

Macro defined by scraps 55, 60e, 62b, 63ad, 64b, 66a, 67ad, 68c, 69d, 70d.
Macro referenced in scrap 43.

Formal description: `(parse-if` *obj*`)`
obj – a Scheme object complying to the form: `(`*expr* obj_1`)` or `(`*expr* obj_1 obj_2`)`
return value: a parse tree node for the `if` statement (see figure 3.8)

First of all, the argument is validated. The rest of the node creation is straightforward. The `if` condition is stored using the `parse-expr` function. The bodies of the consequent and the alternate are evaluated with `parse-objects`.

The `if` statement comes along in two flavors, with or without alternate. If the alternate is not present, we store an empty list in the third element of the node value.

⟨parse-if function 63e⟩ ≡

```
;;; parse if statement
(define parse-if
  (lambda (obj)
    (define pt-node (make-ptn 'if-id)) ; return value
    ; first check for correct argument
    (if (not (list? obj)) ⟨error: wrong if argument 64a⟩)
    (if (not (or (= 3 (length obj)) (= 2 (length obj))))
        ⟨error: wrong if argument 64a⟩)
    ; condition
    (add-ptn-val! pt-node (parse-expr (car obj)))
    ; consequent
    (add-ptn-val! pt-node (parse-objects (list (cadr obj))))
    ; alternate
    (if (= 3 (length obj))
        (add-ptn-val! pt-node (parse-objects (list (caddr obj))))
        ; no alternate
        (add-ptn-val! pt-node '()))
    pt-node))
◇
```

Macro referenced in scrap 63d.

Error and Warning Messages We generate an error message if the argument passed to the `parse-if` function doesn't match the requirements.

⟨error: wrong if argument 64a⟩ ≡

```
(begin
  (error-mes
   "wrong argument type for if statement, two or three element list expected")
  (debug-mes 2 " wrong argument: " obj "") )◇
```

Macro referenced in scrap 63e.

3.4.6 The `parse-case` Function

⟨statement parsing 64b⟩ ≡
⟨parse-case function 64c⟩◇

Macro defined by scraps 55, 60e, 62b, 63ad, 64b, 66a, 67ad, 68c, 69d, 70d.
Macro referenced in scrap 43.

> **Formal description:** `(parse-case` *obj*`)`
>
> *obj* – a Scheme object complying to the form: `(`*expr* $clause_1$ $clause_2$ ... `)`, where *expr* is a Scheme expression and $clause_n$ is a list, which consists of a list of datums and a sequence of expressions
>
> return value: a parse tree node for the `case` statement (see figure 3.9)

After a simple argument check the parsing starts.

⟨parse-case function 64c⟩ ≡

```
;;; parse case statement
(define parse-case
  (lambda (obj)
    ; return value
    (define pt-node (make-ptn 'case-id))
    ; check object
    (if (< (length obj) 2)
        ⟨error: wrong case argument 65c⟩
        ⟨case parsing 65a⟩) ))
◇
```

Macro referenced in scrap 64b.

First we store the key expression in the node value. Then we run over the clauses and build the list of alternates. Special care is taken of the `else` branch that may end a `case` statement.

⟨case parsing 65a⟩ ≡

```
(begin
  (add-ptn-val! pt-node (parse-expr (car obj))) ; case key
  (do ((i 1 (+ i 1)) (tmp '()) (alts '()) (def '()))
      ((= i (length obj)) pt-node)
    (set! tmp (list-ref obj i))
    (if (and (list? tmp) (> (length tmp) 1))
        (cond ((= i (- (length obj) 1)) ; last clause
               (if (eq? (car tmp) 'else)
                   ; with default branch)
                   (set! def (parse-objects (cdr tmp)))
                   ; no default branch
                   (set! alts
                         (append alts (list
                                        (cons ⟨build case datum list 65b⟩
                                              (parse-objects (cdr tmp)))))) )
               (add-ptn-val! pt-node alts)
               (add-ptn-val! pt-node def) )
              ; create alternate pair
              (else (set! alts
                          (append alts (list (cons ⟨build case datum list 65b⟩
                                                   (parse-objects (cdr tmp)))))) ))
        ⟨error: ill-formed case alternate 65d⟩)))
◇
```

Macro referenced in scrap 64c.

The `case` datum in the clauses may either be a list of expressions or a flat expression, we always return the datums wrapped inside a list.

⟨build case datum list 65b⟩ ≡

```
(cond ((list? (car tmp)) ; datum list
       (do ((i 0 (+ i 1)) (dats (car tmp)) (dlist '()))
           ((= i (length dats)) dlist)
         (set! dlist (append dlist (list (parse-expr (list-ref dats i))))) ))
      ; single datum
      (else (list (parse-expr (car tmp)))) )
◇
```

Macro referenced in scrap 65a.

Error and Warning Messages

⟨error: wrong case argument 65c⟩ ≡

```
(begin
  (error-mes "wrong argument for case statement, at least 2-element list expected")
  (debug-mes 2 " wrong argument: " obj "") )◇
```

Macro referenced in scrap 64c.

⟨error: ill-formed case alternate 65d⟩ ≡

```
(begin (error-mes "ill-formed case alternate, at least 2-element list expected")
       (debug-mes 2 " ill-formed alternate: " tmp "") )◇
```

Macro referenced in scrap 65a.

3.4.7 The `parse-cond` Function

⟨statement parsing 66a⟩ ≡
    ⟨parse-cond function 66b⟩◇
Macro defined by scraps 55, 60e, 62b, 63ad, 64b, 66a, 67ad, 68c, 69d, 70d.
Macro referenced in scrap 43.

> **Formal description:** `(parse-cond` *obj*`)`
>
> *obj* – a Scheme object complying to the form: `(`$clause_1$ $clause_2$ `... )`, where $clause_n$ is a list which consists of a test expression and a sequence of expressions
>
> return value: a parse tree node for the `cond` statement (see figure 3.10)

After a simple argument check the parsing starts.

⟨parse-cond function 66b⟩ ≡
```
;;; parse cond statement
(define parse-cond
  (lambda (obj)
    ; return value
    (define pt-node (make-ptn 'cond-id))
    ; check object
    (if (< (length obj) 1)
        ⟨error: wrong cond argument 66d⟩
        ⟨cond parsing 66c⟩) ))
```
◇
Macro referenced in scrap 66a.

We run over the clauses and build the list of alternates. Contrary to the `case` statement, we only have to store an expression, not an expression list.

⟨cond parsing 66c⟩ ≡
```
(do ((i 0 (+ i 1)) (tmp '()) (alts '()) (def '()))
      ((= i (length obj)) pt-node)
    (set! tmp (list-ref obj i))
    (if (and (list? tmp) (> (length tmp) 0))
        (cond ((= i (- (length obj) 1)) ; last clause
               (if (eq? (car tmp) 'else)
                   ; with default branch)
                   (set! def (parse-objects (cdr tmp)))
                   ; no default branch
                   (set! alts (append alts
                                  (list (cons (parse-expr (car tmp))
                                              (parse-objects (cdr tmp)))))))
               (add-ptn-val! pt-node alts)
               (add-ptn-val! pt-node def) )
              ; create alternate pair
              (else
               (set! alts (append alts
                              (list (cons (parse-expr (car tmp))
                                          (parse-objects (cdr tmp)))))) ))
        ⟨error: ill-formed cond alternate 66e⟩ ))
```
◇
Macro referenced in scrap 66b.

Error and Warning Messages

⟨error: wrong cond argument 66d⟩ ≡
```
(begin (error-mes "wrong argument for cond statement, non-empty list expected")
       (debug-mes 2 " wrong argument: " obj "") )◇
```
Macro referenced in scrap 66b.

⟨error: ill-formed cond alternate 66e⟩ ≡
```
(begin (error-mes "ill-formed cond alternate, list expected")
       (debug-mes 2 " ill-formed alternate: " tmp "") )◇
```
Macro referenced in scrap 66c.

3.4.8 The `parse-while` Function

⟨statement parsing 67a⟩ ≡
    ⟨parse-while function 67b⟩◇

Macro defined by scraps 55, 60e, 62b, 63ad, 64b, 66a, 67ad, 68c, 69d, 70d.
Macro referenced in scrap 43.

> **Formal description:** `(parse-while` *obj*`)`
>
> *obj* – a Scheme object (pair) complying to the form: `(`*expr body*`)`
> *expr, body* – arbitrary Scheme expression respectively object
>
> return value: a parse tree node for the `while` statement (see figure 3.12)

All we have to do with the `while` statement is passing its conditional expression to `parse-expr` and the statement body to `parse-objects`. We precede this two steps with a test for a correct object.

⟨parse-while function 67b⟩ ≡
```
;;; parse while macro
(define parse-while
  (lambda (obj)
    ; return value
    (define pt-node (make-ptn 'while-id))
    ; check object
    (if (not (pair? obj))
        ⟨error: wrong while argument 67c⟩
        (begin ; return value
          (add-ptn-val! pt-node (parse-expr (car obj)))
          (add-ptn-val! pt-node (parse-objects (cdr obj))) ))
    pt-node))
```
◇

Macro referenced in scrap 67a.

Error and Warning Messages We print an error message if the argument passed to the `parse-while` is not a pair. The `car` of the pair is the `while` condition, the `cdr` the iteration body.

⟨error: wrong while argument 67c⟩ ≡
```
(begin (error-mes "wrong argument type for while statement, pair expected")
       (debug-mes 2 " wrong argument: " obj "") )◇
```

Macro referenced in scrap 67b.

3.4.9 The `parse-repeat` Function

⟨statement parsing 67d⟩ ≡
    ⟨parse-repeat function 67e⟩◇

Macro defined by scraps 55, 60e, 62b, 63ad, 64b, 66a, 67ad, 68c, 69d, 70d.
Macro referenced in scrap 43.

> **Formal description:** `(parse-while` *obj*`)`
>
> *obj* – a list of Scheme objects complying to the form: `(`obj_1 ... `until` *expr*`)`
> obj_i – arbitrary Scheme objects, $i \geq 0$
> *expr* – Scheme expression for the repeat condition
>
> return value: a parse tree node for the `repeat` statement (see figure 3.13)

Ok, as usual we first perform simple checks on *obj* passed to `parse-repeat`. The the parsing of the `repeat` statement starts.

⟨parse-repeat function 67e⟩ ≡

```
;;; parse repeat macro
;;;
(define parse-repeat
  (lambda (obj)
    (define pt-node (make-ptn 'repeat-id)) ; return value
    ; first check for correct argument
    (if (not (and (list? obj)
                  (> (length obj) 1)
                  (equal? (list-ref obj (- (length obj) 2)) 'until)
                  ))
        ⟨error: wrong repeat argument 68b⟩
        ; now parse the statement
        ⟨repeat parsing 68a⟩)
    pt-node))
◇
```

Macro referenced in scrap 67d.

In the first phase of the `repeat` parsing all statements in the body are passed to `parse-objects` and the resulting nodes are stored in a list.

Thereafter the conditional expression is parsed with `parse-expr` and put into the node value's list as the second element.

⟨repeat parsing 68a⟩ ≡

```
(begin
  (do ( (i 0 (+ i 1)) (objs '()) )
      ((= i (- (length obj) 2))
       ; store parsed repeat body
       (add-ptn-val! pt-node (parse-objects objs)))
    (set! objs (append objs (list (list-ref obj i)))) )
  ; store parse condition
  (add-ptn-val! pt-node (parse-expr (list-ref obj (- (length obj) 1)))) )
◇
```

Macro referenced in scrap 67e.

Error and Warning Messages The object passed to `parse-repeat` has to be a special list. If the check fails the following error message is issued:

⟨error: wrong repeat argument 68b⟩ ≡

```
(begin
  (error-mes "wrong argument type for repeat statement, special list expected")
  (debug-mes 2 " wrong argument: " obj "") ) ◇
```

Macro referenced in scrap 67e.

3.4.10 The `parse-for1` Function

⟨statement parsing 68c⟩ ≡

⟨parse-for1 function 69a⟩◇

Macro defined by scraps 55, 60e, 62b, 63ad, 64b, 66a, 67ad, 68c, 69d, 70d.
Macro referenced in scrap 43.

Formal description: (`parse-for1` *obj*)

obj – a list of Scheme objects complying to the form: (id $expr_1$ $expr_2$ obj_1 ...)
id – symbol representing the variable identifier
$expr_{\{1,2\}}$ – Scheme expressions for the initial and final variable values
obj_i – arbitrary Scheme objects, $i \geq 0$
return value: a parse tree node for the `for1` statement (see figure 3.14)

The `for1` parsing runs along to our normal scheme. First, the function argument is checked.

⟨parse-for1 function 69a⟩ ≡
```
;;; parse for1 macro
(define parse-for1
  (lambda (obj)
    (define pt-node (make-ptn 'for1-id)) ; return value
    ; checks of obj
    (if (< (length obj) 3)
        ⟨error: wrong for1 argument 69c⟩
        ; error checks passed
        ⟨for1 parsing 69b⟩)
    ; return value
    pt-node ))
◇
```
Macro referenced in scrap 68c.

Then all the parts of the `for1` statement are put into the list that constitutes the node value.

⟨for1 parsing 69b⟩ ≡
```
(begin
  ; variable id
  (add-ptn-val! pt-node (get-ptn (make-ptn 'vname-id)
                                 `(,(id-transform (car obj)) ,(car obj))))
  ; initial value
  (add-ptn-val! pt-node (parse-expr (cadr obj)))
  ; final value
  (add-ptn-val! pt-node (parse-expr (caddr obj)))
  ; compile body statements
  (do ( (i 3 (+ i 1)) (objs '()) )
      ((= i (length obj))
       ; store parsed for1 body
       (add-ptn-val! pt-node (parse-objects objs)))
    (set! objs (append objs (list (list-ref obj i)))) ) )
◇
```
Macro referenced in scrap 69a.

Error and Warning Messages The object passed to `parse-for1` has to be a list of at least three elements. An error message is produced if the requirements are not met.

⟨error: wrong for1 argument 69c⟩ ≡
```
(begin
  (error-mes
   "wrong argument type for for1 statement, at least 3-element list expected")
  (debug-mes 2 "wrong argument: " obj "") )◇
```
Macro referenced in scrap 69a.

3.4.11 The `parse-for` Function

⟨statement parsing 69d⟩ ≡
    ⟨parse-for function 70a⟩◇

Macro defined by scraps 55, 60e, 62b, 63ad, 64b, 66a, 67ad, 68c, 69d, 70d.
Macro referenced in scrap 43.

Formal description: `(parse-for` *obj*`)`

obj – a list of Scheme objects of the form: `(`$id\ expr_1\ expr_2\ expr_3\ obj_1\ \ldots$ `)`
id – symbol representing the variable identifier
$expr_{\{1,2,3\}}$ – Scheme expressions for the initial, second and final variable values
obj_i – arbitrary Scheme objects, $i \geq 0$
<u>return value:</u> a parse tree node for the `for` statement (see figure 3.14)

The `for` statement has only one extension to the `for1` macro, the second value of the runtime variable. The rest of the function works analog to the `parse-for1` function.

⟨parse-for function 70a⟩ ≡
```
;;; parse for macro
(define parse-for
  (lambda (obj)
    (define pt-node (make-ptn 'for-id)) ; return value
    ; checks of obj
    (if (< (length obj) 4)
        ⟨error: wrong for argument 70c⟩
        ⟨for parsing 70b⟩)
    ; return value
    pt-node ))
◇
```
Macro referenced in scrap 69d.

⟨for parsing 70b⟩ ≡
```
(begin
  ; variable id
  (add-ptn-val! pt-node (get-ptn (make-ptn 'vname-id)
                                  `(,(id-transform (car obj)) ,(car obj))))
  ; initial value
  (add-ptn-val! pt-node (parse-expr (cadr obj)))
  ; second value
  (add-ptn-val! pt-node (parse-expr (caddr obj)))
  ; final value
  (add-ptn-val! pt-node (parse-expr (cadddr obj)))
  ; compile body statements
  (do ( (i 4 (+ i 1)) (objs '()) )
      ((= i (length obj))
       ; store parsed for1 body
       (add-ptn-val! pt-node (parse-objects objs)) )
    (set! objs (append objs (list (list-ref obj i)))) ) )
◇
```
Macro referenced in scrap 70a.

Error and Warning Messages The object passed to `parse-for` has to be a list of at least four elements. An error message is produced if these requirements are not met.

⟨error: wrong for argument 70c⟩ ≡
```
(begin
  (error-mes
   "wrong argument type for for statement, at least 4-element list expected")
  (debug-mes 2 " wrong argument: " obj "") )◇
```
Macro referenced in scrap 70a.

3.4.12 The `parse-return` Function

⟨statement parsing 70d⟩ ≡
    ⟨parse-return function 71a⟩◇

Macro defined by scraps 55, 60e, 62b, 63ad, 64b, 66a, 67ad, 68c, 69d, 70d.
Macro referenced in scrap 43.

Formal description: (`parse-return` *expr*)
expr – an arbitrary Scheme expression
return value: a parse tree node for the `return` statement (see figure 3.15)

The node for the `return` statement is constructed by storing the return value of `parse-expr` in the node value. We have to use the function `car` on *expr* because it's wrapped inside a list.

⟨parse-return function 71a⟩ ≡

```
;;; parse return macro
(define parse-return
  (lambda (expr)
    ; return value
    (if (null? expr)
        ; empty return
        (get-ptn (make-ptn 'return-id) '())
        (get-ptn (make-ptn 'return-id) (parse-expr (car expr))) ) ))
◇
```

Macro referenced in scrap 70d.

3.5 Expression Parsing

Expressions have their own little grammar. We will convert expressions in Scheme notation into a recursive parse tree node. A nice inheritance of this notation is the solved operator precedence problem. The front end deals with this problem and presents us the resolved expressions.

3.5.1 The `parse-expr` Function

Formal description: `(parse-expr` *expr*`)`
expr – any Scheme expression
<u>return value:</u> a parse tree node for *expr* (see figure 3.17)

We start the expression parsing with the check for nested expressions, go on with known literals and end with variable identifiers.

⟨expression parsing 71b⟩ ≡

```
;;; parse a Scheme expression
(define parse-expr
  (lambda (expr)
    (define pt-node (make-ptn 'expr-id))
    (define ops '())
    (cond
     ⟨parse nested expression 71c⟩
     ⟨parse literals 72c⟩
     ⟨parse identifier 73e⟩ )
    (add-ptn-val! pt-node ops)
    pt-node ))
◇
```

Macro referenced in scrap 43.

3.5.2 Nested Expressions

Nested expressions are either function calls or operator constructs. In Scheme there is no real difference between this two categories thanks to the regularized prefix notation and lambda expressions. But we start at SuchThat, which is imperative and uses infix notation for expressions, and end at C^{++}, where the same holds true. Hence we split them. Operators are limited to unary and binary forms, unlike Scheme expressions: `(+ 1 2 4 8)`.

We examine the `car` field of the expression and decide if it's a operator or a function call by looking up the back end's operators in `op-list`. Depending on this decision we create either a operator name identifier or a function name identifier node.

⟨parse nested expression 71c⟩ ≡

```
((list? expr)
 (cond ((and (not (null? expr))
             (string? (car expr))
             (number? (car expr))
             (char? (car expr))
             (boolean? (car expr)) )
        ⟨error: nested expression starts with literal 72b⟩ )
       ((not (null? expr))
        ; first element: id
        (add-ptn-val!
         pt-node (if (list-memb? (car expr) op-list)
                     (get-ptn (make-ptn 'opname-id)
                              (if (list-memb? (car expr) op-list)
                                  `(,(symbol->string (car expr)) ,(car expr))
                                  `(,(id-transform (car expr)) ,(car expr))))
                     (get-ptn (make-ptn 'fname-id)
                              `(,(id-transform (car expr)) ,(car expr))) ))
        ; build operand list
        (do ((i 1 (+ i 1)))
            ((= i (length expr)))
          (set! ops (append ops (list (parse-expr (list-ref expr i))))) ))
       ; default error
       (else ⟨error: bad expression 72a⟩) ))
◇
```

Macro referenced in scrap 71b.

Error and Warning Messages

⟨error: bad expression 72a⟩ ≡

```
(begin (error-mes "weird expression syntax")
       (debug-mes 2 " expression: " expr "") )◇
```

Macro referenced in scrap 71c.

⟨error: nested expression starts with literal 72b⟩ ≡

```
(begin (error-mes "nested expression starts with literal")
       (debug-mes 2 " expression: " expr "") )◇
```

Macro referenced in scrap 71c.

3.5.3 Literals

We support four kinds of literals: numerical, boolean, character and string literals.

⟨parse literals 72c⟩ ≡

```
⟨parse numerical literal 72d⟩
⟨parse boolean literal 73b⟩
⟨parse character literal 73c⟩
⟨parse string literal 73d⟩
◇
```

Macro referenced in scrap 71b.

3.5.3.1 Numerical Literals

Numerical literals can be identified with the `number?` predicate and are further refined thereafter.

In the node identifier field we store the literal identifier for the appropriate numerical type. The node value is simply the string returned by the standard Scheme function `number->string`.

⟨parse numerical literal 72d⟩ ≡

```
((number? expr)
 (add-ptn-val! pt-node (get-ptn (make-ptn ⟨get numerical type 73a⟩)
                                (number->string expr))) )
◇
```

Macro referenced in scrap 72c.

We don't test the numerical literals according to the level arrangement given in [4] (page 19). We try those types first that are more easily handled by todays micro processors and programming languages to keep the back end faster.

⟨get numerical type 73a⟩ ≡

```
(cond
 ((integer? expr) 'int-id)
 ((real? expr) 'real-id)
 ((rational? expr) 'rational-id)
 ((complex? expr) 'complex-id) )
◇
```

Macro referenced in scrap 72d.

3.5.3.2 Boolean Literals

A boolean literal is represented by either the string `true` or `false`.

⟨parse boolean literal 73b⟩ ≡

```
((boolean? expr)
 (add-ptn-val! pt-node (get-ptn (make-ptn 'bool-id)
                        (if (eq? expr #f) "false" "true"))) )
◇
```

Macro referenced in scrap 72c.

3.5.3.3 Character Literals

Characters are transformed to the equivalent string representation with the Scheme function `string`.

⟨parse character literal 73c⟩ ≡

```
((char? expr)
   (add-ptn-val! pt-node (get-ptn (make-ptn 'char-id) (string expr))) )
◇
```

Macro referenced in scrap 72c.

3.5.3.4 String Literals

Strings are stored *as is* in the node value.

⟨parse string literal 73d⟩ ≡

```
((string? expr)
   (add-ptn-val! pt-node (get-ptn (make-ptn 'string-id) expr)) )
◇
```

Macro referenced in scrap 72c.

3.5.4 Identifiers

A Scheme symbol can only be a variable reference by its identifier name. All we have to do is create an variable identifier node.

⟨parse identifier 73e⟩ ≡

```
((symbol? expr)
 (add-ptn-val! pt-node (get-ptn (make-ptn 'vname-id)
                        `(,(id-transform expr) ,expr)) ))
◇
```

Macro referenced in scrap 71b.

3.6 Parsing SuchThat Declarations

Some of them found their way into our back end. They can't be handled like statements. They introduce new structures or change global behavior, for example the `map` declaration (or *realization*) gives us the possibility to change function and datatype mappings. At least that's what Sibylle had in mind in the first place.

⟨list of declarations 74a⟩ ≡

```
;;; list with SuchThat declarations, used by parse-objects
(st-global . ,parse-global-decl)
(st-local . ,error-decl-scope)
(st-input . ,error-decl-scope)
(st-output . ,error-decl-scope)
(st-map . ,parse-map-decl)
◇
```

Macro referenced in scrap 54b.

Here we keep all keywords that are inherited from SuchThat declarations, they are part of the parser list. The corresponding parser functions also return parse tree nodes, we call them *manipulator nodes*. This is motivated by the fact that these nodes won't be transformed to code in the code generation phase. The nodes are evaluated, and in this process global resources of the compiler are updated, like the global symbol table or mapping tables.

3.6.1 The `parse-global-decl` Procedure

⟨declaration parsing 74b⟩ ≡

⟨parse-global-decl function 74c⟩◇

Macro defined by scraps 74b, 75b, 77a.
Macro referenced in scrap 43.

Formal description: `(parse-global-decl` *decl-list*`)`
decl-list – a list of pairs which comply to the following structure: `(`*id* `.` *type*`)`, where *id* is an identifier symbol and *type* is a type specifier as defined in 1.3
return value: a global declaration parse tree node (see 3.18), containing all the declarations in `decl-list`

Type information occurs only inside declarations. Input, output and local declarations are treated while parsing function definitions, implemented in `parse-define`. That leaves us to cope with global declarations. They are introduced with the special keyword `st-global`, which is part of the parser list.

The procedure examines the argument, and then calls `make-symtab-entry` to create a symbol table as node value, if it's called at global scope. In pass 2 a global declaration node will effect the global symbol table.

⟨parse-global-decl function 74c⟩ ≡

```
;;; parse st-global declaration
(define parse-global-decl
  (lambda (decl-list)
    ; save declaration list in node value
    (if (equal? scope "global")
        (get-ptn (make-ptn 'globaldecl-id)
                 (get-ptn (make-ptn 'symtab-id)
                          (make-symtab-entry '() decl-list)))
        ⟨error: global declarations only at global scope 75a⟩) ))
◇
```

Macro referenced in scrap 74b.

Error and Warning Messages

⟨error: global declarations only at global scope 75a⟩ ≡
```
(begin (error-mes "global declarations only at global scope allowed")
       (debug-mes 2 " misplaced global declarations: " decl-list "") )◇
```
Macro referenced in scrap 74c.

3.6.2 The `parse-map-decl` Function

⟨declaration parsing 75b⟩ ≡
    ⟨parse-map-decl function 75c⟩◇

Macro defined by scraps 74b, 75b, 77a.
Macro referenced in scrap 43.

Formal description: (`parse-map-decl` *decl*)
decl – a three element list, whose first element is either the symbol `function` or `type`, and whose other elements are either identifiers or type specifiers
return value: a mapping parse tree node (see 3.19), containing the declaration in `decl`

We introduced mapping declarations in section 1.2. First we test, if the passed argument is valid. Then the kind of mapping is determined by looking at the symbol at first position in the parameter `decl`.

If mapping is disabled with the flag `enable-map-decl`, a warning is printed and an empty function mapping node is returned.

⟨parse-map-decl function 75c⟩ ≡
```
;;; parse mapping declarations
(define parse-map-decl
  (lambda (decl)
    (define pt-node '())
    (cond (enable-map-decl
           (if (and (list? decl) (not (null? decl)))
               (set! decl (car decl))
               ⟨error: wrong argument for parse-map-decl 76b⟩)
           (if (and (list? decl) (= 3 (length decl)))
               (cond ((eq? 'function (car decl)) ⟨function map parsing 75d⟩)
                     ((eq? 'type (car decl)) ⟨type map parsing 76a⟩)
                     (else ⟨error: unknown map type 76c⟩))
               ⟨error: wrong argument for parse-map-decl 76b⟩) )
          (else ⟨warning: mapping disabled 76e⟩
                (set! pt-node (get-ptn (make-ptn 'funcmap-id) '())) ))
    pt-node))
◇
```
Macro referenced in scrap 75b.

The arguments of a function mapping are two identifiers. We check if they are symbols. Then the two function identifier nodes are put into the value field of the mapping node. We have to omit operators in `op-list` from identifier transformation.

⟨function map parsing 75d⟩ ≡
```
(set! pt-node (make-ptn 'funcmap-id))
; map source
(if (symbol? (cadr decl))
    (if (list-memb? (cadr decl) op-list)
        (add-ptn-val!
         pt-node (get-ptn (make-ptn 'fname-id)
                          `(,(symbol->string (cadr decl)) ,(cadr decl))))
        (add-ptn-val!
         pt-node (get-ptn (make-ptn 'fname-id)
                          `(,(id-transform (cadr decl)) ,(cadr decl)))) )
```

```
    ⟨error: bad mapping parameter 76d⟩)
; map target
(if (symbol? (caddr decl))
    (if (list-memb? (caddr decl) op-list)
        (add-ptn-val!
         pt-node (get-ptn (make-ptn 'fname-id)
                          `(,(symbol->string (caddr decl)) ,(caddr decl))))
        (add-ptn-val!
         pt-node (get-ptn (make-ptn 'fname-id)
                          `(,(id-transform (caddr decl)) ,(caddr decl)))) )
    ⟨error: bad mapping parameter 76d⟩)
◇
```

Macro referenced in scrap 75c.

Type mappings expect an identifier followed by a type specifier as arguments. They are converted into parse tree nodes with the corresponding functions and put into the value field of our mapping node.

⟨type map parsing 76a⟩ ≡

```
(set! pt-node (make-ptn 'typemap-id))
; map source
(if (symbol? (cadr decl))
    (add-ptn-val!
     pt-node (get-ptn (make-ptn 'typename-id)
                      `(,(id-transform (cadr decl)) ,(cadr decl))))
    ⟨error: bad mapping parameter 76d⟩)
;(add-ptn-val! pt-node (parse-type (cadr decl)))
; map target
(add-ptn-val! pt-node (parse-type (caddr decl)))
◇
```

Macro referenced in scrap 75c.

Error and Warning Messages

⟨error: wrong argument for parse-map-decl 76b⟩ ≡

```
(begin
  (error-mes "wrong argument for parse-map-decl, three-element list expected")
  (debug-mes 2 " argument: " decl "") )◇
```

Macro referenced in scrap 75c.

⟨error: unknown map type 76c⟩ ≡

```
(begin (error-mes "unknown map type, <function> or <type> expected")
       (debug-mes 2 " illegal type: " (car decl) "") )◇
```

Macro referenced in scrap 75c.

⟨error: bad mapping parameter 76d⟩ ≡

```
(begin (error-mes "bad mapping parameter, symbol expected")
       (debug-mes 2 " mapping parameters: " (cdr decl) "") )◇
```

Macro referenced in scraps 75d, 76a.

⟨warning: mapping disabled 76e⟩ ≡

```
(begin (warn-mes "mapping disabled")
       (debug-mes 2 " disabled declaration: " decl "") )◇
```

Macro referenced in scrap 75c.

3.6.3 The error-decl-scope Procedure

⟨declaration parsing 77a⟩ ≡
⟨error-decl-scope 77b⟩◇

Macro defined by scraps 74b, 75b, 77a.
Macro referenced in scrap 43.

Formal description: (error-decl-scope *obj*)
obj – list whose car field is st-local, st-input or st-output
return value: unspecified

Global declarations are allowed as stand-alone keyword. The others are permitted only inside the function definition header. Consequently we display an error if we encounter them elsewhere.

⟨error-decl-scope 77b⟩ ≡
```
;;; input, output or local declarations at illegal location
(define error-decl-scope
  (lambda (obj)
    (error-mes "input, output and local declarations not at algorithm header")
    (debug-mes 2 " wrongly placed declaration: " obj "")
    ))
```
◇

Macro referenced in scrap 77a.

3.7 Parsing Type Information

The grammar of type information is specified in section 1.3. We have to build a type tree that represents a type grammar.

⟨type parsing 77c⟩ ≡
⟨parse-type function 77e⟩
◇

Macro referenced in scrap 43.

3.7.1 The parse-type Function

Formal description: (parse-type *type*)
type – a type description conforming to the grammar specifications
return value: a type parse tree node (see figure 3.20)

A string is saved in the node value that encodes various type information (see table 3.1). We use these variables to construct the type string:

⟨type string character encoding 77d⟩ ≡
```
(define fun-type #\f)
(define gen-type #\g)
(define arr-type #\a)
(define con-type #\c)
(define ref-type #\r)
(define val-type #\v)
```
◇

Macro referenced in scraps 77e, 98a, 109b.

We can append the values of the newly defined variables to the initially empty type-inf. Of course, some characters mutually exclude one another.

⟨parse-type function 77e⟩ ≡

```
;;; parse type information
(define parse-type
  (lambda (type)
    (define pt-node (make-ptn 'type-id)) ; return value
    (define npars 0) ; number of type parameters
    (define type-inf "") ; information about type
    ⟨type string character encoding 77d⟩
    (cond
     ; fundamental type
     ((symbol? type) ⟨parse fundamental type 78a⟩ )
     ; derived type
     ((list? type)
      (if (equal? (id-transform (car type)) "Array")
          ⟨parse array type 78b⟩
          ⟨parse container type 79a⟩
          ))
     (else ⟨error: wrong parse-type argument 80a⟩) )
    (add-ptn-val! pt-node type-inf)
    pt-node ))
◇
```

Macro referenced in scrap 77c.

The global list `type-list` contains pairs. The pair's `cdr` field holds the number of type parameters expected by the type, which is denoted in the pair's `car` field. We use `dassoc` to get this value. For fundamental types 0 is returned. If `dassoc` returns `#f`, the type is not known and we mark it as *generic*.

⟨parse fundamental type 78a⟩ ≡

```
; get number of type parameters
(set! nargs (dassoc (id-transform type) type-list))
(cond ((eq? nargs #f) ; not in type-list -> generic type
       (set! type-inf (string gen-type ref-type))
       ; type id name
       (add-ptn-val! pt-node (get-ptn (make-ptn 'typename-id)
                                      `(,(id-transform type) ,type)))
       (add-ptn-val! pt-node '()))
      ((= nargs 0) ; in type-list -> fundamental type
       (set! type-inf (string fun-type ref-type))
       ; type id name
       (add-ptn-val! pt-node (get-ptn (make-ptn 'typename-id)
                                      `(,(id-transform type) ,type)))
       (add-ptn-val! pt-node '()))
      ((> nargs 0) ; container type
       ⟨error: container without type parameters 79b⟩))
◇
```

Macro referenced in scrap 77e.

Arrays have their own syntax. At the second position we find its type and use `parse-type` on it. Then we iterate over the index-ranges for all dimensions. Here we have to discriminate between index-ranges given either as a pair of upper and lower bounds or as a single expression donating the number of elements directly.

⟨parse array type 78b⟩ ≡

```
(if (> (length type) 2)
    (begin
      ; array & call-by-ref
      (set! type-inf (string arr-type ref-type))
      ; array type
      (add-ptn-val! pt-node (parse-type (cadr type)))
      ; array dimensions
```

```
      (do ((i 2 (+ i 1)) (exprs '()) (tmp '()))
          ((= i (length type)) (add-ptn-val! pt-node exprs))
        (set! tmp (list-ref type i))
        ;(cond ((or (not (list? tmp)) ; single index-range
        ;(and (list? tmp) (procedure? (car tmp))) )
        (set! exprs (append exprs (list (parse-expr tmp)))) )
      ; index-range pair
              ;((and (list? tmp) (= 2 (length tmp)))
               ;(set! exprs (append exprs (list (cons (parse-expr (car tmp))
                                                       ;(parse-expr (cadr tmp)))
                                                 ;))) )
    ;
      )
    ⟨error: wrong array type declaration 80b⟩ )
◇
```

Macro referenced in scrap 77e.

Finally, we deal with container types. The `car` field of a container type list stores its name. The remaining fields in the list stpre its type parameters. We run over all type parameters and call `parse-type`.

⟨parse container type 79a⟩ ≡

```
(if (> (length type) 1)
    (begin
      ; get number of type parameters
      (set! nargs (dassoc (id-transform (car type)) type-list))
      (cond ((eq? nargs #f) ⟨error: unknown container type 79d⟩)
            ; correct number of type parameters
            ((eq? nargs (- (length type) 1))
             ; container & call-by-ref
             (set! type-inf (string con-type ref-type))
             ; container name
             (add-ptn-val!
              pt-node (get-ptn (make-ptn 'typename-id)
                               `(,(id-transform (car type)) ,(car type))))
             (do ((i 1 (+ i 1)) (types '()) (tmp '()))
                 ((= i (length type)) (add-ptn-val! pt-node types))
               (set! tmp (list-ref type i))
               (set! types (append types (list (parse-type tmp))))
               ))
            (else ⟨error: wrong number of type parameters for container 79c⟩)) )
    ⟨error: container without type parameters 79b⟩ )
◇
```

Macro referenced in scrap 77e.

Error and Warning Messages

⟨error: container without type parameters 79b⟩ ≡

```
(begin (error-mes "container used without type parameters")
       (debug-mes 2 " container: " type "") )◇
```

Macro referenced in scraps 78a, 79a.

⟨error: wrong number of type parameters for container 79c⟩ ≡

```
(begin (error-mes "wrong number of type parameters for container")
       (debug-mes 2 " container: " type "") )◇
```

Macro referenced in scrap 79a.

⟨error: unknown container type 79d⟩ ≡

```
(begin (error-mes "unknown container type, examine type-list")
       (debug-mes 2 " container: " type "") )◇
```

Macro referenced in scrap 79a.

⟨error: wrong parse-type argument 80a⟩ ≡

```
(begin (error-mes "wrong parse-type argument, symbol or list expected")
       (debug-mes 2 " argument: " type "") )◇
```

Macro referenced in scrap 77e.

⟨error: wrong array type declaration 80b⟩ ≡

```
(begin (error-mes "array type declaration, type and dimension lists expected")
       (debug-mes 2 " bad declaration: " type "") )◇
```

Macro referenced in scrap 78b.

Chapter 4

Pass 2: Code Generation

In the preceding chapter we detailed the parsing process. Its result is completely encapsulated in the parse tree returned by the top level invocation of `parse-objects`, due to our decision to attribute every algorithm with its local symbol tables.

Now we have to generate code for the target language C++. We don't aim at assembly language and therefore we have to focus on other issues than commonly discussed in books on compiler construction. Register allocation, memory management and other low-level problems are left to the back end target language compiler, they have been solved many times before. Most of the code generation is a simple mapping of the Scheme interface grammar to C++ syntax.

SuchThat was created with the building of a library in mind, which is perhaps the most useful application of generic programming. This opinion is shared by Sibylle Schupp, who states in the abstract of [27]:

> Therefore, the most important applications of generic programming are in library design and in the research of algorithm classes.

A prominent example for generic programming is the Standard Template Library ([30]), which has taken the C++ world by storm. One of the main problems we have to tackle in code generation is how algorithms look like. Algorithms play a fundamental part in the STL. Its acceptance in the new C++ standard ([14]) and its increasing importance in every day C++ programming made us chose the algorithm concept of the STL as prototype for our code generator.

"src/code.scm" 81a ≡

⟨special string-append 81b⟩
⟨generate-code function 82b⟩
⟨list of code emitters 82d⟩
⟨list of manipulator functions 98d⟩
◇

The code generator has two function categories with side effects. The first one changes the global variable `icount`, the second one the mapping tables. We therefore need a macro that guarantees the order of evaluation:

⟨special string-append 81b⟩ ≡

```
(define-macro
  (str-seq . args)
  (if (null? args) ""
      (let ((tmp (gensym)))
        `(let ((,tmp ,(car args)))
           (string-append ,tmp (str-seq ,@(cdr args))) ))))
```
◇

Macro referenced in scrap 81a.

4.1 The generate-code Function

The parse-objects function is the central part of the parsing process and builds the parse tree. We pass the parse tree, returned in a node, to the generate-code function. It scans the whole parse tree and generates code for every parse tree node it encounters during tree traversal. The code is kept in a string variable. This enables easy backtracking and patching when needed.

The code corresponding to a parse tree node is generated by a *code emitter*. This is a function that gets the node's value and returns a string representing the value's content in the target language. The code emitters are kept in the list of code emitters: emitter-list.

⟨code emitter list 82a⟩ ≡
```
(define emitter-list
  (map (lambda (e) (cons (cadr e) (caddr e))) pt-nodeids))
◇
```
Macro referenced in scrap 33.

We already associated every code emitter with a node type in pt-nodeids. The Scheme function map is used to set up the list of code emitters, adequate for usage in generate-code

Formal description: (generate-code *ptn*)
ptn – a parse tree node
return value: a string representing the parse tree node in target language code

If expressions appear at top-level, they must be function or procedure calls. We append a semicolon to the end of the expression to satisfy C++ requirements, otherwise we call a code emitter from the emitter table. The technique is the same as the one employed for parser functions.

⟨generate-code function 82b⟩ ≡
```
;;; generate code and return it in a string
(define generate-code
  (lambda (ptn)
    (if (ptn? ptn)
        (if (expr? ptn)
            (emit-strin (str-seq (emit-expr (cdr ptn)) ";"))
            ((dassoc (car ptn) emitter-list) (cdr ptn)) )
        ⟨error: wrong argument for generate-code 82c⟩) ))
◇
```
Macro referenced in scrap 81a.

Error and Warning Messages

⟨error: wrong argument for generate-code 82c⟩ ≡
```
(begin (error-mes "wrong argument for generate-code, parse tree node expected")
       (debug-mes 2 " argument: " ptn "") )◇
```
Macro referenced in scrap 82b.

4.2 List of Code Emitters

Now we describe all the code emitters that are available. Whenever a new target language is chosen, almost all of them have to be rewritten.

⟨list of code emitters 82d⟩ ≡
```
⟨emit-pt function 83⟩
⟨emit-expr function 95c⟩
⟨emit-literal functions 95b⟩
⟨emit-id functions 84a, ... ⟩
⟨emit-type function 98a⟩
⟨emit-definition functions 85a, ... ⟩
⟨emit-statement functions 88b, ... ⟩
(define emit-symtab (lambda (val) "<error: shouldn't get here>"))
◇
```

Macro referenced in scrap 81a.

The rewriting of code emitters is central to retargeting the back end. Therefore we dwell some more on the details of changing them. Figure 4.1 shows a typical code emitter. The input is a parse tree node value. The emitter references the items in the node value `val` with the function `list-ref`. The output is a string, hence the emitter often just concatenates the string representations of the items in `val`.

Code emitters frequently call already defined emitters. Because the items in `val` are normally parse tree nodes, we must keep in mind that these calls are done with the `cdr` field of a node, such that the emitter really gets the node *value*. Typical is the invocation of `emit-expr`: `(emit-expr (cdr (list-ref val i)))`. Of course, this is not necessary for a recursive call of `generate-code`.

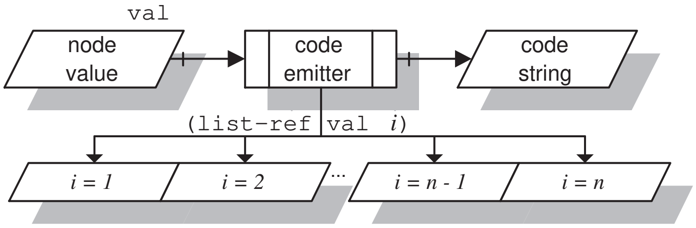

Figure 4.1: functioning of a code emitter

We provide a simple technique to format the generated code with the global variables `icount` and `istep`, in conjunction with the output routines from section 5.5. The functions `ind-inc!` and `ind-dec!` control the indentation. The code emitter must create code strings with the custom output routines to support formatted output.

4.2.1 The `emit-pt` Function

Formal description: (`emit-ptn` *ptn-list*)
ptn-list – a list of parse tree nodes
return value: a string representing the parse tree nodes in target language code

This function has no target specific code, it just works on a parse tree recursion node. The function `generate-code` is called for every node in the value field. The resulting strings are appended to one another.

⟨emit-pt function 83⟩ ≡

```
(define emit-pt
  (lambda (ptn-list)
    (do ((i 0 (+ i 1)) (code ""))
        ((= i (length ptn-list)) code)
      (set! code (str-seq code (generate-code (list-ref ptn-list i))))
      )))
```

◇

Macro referenced in scrap 82d.

4.2.2 Code Emitters for Identifiers

Formal description: (`emit-`*idtype* *val*)
val – the value field of an identifier node
return value: a string representing the identifier, according to target language restrictions

Special treatment is necessary for variable identifiers that denote a single output parameter. They are allocated with `new`, which returns a pointer. We must dereference such identifiers.

⟨emit-id functions 84a⟩ ≡

```
;;; variable name identifier
(define emit-vname
  (lambda (val)
    (if (and (= 1 (length output-symtab))
             (not (null? (get-idtypet (car val) output-symtab)))
             (not (list-memb? (car (get-idtypet (car val) output-symtab))
                              val-list)) )
        (str-seq "(*" (car val) ")")
        (car val)) ))
◇
```

Macro defined by scraps 84abc.
Macro referenced in scrap 82d.

The emit functions for function and operator identifiers check, if the transformed identifier is subject to function mapping. In this case, the mapping target is returned, otherwise the string in the `car` field of the node value.

⟨emit-id functions 84b⟩ ≡

```
;;; function name identifier
(define emit-fname
  (lambda (val)
    (define fn-name (dassoc (car val) func-map))
    (if (eq? #f fn-name) (car val) fn-name) ))
;;; operator identifier
(define emit-opname (lambda (val) (emit-fname val) ))
◇
```

Macro defined by scraps 84abc.
Macro referenced in scrap 82d.

Type names are mapped according to the type mapping table `type-map`. If the type identifier is in a `car` field of this alist, it will be replaced by the string in the `cdr` field. The flag `enable-tm` disables this mechanism.

If the type is realized as a class, it needs the right header files included. Every type that needs special compiler support must name its header files in `type-support`. All strings in this list, associated to the current type, are added to the prefix code.

⟨emit-id functions 84c⟩ ≡

```
;;; typename identifier
(define emit-typename
  (lambda (val)
    (define typename "")
    (define pclist '())
    ; get typename
    (if enable-tm
        (begin
          (set! typename (dassoc (car val) type-map)) ; loop up type-map
          (if (eq? #f typename)
              (set! typename (car val)) ; no match found
              ; get string if node is also stored
              (if (pair? typename) (set! typename (car typename))) ))
        (set! typename (car val)) )
    ; add prefix code of types
    (set! pclist (dassoc typename type-support))
    (if (not (eq? #f pclist))
        (do ((i 0 (+ i 1))) ((= i (length pclist)))
          (add-prefix-code! (list-ref pclist i)) ))
    typename))
◇
```

Macro defined by scraps 84abc.
Macro referenced in scrap 82d.

4.2.3 Code Emitters for Definitions

We have either variable or function definitions. The first ones are converted to simple assignments, but SuchThat algorithms hide behind function definitions.

4.2.3.1 The `emit-var` Function

Formal description: `(emit-var` *val*`)`
val – the value field of a variable definition node (see figure 3.4)
return value: a code string with an equivalent of an assignment in the target language

Variable definitions are reduced to assignments, because the declarations take over the definition part. The semantics of the assignment is defined by the overloaded operator '`=`'.

⟨emit-definition functions 85a⟩ ≡

```
;;; emit code string for variable definition
(define emit-var
  (lambda (val)
    (emit-strin (str-seq (generate-code (car val)) " = "
                         (emit-expr (cdr (cadr val))) ";")) ))
```

◇

Macro defined by scraps 85ab, 88a.
Macro referenced in scrap 82d.

4.2.3.2 The `emit-func` Function

Formal description: `(emit-func` *val*`)`
val – the value field of a function definition node (see figure 3.5)
return value: a code string with an equivalent of a function definition in the target language

Generating code for algorithms is one of the major tasks of the compiler. First we check, if generic types reside inside the symbol tables. The tables are retrieved from the node value. The list of generic types is stored in `gen-list`. If this list is not empty, a template header is created. Now the function signature is produced, followed by the function body. At the beginning of the body all local variables are declared.

⟨emit-definition functions 85b⟩ ≡

```
;;; emit code string for function definition
(define emit-func
  (lambda (val)
    (define tmp "")
    ;(define out "")
    (define gen-list '())
    ; read symbol tables
    (set! input-symtab (cdr (list-ref val 1)))
    (set! output-symtab (cdr (list-ref val 2)))
    (set! local-symtab (cdr (list-ref val 3)))
    ; find generic types
    (set! gen-list (merge-gentypes gen-list input-symtab))
    (set! gen-list (merge-gentypes gen-list output-symtab))
    (set! gen-list (merge-gentypes gen-list local-symtab))
    (str-seq
     ; create code string
     ⟨create template header 86a⟩
     ⟨create function signature 86b⟩
     ; function body
     (emit-strn ") {")
     (ind-inc!)
```

```
        ⟨create local variable declarations 87a⟩
        ⟨create single output parameter 87b⟩
        (generate-code (list-ref val 5)) ; body
        (ind-dec!)
        (emit-strin "}") ) ))
    ◇
```

Macro defined by scraps 85ab, 88a.
Macro referenced in scrap 82d.

We create a template header for functions that have generic datatypes in their symbol tables. The header lists all these generic typenames and thereafter they can be used in the function signature as well as in the body. For example, the generic types `Ring` and `Field` in the symbol tables will lead to this header:

```
template< class Ring, class Field >
```

⟨create template header 86a⟩ ≡

```
    (if (> (length gen-list) 0)
        (str-seq
         (emit-stri "template<")
         (do ((i 0 (+ i 1)) (tp "")) ((= i (length gen-list)) tp)
           (set! tp (str-seq tp "class "
                             (generate-code (list-ref gen-list i)) ))
           (if (not (= i (- (length gen-list) 1)))
               (set! tp (str-seq tp ", ")) ))
         (emit-strn ">")) "")
    ◇
```

Macro referenced in scrap 85b.

The return type is `void` if the number of output parameters is zero or greater than one, otherwise it gets the single output parameter's type. In the latter case we also declare the single output parameter, together with the local variables, in order to introduce its identifier. The return type is of reference type by default, only types inside `val-list` don't get the '`&`' reference sign. Then the function name is generated, this is either a normal identifier or a legal C^{++} operator, prefixed with the keyword `operator`. Finally we scan the formal parameter list of the function definition. Input parameters get a `const` modifier, if they are declared as references.

⟨create function signature 86b⟩ ≡

```
    ; function return type
    (if (= (length output-symtab) 1)
        (begin
          (set! tmp (emit-type (cdr (cadr (list-ref output-symtab 0)))))
          (str-seq (emit-stri tmp)
                   (if (list-memb? tmp val-list) " " " & ")) )
        (emit-stri "void ") )
    ; function name
    (generate-code (car val)) "("
    ; parameter list
    (do ((i 0 (+ i 1)) (pl "")) ((= i (length (list-ref val 4))) pl)
      (set! tmp (get-idtype (get-id-tname (list-ref (list-ref val 4) i))))
      (if (not (null? tmp))
          (cond ((equal? (cdr tmp) "input")
                 (set! pl (str-seq
                           pl (if (list-memb? (car tmp) val-list) "" "const ")
                           (car tmp)
                           (if (list-memb? (car tmp) val-list) " " " & ")
                           (get-id-tname (list-ref (list-ref val 4) i))
                           (if (not (= i (- (length (list-ref val 4)) 1)))
                               ", " "") )))
                ((equal? (cdr tmp) "output")
                 (set! pl (str-seq
```

```
                    pl (car tmp) " & "
                    (get-id-tname (list-ref (list-ref val 4) i))
                    (if (not (= i (- (length (list-ref val 4)) 1)))
                        ", " "") )))
          (else ⟨error: bad function parameter 87c⟩) )
     ⟨error: undefined parameter 87d⟩) )
◇
```

Macro referenced in scrap 85b.

All local variables are declared in front of the function body. This is done by iterating over the local symbol table and emitting the new identifier, preceded by its type.

⟨create local variable declarations 87a⟩ ≡

```
; declare local variables
(do ((i 0 (+ i 1)) (lv "")) ((= i (length local-symtab)) lv)
  (set!
   lv (str-seq ; variable type
       lv (emit-stri (emit-type (cdr (cadr (list-ref local-symtab i)))))
       " " (generate-code (car (list-ref local-symtab i))) ; variable name
       (emit-strn ";") )))
◇
```

Macro referenced in scrap 85b.

Types with call-by-value semantics are no problem. But we have to pay attention to single output parameters with call-by-reference semantics. A new variable has to be allocated on the heap, which is done with C++'s `new` operator. The freshly allocated variable is a pointer and we have to dereference it with the operator '`*`'. The user of functions that allocate objects on the heap must free them explicitly. The alternative to this memory management strategy is garbage collection. See [31], p. 465, for a brief discussion on

⟨create single output parameter 87b⟩ ≡

```
; single output parameter
(if (= 1 (length output-symtab))
    (begin (set! tmp (emit-type (cdr (cadr (list-ref output-symtab 0)))))
           (str-seq
            (emit-stri tmp) " "
            (generate-code (car (list-ref output-symtab 0)))
            (if (not (list-memb? tmp val-list)) ; reference type
                (str-seq " = new " tmp) "")
            (emit-strn ";") ))
    "") ; no single output parameter
◇
```

Macro referenced in scrap 85b.

Error and Warning Messages

⟨error: bad function parameter 87c⟩ ≡

```
(begin (error-mes "function parameter must be either input or output")
       (debug-mes 2 " identifier scope: " (cdr tmp) "")
       (debug-mes 2 " identifier: " (list-ref (list-ref val 4) i) "") )◇
```

Macro referenced in scrap 86b.

⟨error: undefined parameter 87d⟩ ≡

```
(begin (error-mes "undefined identifier, not found in symbol tables")
       (debug-mes 2 " identifier: " (list-ref (list-ref val 4) i) "") )◇
```

Macro referenced in scrap 86b.

4.2.3.3 The `emit-op` Function

The only difference of an operator function definition is the prepended keyword `operator`. So we just do this to the identifier field and call `emit-func`, if the operator is supported by the back end.

⟨emit-definition functions 88a⟩ ≡

```
; emit code string for operator function definition
(define emit-op
  (lambda (val)
    (if (list-memb? (string->symbol (generate-code (car val))) op-list)
        ; identifier name = operator prefixed with keyword "operator"
        (emit-func
         (append (list (get-ptn (make-ptn 'opname-id)
                                (list (str-seq "operator"
                                               (generate-code (car val)))
                                      (get-id-oname (car val)))))
                 (cdr val)) ) ; rest of original value field
        (emit-func val) ))) ; op. not in target language -> standard function
```

◇

Macro defined by scraps 85ab, 88a.
Macro referenced in scrap 82d.

4.2.4 Code Emitters for Statements

These code emitters map parse tree nodes to the back end target language constructs. Most of them are trivial and the descriptions will be brief.

4.2.4.1 The `emit-set!` Function

Formal description: `(emit-set!` *val*`)`
val – the value field of a set! node (see figure 3.6)
return value: a code string with an equivalent of the Scheme `set!` statement in the target language

The `set!` statement is converted into an assignment. The semantics of the assignment depends on the current overloading of the assignment operator.

⟨emit-statement functions 88b⟩ ≡

```
; emit code string for set! statement
(define emit-set!
  (lambda (val)
    (emit-strin (str-seq (generate-code (car val)) " = "
                         (emit-expr (cdr (cadr val))) ";")) ))
```

◇

Macro defined by scraps 88bc, 89ab, 90a, 91a, 92ab, 93ab, 94.
Macro referenced in scrap 82d.

4.2.4.2 The `emit-let` Function

Formal description: `(emit-let` *val*`)`
val – the value field of a `let` node (see figure 3.7)
return value: a code string with an equivalent of the Scheme `let` statement in the target language

We spill out all local variable initializations, which are encoded in a list of `set!` statements. Then the `let` body is generated by calling `generate-code`.

⟨emit-statement functions 88c⟩ ≡

```
; emit code string for let statement
(define emit-let
  (lambda (val)
    (str-seq
     ; local variable initializations
     (do ((i 0 (+ i 1)) (out-str ""))
         ((= (length (car val)) i) out-str)
       (set! out-str (str-seq out-str
                        (generate-code (list-ref (car val) i)))) )
     ; let body
     (generate-code (cadr val)) ) ))
◇
```

Macro defined by scraps 88bc, 89ab, 90a, 91a, 92ab, 93ab, 94.
Macro referenced in scrap 82d.

4.2.4.3 The `emit-begin` Function

Formal description: `(emit-begin` *val*`)`
val – the value field of a `begin` node (see figure 3.11)
<u>return value:</u> a code string with an equivalent of the Scheme `begin` statement in the target language

The node value of a `begin` statement is a parse tree node that contains its body. We pass it on to the `generate-code` function.

⟨emit-statement functions 89a⟩ ≡

```
; emit code string for begin statement
(define emit-begin
  (lambda (val)
    (generate-code val) ))
◇
```

Macro defined by scraps 88bc, 89ab, 90a, 91a, 92ab, 93ab, 94.
Macro referenced in scrap 82d.

4.2.4.4 The `emit-if` Function

Formal description: `(emit-if` *val*`)`
val – the value field of an `if` node (see figure 3.8)
<u>return value:</u> a code string with an equivalent to the Scheme `if` statement in the target language

C++ needs the `if` statement's conditional expression in parentheses, so we have to add them explicitly. This sometimes results in redundant parentheses, but we can't predict if an operator is really an operator or will be mapped to a function. The consequent and optional alternate are properly indented.

⟨emit-statement functions 89b⟩ ≡

```
; emit code string for if statement
(define emit-if
  (lambda (val)
    (str-seq
     ; if condition
     (emit-strin (str-seq "if (" (emit-expr (cdr (car val))) ") {"))
     ; consequent
     (ind-inc!) (generate-code (cadr val)) (ind-dec!)
     (emit-strin "}")
     ; alternate
     (if (not (null? (caddr val)))
         (str-seq
```

```
               (emit-strin "else {")
               (ind-inc!) (generate-code (caddr val)) (ind-dec!)
               (emit-strin "}"))
              "") ) ))
```
◇

Macro defined by scraps 88bc, 89ab, 90a, 91a, 92ab, 93ab, 94.
Macro referenced in scrap 82d.

4.2.4.5 The `emit-case` Function

> **Formal description:** `(emit-case` *val*`)`
>
> *val* – the value field of a `case` node (see figure 3.9)
>
> return value: a code string with an equivalent of the Scheme `case` statement in the target language

We convert the Scheme `case` statement into C++'s `switch` statement. The `case` key is used as the `switch` expression. All alternates make up the `switch` branches. Finally we have to test if an `else` branch exists and expand its body behind a `default:` label.

⟨emit-statement functions 90a⟩ ≡
```
; emit code string for case statement
(define emit-case
  (lambda (val)
    (str-seq
     ; case key
     (emit-strin (str-seq "switch (" (emit-expr (cdr (car val))) ") {"))
     ; alternates
     ⟨create switch branches 90b⟩
     ; else branch
     (if (not (null? (caddr val)))
         (str-seq
          (emit-strin "default:")
          (ind-inc!) (generate-code (caddr val)) (ind-dec!)) )
     (emit-strin "}") ; end of switch
     )))
```
◇

Macro defined by scraps 88bc, 89ab, 90a, 91a, 92ab, 93ab, 94.
Macro referenced in scrap 82d.

Every `case` clause consists of the datums to compare against the `case` key and its body of expressions. After expanding the datum list, the body is converted into a code string, followed by a `break` statement.

⟨create switch branches 90b⟩ ≡
```
(do ((i 0 (+ i 1)) (out-str "") (l (cadr val)))
    ((= i (length l)) out-str)
  (set! out-str (str-seq out-str
                         ⟨expand datum list 90c⟩
                         (ind-inc!)
                         ; alternate
                         (str-seq (generate-code (cdr (list-ref l i)))
                                  (emit-strin "break;"))
                         (ind-dec!)) ))
```
◇

Macro referenced in scrap 90a.

The datum list is converted into a sequence of "`case` *constant-expression*`:`" statements, as described in [31], p. 510.

⟨expand datum list 90c⟩ ≡

```
(do ((j 0 (+ j 1)) (str "") (exprs (car (list-ref l i))))
    ((= (length exprs) j) str)
  (set! str (str-seq str (emit-stri "case ")
                     (emit-expr (cdr (list-ref exprs j)))
                     (emit-strn ":") )))
◇
```

Macro referenced in scrap 90b.

4.2.4.6 The `emit-cond` Function

Formal description: (`emit-cond` *val*)
val – the value field of a `cond` node (see figure 3.10)
return value: a code string with an equivalent of the Scheme `cond` statement in the target language

The Scheme `cond` statement is easily mapped to C++'s `if` statement ([31], p. 510), with optional `else if` and `else` branches.

⟨emit-statement functions 91a⟩ ≡

```
; emit code string for cond statement
(define emit-cond
  (lambda (val)
    (str-seq
     ; cond clauses
     ⟨expand cond clauses 91b⟩
     ; else clause
     ⟨expand cond else clause 91c⟩) ))
◇
```

Macro defined by scraps 88bc, 89ab, 90a, 91a, 92ab, 93ab, 94.
Macro referenced in scrap 82d.

In order to expand the `cond` clauses, the first node value field is scanned element by element. An appropriate `if` or `else if` block is constructed with the clause's test expression and body. We only have to pay attention to the fact that the first C++ block starts with `if` and the other ones with `else if`.

⟨expand cond clauses 91b⟩ ≡

```
(if (> (length (car val)) 0)
    (do ((i 0 (+ i 1)) (out-str "") (tmp '()))
        ((= (length (car val)) i) out-str)
      (set! tmp (list-ref (car val) i))
      (set! out-str (str-seq
                     out-str
                     (emit-strin (str-seq
                                  (if (= i 0) "if (" "else if (")
                                  (emit-expr (cdr (car tmp))) ; clause expr.
                                  ") {"))
                     ; clause body
                     (ind-inc!) (generate-code (cdr tmp)) (ind-dec!)
                     (emit-strin "}") )) )
    "") ; no conditions, only else clause
◇
```

Macro referenced in scrap 91a.

The `else` clause returns its body without any decorations if it's the only clause, otherwise it's wrapped inside an `else` block.

⟨expand cond else clause 91c⟩ ≡

```
(if (not (null? (cadr val)))
    (cond ((null? (car val)) ; else is only clause
           (generate-code (cadr val)) )
          (else
           (str-seq
            (emit-strin "else {")
            (ind-inc!) (generate-code (cadr val)) (ind-dec!)
            (emit-strin "}")) ))
    "") ; no else clause
◇
```

Macro referenced in scrap 91a.

4.2.4.7 The `emit-return` Function

> **Formal description:** `(emit-return` *val*`)`
>
> *val* – the value field of a `return` node (see figure 3.15)
>
> <u>return value:</u> a code string with an equivalent of the SuchThat `return` macro in the target language

The mapping of the `return` macro to C++ is trivial, we convert the `return` expression with `emit-expr` and add the C++ keyword `return`.

⟨emit-statement functions 92a⟩ ≡

```
; emit code string for return macro
(define emit-return
  (lambda (val)
    (str-seq
     (emit-stri "return ")
     (emit-expr (cdr val))
     (emit-strn ";") )))
◇
```

Macro defined by scraps 88bc, 89ab, 90a, 91a, 92ab, 93ab, 94.
Macro referenced in scrap 82d.

4.2.4.8 The `emit-while` Function

> **Formal description:** `(emit-while` *val*`)`
>
> *val* – the value field of a `while` node (see figure 3.12)
>
> <u>return value:</u> a code string with an equivalent of the SuchThat `while` macro in the target language

C++ provides the `while` statement ([31], p. 511), so not much work has to be done. We create a `while` wrapper and expand the body with `generate-code`.

⟨emit-statement functions 92b⟩ ≡

```
; emit code string for while macro
(define emit-while
  (lambda (val)
    (str-seq
     (emit-stri "while (")
     (emit-expr (cdr (car val)))
     (emit-strn ") {")
     (ind-inc!) (generate-code (cadr val)) (ind-dec!) ; body
     (emit-strin "}") )))
◇
```

Macro defined by scraps 88bc, 89ab, 90a, 91a, 92ab, 93ab, 94.
Macro referenced in scrap 82d.

4.2.4.9 The `emit-repeat` Function

> **Formal description:** (`emit-repeat` *val*)
>
> *val* – the value field of a `repeat` node (see figure 3.13)
>
> return value: a code string with an equivalent of the SuchThat `repeat` macro in the target language

The `repeat` macro finds it equivalent in the `do` statement. The conversion runs analog to the `while` macro, but the succession of expression and body expansions is reversed.

⟨emit-statement functions 93a⟩ ≡

```
; emit code string for repeat macro
(define emit-repeat
  (lambda (val)
    (str-seq
     (emit-strin "do {")
     (ind-inc!) (generate-code (car val)) (ind-dec!) ; body
     (emit-stri "} while (!")
     (emit-expr (cdr (cadr val)))
     (emit-strn ");") )))
```

◇

Macro defined by scraps 88bc, 89ab, 90a, 91a, 92ab, 93ab, 94.
Macro referenced in scrap 82d.

4.2.4.10 The `emit-for1` Function

> **Formal description:** (`emit-for1` *val*)
>
> *val* – the value field of a `for1` node (see figure 3.14)
>
> return value: a code string with an equivalent of the SuchThat `for1` macro in the target language

The `for1` macro is a special case of the general `for` statement. The counter represented by the variable identifier is increased by 1, until the final value is reached. This behavior can be mapped straight to C^{++}. The counter type must implement the postfix operator '`++`', because it's used to increase the counter.

⟨emit-statement functions 93b⟩ ≡

```
; emit code string for for1 macro
(define emit-for1
  (lambda (val)
    (str-seq
     (emit-stri "for (")
     (generate-code (car val)) " = " ; counter name
     (emit-expr (cdr (cadr val))) "; " ; initial value
     (generate-code (car val)) " <= " ; counter name
     (emit-expr (cdr (caddr val))) "; " ; final value
     (generate-code (car val)) (emit-strn "++) {") ; inc. counter
     (ind-inc!) (generate-code (cadddr val)) (ind-dec!) ; body
     (emit-strin "}") )))
```

◇

Macro defined by scraps 88bc, 89ab, 90a, 91a, 92ab, 93ab, 94.
Macro referenced in scrap 82d.

4.2.4.11 The `emit-for` Function

> **Formal description:** (`emit-for` *val*)
>
> *val* – the value field of a `for` node (see figure 3.14)
>
> return value: a code string with an equivalent of the SuchThat `for` macro in the target language

The `for` macro is the only one that can't be transformed to C++ without any supplementary code. To illustrate the migration from the Scheme macro to C++, we give a brief example.

```
(for i 10 7 -12
     (display i) (display " ") )
(newline)
```

This tiny code fragment produces the following output:

```
10 7 4 1 -2 -5 -8 -11
```

We perform this transformation to achieve the same result in C++:

```
{
  counter_type _d_ = (7 - 10);
  counter_type _s_ = 1;
  if (_d_ < 0) _s_ = -1;
  for ( i = 10; (i * _s_) <= (-12 * _s_); i += _d_) {
    cout << i << " ";
  }
  cout << endl;
}
```

The variable `_d_` stores the step width, which is computed by subtracting the initial value from the second one. Thus the counter can go up or down and we store the direction of the counter in `_s_`: +1 if it goes up, -1 if it goes down. `_s_` is multiplied with the counter and the final value to get an universal comparison for both directions of the counter. The same technique is applied in the SuchThat macro `for` (see appendix C). Both `_d_` and `_s_` get the type of the counter variable used in the `for` statement.

⟨emit-statement functions 94⟩ ≡

```
; emit code string for for macro
(define emit-for
  (lambda (val)
    (define ctype (get-idtype (get-id-tname (car val)))) ; type of counter
    (if (null? ctype)
        (begin ⟨error: counter variable not declared 95a⟩
               (set! ctype "--err--"))
        (set! ctype (car ctype)))
    (str-seq
     (emit-strin "{")
     (let ((tstr ""))
       (begin
         (ind-inc!)
         (set! tstr (str-seq
                     ; ctype _d_ = b - a;
                     (emit-stri ctype) " _d_ = " (emit-expr (cdr (caddr val)))
                     " - " (emit-expr (cdr (cadr val))) (emit-strn ";")
                     ; ctype _s_ = 1;
                     (emit-stri ctype) (emit-strn " _s_ = 1;")
                     (emit-strin "if (_d_ < 0) _s_ = -1;")
                     ; for header
                     (emit-stri "for (")
                     ; initial value
                     (generate-code (car val)) " = "
                     (emit-expr (cdr (cadr val))) "; "
                     ; end condition
                     (generate-code (car val)) " * _s_ <= _s_ * "
                     (emit-expr (cdr (cadddr val))) "; "
                     ; increase statement
                     (generate-code (car val))
```

```
                    (emit-strn " += _d_) {") ; inc. counter
                    ; for body
                    (ind-inc!) (generate-code (list-ref val 4)) (ind-dec!)
                    (emit-strin "}"))) (ind-dec!)) tstr)
        (emit-strin "}") )))
```

◇

Macro defined by scraps 88bc, 89ab, 90a, 91a, 92ab, 93ab, 94.
Macro referenced in scrap 82d.

Error and Warning Messages

⟨error: counter variable not declared 95a⟩ ≡

```
(begin (error-mes "counter variable not declared")
       (debug-mes 2 " variable: " (get-id-tname (car val)) "") )◇
```

Macro referenced in scrap 94.

4.2.5 Code Emitters for Literals

The code emitting functions for literals are very similar and are therefore described all together.

Formal description: (`emit-`*literal val*)
val – the literal's string representation
<u>return value:</u> a string representing the literal, according to target language syntax

Character and string literals are wrapped in `'` and `"` characters, respectively. The other literals can return their string representation unchanged.

⟨emit-literal functions 95b⟩ ≡

```
(define emit-bool (lambda (val) val ))
(define emit-char (lambda (val) (str-seq "'" val "'") ))
(define emit-string (lambda (val) (str-seq "\"" val "\"") ))
(define emit-int (lambda (val) val ))
(define emit-rational (lambda (val) val ))
(define emit-real (lambda (val) val ))
(define emit-complex (lambda (val) val ))
◇
```

Macro referenced in scrap 82d.

The emitters for rational and complex literals may be subject to changes when a C^{++} support library is plugged in.

4.2.6 The `emit-expr` Function

Formal description: (`emit-expr` *val*)
val – the value field of an expression node (see figure 3.17)
<u>return value:</u> a string representing the expression node

To emit an expression is a little bit more complex. We therefore sketched the algorithm in a flowchart, see figure 4.2.

Literals and variable identifiers are handled first. They pose no problem, we call `generate-code` with the expression's first node value, which is either a literal or a variable identifier node. Otherwise we have to deal with an operator expression or a function call.

⟨emit-expr function 95c⟩ ≡

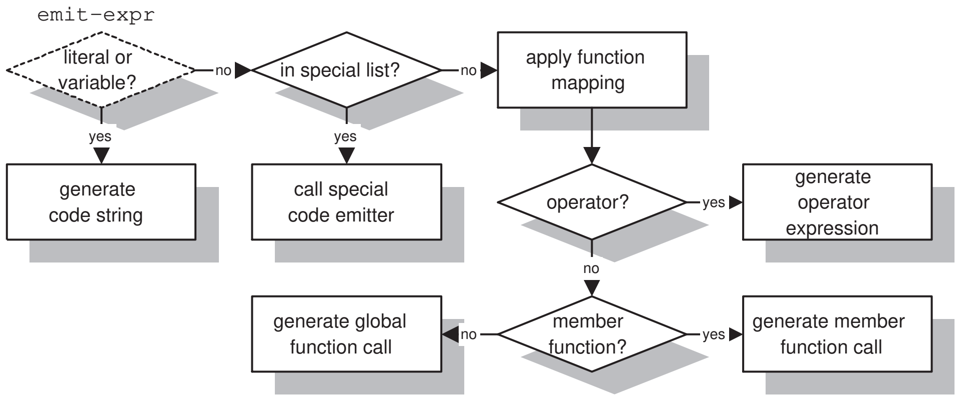

Figure 4.2: flowchart of the `emit-expr` function

```
(define emit-expr
  (lambda (val)
    (define expr-name "")
    (if (or (literal? (car val)) (var-id? (car val)))  ; literal or variable
        (generate-code (car val))
        ; function call
        ⟨function call or operator expression 96a⟩) ))
◇
```

Macro referenced in scrap 82d.

Now we apply the function mapping table `func-map` to the identifier of the operator or function. After that, we can really decide if we have to deal with a function call or an operator expression. A global function call consists of the function name and the parameter list, enclosed in parentheses. A member function call uses C++'s dot notation.

⟨function call or operator expression 96a⟩ ≡

```
; special function check with original identifier name
(if (dassoc (get-id-oname (car val)) sfunc-list)
    ; call special code emitter
    ((dassoc (cadr (cdr (car val))) sfunc-list) (cadr val))
    ; otherwise resolve operator or function expression
    (begin
      ; get real expression name
      (set! expr-name (dassoc (generate-code (car val)) func-map))
      (if (eq? expr-name #f) (set! expr-name (generate-code (car val))))
      (if (list-memb? (string->symbol expr-name) op-list)
          ⟨emit operator expression 96b⟩
          (if (list-memb? expr-name member-funcs)
              ⟨emit member function call 97a⟩
              ; global function call
              (str-seq expr-name "(" ⟨emit parameter list 97b⟩ ")") ))))
◇
```

Macro referenced in scrap 95c.

Operators are transformed from Scheme's prefix notation into the infix notation used for C++'s binary operators. Unary operators are left as is.

⟨emit operator expression 96b⟩ ≡

```
(cond ((= (length (cadr val)) 1) ; unary operator
       (str-seq "(" expr-name " " ; op.
                (emit-expr (cdr (car (cadr val)))) ")" )) ; expr
      ((= (length (cadr val)) 2) ; binary operator
       (str-seq "(" (emit-expr (cdr (car (cadr val)))) ; first expr
```

```
                    " " expr-name " "; op.
                    (emit-expr (cdr (cadr (cadr val)))) ")" )) ; second expr
          (else "")) ; shouldn't get here
     ◇
```

Macro referenced in scrap 96a.

A member function call is composed of the object it's called with, the function name and the parameters. The function name is glued to the calling object with the typical dot notation.

⟨emit member function call 97a⟩ ≡

```
     (if (= (length (cadr val)) 0)
         (begin ⟨error: member functions needs object 97c⟩ "--err--")
         (str-seq
          ; object to call member function on
          (emit-expr (cdr (list-ref (cadr val) 0))) "." expr-name "("
          ; parameters
          (do ((i 1 (+ i 1)) (op-list ""))
              ((= (length (cadr val)) i) op-list)
            ; get expression
            (set! op-list (str-seq op-list
                                        (emit-expr (cdr (list-ref (cadr val) i)))))
            ; no comma after last expression
            (if (< i (- (length (cadr val)) 1))
                (set! op-list (str-seq op-list ", ")))) ")" ))
     ◇
```

Macro referenced in scrap 96a.

The parameter list is created by iterating over the parameter list in the second node value field. This list contains expression nodes, which are converted into strings with `emit-expr`. They are separated by commas, except for the last one.

⟨emit parameter list 97b⟩ ≡

```
     (do ((i 0 (+ i 1)) (op-list ""))
         ((= (length (cadr val)) i) op-list)
       ; get expression
       (set! op-list (str-seq op-list (emit-expr (cdr (list-ref (cadr val) i)))))
       ; no comma after last expression
       (if (< i (- (length (cadr val)) 1))
           (set! op-list (str-seq op-list ", "))) )
     ◇
```

Macro referenced in scrap 96a.

Error and Warning Messages

⟨error: member functions needs object 97c⟩ ≡

```
     (begin (error-mes "member functions needs object")
            (debug-mes 2 " function: " expr-name "") )◇
```

Macro referenced in scrap 97a.

4.2.7 The `emit-type` Function

Formal description: (`emit-type` *val*)
val – the value field of a type node (see figure 3.20)
return value: a string representing the type node

Type nodes only appear in symbol tables, therefore the function `emit-type` is normally called from `emit-func`. We need the type strings in the function's signature and for defining variables.

Fundamental and generic types are flat types. We pass the first node value to `generate-code`, where the code emitter for type identifiers will be called. `emit-typename` does the type identifier mapping if necessary. Array and container types are treated separately.

⟨emit-type function 98a⟩ ≡
```
; emit type string
(define emit-type
  (lambda (val)
    ⟨type string character encoding 77d⟩
    (cond ((list-memb? fun-type (string->list (caddr val)))
           (generate-code (car val)))
          ((list-memb? gen-type (string->list (caddr val)))
           (generate-code (car val)))
          ((list-memb? arr-type (string->list (caddr val)))
           ⟨create array type string 98b⟩ )
          ((list-memb? con-type (string->list (caddr val)))
           ⟨create container type string 98c⟩) )))
◇
```
Macro referenced in scrap 82d.

The type information of an array consists of the array's type and its index ranges. The type string is created by calling `emit-type` recursively. Now all the index ranges are put between brackets[1].

⟨create array type string 98b⟩ ≡
```
(str-seq
 (emit-type (cdr (car val)))
 (do ((i 0 (+ i 1)) (t "")) ((= i (length (cadr val))) t)
   (set! t (str-seq t "["))
   (if (and (expr? (car (list-ref (cadr val) i)))
            (expr? (cdr (list-ref (cadr val) i))))
       ; dimension bounds
       (set! t (str-seq
                t (emit-expr (cdr (cdr (list-ref (cadr val) i))))
                "-" (emit-expr (cdr (car (list-ref (cadr val) i)))) "+1"))
       ; dimension size
       (set! t (str-seq t (emit-expr (cdr (list-ref (cadr val) i))))) )
   (set! t (str-seq t "]")) ))
◇
```
Macro referenced in scrap 98a.

Container types come with a list of type parameters. They are enclosed in relation signs '`< >`', preceded by the container name.

⟨create container type string 98c⟩ ≡
```
(str-seq
 (generate-code (car val)) "< "
 (do ((i 0 (+ i 1)) (t "")) ((= i (length (cadr val))) t)
   (set! t (str-seq t (emit-type (cdr (list-ref (cadr val) i)))))
   (if (not (= (- (length (cadr val)) 1) i))
       (set! t (str-seq t ", "))) )
 " >")
◇
```
Macro referenced in scrap 98a.

4.3 Manipulator Functions

Manipulator functions don't generate code directly. They work on manipulator nodes and evaluate their value fields. This leads to updated global resources. They all have their roots in SuchThat declarations. They must return empty strings, or `generate-code` will get into trouble.

⟨list of manipulator functions 98d⟩ ≡
```
⟨update-global manipulator 99a⟩
⟨update-funcmap manipulator 99d⟩
⟨update-typemap manipulator 100a⟩
◇
```
Macro referenced in scrap 81a.

[1]at the moment, index ranges given by lower and upper bound are not implemented

4.3.1 The `update-global` Manipulator

> **Formal description:** `(update-global` *val*`)`
>
> *val* – the value field of a global declaration node (see figure 3.14), which is a symbol table
>
> return value: a code string, consisting of variable declarations from the symbol table in emphval

This manipulator changes the global symbol table. All entries in the node's symbol table are inserted into `global-symtab`. This manipulator also emits code, because the global variables have to be declared.

⟨update-global manipulator 99a⟩ ≡
```
(define update-global
  (lambda (val)
    (do ((i 0 (+ i 1)) (gv "") (tmp "")) ((= (length (cdr val)) i) gv)
      ⟨check for already used global identifiers 99b⟩
      ; update symbol table
      (set! global-symtab (append global-symtab (list (list-ref (cdr val) i))))
      ; emit global variable declarations
      (set! gv (str-seq
                 ; identifier type
                 gv (emit-stri (emit-type (cdr (cadr (list-ref (cdr val) i)))))
                 ; identifier name
                 " " (generate-code (car (list-ref (cdr val) i)))
                 (emit-strn ";"))) )))
◇
```
Macro referenced in scrap 98d.

We scan the global symbol table to avoid redefinitions of already used identifiers. They would lead to compile errors within the target language compiler.

⟨check for already used global identifiers 99b⟩ ≡
```
(set! tmp (get-id-tname (car (list-ref (cdr val) i))))
(do ((j 0 (+ j 1))) ((= j (length global-symtab)))
  (if (equal? (get-id-tname (car (list-ref global-symtab j))) tmp)
      ⟨error: global identifier already used 99c⟩))
◇
```
Macro referenced in scrap 99a.

Error and Warning Messages

⟨error: global identifier already used 99c⟩ ≡
```
(begin (error-mes "global identifier already used")
       (debug-mes 2 " identifier: " tmp "") )
◇
```
Macro referenced in scrap 99b.

4.3.2 The `update-funcmap` Manipulator

> **Formal description:** `(update-funcmap` *val*`)`
>
> *val* – the value field of a function mapping declaration node (see figure 3.19)
>
> return value: an empty string

Function mappings can be changed dynamically. This manipulator changes the mapping table `func-map`. Already used mappings are replaced, and otherwise a new entry is created.

⟨update-funcmap manipulator 99d⟩ ≡

```
(define update-funcmap
  (lambda (val)
    (define lf (length func-map))
    ; get position of source name in function map
    (define pos (do ((i 0 (+ i 1)))
                    ((or (= lf i) (equal? (get-id-tname (car val))
                                          (car (list-ref func-map i)))) i)))
    (if (= pos lf) ; no entry with same name
        (set! func-map (append func-map
                               (list (cons (get-id-tname (car val))
                                           (get-id-tname (cadr val))))))
        ; remove entry
        (set! func-map (append (list (cons (get-id-tname (car val))
                                           (get-id-tname (cadr val))))
                               (list-tail func-map (+ pos 1))
                               (list-tail (reverse func-map) (- lf pos)))) )
    "" ))
◇
```

Macro referenced in scrap 98d.

4.3.3 The `update-typemap` Manipulator

Formal description: `(update-typemap` *val*`)`
val – the value field of a type mapping declaration node (see figure 3.19)
return value: an empty string

Type mappings can also be changed dynamically. This manipulator changes the mapping table `type-map`. Already used mappings are replaced or a new entry is created.

⟨update-typemap manipulator 100a⟩ ≡

```
(define update-typemap
  (lambda (val)
    (define lf (length type-map))
    ; get position of source name in type map
    (define pos (do ((i 0 (+ i 1)))
                    ((or (= lf i) (equal? (get-id-tname (car val))
                                          (car (list-ref type-map i)))) i)))
    (if (= pos lf) ; no entry with same name
        (set! type-map (append type-map
                               (list (cons (get-id-tname (car val))
                                           ⟨create type-map cdr field 100b⟩))))
        ; remove entry
        (set! type-map (append (list (cons (get-id-tname (car val))
                                            ⟨create type-map cdr field 100b⟩))
                               (list-tail type-map (+ pos 1))
                               (list-tail (reverse type-map) (- lf pos)))) )
    ⟨refresh type map 101⟩
    "" ))
◇
```

Macro referenced in scrap 98d.

When a flat type is mapped, we simply compute its string representation. Container and array types need recomputation whenever a type mapping occurs. Therefore we also save the type node, from which the string representation can be reconstructed.

⟨create type-map `cdr` field 100b⟩ ≡

```
(let ((out '()) (save enable-tm))
  (set! enable-tm #f)
  (set! out (if (> (length (cadr (cdr (cadr val)))) 0)
```

```
                    ; save type node for nested types
                    (cons (emit-type (cdr (cadr val))) ; type string
                          (cadr val)) ; value field of type node
                    ; only string necessary for flat types
                    (emit-type (cdr (cadr val)))) )
      (set! enable-tm save)
      out)
    ◇
```

Macro referenced in scrap 100a.

After a type mapping, all code strings of nested types have to be recomputed. We check this and call `emit-type` on them.

⟨refresh type map 101⟩ ≡

```
    (do ((i 0 (+ i 1))) ((= i (length type-map)))
      (if (pair? (cdr (list-ref type-map i)))
          (set-car! (cdr (list-ref type-map i))
                    (emit-type (cdr (cdr (cdr (list-ref type-map i)))))) ))
    ◇
```

Macro referenced in scrap 100a.

Chapter 5

Miscellaneous **Scheme** Procedures and Functions

In this chapter we describe all the auxiliary functions and procedures that are being used throughout the whole program, but haven't been discussed on their own.

5.1 Identifier Transformation

Now we actually do the identifier transformation by applying the identifier mapping table `id-map` to Scheme symbols. The function `id-transform` is used wherever an identifier is needed: assignments, definitions, expressions ...

`"src/misc.scm"` 102a ≡
    ⟨identifier transformation 102b⟩◇

File defined by scraps 102a, 104a, 107a, 109c, 111b, 113b.

Formal description: `(id-transform` *symb*`)`

symb – Scheme symbol representing an identifier complying to the specification in section 1.1

return value: a string with the transformed identifier for proper usage in the target language

First of all, the symbol is converted into a string. The resulting string is transformed in two passes:

pass one: reproduce capital letters if Scheme input is case insensitive

pass two: replace all characters of the string that appear in the identifier mapping table with the corresponding form

⟨identifier transformation 102b⟩ ≡

```
;;; transform identifier from scm interface to target language
(define id-transform
  (lambda (symb)
    (define out "")
    (if (symbol? symb)
        (begin
          (set! out (symbol->string symb))
          ; two pass transformation
          ⟨reproduce capital letters 103a⟩
          ⟨character mapping 103b⟩
          out) ; return value
        (begin
```

```
            ⟨error: wrong id-transform argument 103d⟩
            "--err--") ) )) ; error identifier
   ◇
```
Macro referenced in scrap 102a.

Capital letters are reproduced with the help of the case character '#'. The algorithm is very easy. Every occurrence of a case characters preceded by an alphabetic character is replaced by the upper case version of the alphabetic character.

⟨reproduce capital letters 103a⟩ ≡
```
   (if (not enable-case-sensitive)
       (do ( (i 0 (+ i 1)) (tmp "") )
           ((= i (string-length out)) (set! out tmp))
         (if (char-alphabetic? (string-ref out i))
             (if (and (< i (- (string-length out) 1))
                      (char=? #\# (string-ref out (+ i 1))))
                 ; reproduce capital letter
                 (begin
                   (set! tmp (string-append
                              tmp (string (char-upcase (string-ref out i)))))
                   (set! i (+ i 1)))
                 ; force lower case
                 (set! tmp (string-append
                            tmp (string (char-downcase (string-ref out i))))))
             ; just copy character
             (set! tmp (string-append
                        tmp (string (string-ref out i))))
             )))
   ◇
```
Macro referenced in scrap 102b.

In the second pass we simply scan the output string of pass one for characters in the identifier mapping table.

⟨character mapping 103b⟩ ≡
```
   (do ( (i 0 (+ i 1)) (tmp "") (str "") (p ⟨right form position 103c⟩) )
       ((= i (string-length out)) (set! out tmp))
     (begin
       (set! str (get-nval (string-ref out i) p id-map))
       (if (eq? str #f)
           (set! tmp (string-append tmp (string (string-ref out i))))
           (set! tmp (string-append tmp str)))
       ))
   ◇
```
Macro referenced in scrap 102b.

Matches are substituted by either the short or the long form for this particular character. We use our function `get-nval` to get the selected form. It's the second or the third element in the identifier mapping table.

⟨right form position 103c⟩ ≡
```
   (if (eq? id-map-long #f) 1 2)◇
```
Macro referenced in scrap 103b.

Error and Warning Messages An error message is produced if the object *symb* passed to the `id-transform` function is not a symbol.

⟨error: wrong id-transform argument 103d⟩ ≡
```
   (begin (error-mes "wrong argument type for id-transform, symbol expected")
          (debug-mes 2 " wrong argument: " symb "") )◇
```
Macro referenced in scrap 102b.

5.2 Parse Tree Operations

"src/misc.scm" 104a ≡
⟨parse tree operations 104b, ... ⟩◇

File defined by scraps 102a, 104a, 107a, 109c, 111b, 113b.

> **Formal description:** (`get-id-[t|o]name` *ptn*)
>
> *ptn* – a parse tree node
>
> return value: `get-id-tname`: the string representing the transformed identifier; `get-id-oname`: the original identifier symbol; `#f` if *ptn* is not an identifier node

These functions avoid the direct access of the values inside an identifier node. That's ingenious because it's done frequently and an error source.

⟨parse tree operations 104b⟩ ≡

```
(define get-id-tname
  (lambda (ptn)
    (if (and (ptn? ptn) (id? ptn))
        (car (cdr ptn)) #f)))
(define get-id-oname
  (lambda (ptn)
    (if (and (ptn? ptn) (id? ptn))
        (cadr (cdr ptn)) #f)))
```
◇

Macro defined by scraps 104bc, 105abcd, 106.
Macro referenced in scrap 104a.

5.2.1 Node Creation

To avoid nasty bugs that originate from a corrupted parse tree, we provide functions and procedures to create and modify parse tree nodes.

> **Formal description:** (`make-ptn` *node-id*)
>
> *node-id* – node identifier symbol for node type
>
> return value: a parse tree node that has the node identifier field initialized to the string from `pt-nodeids` according to *node-id*

We create a pair with `cons` and retrieve the identifier string from `pt-nodeids`. If the symbol *node-id* is not inside `pt-nodeids` the string `"PTN: unknown"` for an unknown identifier symbol is used.

⟨parse tree operations 104c⟩ ≡

```
(define make-ptn
  (lambda (node-id)
    (if (eq? (get-nval node-id 1 pt-nodeids) #f)
        ; unknown node identifier
        (begin
          (warn-mes "unknown node identifier symbol encountered")
          (cons "PTN error" '()) )
        (cons (get-nval node-id 1 pt-nodeids) '()) )))
```
◇

Macro defined by scraps 104bc, 105abcd, 106.
Macro referenced in scrap 104a.

> **Formal description:** (`get-ptn` *pt-node val*)
>
> *pt-node* – properly initialized parse tree node
> *val* – parse tree node value
>
> return value: a parse tree node that has the node value set to *val*, but keeps the node identifier from *pt-node*

Be careful when using this function. All previously stored node values are stripped from the parse tree node passed as argument.

⟨parse tree operations 105a⟩ ≡
```
(define get-ptn (lambda (pt-node val) (cons (car pt-node) val) ))
```
◇

Macro defined by scraps 104bc, 105abcd, 106.
Macro referenced in scrap 104a.

> **Formal description:** `(set-ptn-val!` *pt-node val*`)`
>
> *pt-node* – properly initialized parse tree node
> *val* – parse tree node value
>
> return value: unspecified

This procedure works just like `get-ptn`, but overwrites the node value itself. The exclamation mark indicates the function's mutating character.

⟨parse tree operations 105b⟩ ≡
```
(define set-ptn-val! (lambda (pt-node val) (set-cdr! pt-node val) ))
```
◇

Macro defined by scraps 104bc, 105abcd, 106.
Macro referenced in scrap 104a.

> **Formal description:** `(add-ptn-val!` *pt-node obj*`)`
>
> *pt-node* – properly initialized parse tree node
> *obj* – arbitrary Scheme object
>
> return value: unspecified

This procedure is useful for parse tree nodes that have lists as their node value. You can simply add an object to this list by calling the procedure with the already initialized parse tree node and the object that should be inserted into the value list. The node value is modified directly. The new object is appended at the end of the current node value list.

⟨parse tree operations 105c⟩ ≡
```
(define add-ptn-val!
  (lambda (pt-node obj)
    (set-cdr! pt-node (append (cdr pt-node) (list obj))) ))
```
◇

Macro defined by scraps 104bc, 105abcd, 106.
Macro referenced in scrap 104a.

5.2.2 Predicates for Parse Tree Node Identification

> **Formal description:** `(ptn?` *obj*`)`
>
> *obj* – any Scheme object
>
> return value: `#t` if *obj* is a parse tree node, otherwise `#f`

This predicate tests if *obj* is a parse tree node. It uses the string prefix `"PTN"` in the node's `car` field to identify a parse tree node.

⟨parse tree operations 105d⟩ ≡
```
(define ptn?
  (lambda (obj)
    (if (and (pair? obj)
             (string? (car obj))
             (>= (string-length (car obj)) 3)
             (equal? "PTN" (substring (car obj) 0 3)))
        #t #f ) ))
```
◇

Macro defined by scraps 104bc, 105abcd, 106.
Macro referenced in scrap 104a.

The next few predicates all do exactly the same. They test if the argument is a certain parse tree node. In order to obtain this information, the predicates first test if the object is a parse tree node at all. Thereafter the node identifier string is checked and the answer is returned.

Formal description: `(node-type? ptn)`

ptn – a parse tree node

return value: `#t` if *ptn* is a parse tree node of `node-type`, otherwise `#f`

⟨parse tree operations 106⟩ ≡

```
(define parse-tree?
  (lambda (ptn)
    (if (and (ptn? ptn) (equal? "PTN Parse Tree" (car ptn) ))
        #t #f) ))
(define literal?
  (lambda (ptn)
    (if (and (ptn? ptn) (>= (string-length (car ptn)) 12)
             (equal? "PTN Literal:" (substring (car ptn) 0 12)))
        #t #f) ))
(define expr?
  (lambda (ptn)
    (if (and (ptn? ptn) (equal? "PTN Expression" (car ptn) ))
        #t #f) ))
(define id?
  (lambda (ptn)
    (if (and (ptn? ptn) (>= (string-length (car ptn)) 15)
             (equal? "PTN Identifier:" (substring (car ptn) 0 15)))
        #t #f) ))
(define type-id?
  (lambda (ptn)
    (if (and (ptn? ptn) (equal? "PTN Identifier: Type" (car ptn) ))
        #t #f) ))
(define func-id?
  (lambda (ptn)
    (if (and (ptn? ptn) (equal? "PTN Identifier: Function" (car ptn) ))
        #t #f) ))
(define op-id?
  (lambda (ptn)
    (if (and (ptn? ptn) (equal? "PTN Identifier: Operator" (car ptn) ))
        #t #f) ))
(define var-id?
  (lambda (ptn)
    (if (and (ptn? ptn) (equal? "PTN Identifier: Variable" (car ptn) ))
        #t #f) ))
(define func-def?
  (lambda (ptn)
    (if (and (ptn? ptn) (equal? "PTN Definition: Function" (car ptn)))
        #t #f) ))
(define op-def?
  (lambda (ptn)
    (if (and (ptn? ptn) (equal? "PTN Definition: Operator" (car ptn)))
        #t #f) ))
(define type?
  (lambda (ptn)
    (if (and (ptn? ptn) (equal? "PTN Type" (car ptn)))
        #t #f) ))
(define symbol-table?
  (lambda (ptn)
```

```
    (if (and (ptn? ptn) (equal? "PTN Symbol Table" (car ptn)))
        #t #f) ))
◇
```

Macro defined by scraps 104bc, 105abcd, 106.
Macro referenced in scrap 104a.

5.3 Symbol Table Operations

"src/misc.scm" 107a ≡

⟨symbol table operations 107b, ... ⟩◇

File defined by scraps 102a, 104a, 107a, 109c, 111b, 113b.

Formal description: (make-symtab-entry *symtab decl-list*)
symtab – a symbol table
decl-list – a list of pairs which comply to the following structure: (*id* . *type*), where *id* is an identifier symbol and *type* is a type specifier as defined in section 1.3
return value: the updated symbol table

Whenever an entry in one of the symbol tables is made, the following function is called. After the common argument checks we create the new symbol table entries and append the resulting list to the symbol table.

⟨symbol table operations 107b⟩ ≡

```
(define make-symtab-entry
  (lambda (symtab decl-list)
    ; argument check
    (if (not (list? decl-list))
        (begin ⟨error: wrong argument for make-symtab-entry 108a⟩
               symtab) ; unchanged symbol table
        (append symtab ⟨build symbol table entries 107c⟩) )))
◇
```

Macro defined by scraps 107b, 108c, 109a.
Macro referenced in scrap 107a.

The declaration argument is a list of pairs. Each pair holds an identifier and a type. We put them one by one into the list `new-entries`.

⟨build symbol table entries 107c⟩ ≡

```
(do ((i 0 (+ i 1))
     (new-entries '()) ; result list
     (tmp '()) (pt-node '()) (entry '())) ((= i (length decl-list)) new-entries)
  (set! tmp (list-ref decl-list i))
  (cond ((and (pair? tmp) (symbol? (car tmp)))
         ; id name
         (set! pt-node (make-ptn 'vname-id))
         (set-ptn-val! pt-node `(,(id-transform (car tmp)) ,(car tmp)))
         (set! entry (list pt-node))
         ; id type
         (set! pt-node (parse-type (cdr tmp)))
         (set! entry (append entry (list pt-node)))
         ; add entry to list of new entries
         (set! new-entries (append new-entries (list entry))) )
        (else ⟨error: wrong item in declaration list 108b⟩) ))
◇
```

Macro referenced in scrap 107b.

Error and Warning Messages The argument passed to `make-symtab-entry` must be a list. Other calls result in this message.

⟨error: wrong argument for make-symtab-entry 108a⟩ ≡
```
(begin (error-mes "wrong argument for make-symtab-entry, list expected")
       (debug-mes 2 " wrong argument: " decl-list "") )◇
```
Macro referenced in scrap 107b.

⟨error: wrong item in declaration list 108b⟩ ≡
```
(begin (error-mes "wrong item in declarations of make-symtab-entry, pair expected")
       (debug-mes 2 " wrong argument: " tmp "") )◇
```
Macro referenced in scrap 107c.

> **Formal description:** (`get-idtype` *str*)
>
> *str* – the string of a transformed identifier
>
> <u>return value:</u> a pair, which `car` field is a type string and the `cdr` field is string, denoting the scope of the identifier `str`; if the identifier doesn't exist: the empty list

This function looks up an identifier's type in the symbol tables. The tables are searched in this order: input, output, local and finally the global symbol table. A single table is searched with the auxiliary function `get-idtypet`.

⟨symbol table operations 108c⟩ ≡
```
(define get-idtype
  (lambda (str)
    (define out '())
    (set! out (get-idtypet str input-symtab))
    (if (not (null? out)) (set-cdr! out "input")
        (begin
          (set! out (get-idtypet str output-symtab))
          (if (not (null? out)) (set-cdr! out "output")
              (begin
                (set! out (get-idtypet str local-symtab))
                (if (not (null? out)) (set-cdr! out "local")
                    (begin
                      (set! out (get-idtypet str global-symtab))
                      (if (not (null? out)) (set-cdr! out "global") )
                      ))))))
    out))
⟨search identifier in symbol table 108d⟩
◇
```
Macro defined by scraps 107b, 108c, 109a.
Macro referenced in scrap 107a.

This auxiliary function scans every entry in a symbol table and compares its identifier string against the given string. If a match is found, a pair with the identifier's associated type in the `car` field is created. The empty list is returned if the identifier is not a member of the symbol table.

⟨search identifier in symbol table 108d⟩ ≡
```
(define get-idtypet
  (lambda (str table)
    (do ((i 0 (+ i 1)) (tmp '()))
        ((or (= i (length table)) (not (null? tmp))) tmp)
      (if (equal? (get-id-tname (car (list-ref table i))) str)
          (set! tmp (cons (emit-type (cdr (cadr (list-ref table i))))
                          "")) )) ))
◇
```
Macro referenced in scrap 108c.

Formal description: (`merge-gentypes` *l symtab*)
l – a list of strings representing generic types
symtab – a symbol table
return value: the list, which contains the merged generic types from `symtab` and `l`

This function scans a symbol table and searches all generic types that occur inside it. These types are merged with those from the input parameter `l`.

⟨symbol table operations 109a⟩ ≡
```
(define merge-gentypes
  (lambda (l symtab)
    (do ((i 0 (+ i 1)) (tmp '())) ((= i (length symtab)) l)
      (set! tmp (gen-types (cadr (list-ref symtab i))))
       (do ((j 0 (+ j 1))) ((= j (length tmp)))
        (if (not (list-memb? (list-ref tmp j) l))
            (set! l (append l (list (list-ref tmp j)))))
        )) ))
```
⟨search generics in type node 109b⟩
◇

Macro defined by scraps 107b, 108c, 109a.
Macro referenced in scrap 107a.

This auxiliary function examines a type node's encoding. Nested types are examined recursively. Generic and fundamental types return the type identifier or the empty list, respectively.

⟨search generics in type node 109b⟩ ≡
```
(define gen-types
  (lambda (ptn)
```
⟨type string character encoding 77d⟩
```
    (define val (cdr ptn))
    (cond ((list-memb? fun-type (string->list (caddr val))) '())
          ((list-memb? gen-type (string->list (caddr val))) (list (car val)))
          ((list-memb? arr-type (string->list (caddr val))) (gen-types (car val)))
          ((list-memb? con-type (string->list (caddr val)))
           (do ((i 0 (+ i 1)) (l '())) ((= i (length (cadr val))) l)
             (set! l (append l (gen-types (list-ref (cadr val) i)))) ))
          ) ))
```
◇

Macro referenced in scrap 109a.

5.4 Various List Utility Functions

`"src/misc.scm"` 109c ≡
⟨list utility procedures 109d, ... ⟩◇

File defined by scraps 102a, 104a, 107a, 109c, 111b, 113b.

We make heavy use of associative lists in our back end. It would be a nuissance to access the `cdr` field of the returned pair from `assq`, `assv` or `assoc` "by hand" every time. We therefore define the procedures `dassq`, `dassv` and `dassoc`. They return the `cdr` field of a pair returned by the corresponding Scheme standard procedures.

Formal description: (`dass[q|v|oc]` *obj alist*)
obj – arbitrary Scheme object
alist – a list of pairs (associative list)
return value: the `cdr` field of the first pair in *alist*, whose `car` field is *obj*; `#f` if no such `car` field exists

⟨list utility procedures 109d⟩ ≡

```
;;; dassq - just return cdr field from assq
(define dassq
  (lambda (obj alist)
    (set! p  (assq obj alist)) ; to avoid second evaluation
    (if (eq? #f p) #f (cdr p)) ))
;;; dassv - just return cdr field from assv
(define dassv
  (lambda (obj alist)
    (set! p  (assv obj alist)) ; to avoid second evaluation
    (if (eq? #f p) #f (cdr p)) ))
;;; dassoc - just return cdr field from assoc
(define dassoc
  (lambda (obj alist)
    (set! p  (assoc obj alist)) ; to avoid second evaluation
    (if (eq? #f p) #f (cdr p)) ))
◇
```

Macro defined by scraps 109d, 110ab.
Macro referenced in scrap 109c.

Sometimes is useful to have *extended* alists at hand. Instead of a list of pairs it's a list of equal-length lists.

Formal description: `(get-nval` *obj pos xalist*`)`

obj – arbitrary Scheme object
pos – expression evaluating to a positive integer
xalist – a list of lists (extended associative list)

return value: the element at position *pos* of the first list in *xalist*, whose `car` field is *obj*; `#f` if no such `car` field exists

This function looks for *obj* in the `car` field of all lists that are inside *xalist*. When a match is found the object at position *p* inside this list is returned.

⟨list utility procedures 110a⟩ ≡

```
;;; get nth value out of extended alist
(define get-nval
  (lambda (obj pos xalist)
    (do ( (i 0 (+ i 1)) )
        ((cond ((= i (length xalist)) #t)
               ((equal? (car (list-ref xalist i)) obj) #t)
               (#t #f))
         (cond ((= i (length xalist)) #f)
               (#t (list-ref (list-ref xalist i) pos))
              )))))
◇
```

Macro defined by scraps 109d, 110ab.
Macro referenced in scrap 109c.

Formal description: `(list-memb?` *obj l*`)`

obj – arbitrary Scheme object
l – any Scheme list

return value: `#t`, if *obj* is a member of *l*, otherwise `#f`

The next function simply tests if an object is part of a list. The auxiliary function `flist-memb?` exists only for performance considerations. We want to avoid the argument check on consecutive function calls.

⟨list utility procedures 110b⟩ ≡

```
;;; test if object is in list
(define list-memb?
  (lambda (obj l)
    (if (not (list? l)) ; argument check
        (begin ⟨error: wrong list-memb? argument 111a⟩ #f))
    (cond ((null? l) #f)
          ((equal? (car l) obj) #t)
          (else (flist-memb? obj (cdr l))) ) ))
(define flist-memb? ; to avoid argument check
  (lambda (obj l)
    (cond ((null? l) #f)
          ((equal? (car l) obj) #t)
          (else (flist-memb? obj (cdr l))) ) ))
◇
```

Macro defined by scraps 109d, 110ab.
Macro referenced in scrap 109c.

The second argument is tested with `list?` for validity to avoid access violations.

Error and Warning Messages

⟨error: wrong list-memb? argument 111a⟩ ≡

```
(begin (error-mes "wrong second argument for list-memb? function, list expected")
       (debug-mes 2 " wrong argument: " l "") )◇
```

Macro referenced in scrap 110b.

5.5 Custom Output Routines

We use the global variable `debug-level` to control how much debugging output is generated during the compilation. The `debug-mes` procedure provides a convenient and uniform way to produce these messages.

`"src/misc.scm"` 111b ≡

⟨custom output routines 111c, ... ⟩◇

File defined by scraps 102a, 104a, 107a, 109c, 111b, 113b.

> **Formal description:** (`debug-mes` *level pre obj suf*)
>
> *level* – integer indicating at which `debug-level` the message should appear
> *pre* – prefix string of the debug message
> *obj* – arbitrary Scheme object
> *suf* – suffix string of the debug message
>
> return value: unspecified

All the procedure does is test if the `debug-level` is set to a value greater than or equal to *level*. If the condition is fulfilled the three remaining parameters are displayed in sequence and newline is added.

⟨custom output routines 111c⟩ ≡

```
(define debug-mes
  (lambda (level pre obj suf)
    (if (>= debug-level level)
        (begin
          (display pre) (display obj) (display suf)
          (newline))) ))
◇
```

Macro defined by scraps 111c, 112abd, 113a.
Macro referenced in scrap 111b.

Error and warning messages are also often used and we provide a similar interface for their generation as with debug messages. The global counters `error-cnt` and `warn-cnt` are increased with every invocation of the error or warning message display procedure.

> **Formal description:** `([warn|error]-mes` *mes*`)`
>
> *mes* – message to display
>
> return value: unspecified

⟨custom output routines 112a⟩ ≡
```
(define error-mes
  (lambda (mes)
    (set! error-cnt (+ error-cnt 1))
    (if (> debug-level 0)
        (begin
          (display "Error") ⟨display scope and algorithm 112c⟩ (display ": ")
          (display mes) (newline)) )))
◇
```
Macro defined by scraps 111c, 112abd, 113a.
Macro referenced in scrap 111b.

⟨custom output routines 112b⟩ ≡
```
(define warn-mes
  (lambda (mes)
    (set! warn-cnt (+ warn-cnt 1))
    (if (> debug-level 0)
        (begin
          (display "Warning") ⟨display scope and algorithm 112c⟩ (display ": ")
          (display mes) (newline)) )))
◇
```
Macro defined by scraps 111c, 112abd, 113a.
Macro referenced in scrap 111b.

They both display the scope at which the error occurs, possibly with the last, respectively current algorithm name.

⟨display scope and algorithm 112c⟩ ≡
```
(display (cond ((equal? scope "global")
                (if (equal? error-alg "") " at global scope"
                    (string-append " after algorithm " error-alg)))
               ((equal? scope "algorithm")
                (if (equal? error-alg "") " at algorithm scope"
                    (string-append " inside algorithm " error-alg)))) )
◇
```
Macro referenced in scraps 112ab.

> **Formal description:** `(ind-inc!)` or `(ind-dec!)`
>
> return value: an empty string

`ind-inc!` and `ind-dec!` don't create any output, but they control how the code string created with the following functions is indented. `ind-inc!` increases the indentation by `istep`, `ind-dec!` decreases it by the same amount.

⟨custom output routines 112d⟩ ≡
```
;;; increase indentation counter
(define ind-inc!
  (lambda ()
    (set! icount (+ icount istep)) "" ))
;;; decrease indentation counter
(define ind-dec!
  (lambda ()
    (set! icount (- icount istep)) "" ))
◇
```

Macro defined by scraps 111c, 112abd, 113a.
Macro referenced in scrap 111b.

Formal description: (`emit-str[i|n|in]` *line*)
line – an unformatted code string
return value: a code string

These functions do some magic with the unformatted code string *line*. `emit-strn` adds a newline character to *line*. `emit-stri` puts `icount` space characters in front of *line*. And, guess what, `emit-strin` does the compositions of both functions. The main purpose of these functions is to standardize the generation of formatted code strings.

⟨custom output routines 113a⟩ ≡

```
;;; emit string with correct indentation & newline
(define emit-strin
  (lambda (line)
    (string-append (make-string icount #\space) line "\n") ))
;;; emit string with newline
(define emit-strn
  (lambda (line)
    (string-append line "\n") ))
;;; emit string with correct indentation
(define emit-stri
  (lambda (line)
    (string-append (make-string icount #\space) line) ))
◇
```

Macro defined by scraps 111c, 112abd, 113a.
Macro referenced in scrap 111b.

5.6 Odds and Ends

`"src/misc.scm"` 113b ≡
⟨odds and ends 113c, … ⟩◇

File defined by scraps 102a, 104a, 107a, 109c, 111b, 113b.

5.6.1 Derive Output from Input File Name

Formal description: (`derive-oname` *in*)
in – string representing a Scheme file name
return value: string with file extension `.cpp`

If the input file name *in* has the extension `.scm`, it will be replaced by the extension `.cpp`. Otherwise the latter will just be appended to the input file name. Here are two examples:

- `fooname.scm` becomes `fooname.cpp`
- `NameWithoutExtension` becomes `NameWithoutExtension.cpp`

Note: the extension comparison is case insensitive.

⟨odds and ends 113c⟩ ≡

```
(define derive-oname
  (lambda (in)
    (if (and (>= (string-length in) 5)
             (string-ci=?
              (substring in (- (string-length in) 4) (string-length in))
              ".scm"))
        ; replace extension ".scm" with ".cpp"
```

```
            (string-append (substring in 0 (- (string-length in) 3)) "cpp")
            ; otherwise just append ".cpp" to file name
            (string-append in ".cpp") )))
    ◇
```

Macro defined by scraps 113c, 114ab.
Macro referenced in scrap 113b.

5.6.2 Prefix Code Operations

Formal description: `(write-prefix-code)`

return value: string with every item of `prefix-code` in a single line

`write-prefix-code` retrieves every item from `prefix-code` and appends a newline character to it. The resulting string is ready to be included in the final code.

⟨odds and ends 114a⟩ ≡

```
(define write-prefix-code
  (lambda ()
    (recurse-pc prefix-code) ))
(define recurse-pc
  (lambda (pc)
    (if (null? pc) "\n"
        (string-append (car pc) "\n" (recurse-pc (cdr pc))) )))
◇
```

Macro defined by scraps 113c, 114ab.
Macro referenced in scrap 113b.

Formal description: `(add-prefix-code!` *pc-str*`)`

pc-str – string with a complete line of code

return value: unspecified

A single line of code is appended to the list `prefix-code` with this function. The code line is only appended, if it's not already inside `prefix-code`.

⟨odds and ends 114b⟩ ≡

```
(define add-prefix-code!
  (lambda (pc-str)
    (if (not (list-memb? pc-str prefix-code))
        (set! prefix-code (append prefix-code (list pc-str)))) ))
◇
```

Macro defined by scraps 113c, 114ab.
Macro referenced in scrap 113b.

Chapter 6

Customizing the Back End

In this chapter we show how we fulfilled the promise of a configurable back end. We mentioned the two means throughout the text:

1. initializer lists
2. special functions (easily extended code emitters)

The initializer lists reside in the directory `init`, relative to the location of our back end.

6.1 The Identifier Mapping Table

In section 1.1, we described thoroughly how the transformation of SuchThat identifiers to Scheme and finally to C^{++} works. The back end uses the table `id-map.ini`. It is read in every time ScmToCpp is invoked, and the user can change it to fit his/her desires.

"init/id-map.ini" 115 ≡

```
;;; identifier mapping table
;;;
; all characters present in the interface but not in the target language
; should have an entry of the form: (char "short form" "long form")

((#\\ "_BS_" "_BackSlash_")
 (#\$ "_DS_" "_DollarSign_")
 (#\^ "_H_"  "_Hat_")
 (#\! "_EM_" "_ExclamationMark_")
 (#\% "_PS_" "_PercentSign_")
 (#\& "_A_" "_Ampersand_")
 (#\* "_S_" "_Star_")
 (#\/ "_FS_" "_ForwardSlash_")
 (#\: "_C_" "_Colon_")
 (#\< "_L_" "_Less_")
 (#\= "_E_" "_Equal_")
 (#\> "_G_" "_Greater_")
 (#\? "_QM_" "_QuestionMark_")
 (#\~ "_T_" "_Tilde_")
 (#\. "_D_" "_Dot_")
 (#\+ "_P_" "_Plus_")
 (#\- "_M_" "_Minus_")
 (#\# "_SH_" "_Sharp_")
 )
```
◇

The table contains three element lists. The first element is the character that should undergo the mapping operation. It is followed by two strings which give the short respectively long form to which the character should be mapped. Be careful to use only strings that are themselves legal identifiers in the target language. Of course we aim at C++. It's possible to use the empty string '""' as short or long form. This enables us to let some characters vanish completely from the identifier. But we must pay attention not to overuse this feature for the resulting identifiers must be unique.

6.2 Types

The list `type-list` contains all non-generic types. The alist is read from the file `type-list.ini`. It's very easy to add a new type to the list. The `car` field of an entry in `type-list` is the type name as a string, the `cdr` field is the number of type parameters. Fundamental types need no type parameters, so they get the value 0.

"init/type-list.ini" 116a ≡

```
;;; list of non-generic types: name - number of type parameters
(; fundamental types have no type parameters
 ("Boolean" . 0)
 ("Character" . 0)
 ("String" . 0)
 ("Integer" . 0)
 ("Rational" . 0)
 ("Real" . 0)
 ("Complex" . 1)
 ; container types and the number of expected type parameters
 ("List" . 1)
 ("Vector" . 1)
 ("Map" . 2)
 ⟨example: new type 124a⟩
 )
◇
```

Types that are not immanent to the target language require special support by the back end compiler. For C++, we include header files that contain the type's class interface. This is necessary for the `string` class, as well as for the STL containers `vector`, `list` and `map`. The list `type-support` is made of pairs. The `car` field of a pair holds the final type string. In the `cdr` field, a list of strings is stored. Every string in this list is a code line in the target language. They are passed to `add-prefix-code!` when the type appears in the source code.

"init/type-support.ini" 116b ≡

```
;;; list of final types and their support files
(("string" . ("#include <string>"))
 ("vector" . ("#include <vector>"))
 ("list" . ("#include <list>"))
 ("map" . ("#include <map>"))
 ⟨example: new type support 124b⟩
 )
◇
```

The standard calling convention for functions is call-by-reference. It can be overridden with the file `val-list.ini`. The entries in the list, which constitutes this file, are final type names[1]. They are handled with call-by-value semantics.

"init/val-list.ini" 116c ≡

```
;;; list of all types that should use call-by-value semantics
("bool" "char" "int" "float" "double")
◇
```

[1]after the type mapping mechanism is applied

6.3 Member Functions

New types are implemented as classes in the back end system. A lot of the type's functionality is located in its member functions. SCMTOCPP creates global functions by default. The entries of the list `member-funcs` are strings. They represent function names. These functions are converted into member function calls, overriding global function creation. The initial entries are common member functions of containers.

"init/member-funcs.ini" 117a ≡

```
;;; expand these functions to member function calls
("begin" "end" "rbegin" "rend"
 "push_front" "push_back"
 "pop_front" "pop_back"
 "list" "vector" "map"
 ⟨example: member funcs 125b⟩)
◇
```

6.4 Function and Type Mappings

Mapping declarations (see section 1.2) store their status in the lists `func-map` and `type-map`. They are initialized at compiler startup with the following two files.

6.4.1 Function Mapping

The function type map is an alist. The entries in the pairs are strings. The `car` field names the function to be mapped and the `cdr` field the mapping target. Both can be either normal function names or operator names.

We use the function map to adapt Scheme expressions to C++ syntax. Thus it's not wise to disable function mapping with the compiler switch `-fm`.

"init/func-map.ini" 117b ≡

```
;;;
;;; table of default function and operator mappings
;;;
(("not" . "!")
 ("not-eq?" . "!=")
 ("=" . "==") ("eq?" . "==") ("eqv?" . "==") ("equal?" . "==")
 ("or" . "||")
 ("and" . "&&") )
◇
```

6.4.2 Type Mapping

The type map is also an alist. The `car` field holds the transformed identifier string of the type to be mapped. The `cdr` field can be more complex. In its simple form, it contains a string. We use this form to map the SUCHTHAT default types to C++ types. The second possibility is pair, whose `car` field is a type string, as usual. The `cdr` field holds a type node. This construct is used by the back end to map identifiers to nested types. It's uncommon to write this in the initializer list by hand, but perfectly legal. We show an example on how such a pair looks like, it maps the identifier `matrix_type` to a container.

```
("matrix_type" . ("vector< vector < int > >" . (v#ector (v#ector i#nteger))) )
```

"init/type-map.ini" 117c ≡

```
;;; map for type identifier mapping
(("Boolean" . "bool")
 ("Character" . "char")
 ("String" . "string")
 ("Integer" . "int")
```

```
 ("Real" . "double")
 ("Vector" . "vector")
 ("List" . "list")
 ("Map" . "map")
)
◇
```

6.5 The Operator List

In this list we name all operators that are legal in the back end. They will not be processed by `id-transform` and may be used as function names. They are treated specially in expressions. They cause no function call but an operator expression.

It makes no sense to list all legal C++ operators, for many of them have only meaning in the context of C++'s class structure, its pointer arithmetic and other aspects that are not part of SuchThat.

"init/op-list.ini" 118a ≡

```
;;; list of supported operators
(+ - * / = += -= /= *= <<= >>= < > <= >= == != ! \% << >> ^ || &&)
◇
```

6.6 Special Functions

The second option to extend the back end are *special functions*. They are called by the code emitter for expressions (see section 4.2.6). All special functions are listed in `sfunc-list`.

⟨list of special functions' code emitters 118b⟩ ≡

```
;;; list of all function names that have a special code emitter
(define sfunc-list
  `((array-ref . ,emit-array-ref) (vector-ref . ,emit-array-ref)
    (newline . ,emit-newline)
    (display . ,emit-display)
    ))
◇
```

Macro referenced in scrap 118c.

The `car` field of an entry in this alist holds the Scheme symbol of the special function's name. The `cdr` field holds the quasiquoted special function itself.

How do special functions work? They get their only input parameter from the second element of an expression node's value field. This is a list of expression nodes. So what we're dealing with here are function calls that are expanded in an unusual way. The advantage of this approach is that we don't have to introduce new node types to handle these function calls. Special functions are really specialized code emitters. Like code emitters, they return a code string. Because they are called by `emit-expr`, we must not add a semicolon at the end.

What is the main application field of this feature? We already mentioned that data types have operations associated with them. In an object oriented environment like C++ or Java, they are called member functions or methods. Special functions serve as a handy tool to enhance our back end in this regard.

How are special functions installed? Special functions and their list `sfunc-list` reside in the file `code-ext.scm`. We need to rebuild the compiler when new functions are added. That's not really a problem, because they are extended when a new back end library is supplemented[2].

"src/code-ext.scm" 118c ≡

⟨special functions' code emitters 119a, ... ⟩
⟨list of special functions' code emitters 118b⟩
◇

[2] and this shouldn't happen once a week ...

6.6.1 Examples of Special Functions

The examples for special functions are vital for every back end. They illustrate how we drastically increase the back end's functionality with little effort.

The first example converts the SLIB function `array-ref` to C++ syntax. Not much is done, the array's name is succeeded with the indices, enclosed in brackets. Vector access looks the same in C++, we can use this function for them, too.

⟨special functions' code emitters 119a⟩ ≡
```
(define emit-array-ref
  (lambda (val)
    (string-append
     (emit-expr (cdr (car val))) ; array name
     ; array indices
     (do ((i 1 (+ i 1)) (s "")) ((= i (length val)) s)
       (set! s (string-append s "[" (emit-expr (cdr (list-ref val i))) "]")) )
     ) ))
◇
```
Macro defined by scraps 119ab.
Macro referenced in scrap 118c.

The second example enriches our back end with output abilities. The Scheme special forms `newline` and `display` are mapped to C++ output constructs. `display` is even allowed with multiple arguments, thanks to C++ output streams.

⟨special functions' code emitters 119b⟩ ≡
```
(define emit-newline
  (lambda (val)
    "cout << endl" ))
(define emit-display
  (lambda (val)
    (string-append
     "cout"
     (do ((i 0 (+ i 1)) (s "")) ((= i (length val)) s)
       (set! s (string-append s " << " (emit-expr (cdr (list-ref val i))) )) )
     ) ))
◇
```
Macro defined by scraps 119ab.
Macro referenced in scrap 118c.

Chapter 7

Looking Beyond

This paper presented a Scheme interface, which constitutes a portable SuchThat basis. It's used as the output of the type checker and as input of the code generator. We elaborated ScmToCpp, a configurable back end with C++ as target language. The extension of the back end is facilitated by its design and several build-in features.

What's next? The first thing to do is writing a back end support library such that all default types are handled adequately. This involves the implementation of SuchThat default types in C++ classes. A nice example is Sebastian Wedeniwski's multiple precision library for integers ([34]), Piologie. We might even take this one step further. Barton and Nackman proposed a class concept called *categories* ([3], p. 480), which may serve as foundation of SuchThat's mathematical structures. In other terms, the back end language enforces SuchThat's type system with special idioms. On the back end side the classes are imbedded with special functions and addition in the type list.

In its current state, the back end relies heavily on the target language compiler. The compiler does the checks for valid function overloading and if functions are declared at all. It would be more sensible to put the known functions of a back end library into a database[1] and check the functions in ScmToCpp. This might lead to an automated usage of special functions.

The mapping mechanism may be subject to subsequent changes. Growing experience with all parts of the SuchThat system will show if the rudimentary implementation is sufficient or a more sophisticated strategy is needed.

ScmToCpp is intended as a test bed of SuchThat. Therefore not much thought was put into one aspect: performance. The implementation language is one facet of this theme. Scheme enables fast development and easy testing, because it's an interpreted language. But inherent to all interpreted languages is the performance penalty. A mature SuchThat basis will fuel the development of fast, efficient implementations, maybe in C++ or with well known compiler compilers like Bison or Yacc.

The current SuchThat system does a lot of work more than once. First, the SuchThat source code is converted into Scheme, which requires lexical analysis and parsing. Next comes the type checker, which builds a symbol table. It aims at Scheme, too. Finally ScmToCpp parses the Scheme input once again, also building symbol tables and another intermediate representation, the parse tree. The melting of some phases in the SuchThat compilation could enhance the whole system significantly. For example, instead of generating Scheme output, a combination of type checker and back end parser permits the direct creation of the parse tree.

Perhaps another back end language is desirable for various reasons. Java in its current form (JDK 1.1) is no serious candidate. It lacks too much expressive power for the task of generic programming. The extensions proposed by Liskov et. al. ([2]) or Odersky and Wadler ([24]) show that not all hope is lost on Java. Two remarks in our paper hinted to another promising alternative: Ada95. It has a long tradition and the new features since its revision in 1995 puts it ahead even of C++. One of its major goals is the coupling of performance and stability. SuchThat's

[1] maybe into SuchThat's `adb`

incorporation of the specification into the program text is, to some extent, expression of the yearning for stability.

All mentioned target languages support exception handling, which is a new way of writing robust programs. New research should examine how apt exception handling is for inclusion into SuchThat.

An entirely different approach would be to generate Java bytecode instead of a high level language. The continuing Java hype grants enough support for this platform. And wouldn't it be nice to use Java's networking and multithread capabilities to explore new roads in computer algebra? The feasibility of such a project is demonstrated by two Ada95 compilers that generate Java bytecode ([12] and [1]). They even allow applets to be written in Ada95!

The long-term goal of the SuchThat project is the building of a generic library of computer algebra algorithms – WLOG. A comparable, existing system is SAC-2 ([18]). But it suffers from all the problems that non-generic programs have in common. Algorithms are written in many specializations for every combination of datatypes. For this reason, the usage of the library gets very cumbersome. The way out of this dilemma is *lifting* (see [27], p. 22). The creation of WLOG will hopefully exhibit the benefits and potency of generic programming.

Appendix A

Command Line Syntax of `scm2cpp`

The command line syntax with `+si` present: `scm2cpp [option ...]`
The command line syntax without `+si`: `scm2cpp [option ...] input-file`

option	**effect**
`-tm`	turns off type mapping
`+tm`	turns on type mapping (default)
`-fm`	turns off function and operator mapping
`+fm`	turns on function and operator mapping (default)
`-md`	disables the mapping declaration `st-map`
`+md`	enables the mapping declaration `st-map` (default)
`-li`	enables short form of identifier mapping (default)
`+li`	enables long form of identifier mapping
`-cs`	selects case insensitive Scheme implementation (default)
`-cs`	selects case sensitive Scheme implementation
`-si`	read input from file (default)
`+si`	read input from `stdin` (no `input-file` needed)
`-so`	write output to file (default)
`+so`	write output to `stdout`
`-ac`	disable ANSI C++ output
`+ac`	enable ANSI C++ output (default)
`-o` name	sets the output file name
`-dl` n	sets the verboseness level of messages to n (default: 2)
`-st` n	sets output code indentation to n (default: 2)
`-rd` dir	sets the root directory of scm2cpp (needed to find ini files)

Appendix B

Examples

All examples are tested on a Windows NT machine with the Visual C^{++} compiler, version 5.0. ScmToCpp was compiled to C with `gsc` of the Gambit-C package and the executable was produced with Visual C^{++} 5.0.

B.1 Example 1: Factorial

The first example is a rather trivial one. It defines a generic factorial function and displays the factorials from 40 to 50.

"examples/factorial.st" 123a ≡

```
Global:    Let N be a commutative ring with unit.
Algorithm: n! := factorial(n)
Input:     n ∈ N, with n ≥ 0.
Output:    n! ∈ N such that n! = ∏_{i=1}^{n} i.
Local:     i ∈ N.
(1) [init n!]
    n! := 1.
(2) [compute factorial]
    for i=1,...,n do { n! := n! · i }.
(3) return n!; stop □

Algorithm: main
Local:     i ∈ Natural.
(1) [show factorials]
    for i=40,...,50 do { display(factorial(i)); newline() }.
◇
```

This SuchThat source code will emit the following intermediate representation in our Scheme interface:

"examples/factorial.scm" 123b ≡

```
(define factorial
  (lambda (n)
    (st-input (n . n#))
    (st-output (n! . n#))
    (st-local (i . n#))
    (call-with-current-continuation
     (lambda (return)
       (let ((n! 1))
         (for1 i 1 n (set! n! (* n! i)))
         (return n!) )))))
(define main
  (lambda ()
    (st-local (i . n#atural))
```

```
        (call-with-current-continuation
         (lambda (return)
           (for1 i 40 50
                 (display (factorial i))
                 (newline)) ))))
    ◇
```

The problem with this example is the C++ `int` type. The factorials from 40 to 50 with `int`s result in overflow errors. Therefore we will try to use Sebastian Wedeniwski's `Natural` class.

How do we set up the back end to handle the new type? First, we introduce `Natural` as a new fundamental type in `type-list.ini`:

⟨example: new type 124a⟩ ≡

```
    ("Natural" . 0)◇
```

Macro referenced in scrap 116a.

The `Natural` class needs the header file `natural.h`, so we add it to the type support list:

⟨example: new type support 124b⟩ ≡

```
    ("Natural" . ("#include \"natural.h\""))◇
```

Macro referenced in scrap 116b.

The output of a compile run with ScmToCpp looks like this:

```
// File created by scm2cpp
// examples/factorial.scm -> examples/factorial.cpp

#include "natural.h"
#include <iostream>
using namespace std;

template<class N>
N & factorial(const N & n) {
  N i;
  N (*n_EM_) = new N;
  (*n_EM_) = 1;
  for (i = 1; i <= n; i++) {
    (*n_EM_) = ((*n_EM_) * i);
  }
  return (*n_EM_);
}
void main() {
  Natural i;
  for (i = 40; i <= 50; i++) {
    cout << factorial(i);
    cout << endl;
  }
}

// --- EOF by scm2cpp ---
```

The executable produces this output:

```
815915283247897734345611269596115894272000000000
33452526613163807108170062053440751665152000000000
1405006117752879898543142606244511569936384000000000
60415263063373835637355132068513997507264512000000000
2658271574788448768043625811014615890319638528000000000
119622220865480194561963161495657715064383733760000000000
5502622159812088949850305428800254892961651752960000000000
258623241511168180642964355153611979969197632389120000000000
12413915592536072670862289047373375038521486354677760000000000
608281864034267560872252163321295376887552831379210240000000000
30414093201713378043612608166064768844377641568960512000000000000
```

B.2 Example 2: Henrici/Brown Algorithm

The second example defines a generic Henrici/Brown fraction addition.

"examples/hb+.st" 125a ≡

```
Global:    let I be gcdDomain;
           let Q be FractionField of I;
           let A be AmpleSet of I.
Algorithm: t := operator+(r, s)
Input:     r, s ∈ Q such that r, s normalized.
Output:    t ∈ Q such that t = r + s and t normalized.

(1) [r = 0 or s = 0 ?]
    if r = 0 then { t := s; return t};
    if s = 0 then { t := r; return t}.
(2) [get numerators and demominators]
    r1 := num(r); r2 := denom(r);
    s1 := num(s); s2 := denom(s).
(3) [r and s ∈ I]
    if (r2 = 1 /\ s2 = 1) then { t := fract(r1 + s1, 1); return t}.
(4) [r or s ∈ I]
    if r2 = 1 then { t := fract(r1 · s2 + s1, s2); return t};
    if s2 = 1 then { t := fract(s1 · r2 + r1, r2); return t}.
(5) [general case]
    d := gcd(r2, s2);
    if d = 1 then { t := fract(r1 · s2 + r2 · s1, r2 · s2); return t};
    if d ≠ 1 then {
      r2* := r2/d; s2* := s2/d;
      t1 := r1 · s2* + s1 · r2*; t2 := r2 · s2*;
      if t1 = 0 then { t := 0; return t };
      if t1 ≠ 0 then {
        e := gcd(t1, d);
        t1* := t1/e; t2* := t2/e;
        t := fract(t1*, t2*) }} □

Algorithm: r1 := num(r)
Input:     r ∈ Q
Output:    r1 ∈ I with r1 the numerator of r.

Algorithm: r2 := denom(r)
Input:     r ∈ Q
Output:    r2 ∈ I with r2 the denominator of r.

Algorithm: r := fract(r1, r2)
Input:     r1, r2 ∈ I, with r2 ≠ 0.
Output:    r ∈ Q with r = r1/r2 and r is normalized.

Algorithm: r := operator/(r1, r2)
Input:     r1, r2 ∈ I, with r2 ≠ 0 and r2 divides r1 exactly.
Output:    r ∈ I with r = r1/r2.

Algorithm: r := gcd(r1, r2)
Input:     r1, r2 ∈ I.
Output:    r ∈ I with r is the greatest common divisor of r1 and r2.
◇
```

The input for the compiler uses function mappings for `num`, `denom` and `fract`. They are mapped to complete names, which must be implemented by the back end library at the moment. Maybe generic versions will be available in the future. `numerator` and `denominator` are typical member functions, so we register them in the inital file `member-funcs.ini`.

⟨example: member funcs 125b⟩ ≡

```
"numerator" "denominator"◇
```

Macro referenced in scrap 117a.

It's no good idea to overload the operator '+' globally, therefore we map the function name to `hb_plus`.

"examples/hb+.scm" 126 ≡

```
; input for scm2cpp
(st-map (function num numerator))
(st-map (function denom denominator))
(st-map (function fract fraction))
(st-map (function + hb_plus))
(define (operator +)
  (lambda (r s)
    (st-input (r . q#) (s . q#))
    (st-output (t . q#))
    (st-local (r1 . i#) (r2 . i#) (s1 . i#) (s2 . i#) (t1 . i#) (t2 . i#))
    (st-local (r2\star . i#) (s2\star . i#) (t1\star . i#) (t2\star . i#))
    (st-local (d . i#) (e . i#))
    (call-with-current-continuation
     (lambda (return)
       (st-map (function + +))
       (let ((t 0) (r1 0) (r2 0) (s1 0) (s2 0) (t1 0) (t2 0)
                   (r2\star 0) (s2\star 0) (t1\star 0) (t2\star 0) (d 0) (e 0))
         ; (1)
         (if (= r 0) (begin (set! t s) (return t)) )
         (if (= s 0) (begin (set! t r) (return t)) )
         ; (2)
         (set! r1 (num r))
         (set! r2 (denom r))
         (set! s1 (num s))
         (set! s2 (denom s))
         ; (3)
         (if (and (= r2 1) (= s2 1))
             (begin
               (set! t (fract (+ r1 s1) 1)) (return t)) )
         ; (4)
         (if (= r2 1)
             (begin
               (set! t (fract (+ (* r1 s2) s1) s2)) (return t)) )
         (if (= s2 1)
             (begin
               (set! t (fract (+ (* s1 r2) r1) r2)) (return t)) )
         ; (5)
         (set! d (gcd r2 s2))
         (if (= d 1)
             (begin
               (set! t (fract (+ (* r1 s2) (* r2 s1)) (* r2 s2))) (return t)) )
         (if (not (= d 1))
             (begin
               (set! r2\star (/ r2 d))
               (set! s2\star (/ s2 d))
               (set! t1 (+ (* r1 s2\star) (* s1 r2\star)))
               (set! t2 (* r2 s2\star))
               (if (= t1 0) (begin (set! t 0) (return t)) )
               (if (not (= t1 0))
                   (begin
                     (set! e (gcd t1 d))
                     (set! t1\star (/ t1 e))
                     (set! t2\star (/ t2 e))
                     (set! t (fract t1\star t2\star)) ))))
```

```
              (return t) )))))
```
◇

We get this output from a `scm2cpp` run:

```
// File created by scm2cpp
// examples/hb+.scm -> examples/hb+.cpp

#include <iostream>
using namespace std;

template<class Q, class I>
Q & hb_plus(const Q & r, const Q & s) {
  I d;
  I e;
  I r2_BS_star;
  I s2_BS_star;
  I t1_BS_star;
  I t2_BS_star;
  I r1;
  I r2;
  I s1;
  I s2;
  I t1;
  I t2;
  Q (*t) = new Q;
  (*t) = 0;
  r1 = 0;
  r2 = 0;
  s1 = 0;
  s2 = 0;
  t1 = 0;
  t2 = 0;
  r2_BS_star = 0;
  s2_BS_star = 0;
  t1_BS_star = 0;
  t2_BS_star = 0;
  d = 0;
  e = 0;
  if ((r == 0)) {
    (*t) = s;
    return (*t);
  }
  if ((s == 0)) {
    (*t) = r;
    return (*t);
  }
  r1 = r.numerator();
  r2 = r.denominator();
  s1 = s.numerator();
  s2 = s.denominator();
  if (((r2 == 1) && (s2 == 1))) {
    (*t) = fraction((r1 + s1), 1);
    return (*t);
  }
  if ((r2 == 1)) {
    (*t) = fraction(((r1 * s2) + s1), s2);
    return (*t);
  }
  if ((s2 == 1)) {
    (*t) = fraction(((s1 * r2) + r1), r2);
```

```
    return (*t);
  }
  d = gcd(r2, s2);
  if ((d == 1)) {
    (*t) = fraction(((r1 * s2) + (r2 * s1)), (r2 * s2));
    return (*t);
  }
  if ((! (d == 1))) {
    r2_BS_star = (r2 / d);
    s2_BS_star = (s2 / d);
    t1 = ((r1 * s2_BS_star) + (s1 * r2_BS_star));
    t2 = (r2 * s2_BS_star);
    if ((t1 == 0)) {
      (*t) = 0;
      return (*t);
    }
    if ((! (t1 == 0))) {
      e = gcd(t1, d);
      t1_BS_star = (t1 / e);
      t2_BS_star = (t2 / e);
      (*t) = fraction(t1_BS_star, t2_BS_star);
    }
  }
  return (*t);
}

// --- EOF by scm2cpp ---
```

This examples points out another problem for future research. The function has two type parameters: `I` and `Q`. But only `Q` is named in the signature. The C++ compiler can't deduce the right function instantiation without interference of the user. Few compilers really support the new syntax for template functions. An example application of the function `hb_plus` shows an explicit template function instantiation:

```
#include "integer.h" // Piologie Integer
#include "rational"  // simple Rational template implementation
#include "hb+.cpp"   // Henrici/Brown algorithm from scm2cpp
#include <iostream>

using namespace std;
void main() {
  Rational<Integer> p(1, 99), q(8, 99);
  cout << p << " + " << q << " = ";
  cout << hb_plus<Rational<Integer>, Integer>(p, q) << endl;
  p = Rational<Integer>(4, 3); q = Rational<Integer>(5, 8);
  cout << p << " + " << q << " = ";
  cout << hb_plus<Rational<Integer>, Integer>(p, q) << endl;
}
```

Output:

```
1/99 + 8/99 = 1/11
4/3 + 5/8 = 47/24
```

Appendix C

SuchThat Macros

These macros simulate imperative statements, taken from Aldes, in Scheme. They use Scheme's ability to derive new expression types (see [4], p. 40).

"src/stmac.scm" 129 ≡

```
;;; while macro
(define-syntax while
  (syntax-rules ()
              ((while test body ...)
               (do ()
                   ((not test))
                 body ...) )))
;;; repeat macro
(define-syntax repeat
    (syntax-rules ()
                ((repeat body ...)
                 (begin body ...
                        (do ()
                            (#f)
                          body ...)))
                ((repeat body ... until test)
                 (begin body ...
                        (do ()
                            (test)
                          body ...)))))
;;; for macro
(define-syntax for
  (syntax-rules (d s)
   ((for1 i t1 t2 t3 body ...)
    (begin (set! i t1)
           (let* ((d (- t2 t1))
                  (s (if (> d 0) 1 (- 1))))
             (while (<= (* s i) (* s t3))
                    body ... (set! i (+ i d))))
           ))))
;;; for1 macro
(define-syntax for1
  (syntax-rules ()
              ((for1  i t1 t3 body ...)
               (begin (set! i t1)
                      (while (<= i t3) body ... (set! i (+ i 1)))
                      ))))
```

◇

Appendix D

Makefile for ScmToCpp

The current makefile works with GNU `make` and Microsoft's `nmake`. The limitations of `nmake` led to some rough edges. Targets:

doc:	creates DVI documentation
ps:	creates PS documentation
scmsrc:	creates SCM source files
exe:	creates `scm2cpp` executable
all:	creates executable and full documentation
regress:	creates regression test output files
report:	creates reports of changes between output and canon files
accept:	creates canon files from previous output files
clean:	remove garbage files

"Makefile" 130 ≡

```
########################################
# Makefile for scm2cpp
########################################
# please use the appropriate directory delimiter: \ or / (NT vs. Unix)
DD = \

WEBSRC = scm2cpp.w
WEBFILES = parse.w code.w
SCMSRC = main.scm
SCMFILES = parse.scm code.scm code-ext.scm misc.scm scm2cpp.scm

EXE = scm2cpp
EXESUF = .exe

DOCDIR = doc
SRCDIR = src
EXEDIR = runtime
TSTDIR = test

# set your commands
CP = copy
RM = del
MV = move
DIFF = fc

########################################
# documentation specific part
########################################
dd = doc$(DD)

# create documentation
```

```
doc: $(dd)$(WEBSRC:.w=.dvi)
ps: $(dd)$(WEBSRC:.w=.ps)

# call dvips to create ps file
$(dd)$(WEBSRC:.w=.ps): $(dd)$(WEBSRC:.w=.dvi)
        cd $(DOCDIR)
        dvips -o $(WEBSRC:.w=.ps) $(WEBSRC:.w=.dvi)
        cd ..

# call latex to create dvi file
$(dd)$(WEBSRC:.w=.dvi): $(dd)$(WEBSRC:.w=.tex)
        cd $(DOCDIR)
        latex \nonstopmode \input $(WEBSRC:.w=.tex)
        cd ..

# run stweb on web file
$(dd)$(WEBSRC:.w=.tex): $(WEBSRC) $(WEBFILES)
        $(CP) *.w $(dd)
        cd $(dd)
        stweb -o $(WEBSRC)
        cd ..

########################################
# code specific part
########################################
link = $(SRCDIR)$(DD)parse.scm \
       $(SRCDIR)$(DD)code.scm \
       $(SRCDIR)$(DD)code-ext.scm \
       $(SRCDIR)$(DD)misc.scm \
       $(SRCDIR)$(DD)scm2cpp.scm
main = $(SRCDIR)$(DD)main.scm

gambloc = c:/programme/gambitc

# static version
gamblib = libgambc.lib
cflags =
# dynamic version
#gamblib = gambc.lib
#cflags = -D___SHARED

# make scm source (from web file)
scmsrc: $(link) $(main)
$(link) $(main): $(WEBSRC) $(WEBFILES)
        stweb -c -t $(WEBSRC)

# make c source
csrc: $(main:.scm=.c)
$(main:.scm=.c): scmsrc
        gsc $(link) $(main)

# make executable
$(EXEDIR)/$(EXE)$(EXESUF): csrc
        cl $(cflags) -MD -I$(gambloc) -o $@ \
        $(link:.scm=.c) $(main:.scm=.c) $(main:.scm=_.c) \
        $(gambloc)/$(gamblib)
        $(MV) *.obj

exe: $(EXEDIR)/$(EXE)$(EXESUF)
```

```
########################################
# regression tests
########################################
test: test$(DD)out$(DD)*.out
	gsi test$(DD)in$(DD)parse.in > test$(DD)out$(DD)parse.out
	gsi test$(DD)in$(DD)code.in > test$(DD)out$(DD)code.out
	gsi test$(DD)in$(DD)code-ext.in > test$(DD)out$(DD)code-ext.out
	gsi test$(DD)in$(DD)misc.in > test$(DD)out$(DD)misc.out
	gsi test$(DD)in$(DD)scm2cpp.in > test$(DD)out$(DD)scm2cpp.out

accept: test
	$(CP) test$(DD)out$(DD)*.out test$(DD)canon$(DD)

report: test
	echo "---------- testing unit PARSE ----------"
	$(DIFF) test$(DD)out$(DD)parse.out test$(DD)canon$(DD)parse.out
	$(DIFF) test$(DD)out$(DD)parse.out test$(DD)canon$(DD)parse.out\
		> test$(DD)report$(DD)parse.rpt
	echo "---------- testing unit CODE ----------"
	$(DIFF) test$(DD)out$(DD)code.out test$(DD)canon$(DD)code.out
	$(DIFF) test$(DD)out$(DD)code.out test$(DD)canon$(DD)code.out\
		> test$(DD)report$(DD)code.rpt
	echo "---------- testing unit CODE-EXT ----------"
	$(DIFF) test$(DD)out$(DD)code-ext.out test$(DD)canon$(DD)code-ext.out
	$(DIFF) test$(DD)out$(DD)code-ext.out test$(DD)canon$(DD)code-ext.out\
		> test$(DD)report$(DD)code-ext.rpt
	echo "---------- testing unit MISC ----------"
	$(DIFF) test$(DD)out$(DD)misc.out test$(DD)canon$(DD)misc.out
	$(DIFF) test$(DD)out$(DD)misc.out test$(DD)canon$(DD)misc.out\
		> test$(DD)report$(DD)misc.rpt
	echo "---------- testing unit SCM2CPP ----------"
	$(DIFF) test$(DD)out$(DD)scm2cpp.out test$(DD)canon$(DD)scm2cpp.out
	$(DIFF) test$(DD)out$(DD)scm2cpp.out test$(DD)canon$(DD)scm2cpp.out\
		> test$(DD)report$(DD)scm2cpp.rpt

########################################

# source code & documentation
web: src ps
all: exe web

# remove unnecessary files
clean:
	$(RM) *~ *.o *.obj *.tex
	$(RM) test\out\*
```
◇

Appendix E

Unit and Regression Testing

We follow Adrian McCarthy's scheme to test software projects ([19]). The units of our compiler match with its file structure. Every Scheme file is a single unit. A short Scheme program extracts all definitions from a Scheme file. We tailor the program's output to our needs and use it in the input files for the unit testing. The input files for the unit and regression testing are appended at the end.

"src/scm/scanfile.scm" 133a ≡

```
(define scanfile
  (lambda (file)
    (define port (open-input-file file)) ; open file
    (display "file: ") (display file) (newline)
    (do ( (obj (read port) (read port)) ) ((eof-object? obj))
      (if (eq? 'define (car obj))
          (cond ((and (list obj) ; function definition
                      (= 3 (length obj)) (pair? (caddr obj))
                      (eq? 'lambda (car (caddr obj))))
                 (display "(display \"testing function: ")
                 (display (cadr obj)) (display "\")")
                 (display " (newline)") (newline)
                 (display ";(display (")
                 (display (cadr obj)) (display " ")
                 (display (cadr (caddr obj))) (display "))")
                 (display " (newline)") (newline) )
                ; var. declaration
                ((and (list obj) (= 3 (length obj))) ; variable declaration
                 (display "(display \"testing definition: ")
                 (display (cadr obj)) (display "\")")
                 (display " (newline)") (newline)
                 (display "(display ") (display (cadr obj))
                 (display ")") (display " (newline)") (newline) ))))
    (close-input-port port) ))
(scanfile "parse.scm")
(scanfile "code.scm")
(scanfile "code-ext.scm")
(scanfile "misc.scm")
(scanfile "scm2cpp.scm")
◇
```

"src/scm/loader.scm" 133b ≡

```
(begin ⟨load source files 141⟩
       (define argv (lambda () '("scm2cpp" "examples/hb+.scm"))) )◇
```

This Scheme file is intended for experimenting with SCMTOCPP in a Scheme interpreter. It simulates a command line call by replacing the function argv. The redefinition returns a list of strings. The command line parameters must be written into this list by hand.

"test/in/parse.in" 134a ≡

```
(begin ⟨load source files 141⟩
       (display "testing function: parse-objects") (newline)
       (pp (parse-objects '())) (newline)
       (pp (parse-objects '((set! s 0) x y))) (newline)
       (display "testing function: parse-define") (newline)
       (pp (parse-define '(x 0))) (newline)
       ;(display (parse-define (obj))) (newline)
       (display "testing function: parse-let") (newline)
       (set! scope "algorithm")
       (pp (parse-let '(((i 0))))) (newline)
       (pp (parse-let '(((i 0)) x y))) (newline)
       (set! scope "global")
       (display "testing function: parse-set!") (newline)
       (pp (parse-set! '(x 1))) (newline)
       (display "testing function: parse-begin") (newline)
       (pp (parse-begin '())) (newline)
       (pp (parse-begin '(x y))) (newline)
       (display "testing function: parse-if") (newline)
       (pp (parse-if '(t c a))) (newline)
       (pp (parse-if '(t c))) (newline)
       (display "testing function: parse-case") (newline)
       (pp (parse-case '("key" (("a" "b") x y) (else z)))) (newline)
       (pp (parse-case '(#\k (else x y)))) (newline)
       (display "testing function: parse-cond") (newline)
       (pp (parse-cond '(("a" x y) (else z)))) (newline)
       (pp (parse-cond '(((t) x y z) (else a)))) (newline)
       (display "testing function: parse-while") (newline)
       (pp (parse-while '(66 (set! x 0) y z))) (newline)
       (display "testing function: parse-repeat") (newline)
       (pp (parse-repeat '(x (set! y 1) z until 4711))) (newline)
       (display "testing function: parse-for1") (newline)
       (pp (parse-for1 '(i 1 10 a b c))) (newline)
       (display "testing function: parse-for") (newline)
       (pp (parse-for '(i 1 -2 -99 a b c))) (newline)
       (display "testing function: parse-return") (newline)
       (pp (parse-return '((< x 0)))) (newline)
       (display "testing function: parse-expr") (newline)
       (pp (parse-expr '(not (/ x (- 7 a))))) (newline)
       (display "testing function: parse-global-decl") (newline)
       (pp (parse-global-decl '((x . int) (f# . i#nteger)))) (newline)
       (display "testing function: parse-map-decl") (newline)
       (pp (parse-map-decl '((function not !=)))) (newline)
       (pp (parse-map-decl '((type mtype (v#ector (l#ist t#)))))) (newline)
       (display "testing function: error-decl-scope") (newline)
       (pp (error-decl-scope '(dummy))) (newline)
       (display "testing function: parse-type") (newline)
       (pp (parse-type '(int))) (newline)
       (pp (parse-type 'i#nteger)) (newline)
       (display "testing definition: parser-list") (newline)
       (pp parser-list) (newline)
       )
◇
```

"test/in/code.in" 134b ≡

```
(begin ⟨load source files 141⟩
       (display "testing function: generate-code") (newline)
       (display (generate-code
            '("PTN Parse Tree" . (("PTN Statement: set!" .
              (("PTN Identifier: Variable" . ("s" s))
               ("PTN Expression" . (("PTN Literal: Integer" . "0") ())))))
```

```
      ("PTN Expression" . (("PTN Identifier: Variable" . ("x" x)) ()))
      ("PTN Expression" . (("PTN Identifier: Variable" . ("y" y)) ())))))
     )) (newline)
(display "testing function: emit-pt") (newline)
(display (emit-pt
     '(("PTN Statement: set!" .
       (("PTN Identifier: Variable" . ("s" s))
        ("PTN Expression" . (("PTN Literal: Integer" . "0") ()))))
       ("PTN Expression" . (("PTN Identifier: Variable" . ("x" x)) ()))
       ("PTN Expression" . (("PTN Identifier: Variable" . ("y" y)) ())))
     )) (newline)
(display "testing function: emit-expr") (newline)
(display (emit-expr
     '(("PTN Identifier: Operator" . ("-" -))
       (("PTN Expression" . (("PTN Literal: Integer" . "7") ()))
        ("PTN Expression" . (("PTN Identifier: Variable" . ("a" a)) ()))))
     )) (newline)
(display "testing function: emit-bool") (newline)
(display (emit-bool "true")) (newline)
(display "testing function: emit-char") (newline)
(display (emit-char "x")) (newline)
(display "testing function: emit-string") (newline)
(display (emit-string "string")) (newline)
(display "testing function: emit-int") (newline)
(display (emit-int "int")) (newline)
(display "testing function: emit-rational") (newline)
(display (emit-rational "1/2")) (newline)
(display "testing function: emit-real") (newline)
(display (emit-real "1.7")) (newline)
(display "testing function: emit-complex") (newline)
(display (emit-complex "4+2i")) (newline)
(display "testing function: emit-vname") (newline)
(display (emit-vname '("a" a))) (newline)
(display "testing function: emit-fname") (newline)
(display (emit-fname '("sin" sin))) (newline)
(display "testing function: emit-opname") (newline)
(display (emit-opname '("+" +))) (newline)
(display "testing function: emit-typename") (newline)
(display (emit-typename '("Integer" i#nteger))) (newline)
(display "testing function: emit-type") (newline)
(display (emit-type
     '(("PTN Identifier: Type" . ("List" l#ist))
       (("PTN Type" . (("PTN Identifier: Type" . ("T" t#)) () "gr")))
       "cr")
     )) (newline)
(display "testing function: emit-var") (newline)
(display (emit-var
     '(("PTN Identifier: Variable" .("x" x))
       ("PTN Expression" . (("PTN Literal: Integer" . "0") ())))
     )) (newline)
(display "testing function: emit-func") (newline)
;(display (emit-func (val))) (newline)
(display "testing function: emit-op") (newline)
;(display (emit-op (val))) (newline)
(display "testing function: emit-set!") (newline)
(display (emit-set!
     '(("PTN Identifier: Variable" . ("x" x))
       ("PTN Expression" . (("PTN Literal: Integer" . "0") ())))
     )) (newline)
(display "testing function: emit-let") (newline)
```

```
(display (emit-let
     '((("PTN Statement: set!" . (
        ("PTN Identifier: Variable" . ("i" i))
        ("PTN Expression" . (("PTN Literal: Integer" . "0") ()))))
        ("PTN Statement: set!" . (
        ("PTN Identifier: Variable" . ("r" i))
        ("PTN Expression" . (("PTN Literal: Real" . "1.8") ())))))
       ("PTN Parse Tree" . ()))
     )) (newline)
(display "testing function: emit-begin") (newline)
(display (emit-begin
     '("PTN Parse Tree" . (
       ("PTN Expression" . (("PTN Identifier: Variable" . ("x" x)) ()))
       ("PTN Expression" . (("PTN Identifier: Variable" . ("y" y)) ()))))
     )) (newline)
(display "testing function: emit-if") (newline)
(display (emit-if
     '(("PTN Expression" . (("PTN Identifier: Variable" . ("t" t)) ()))
       ("PTN Parse Tree" . (
        ("PTN Expression" . (("PTN Identifier: Variable" . ("c" c)) ()))))
       ("PTN Parse Tree" . (
        ("PTN Expression" . (("PTN Identifier: Variable" . ("a" a)) ()))))
     ))) (newline)
(display "testing function: emit-case") (newline)
(display (emit-case
     '(("PTN Expression" . (("PTN Literal: String" . "key") ()))
       (((("PTN Expression" . (("PTN Literal: String" . "a") ()))
          ("PTN Expression" . (("PTN Literal: String" . "b") ()))) .
         ("PTN Parse Tree" . (
          ("PTN Expression" .(("PTN Identifier: Variable" . ("x" x)) ()))
          ("PTN Expression" .(("PTN Identifier: Variable" . ("y" y)) ())))))))
         ("PTN Parse Tree" . (
          ("PTN Expression" .(("PTN Identifier: Variable" . ("z" z))
           ())))))
     )) (newline)
(display "testing function: emit-cond") (newline)
(display (emit-cond
     '(((("PTN Expression" . (("PTN Literal: String" . "b") ())) .
         ("PTN Parse Tree" . (
          ("PTN Expression" .(("PTN Identifier: Variable" . ("x" x)) ()))
          ("PTN Expression" .(("PTN Identifier: Variable" . ("y" y)) ())))))))
         ("PTN Parse Tree" . (
          ("PTN Expression" .(("PTN Identifier: Variable" . ("z" z))
           ())))))
     )) (newline)
(display "testing function: emit-return") (newline)
(display (emit-return
     '("PTN Expression" .(("PTN Identifier: Variable" . ("x" x)) ()))
     )) (newline)
(display "testing function: emit-while") (newline)
(display (emit-while
     '(("PTN Expression" . (("PTN Literal: Boolean" . "true") ()))
       ("PTN Parse Tree" . (
        ("PTN Expression" .(("PTN Identifier: Variable" . ("x" x)) ()))
        ("PTN Expression" .(("PTN Identifier: Variable" . ("y" y)) ())))))
     )) (newline)
(display "testing function: emit-repeat") (newline)
(display (emit-repeat
     '(("PTN Parse Tree" . (
         ("PTN Expression" .(("PTN Identifier: Variable" . ("x" x)) ()))
```

```
                ("PTN Expression" .(("PTN Identifier: Variable" . ("y" y)) ())))
               ("PTN Expression" . (("PTN Literal: Boolean" . "true") ())))
            )) (newline)
        (display "testing function: emit-for1") (newline)
        (display (emit-for1
            '(("PTN Identifier: Variable" . ("i" i))
              ("PTN Expression" .(("PTN Literal: Integer" . "-7") ()))
              ("PTN Expression" .(("PTN Literal: Integer" . "49") ()))
              ("PTN Parse Tree" . (
               ("PTN Expression" .(("PTN Identifier: Variable" . ("x" x)) ()))
               ("PTN Expression" .(("PTN Identifier: Variable" . ("y" y)) ()))))
            )) (newline)
        (display "testing function: emit-for") (newline)
        (display (emit-for
            '(("PTN Identifier: Variable" . ("i" i))
              ("PTN Expression" .(("PTN Literal: Integer" . "1") ()))
              ("PTN Expression" .(("PTN Literal: Integer" . "3") ()))
              ("PTN Expression" .(("PTN Literal: Integer" . "37") ()))
              ("PTN Parse Tree" . (
               ("PTN Expression" .(("PTN Identifier: Variable" . ("x" x)) ()))
               ("PTN Expression" .(("PTN Identifier: Variable" . ("y" y)) ()))))
            )) (newline)
        (display "testing function: emit-symtab") (newline)
        (display (emit-symtab 'val)) (newline)
        (display "testing function: update-global") (newline)
        (display (update-global
            '("PTN Symbol Table" .(
              (("PTN Identifier: Variable" . ("x" x))
              ("PTN Type" . (("PTN Identifier: Type" . ("int" int)) () "gr")))))
            )) (newline)
        (display "testing function: update-funcmap") (newline)
        (pp func-map) (newline)
        (display (update-funcmap
            '(("PTN Identifier: Function" . ("add1" add1))
              ("PTN Identifier: Function" . ("++" ++)))
            )) (newline)
        (pp func-map) (newline)
        (display "testing function: update-typemap") (newline)
        (pp type-map) (newline)
        (display (update-typemap
            '(("PTN Identifier: Type" . ("mtype" mtype))
               ("PTN Type" . (
                ("PTN Identifier: Type" . ("Vector" v#ector))
                (("PTN Type" . (
                 ("PTN Identifier: Type" . ("List" l#ist))
                 (("PTN Type" . (("PTN Identifier: Type" .("T" t#)) () "gr")))
                 "cr")))
                "cr")))
            )) (newline)
        (pp type-map) (newline)
        )
```

◇

"test/in/code-ext.in" 137 ≡

```
(begin ⟨load source files 141⟩
        (display "testing function: emit-array-ref") (newline)
        (display (emit-array-ref
            '(("PTN Expression" .(("PTN Identifier: Variable" . ("x" x))))
              ("PTN Expression" .(("PTN Literal: Integer" . "37") ())))
            )) (newline)
        (display "testing function: emit-newline") (newline)
```

```
        (display (emit-newline '())) (newline)
        (display "testing function: emit-display") (newline)
        (display (emit-display
             '(("PTN Expression" .(("PTN Identifier: Variable" . ("x" x))))
               ("PTN Expression" .(("PTN Literal: Integer" . "37") ())))
             )) (newline)
        (display "testing definition: sfunc-list") (newline)
        (pp sfunc-list) (newline)
        )
◇
```

"test/in/misc.in" 138 ≡

```
(begin ⟨load source files 141⟩
        (display "testing function: id-transform") (newline)
        (display (id-transform 's#uck$$*)) (newline)
        (display "testing function: get-id-tname") (newline)
        (display (get-id-tname
             '("PTN Identifier: Type" . ("Vector" v#ector)))) (newline)
        (display "testing function: get-id-oname") (newline)
        (display (get-id-oname
             '("PTN Identifier: Type" . ("Vector" v#ector)))) (newline)
        (display "testing function: make-ptn") (newline)
        (display (make-ptn 'vname-id)) (newline)
        (display "testing function: get-ptn") (newline)
        (display (get-ptn (make-ptn 'vname-id) "hello")) (newline)
        (display "testing function: set-ptn-val!") (newline)
        ;(display (set-ptn-val! (pt-node val))) (newline)
        (display "testing function: add-ptn-val!") (newline)
        ;(display (add-ptn-val! (pt-node obj))) (newline)
        (display "testing function: ptn?") (newline)
        (display (ptn? (make-ptn 'vname-id))) (newline)
        (display "testing function: parse-tree?") (newline)
        (display (parse-tree? (make-ptn 'pt-id))) (newline)
        (display "testing function: literal?") (newline)
        (display (literal? (make-ptn 'bool-id))) (newline)
        (display "testing function: expr?") (newline)
        (display (expr? (make-ptn 'expr-id))) (newline)
        (display "testing function: id?") (newline)
        (display (id? (make-ptn 'vname-id))) (newline)
        (display "testing function: type-id?") (newline)
        (display (type-id? (make-ptn 'typename-id))) (newline)
        (display "testing function: func-id?") (newline)
        (display (func-id? (make-ptn 'fname-id))) (newline)
        (display "testing function: op-id?") (newline)
        (display (op-id? (make-ptn 'opname-id))) (newline)
        (display "testing function: var-id?") (newline)
        (display (var-id? (make-ptn 'vname-id))) (newline)
        (display "testing function: func-def?") (newline)
        (display (func-def? (make-ptn 'func-id))) (newline)
        (display "testing function: op-def?") (newline)
        (display (op-def? (make-ptn 'op-id))) (newline)
        (display "testing function: type?") (newline)
        (display (type? (make-ptn 'type-id))) (newline)
        (display "testing function: symbol-table?") (newline)
        (display (symbol-table? (make-ptn 'symtab-id))) (newline)
        (display "testing function: make-symtab-entry") (newline)
        (pp (make-symtab-entry
             global-symtab '((i . i#nteger) (f . f#ield)) )) (newline)
        (set! global-symtab (make-symtab-entry
                             global-symtab '((i . i#nteger) (f . f#ield))))
        (display "testing function: get-idtype") (newline)
```

```
(display (get-idtype "i")) (newline)
(display "testing function: get-idtypet") (newline)
(display (get-idtypet "f" global-symtab)) (newline)
(display "testing function: merge-gentypes") (newline)
(display (merge-gentypes '() global-symtab)) (newline)
(display "testing function: gen-types") (newline)
(display (gen-types
     '("PTN Type" . (
         ("PTN Identifier: Type" . ("Vector" v#ector))
         (("PTN Type" . (
          ("PTN Identifier: Type" . ("List" l#ist))
          (("PTN Type" . (("PTN Identifier: Type" .("T" t#)) () "gr")))
          "cr")))
         "cr"))
      )) (newline)
(display "testing function: dassq") (newline)
(display (dassq 'x '((a . 1) (x . x)))) (newline)
(display (dassq 'y '((a . 1) (x . x)))) (newline)
(display "testing function: dassv") (newline)
(display (dassv 'x '((a . 1) (x . x)))) (newline)
(display (dassv 'y '((a . 1) (x . x)))) (newline)
(display "testing function: dassoc") (newline)
(display (dassoc 'x '((a . 1) (x . x)))) (newline)
(display (dassoc 'y '((a . 1) (x . x)))) (newline)
(display "testing function: get-nval") (newline)
(display (get-nval 'x 2 '((x 1 2 3)))) (newline)
(display "testing function: list-memb?") (newline)
(display (list-memb? "s" '("a" "b" "s"))) (newline)
(display (list-memb? "s" '("a" "b" "c"))) (newline)
(display "testing function: flist-memb?") (newline)
;(display (flist-memb? (obj l))) (newline)
(display "testing function: debug-mes") (newline)
(display (debug-mes 0 "pre" 'obj "suf")) (newline)
(display "testing function: error-mes") (newline)
(display (error-mes "--err--")) (newline)
(display "testing function: warn-mes") (newline)
(display (warn-mes "--warn--")) (newline)
(display "testing function: ind-inc!") (newline)
(display (ind-inc!)) (newline)
(display "testing function: ind-dec!") (newline)
(display (ind-dec!)) (newline)
(display "testing function: emit-strin") (newline)
(display (emit-strin "line1")) (newline)
(display "testing function: emit-strn") (newline)
(display (emit-strn "line2")) (newline)
(display "testing function: emit-stri") (newline)
(display (emit-stri "line3")) (newline)
(display "testing function: derive-oname") (newline)
(display (derive-oname "in.scm")) (newline)
(display (derive-oname "in.scmx")) (newline)
(display "testing function: add-prefix-code!") (newline)
(display (add-prefix-code! "pc-str")) (newline)
(display "testing function: write-prefix-code") (newline)
(display (write-prefix-code)) (newline)
(display "testing function: recurse-pc") (newline)
;(display (recurse-pc (pc))) (newline)
)
```

◇

"test/in/scm2cpp.in" 139 ≡

(begin ⟨load source files 141⟩

```
(display "testing definition: enable-stdin") (newline)
(display enable-stdin) (newline)
(display "testing definition: enable-stdout") (newline)
(display enable-stdout) (newline)
(display "testing definition: icount") (newline)
(display icount) (newline)
(display "testing definition: istep") (newline)
(display istep) (newline)
(display "testing definition: ansi-c++") (newline)
(display ansi-c++) (newline)
(display "testing definition: enable-tm") (newline)
(display enable-tm) (newline)
(display "testing definition: enable-fm") (newline)
(display enable-fm) (newline)
(display "testing definition: enable-map-decl") (newline)
(display enable-map-decl) (newline)
(display "testing definition: prefix-code") (newline)
(display prefix-code) (newline)
(display "testing definition: recurse-level") (newline)
(display recurse-level) (newline)
(display "testing definition: scope") (newline)
(display scope) (newline)
(display "testing definition: error-cnt") (newline)
(display error-cnt) (newline)
(display "testing definition: warn-cnt") (newline)
(display warn-cnt) (newline)
(display "testing definition: error-alg") (newline)
(display error-alg) (newline)
(display "testing definition: debug-level") (newline)
(display debug-level) (newline)
(display "testing definition: global-symtab") (newline)
(display global-symtab) (newline)
(display "testing definition: input-symtab") (newline)
(display input-symtab) (newline)
(display "testing definition: output-symtab") (newline)
(display output-symtab) (newline)
(display "testing definition: local-symtab") (newline)
(display local-symtab) (newline)
(display "testing definition: enable-case-sensitive") (newline)
(display enable-case-sensitive) (newline)
(display "testing definition: id-map") (newline)
(pp id-map) (newline)
(display "testing definition: id-map-long") (newline)
(pp id-map-long) (newline)
(display "testing definition: op-list") (newline)
(pp op-list) (newline)
(display "testing definition: type-list") (newline)
(pp type-list) (newline)
(display "testing definition: type-support") (newline)
(pp type-support) (newline)
(display "testing definition: val-list") (newline)
(pp val-list) (newline)
(display "testing definition: func-map") (newline)
(pp func-map) (newline)
(display "testing definition: type-map") (newline)
(pp type-map) (newline)
(display "testing definition: root-dir") (newline)
(pp root-dir) (newline)
(display "testing function: scm2cpp") (newline)
;(display (scm2cpp ())) (newline)
```

```
            (display "testing definition: pt-nodeids") (newline)
            (pp pt-nodeids) (newline)
            (display "testing definition: emitter-list") (newline)
            (pp emitter-list) (newline)
            )
     ◇
```

⟨load source files 141⟩ ≡

```
     (load "src/parse.scm")
     (load "src/code.scm")
     (load "src/code-ext.scm")
     (load "src/misc.scm")
     (load "src/scm2cpp.scm")
     ⟨read init files 39f⟩
     ◇
```

Macro referenced in scraps 133b, 134ab, 137, 138, 139.

Appendix F

Indices

F.1 File Names

F.2 Macros

F.3 Identifiers

Bibliography

[1] Aonix: *Object Ada for Windows*, San Francisco, 1997.
available at: `http://www.thomsoft.com/Products/Ada/factsht.html`

[2] Joseph A. Bank, Barbara Liskov, Andrew C. Myers: *Parameterized Types and Java*, Massachusetts Institute of Technology, 1996.

[3] John J. Barton, Lee R. Nackman: *Scientific and Engineering C++*, Addison-Wesley Publishing Company, 1994.

[4] William Clinger, Jonathan Rees (Editors) et. al.: *Revised^4 Report on the Algorithmic Language Scheme*, 1991.
available at: `ftp://ftp.swiss.ai.mit.edu/pub/scheme-reports/r4rs.ps`

[5] Todd R. Eigenschink, Dave Love, Aubrey Jaffer: *SLIB, The Portable Scheme Library*, 1997.
available at: `ftp://ftp.swiss.ai.mit.edu/pub/scm/slib2a7.tar.gz`

[6] Dawson E. Engler and Todd A. Proebsting: *DCG: An Efficient, Retargetable Dynamic Code Generation System*, ACM SIGPLAN Notices, p. 263-272, November 1994.

[7] David Flanagan: *Java in a Nutshell*, O'Reilly & Associates, Inc., 1996.

[8] Marc Feeley: *Gambit-C, version 2.5.1*, 1997.
available at: `ftp://ftp.iro.umontreal.ca/pub/parallele/gambit`

[9] Christopher Fraser, David Hanson: *A Retargetable C Compiler: Design and Implementation*, The Benjamin/Cummings Publishing Company, Inc., 1995.

[10] Mark Galassi: *Guile 1.0*, 1997.
available at: `ftp://prep.ai.mit.edu/pub/gnu/guile-1.0.tar.gz`

[11] James Gosling, Bill Joy, Guy Steele: *The Java Language Specification*, Addison-Wesley, 1996.
available at: `ftp://ftp.javasoft.com/docs/langspec-1.0`

[12] Intermetrics: *AppletMagic*, 1997.
available at: `http://www.appletmagic.com`

[13] Aaron Kershenbaum, David Musser, Alexander Stepanov: *Higher Order Imperative Programming*, Rensselaer Polytechnic Institute, Technical Report 88-10, April 1988.

[14] Andrew Koenig (contact person): *Working Paper for Draft Proposed International Standard for Information Systems–Programming Language C++*, Accredited Standards Committee X3, INFORMATION PROCESSING SYSTEMS, Doc No:X3J16/96-0225 WG21/N1043, December 1996.
available at: `http://www.setech.com/x3.html`

[15] Helmut Kopka: *LaTeX, Eine Einführung*, Addison-Wesley Publishing Company, 4. Auflage, 1992.

[16] Stanley B. Lippman: *C++ Primer, 2nd Edition*, Addison-Wesley Publishing Company, 1991.

[17] Rüdiger G. K. Loos and George E. Collins: *Revised Report on the Algorithm Description Language ALDES*, Technischer Report WSI 92-14, Fakultät für Informatik der Eberhard-Karls-Universität Tübingen, 1992.

[18] Rüdiger G. K. Loos and George E. Collins: *Specifications and Index of SAC-2 Algorithms*, Technischer Report WSI 90-4, Fakultät für Informatik der Eberhard-Karls-Universität Tübingen, 1990.

[19] Adrian McCarthy: *Unit and Regression Testing*, Dr. Dobb's Journal, p. 18-20, February 1997.

[20] Brian L. Meek: *Programming Languages: Towards Greater Commonality*, ACM SIGPLAN Notices, p. 49-57, April 1994.

[21] Brian L. Meek: *A taxonomy of datatypes*, ACM SIGPLAN Notices, p. 159-167, September 1994.

[22] David R. Musser, Atul Saini: *STL Tutorial and Reference Guide*, Addison-Wesley Publishing Company, 1996.

[23] Object Space Developer Team: *JGL - the Generic Collection Library*, 1996.
available at `http://www.objectspace.com/jgl`

[24] Martin Odersky, Philip Wadler: *Pizza into Java, Translating theory into practice*, Universities of Karlsruhe and Glasgow, 1996.
available at: `http://wwwipd.ira.uka.de/~pizza`

[25] Sun Microsystems: *The Java Development Kit (JDK)*, 1997.
available at: `http://java.sun.com/products/jdk/1.1`

[26] Tools.h++ Team: *Tools.h++, Foundation Class Library for C++ Programming, Class Reference, Version 7*, Rogue Wave Software, Corvallis, Oregon USA, March 1996.

[27] Sibylle Schupp: *Generic programming* — SuchThat *one can build an algebraic library*, Dissertation am Wilhelm-Schickard-Institut für Informatik der Eberhard-Karls-Universität Tübingen, 1996.

[28] Silicon Graphics Developer Team: *Standard Template Library Programmer's Guide*, 1996.
available at: `http://www.sgi.com/Technology/STL`

[29] George Springer and Daniel P. Friedman: *Scheme and the Art of Programming*, The MIT Press, Massachusetts Institute of Technology, 1990.

[30] Alexander Stepanow, Meng Lee: *The Standard Template Library*, Hewlett-Packard Company, Palo Alto, 1995.

[31] Bjarne Stroustrup: *The C++ Programming Language, 2nd Edition*, Addison-Wesley Publishing Company, 1991.

[32] S. Tucker Taft, Robert A. Duff (Editors): *Ada 95 Reference Manual*, Lecture Notes in Computer Science, Vol. 1246, Springer-Verlag, 1997.

[33] Sebastian Wedeniwski: *Generische Matrizen und die Implementation schneller Matrixoperationen*, Diplomarbeit am Wilhelm-Schickard-Institut für Informatik der Eberhard-Karls-Universitä Tübingen, Oktober 1996.

[34] Sebastian Wedeniwski: *Piologie, Eine exakte arithmetische Bibliothek in C++*, Technischer Report WSI 96-35, Wilhelm-Schickard-Institut für Informatik der Eberhard-Karls-Universität Tübingen, 1996.

[35] William M. Waite, Gerhard Goos: *Compiler construction*, Springer-Verlag, 1984.

[36] Niklaus Wirth: *Grundlagen und Techniken des Compilerbaus*, Addison-Wesley, New York, 1984.